

ECE 445
SENIOR DESIGN LABORATORY
FINAL REPORT

Project Polaris: An Air And Ground Autonomous Collaboration System

Team #10

XINYU LIAN
(lian7@illinois.edu)
NACHUAN WANG
(nachuan3@illinois.edu)
NENG XIONG
(nengx2@illinois.edu)
ZHONGBO ZHU
(zhongbo2@illinois.edu)

Instructor: Prof. Hongwei Wang

TA: Chunzeng Luo

May 14, 2022

Contents

1	Introduction	1
1.1	Purpose	1
1.2	Functionality	1
1.3	Subsystem Overview	1
2	Design	2
2.1	Equations & Simulations	2
2.1.1	Motor Control	2
2.1.2	Vision	3
2.2	Design Alternatives	3
2.2.1	Robust Image Processing Pipeline	3
2.2.2	Perspective Transform of Video Frames	6
2.2.3	Collision Detection	7
2.3	Design Description & Justification	8
2.3.1	Camera Subsystem	8
2.3.2	Image Processing Subsystem	8
2.3.3	Path Generation Subsystem	9
2.3.4	Vehicle Subsystem	11
2.4	Subsystem Diagrams & Schematics	14
2.4.1	Camera Subsystem	14
2.4.2	Image Processing Subsystem	14
2.4.3	Path Generation Subsystem	14
2.4.4	Vehicle Subsystem	15
3	Cost & Schedule	15
3.1	Cost	15
3.2	Schedule	16
4	Requirements & Verification	17
4.1	Complete Requirements	17
4.1.1	Camera Subsystem	17
4.1.2	Image Processing Subsystem	18
4.1.3	Path Generation Subsystem	18
4.1.4	Vehicle Subsystem	18
4.2	Appropriate Verification Procedures	18
4.2.1	Camera Subsystem	18
4.2.2	Image Processing Subsystem	19
4.2.3	Path Generation Subsystem	19
4.2.4	Vehicle Subsystem	20
4.3	Quantitative Results	21
4.3.1	Camera Subsystem	21
4.3.2	Image Processing Subsystem	21
4.3.3	Path Generation Subsystem	21

4.3.4	Vehicle Subsystem	22
5	Conclusion	23
5.1	Accomplishments	23
5.2	Uncertainties	23
5.3	Future Work / Alternatives	23
5.4	Ethical Considerations	24
	References	i
	Appendix A	i

1 Introduction

1.1 Purpose

With the rapid development of artificial intelligence, drones and unmanned vehicles are found applicable in many different scenarios in people's daily lives, industry and research for tasks such as package delivery, dangerous situation probing and scientific research operations. They have therefore appeared in our lives and brought us convenience. However, in some real-world situations, drones and unmanned vehicles can only solve partial tasks.

The purpose of our project is to design and verify an air-ground cooperation system that makes use of both drone and unmanned vehicles to achieve complex tasks. Specifically, it uses a drone to carry sensing equipment that collects data of the environment and uses the data to guide the movement of the unmanned vehicle.

1.2 Functionality

We have built a prototype that uses a battery-driven car to represent the unmanned vehicle, a hanging camera to represent the sensor carried by the drone and a maze to represent the environment. We also use a server computer to process image data and send instructions to the car. The prototype has the following functions:

- The computer server can establish a high-speed WLAN connection with the car and the camera, which enable the camera to pass 1024x1440 RGB image at 20 fps and the car to receive 256B instruction within 0.05 second. This function makes sure that the system has a short overall reaction time.
- The computer server can process the image correctly to identify maze walls, target and car. Based on this information, it can generate a correct path for the car. This function makes sure the correct operation of our system.
- The car executes instructions with an average error of less than 10%. This function also ensures the correct operation of our system.

1.3 Subsystem Overview

As shown in the diagram 1, the maze information is first collected by the Camera Subsystem as image data, then the Image Processing Subsystem process the image data and generates an integrated data set that contains both the abstracted information of maze and vehicles' status. The Path Generation Subsystem will calculate the optimal path based on the data set. Finally, the Vehicle Subsystem will generate instructions for the path and execute those instructions as accurately as possible to reach the target position.

