Manuel Gozzi, Luisa Chiu

Ventura

12/11/2023

Using Computer Vision to Determine Robot Position and Orientation and Algorithm Development for a Maze Solving Robot

Introduction:

We implemented a maze-solving algorithm that also tracks the rotation and position of a robot using an overhead camera. The problem we are solving is both maze automation and the capacity to replace onboard robot sensors with cheap cameras that stitch together pathways solely using video footage and markers on the robots. This is an interesting problem since it involves isolating the robot from its surroundings, calculating the best path, monitoring the position of the robot, interfacing via Bluetooth or other means of remote data transmission, and ensuring that the path is smooth and optimal. This is a challenging problem since environments can be pretty dynamic. This will affect the thresholding that is required to discriminate against boundaries. Our approach was to threshold the boundaries and find the best possible path from the start to the finish. We then pad this path to account for the width of the robot itself while traveling across this path to ensure that it will avoid any interaction with the wall at all. After this, our goal was to create this scenario in real life by dynamically dispatching commands based on the live position and orientation of a physical robot in a maze. Unfortunately due to technical issues, this feature wasn't able to be fully implemented.

Background

(Rastogi, 2020) Previous approaches to this problem involve voxelizing the maze into specific signatures. For example, a wall on the left, a wall on the right, a wall on both sides, a dead end, etc. This approach would apply to our scenario if the data was perfectly curated as a perfectly square map. This, unfortunately, isn't well suited for our implementation since lighting conditions, incorrect dimensions, and nonsquare paths would be common in the problem that we're trying to solve. Instead, we decided to utilize breadth-first search, which is computationally more expensive but allows us to solve for non-rectangular paths. For robot tracking, we found a paper that talked about using AruCo marker data to assist in navigating an unmanned aerial system. (Qiu, Z) Their idea was to use a camera to track the AruCo marker pose for landing purposes. (Qiu, Z) They communicated with the UAV using remote commands to move it around in conjunction with AruCo marker data to determine the pose of the vehicle. (Qiu, Z) This seems like a valid approach for our scenario, remotely dispatching the commands based on the pose of the robot.

Data and methods

Our first data points involved using images of mazes straight from Google images. This was done to expedite the development of our path-finding algorithm since a real-life maze had not yet been created. Moving on from this system, we expanded to real-life depictions of a maze.

(Akshay, 2017) We implemented a breadth-first search algorithm for the path-solving portion of this project. We combined this with the orientation and position info based on the markers on the robot to create a live vector of the robot's orientational data.

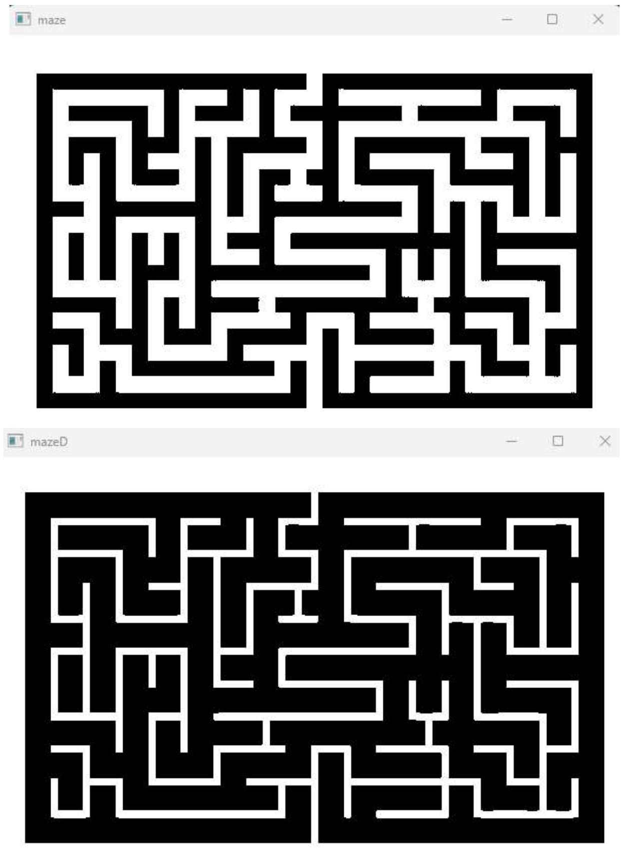


Figure 1. Maze Erosion

Instead of detecting boundaries after solving the maze, we decided to erode the map before running the path-finding algorithm. Since we know that we don't want to create a path that is not possible, in other words too small, and would like to not compute for unnecessary points, the points that are too close to the walls, we decided to erode the maze mapping. This involved using a square kernel that turned previously white pixels in the maze into black pixels. This essentially reduced the area of the maze that had to be analyzed. The result after dilating the original maze (maze) can be seen in the eroded maze (mazeD).