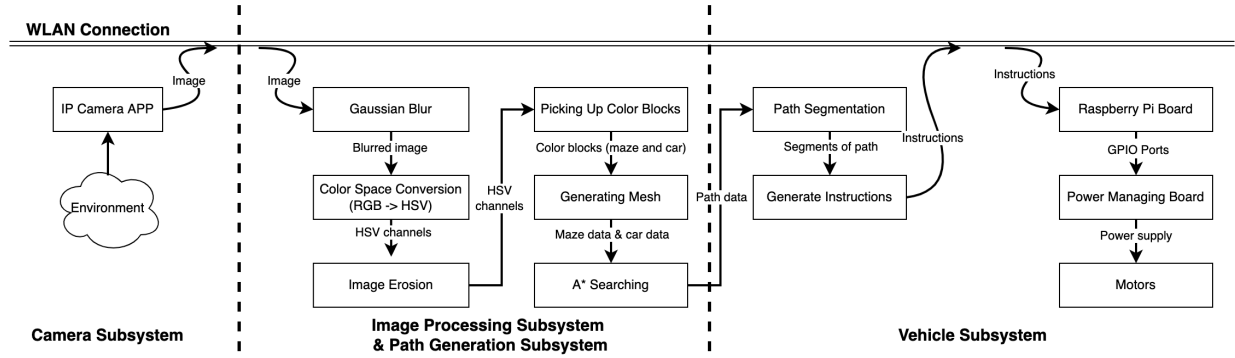


Figure 1: Block Diagram

2 Design

2.1 Equations & Simulations

2.1.1 Motor Control

It is impossible to control the exact rotation speed of the motor by simply providing an angular velocity. Instead, the power managing chip only accepts an integer level ranging from 0 to 100 as the representation of output power. Therefore, to achieve an accurate control, we have to measure the relation between speed, integer level and motor operation time.

Instead of directly measuring the speed, we measured the relationship between the displacement of the car and the motor's operation time, and they demonstrate a good linear relationship. After some further experiments, we chose 30 as the level of output power and measured the displacement/angular displacement versus time at this level. Measured data and its standard deviation are shown in 2, 3 and 4, 5.

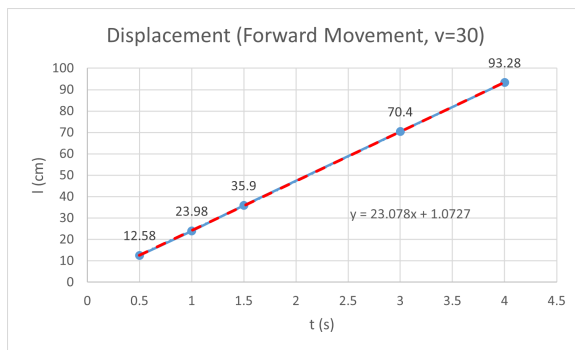


Figure 2: Forward Movement vs. Time

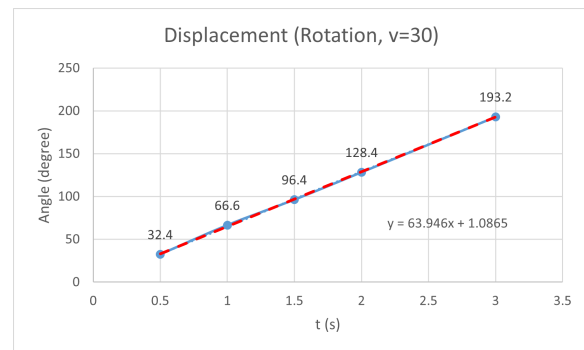


Figure 3: Rotation vs. Time

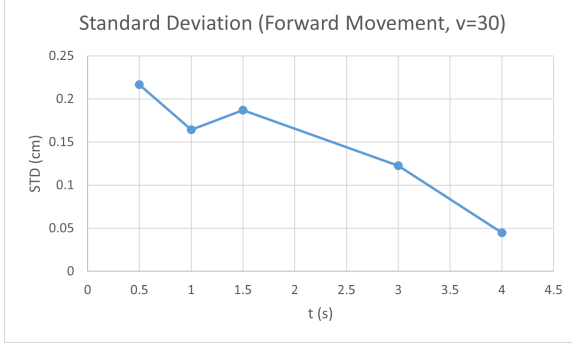


Figure 4: Forward Movement STD

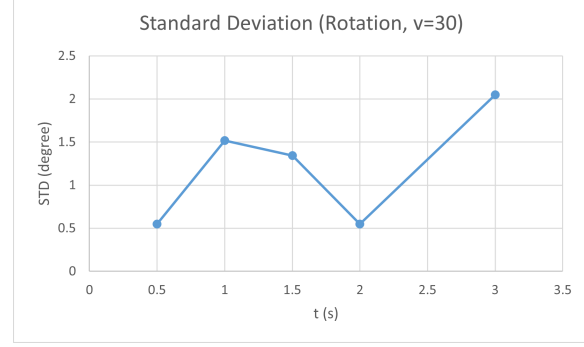


Figure 5: Rotation STD

By fitting data points into linear equations, we got formulas for motor control:

$$displace_{forward} = 23.08t + 1.07 \quad (1)$$

$$displace_{rotate} = 63.95t + 1.09 \quad (2)$$

It is worth noticing that their coefficients might vary in corresponding to the texture of the ground and battery conditions, so we wrote a program to correct these coefficients under different environments.

2.1.2 Vision

The vision algorithm we propose can be expressed in Equation (3), where the l means the labels we assign to objects in the maze. The function f represents a pipeline of image processing functions, such that $f(x, y) = Classify \circ GenMesh \circ ImageFilter(x, y)$. The *ImageFilter* function will generate a clean enough frame which only contains simple colors blocks. Then we use *GenMesh* to generate sparse square meshes and their labels based on the pixel majority vote within the mesh region which typically has size 8×8 or 4×4 . Finally, we can assign labels to each input pixel coordinate (x, y) by checking which mesh region it belongs to.

$$\begin{aligned} l &= f(x, y) \\ l &\in \{Road, Wall, CarFront, CarEnd, Destination\} \end{aligned} \quad (3)$$

2.2 Design Alternatives

2.2.1 Robust Image Processing Pipeline

In the design document, we proposed the first version of the image processing algorithm, in which we define the *ImageFilter* function as a combination of *GaussianBlur* and *Thresholding*. In experiments, we found that real-world lighting conditions are way more complicated than our expectations, and the ground color is not pure gray. The first version of the vision algorithm could work well for our ideal sample image but easily failed in the tests using real-time captured video frames.

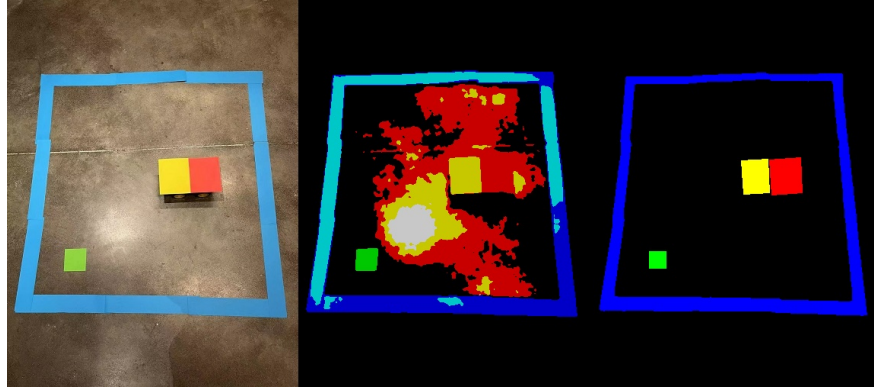


Figure 6: Effect of New Robust Image Processing Algorithm

Figure 6 shows the original image, the output of the old image filtering algorithm, and the output of the improved robust image filtering algorithm (from left to right). You can see that the highlight region on the ground and the complicated color distribution make the old method generate a considerable amount of outlier pixels.

Our new method was designed to deal with real-world images, which can be complicated and full of unnecessary details. To pick out the color regions we want, we first transform the image's color space from RGB to HSV because, in HSV color space, colors are better separated so that we are able to check if the pixel has color in the target range. Then, to cancel the effect of unnecessary information in an image, we apply an image erosion algorithm to filter out details and reserve major color regions.

HSV color space represents color by hue, saturation, and value. Here, value means the brightness of this color. Figure 8 shows the difference between RGB color space and HSV color space. The HSV color space was invented to suit better the human eye's perception of color [1], which is more intuitive, allowing us to tune the detection parameters based on the test field easily.

The color erosion algorithm is based on image convolution. A kernel(a matrix of odd size(3,5,7)) is convolved with the image, and a pixel in the original image will be made to zero if any pixel under the kernel has zero value [2]. For colored images, the above procedure is replicated for all three channels. Color erosion can help us to eliminate minor noisy outlier points.