For the real-life maze application, we created a maze using green tape and white paper, as seen in Figure 2 below.

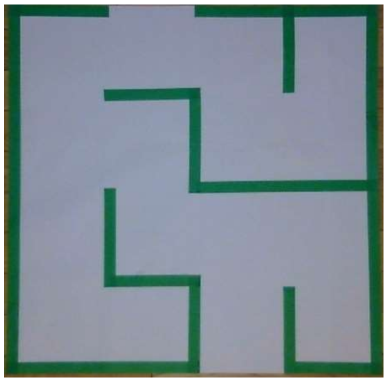


Figure 2. Real-Life 4x4 Maze

The robot was DFRobot's Maqueen wheeled robot that uses a micro-bit to control its motors and other sensors we did not need to utilize. We designed and 3-D printed a bracket for mounting an AruCo marker on top of the robot. The AruCo marker we used can be seen below, along with it taped onto the bracket mounted on top of the robot.

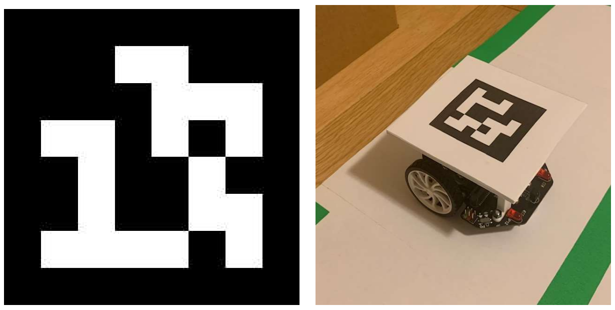


Figure 3. AruCo Marker Mounted on Robot

Next, we found the location and angles of the robot using the AruCo markers and image pixels. The top left corner in the maze image is defined as the origin (0,0), and calculating the center of the AruCo marker was done using OpenCV's Aruco detectMarkers() function that yielded the coordinates of each marker corner.

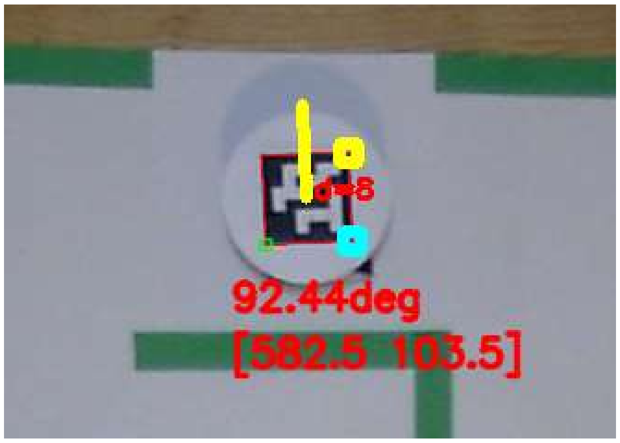


Figure 4. AruCo Parameters

We used an atan2 function to determine the angle rotation of the robot, which corresponds to the following angle coordinates.

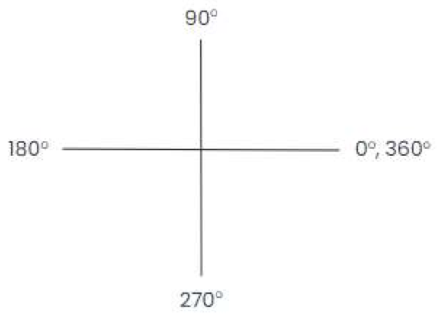


Figure 5. Angle Coordinate System

In terms of the remote dispatch portion, we used MakeCode to flash code onto the micro:bit. We tried to interact with the micro:bit Bluetooth feature, but it did not work. Our first issue was that we had a v1 micro:bit, which ran out of memory quickly when we used Bluetooth. Additionally, we tried to use MakeCode's extension, BlockyTalky-BLE, which allows the micro:bit to receive data via Bluetooth, but it did not work either because the micro:bit ran out of memory before the extension could start. We wanted to use it to send remote commands to the micro:bit from our computers, but without it working properly and with no memory on the micro:bit, we were unable to get its characteristic UUID to send any data.

Due to the Bluetooth not working properly, we wanted to try hardcoding the robot to navigate through the maze for position and rotation analysis, but the gears in the motors had a clicking issue that threw off the wheel rotation and caused over or under-rotation. As an alternative, we decided to take static images as if the robot were navigating through the maze to analyze the position and orientation of the robot. We took 13 images imitating each step the robot would need to take if it were to solve the maze.

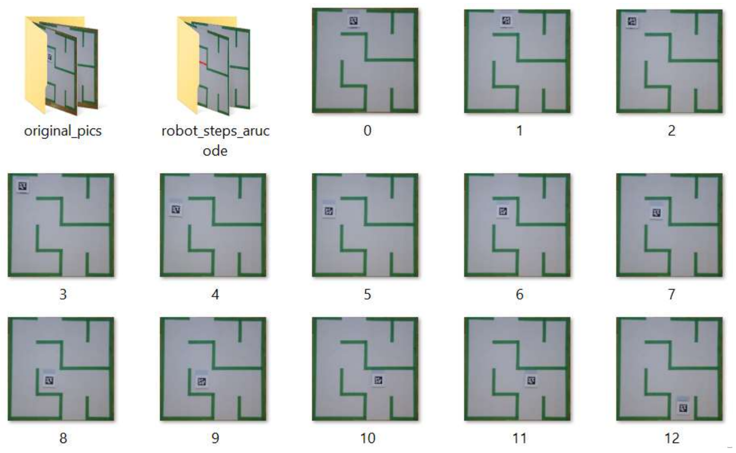


Figure 6. Snapshot of 13 Steps Taken by Robot

For example, steps 0 to 1: rotate 90 degrees clockwise; steps 2 to 3: no rotation, move forward, etc. These were static images that did not display intermediate movement when the robot moved from one grid space to another.

Next, we used pixel coordinates to draw gridlines on an empty maze image to confirm the robot location moved to its expected grid space. As seen below, there is a comparison between OpenCV image axes and Python Numpy array axes. Since this computer vision project is to assist Luisa's thesis, we chose to classify the grid spaces according to the numpy array axes. For example, if the robot was located in the top right grid space, it is at coordinate (0, 3) (Numpy coordinates) rather than (3,0) (OpenCV coordinates).

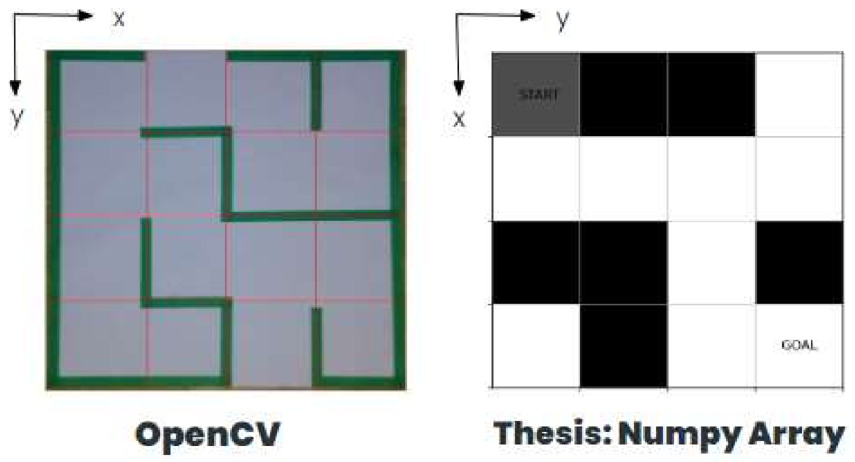


Figure 7. OpenCV vs Numpy Array Coordinate System Differences

We developed a function that checked pixel coordinates against grid spaces to output a tuple representing maze coordinates. For the rotation portion, we compared the frames for two adjacent moves, finding the angle differences and direction of rotation (i.e., clockwise or counter-clockwise). An example of adjacent move frames being compared can be seen below.