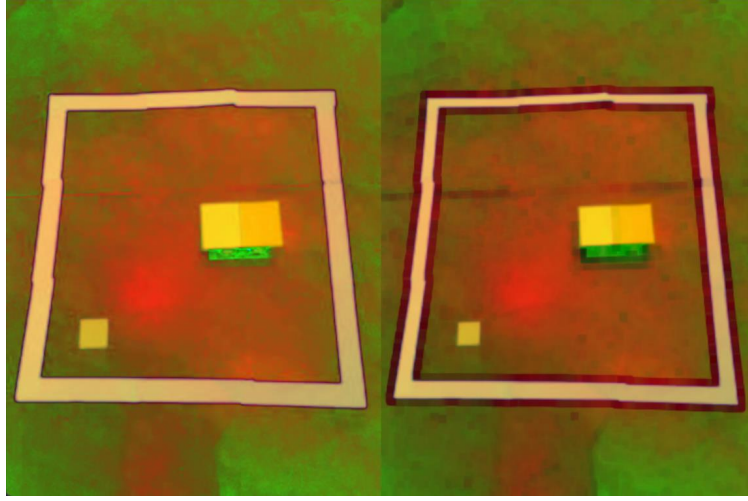


Figure 7: Effect of Image Erosion

If you look closely at the image of our test field, the color of the ground is complicated and noisy, with many pellet-like black points. The color erosion is designed for such cases, and Figure 7 shows the effect of the erosion algorithm. Please note that the image erosion should be done in the HSV color space to achieve the best result, as shown in this figure. This figure shows that all those minor outlier points are mixed with their near region, reducing the chance of having outlier points in the output of *ImageFilter*.

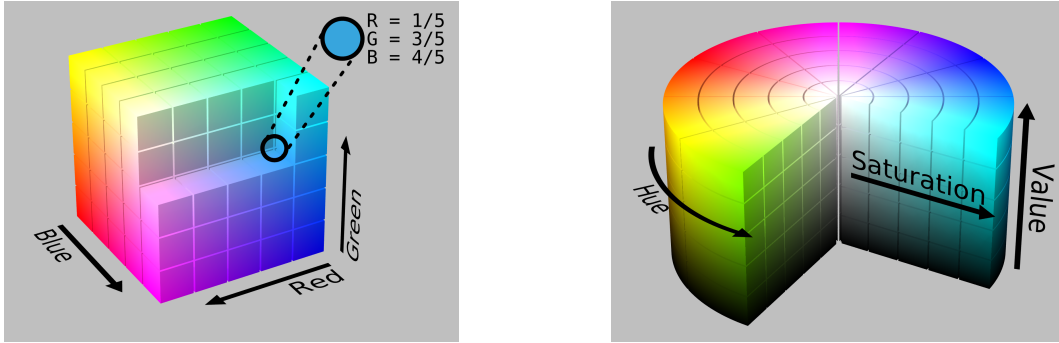


Figure 8: RGB Cube (L) and HSV Cylinder (R)

We define the new *ImageFilter*(x, y) function in the algorithm 3. We map the original image to a cleaner image which contains a simple set of colors:

$$getColorSet(img) = \{carEndColor, carFrontColor, destColor, wallColor, black\}$$

so that we are able to make an analysis of this output image. Note that the *inRange*(*img*, *range*) function checks if each pixel value is in the range and outputs a binary map that shows if each pixel is in the input color range.

After executing the *ImageFilter*, we can use this processed image to generate meshes. We first divide the entire image by 8×8 or 4×4 square meshes and classify the label for

each mesh. The output labels for each mesh will be the source information of our later path generation algorithm. We adopt the mesh generation algorithm for efficiency because if the path generation module can accept a much smaller dimension of mesh labels instead of having one label for each pixel, the computational complexity can be reduced. To generate labels for each mesh, we choose the majority vote mechanism, which means that most pixel values in this mesh can decide the property/label of this mesh. This mesh generation algorithm is highly repetitive and perfect for GPU acceleration. Thanks to NVIDIA CUDA technology and its python support called Cupy, we can write CUDA code for mesh generation and compile it in runtime using *cupy.RawKernel*, which returns an interface function of CUDA kernel for us. Figure 19 shows the output of mesh generation.

2.2.2 Perspective Transform of Video Frames

To extend the applicability of our system, we no longer limit the plane to be directly above the maze but at any angle above it. So we added quadrangle correction to the resulting photo-stream, which is to turn the distorted image in the camera into a normal view of the original image. First, we need the coordinates of the four angles to get the magnitude and angle of correction. So we put a black and white filter on the image to identify the separated edges 9. Then we smooth the contour 10 to a simplified contour 11. After we get the contour, we can get the positions of the four vertices and correct the figure 12.