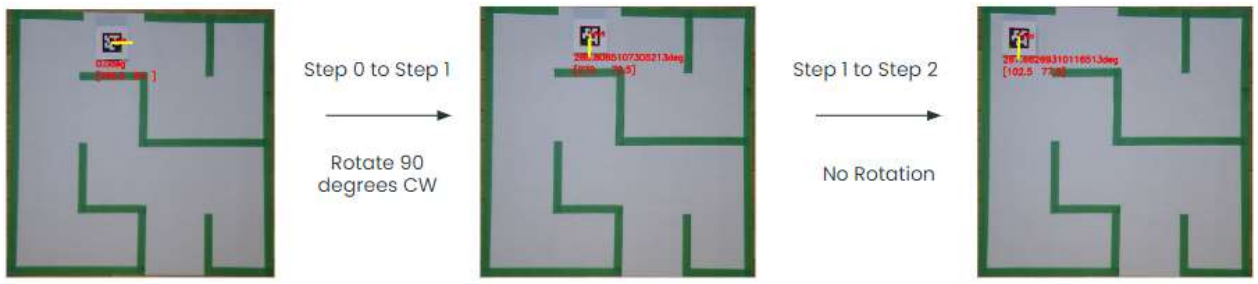
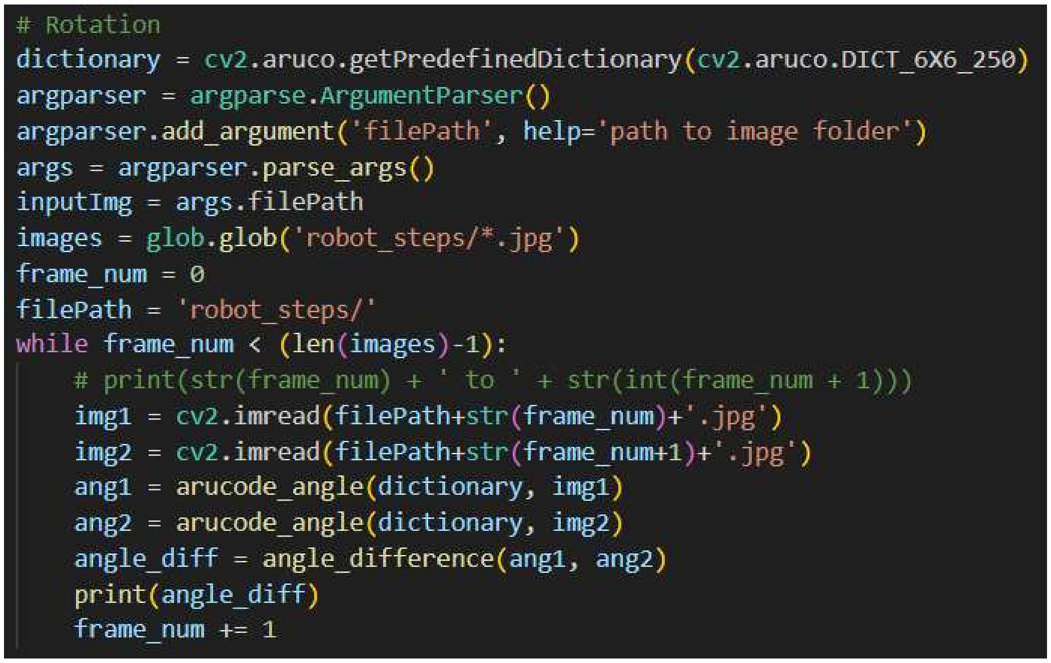


Figure 8. Robot Rotation: Steps 0 to 2

The angle differences will not strictly be 90 degrees, so a tolerance of 5 degrees was used to determine whether or not the robot rotated. The code for comparison between the two frames is shown in Figure 9.



```python
# Rotation
dictionary = cv2.aruco.getPredefinedDictionary(cv2.aruco.DICT_6X6_250)
argparser = argparse.ArgumentParser()
argparser.add_argument('filePath', help='path to image folder')
args = argparser.parse_args()
inputImg = args.filePath
images = glob.glob('robot_steps/*.jpg')
frame_num = 0
filePath = 'robot_steps/'
while frame_num < (len(images)-1):
    # print(str(frame_num) + ' to ' + str(int(frame_num + 1)))
    img1 = cv2.imread(filePath+str(frame_num)+'.jpg')
    img2 = cv2.imread(filePath+str(frame_num+1)+'.jpg')
    ang1 = arucode_angle(dictionary, img1)
    ang2 = arucode_angle(dictionary, img2)
    angle_diff = angle_difference(ang1, ang2)
    print(angle_diff)
    frame_num += 1
```