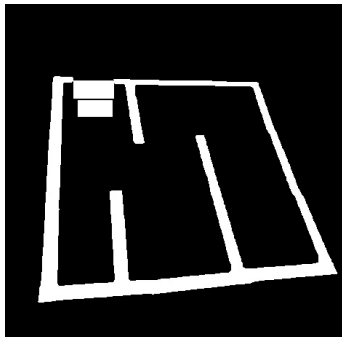


Figure 9: detect the outline1

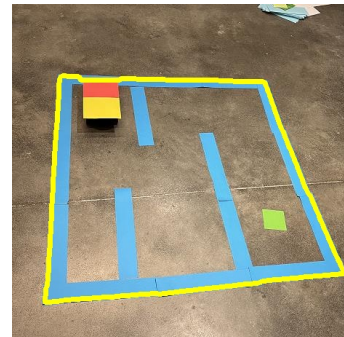


Figure 10: detect the outline2

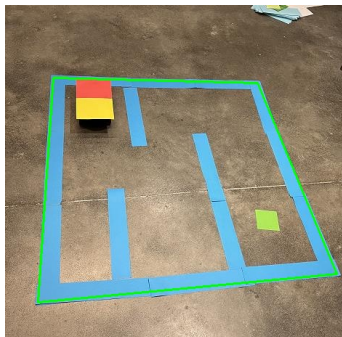


Figure 11: detect the outline3

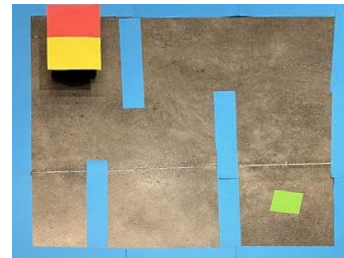


Figure 12: detect the outline4

2.2.3 Collision Detection

Overview When we first discussed the design of our project, we considered the vehicle as a mathematical dot but not an actual physical object with a three-dimensional size. To write search algorithms for a dot is much simpler; however, in the real world, it involves checking the distance from the vehicle to the obstacles to avoid collisions. Therefore, we took distance detection into account when writing the A* search algorithm into account.

Local Collision Detection Matrix The A* search algorithm runs on a well-established maze class. This class contains information about the cart position, destination position, and obstacle position, which are represented in the form of horizontal and vertical coordinates of the mesh.

When computing the path, each step involves putting all valid neighbor grids of a current location into the search frontier, which is the stage where we can perform the obstacle distance confirmation. Instead of directly using the output from *getNeighbors* for our search frontier, we add a layer of *checkDisToObstacle* to ensure there is no obstacle inside the local collision detection matrix centered at the current location with one plus twice the given collision radius as size.

The illustration of the local collision detection matrix is shown in Figure 13. Inside the representative mesh of the maze, there are mesh blocks labeled O to represent obstacles. With a given collision radius of 3 mesh blocks, the local collision detection matrix include a square matrix with size $7 = 3(\text{collision_radius}) \times 2 + 1$. Then we scan the whole matrix to see if there is any obstacle cell in the matrix; if so, the current cell will not be counted as a valid move.

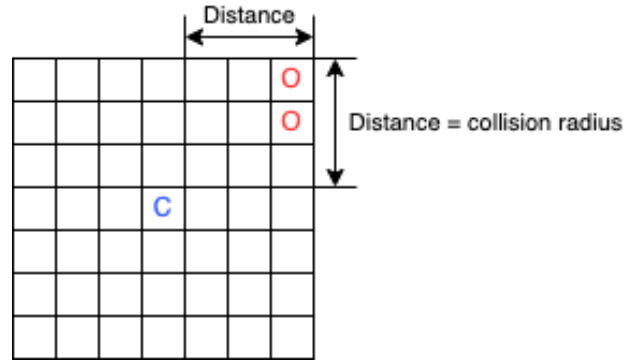


Figure 13: Local Collision Detection Matrix

Calculation of Collision Radius The collision radius is calculated by measuring the length of the car.

2.3 Design Description & Justification

2.3.1 Camera Subsystem

The camera subsystem will be used to take pictures of the whole landscape and pass the captured video frames to a server for further analysis via socket. The camera subsystem shown in Figure 20 consists of two sides, the receiver side and the sender side. For the sender, the software has the ability to fetch frames taken by the camera and serialize the multimedia data by video encoders so that the network adapter can send the video byte stream to our PC. For the receiver, our PC will decode the video frames so that our image processing subsystem can begin its analysis.

An IP camera app can take the role of the image sender. For the receiver side, we choose OpenCV to capture video frames. OpenCV has provided the *cv2.VideoCapture* interface, which can take a network address as an input parameter, which allows us to get data from our cell phone camera in a wireless way.

To enable our camera subsystem, we should first install a router in the test room, or we can use the cell phone hot-spot as an alternative solution. Then we make the server, the cell phone and also the car control board connect to the same router, so that they are in the same network.

2.3.2 Image Processing Subsystem

The image processing subsystem, which runs on a separate server, will be used to transfer the image from the camera subsystem to all the information needed by the scheduling subsystem. The generated information should be accurate and specific to ensure the stability of the whole system. It will at least include the location of the maze's obstacles and the posture of the SUGVs.

To achieve the goal, we plan to use OpenCV (an open-source computer vision and machine learning software library) to detect objects in the maze. We have talked about the technical details in the section on design alternatives. In short, we have adopted image color space transformation, color erosion, and color range checks to realize effective information extraction from the image.

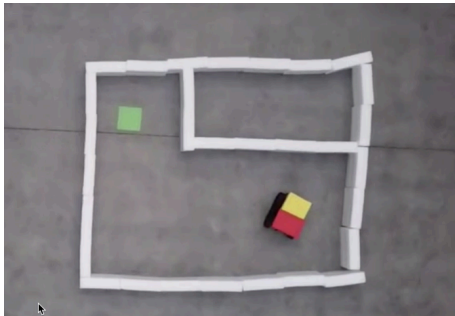


Figure 14: Input of I-P subsystem

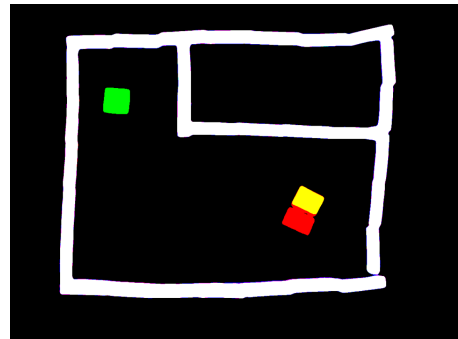


Figure 15: Detection and image cleaning

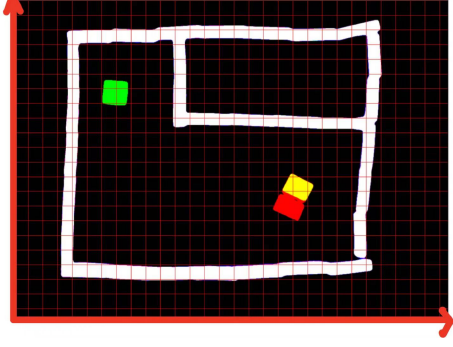


Figure 16: Add mesh

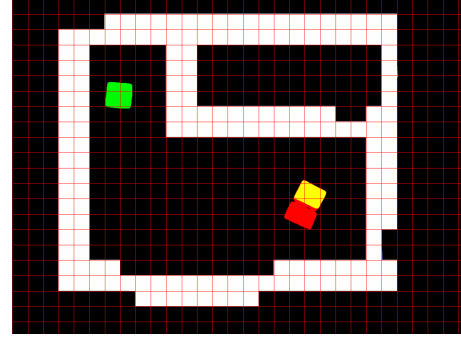


Figure 17: Filling color

We take the lower-left corner of the picture as the center coordinate to construct a two-dimensional mesh coordinate system such as Figure 16. Then we can use the coordinate system we built to get the location of all the maze's obstacles.

The last part in this section is already enough to provide the information required by the scheduling subsystem. One more concern is the speed of image processing. Higher speed is required to reduce the latency. Therefore, we have used CUDA acceleration for mesh generation. We have provided details about it in the previous section, the design alternative section.

2.3.3 Path Generation Subsystem

The Scheduling Subsystem has two parts: the path planning and the scheduling protocol when we have multiple cars on the ground.

Theoretical Foundation For the path planning module, we choose the A-Star algorithm [3] since the system could easily obtain the position of the SUGVs and their destinations from the captured images (Figure 18). Moreover, with the information on SUGV position and the location of the destination, the system could calculate the distance from the SUGV to its starting point, and the heuristic: the distance from SUGV to the destination, thus making the A-Star algorithm the most efficient choice.

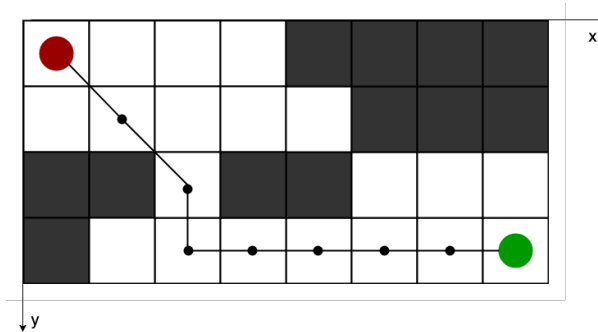


Figure 18: A* Algorithm

Without loss of generality, we consider the terrain a square grid with many obstacles, and we are given a starting red cell (the initial position of the SUGV) and a target green cell (the destination). We want to determine the shortest possible path from the starting cell (if exists) to the target cell.