Figure 9. Snapshot of Code for Angle Difference Analysis

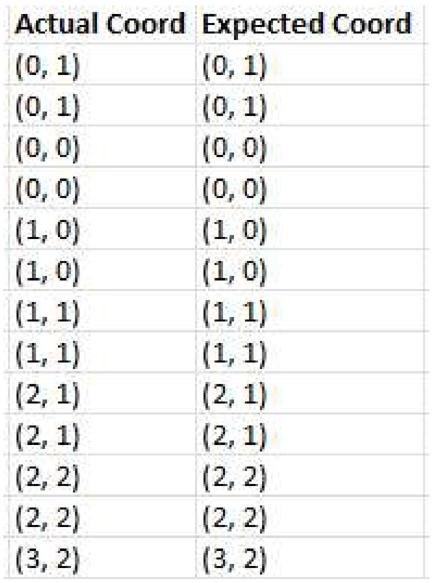
We used a while loop and iterated through two frames at once to find angle differences and directions of rotation. In the next section, we will discuss the results obtained from using our methods.

Evaluation

Since our problem is well-defined, our metrics for optimal path finding were quite straightforward. By definition, breadth-first search does indeed find the shortest path. We evaluated our results by testing to see if the path-finding algorithm can find a path and reconstruct the optimal path back to the starting position. Analyzing the accuracy of images and environments of less-than-ideal conditions was another story. If a material wasn’t in the color profile that our system desired, it would be incorrectly marked as a boundary when it was irrelevant to the boundary mapping.

In terms of the location and rotation methods, we defined expected values for coordinates, angles, and direction by looking at the 13 frames and figuring out directions for the robot to successfully navigate the maze.

For location detection, we analyzed whether or not the program recognized the robot in its expected grid space. The table displaying the actual and expected coordinates can be seen below.

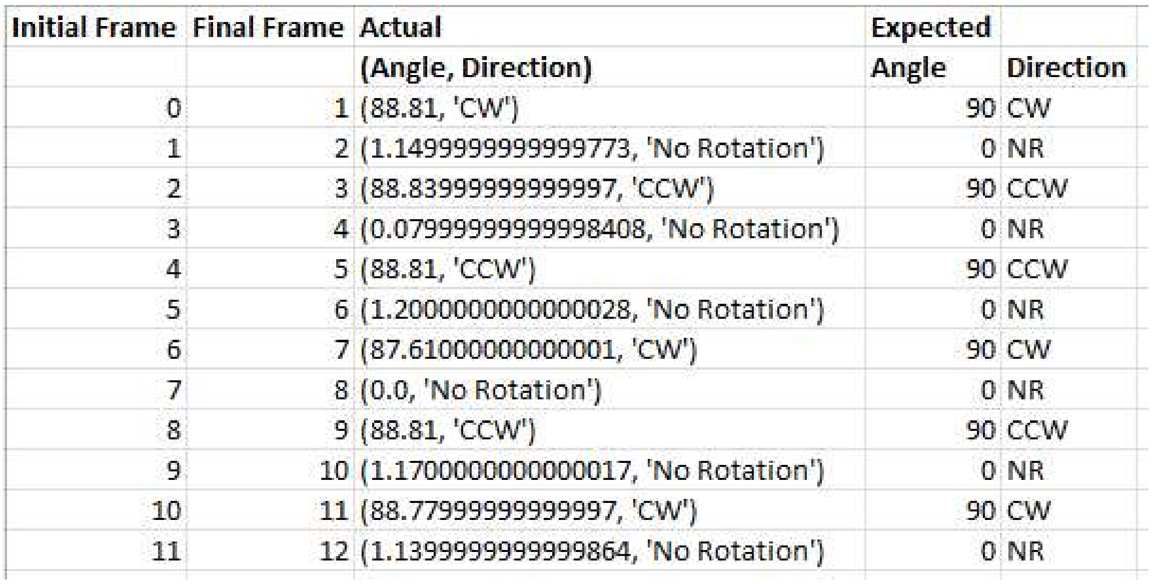


Actual Coord	Expected Coord
(0, 1)	(0, 1)
(0, 1)	(0, 1)
(0, 0)	(0, 0)
(0, 0)	(0, 0)
(1, 0)	(1, 0)
(1, 0)	(1, 0)
(1, 1)	(1, 1)
(1, 1)	(1, 1)
(2, 1)	(2, 1)
(2, 1)	(2, 1)
(2, 2)	(2, 2)
(2, 2)	(2, 2)
(3, 2)	(3, 2)

Table 1. Gridspace Coordinates Data

The table showed that the actual and expected data points matched completely. As stated earlier, the gridlines for the maze were determined using image pixels and logic was implemented to check if the marker center was within a grid square. This grid coordinate checking procedure was a good confirmation method to make sure the has reached its expected location in the physical maze space.