What the A-Star Search Algorithm does is that at each step, it picks the node from the priority queue of frontier according to a value f , which is a parameter equal to the sum of two other parameters $f = g + h$. At each step, it picks the cell having the lowest f , then checks and pushes the neighbors of that cell to the frontier priority queue. In the meantime, we set the parent of these neighbors to the picked cell.

The value g and h are defined as below.

- g is the movement cost to move from the starting point to a given square on the grid, following the path generated to get there.
- h is the estimated movement cost to move from that given square on the grid to the final destination which is often referred to as the heuristic.

In our project, we use Manhattan Distance to calculate the approximation heuristics and use dynamic programming to reduce the time to calculate g .

$$h = |current_cell.x - goal.x| + |current_cell.y - goal.y|$$

For the neighbors we check for the following conditions.

- If neighbor is the target cell, stop search.
- Else, compute both g and h for neighbor and set $neighbor.f = neighbor.g + neighbor.h$.
- If a node with the same position as neighbor in the frontier priority queue which has a lower f than neighbor, skip this neighbor.
- If a node with the same position as neighbor is already a parent which has a lower f than neighbor, skip this neighbor; otherwise, add the node to the frontier priority queue.

Finally, if we reach a point where the frontier priority queue is empty, then there is no path from the starting cell to the target. Otherwise, if we reach the target cell, we simply backtrack the parent of the target cell until we get back to the start and get the planned path.

Actual Implementation As illustrated in the component diagram 22, this module consists of three parts - the maze builder, the A-star Search Algorithm, and the Path Search Module. The Mesh Generation Module transfers the processed image of the maze and the vehicle to a mesh represented by a 2-Dimension array with labels in Table1.

To fully understand the output of the Mesh Generation Module, we can plot the space, obstacle, head & tail of the vehicle and the destination in black, white, yellow, red, and

Flag	Meaning
0	Empty space
1	Wall (Obstacle)
2	Head of Vehicle
3	Tail of Vehicle
4	Destination

Table 1: Flag Correspondence

green to get the illustration of mesh output. The plot to the left is the result after image processing, and the plot to the right is the output from Mesh Generation Module.

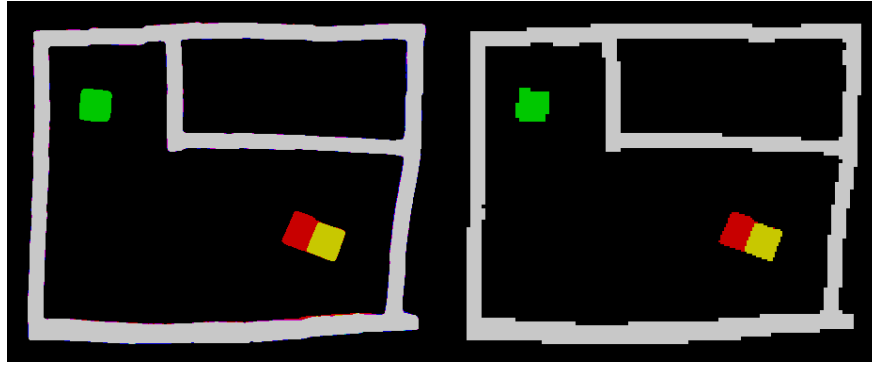


Figure 19: Mesh Gen Output

After receiving the output of the Mesh Generation Module, we can transform it into a maze object with the maze builder. This class initializes by filling a sequence of properties of the maze, recording all the information the search algorithm needs to access. Then, a group of class functions returns the information, such as whether the current mesh cell is wall, space, or destination. One of these help functions returns the valid move of a block, thus providing the validate neighbors of a given (col, row) . Also, another function checks whether the surrounding (with a provided distance) of given (col, row) collides with an obstacle so that if the vehicle gets to the point, it crashes with obstacles.

After the maze information is transferred into the new class, we can implement the actual A* algorithm. The pseudo-code is shown in Algorithm 6. With this program, we can obtain the optimal path to the desired destination from a given start point.

2.3.4 Vehicle Subsystem

The Vehicle Subsystem receives path data from the Path Generation System, cuts the path into segments and generates instructions for each segment to guide the car's motion.

Vehicle The vehicle consists of one control board, one power managing board, one battery set, four motors, a plastic skeleton, and wires. We choose Raspberry Pi 3 Model B+ as the control board. The board has two main functions, one is receiving instructions from the server through WLAN connection, the other is generating control signals and delivering them to the power managing board through GPIO ports. By running simple Python scripts on the Linux system, we can easily read data from the connectivity module and deliver motor control signals through the GPIO ports accordingly. The control board also receives a 5V power supply from the power managing board through GPIO header. The power managing board is designed by the car seller. We used the 5V voltage supply pin to power the control board and 18, 23 PWM ports and 22, 24, 25, 27 GPIO ports to send motor control signals. The power supply of four motors is managed by two TB6612FNG chips, they enable us to control the speed and direction of motor rotation through GPIO ports.