For the rotation method, the following data table displays the actual and expected angle and direction between two frames.



Initial Frame	Final Frame	Actual	Expected	
		(Angle, Direction)	Angle	Direction
0	1	(88.81, 'CW')	90	CW
1	2	(1.1499999999999773, 'No Rotation')	0	NR
2	3	(88.83999999999997, 'CCW')	90	CCW
3	4	(0.07999999999998408, 'No Rotation')	0	NR
4	5	(88.81, 'CCW')	90	CCW
5	6	(1.2000000000000028, 'No Rotation')	0	NR
6	7	(87.61000000000001, 'CW')	90	CW
7	8	(0.0, 'No Rotation')	0	NR
8	9	(88.81, 'CCW')	90	CCW
9	10	(1.1700000000000017, 'No Rotation')	0	NR
10	11	(88.77999999999997, 'CW')	90	CW
11	12	(1.1399999999999864, 'No Rotation')	0	NR

Table 2. Robot Rotation Angle and Direction Data

As seen above, the angle differences for actual and expected were very similar, and the directions matched completely. Due to the nature of manually rotating the robot, the angles were not exactly 90 degrees as expected, but the angle values were close enough and small errors were negligible. Even if we were to hardcode the robot moving through the maze, it would not be a perfect 90-degree turn without closed-loop control or feedback from a live-feed video. The angle difference tolerance of 5 degrees worked well, as the angle differences of about 1 degree were counted as no rotation.

To conclude this section, our data and analysis show that our methods were successful in solving the maze efficiently and checking the results of the robot actions taken (rotation and location).

Results

Overall our results seemed to be promising. Though there is a lot of room for improvement, such as skipping over data points strategically to reduce computational costs, our system does provide a well-suited solution for paths short to medium-length paths.

As stated previously, our first iteration of the project involved using Bluetooth and remote dispatch. However, since that did not work out, our second iteration of the project using static images allowed us to continue analyzing the robot's location and rotation. The location and rotation evaluations shown in the section above indicated that the AruCo marker was a great method for tracking the robot. Initially, we were looking at the world coordinates of the robot using the AruCo marker, hoping to implement logic such that if the robot moved x millimeters, it moved to the next grid space. However, trying to translate the corner markers proved to be a challenge, and it was not yielding great results due to camera distortion, which made the data noisy. After consulting with our professor, we decided to use image pixels rather than real-world coordinates. This method was very successful, as our actual and expected values matched.

Outlook

At first, our expectations were high for this project since we wanted to develop a machine learning system that could solve the maze instead with onboard sensors, however, this turned out to be far beyond our reach due to time constraints imposed by competing coursework. We would have enjoyed implementing the remote execution system on a real robot to see it complete the maze however we have the fundamental steps towards that goal established. Even something as simple as upgrading to a new version of the microbit might make this possible. Most of the research and code that we needed has already been written to dispatch these commands, along with interpreting in real time on the robot side. We learned a good deal from this project, especially in the realm of hardware and wireless connectivity. Being able to explore how Bluetooth functions for devices such as a microbit was incredibly interesting. Exposing the specific UUIDs for communication and creating a handshake was surprisingly simple and fun to figure out via numerous failed Python scripts. If this project's time frame were to be expanded, we would certainly implement the remote dispatch feature and hopefully work on making a more robust filtering system to avoid taking into account non-obstacles in our path-finding system.

References

BFS Maze Solver using OpenCV. (2017). YouTube. Retrieved December 11, 2023, from https://youtu.be/-keGVhYmY2c.

Rastogi, O. (2020a, March 28). Maze Finder using OpenCV Python. Medium. https://omrastogi.medium.com/maze-finder-using-opencv-python-7dc5023ccd02

Qiu, Z., et al. “A Global Aruco-Based LIDAR Navigation System for UAV Navigation in GNSS-Denied Environments.” MDPI, Multidisciplinary Digital Publishing Institute, 19 Aug. 2022, doi.org/10.3390/aerospace9080456