Path Segmentation and Instruction Generation The path data from the Path Generation System, which is represented as a list of integer coordinates, should be converted into a sequence of instructions in order to guide the vehicle, so we wrote the function to perform the conversion. Since the path data is discrete, the function first detects continuous segments in a list of points, then for each segment, it generates a pair of rotation and forward movement instructions and stores it in the queue. The instruction pair for each segment is calculated by *segmentInst* function 4.

Control Strategy Though we have paid much effort in improving the stability of the image processing pipeline, the measured data may still have small fluctuation, therefore, the control strategy must be carefully designed to handle such fluctuation. We used the *getCarState* function 1 to measure the precise position and direction of the car, and used the *autoControl* program 2 to control the car.

The *getCarState* function 1 will take n times of measurements $Data = [m_1, m_2, \dots, m_n]$, calculate their mean value $M = Mean(Data)$, and pick the measurement that is closest to the mean value M . This approach can minimize the fluctuation of data.

Algorithm 1 Measure Car's State (*getCarState*)

Require: times of measurement n

Ensure: vehicle's position $\vec{p}$ and direction $\vec{d}$

- 1: $Data \leftarrow$ Measure n data points
 - 2: $M \leftarrow Mean(Data)$
 - 3: $m_{best} \leftarrow argmin_{m_i}(abs(M - m_i))$
 - 4: Convert m_{best} to $\vec{p}, \vec{d}$
-

The *autoControl* program 2 send instructions to the car and monitor its execution. If the error is too large, the program will send extra correction instructions to the car.

Algorithm 2 Autonomous Control (*autoControl*)

Require: Instruction queue Q **Ensure:** None

```
1: while  $Q$  is not empty do
2:   Current car's state  $\vec{p}_{curr}, \vec{d}_{curr} \leftarrow getCarState()$ 
3:    $instruction \leftarrow Q.pop()$ 
4:    $execute(instruction)$ 
5:   New car's state  $\vec{p}_{new}, \vec{d}_{new} \leftarrow getCarState()$ 
6:   Calculate the difference between actual movement and target value
7:   while Number of correction time does not exceed limit do
8:     if Error rate too large (actual threshold may vary) then
9:        $instruction_{correction} \leftarrow$  Generate feedback control instruction
10:       $execute(instruction_{correction})$ 
11:      Calculate again the error rate
12:    else
13:      Break loop
14:    end if
15:  end while
16: end while
```

Motor Control Coefficients Correction As described in section 2.1.1, the coefficients of the motor control formula may be influenced by many factors. Therefore, we wrote a program to correct the coefficients by giving the vehicle several test instructions and measuring the result using *getCarState* function 1 with a very high n (7-9). We run the correction program before putting the car into a new environment to get the correct motor control formula.

2.4 Subsystem Diagrams & Schematics

2.4.1 Camera Subsystem

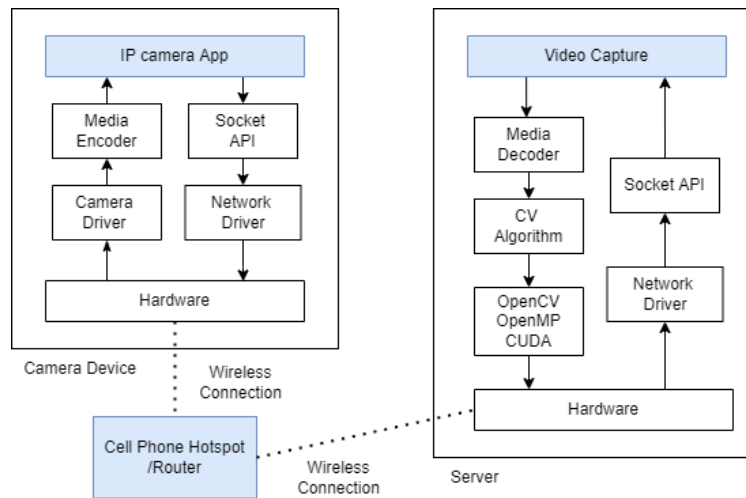


Figure 20: Camera subsystem architecture, with the blue blocks representing three key components of the network: sender, router and receiver.

2.4.2 Image Processing Subsystem

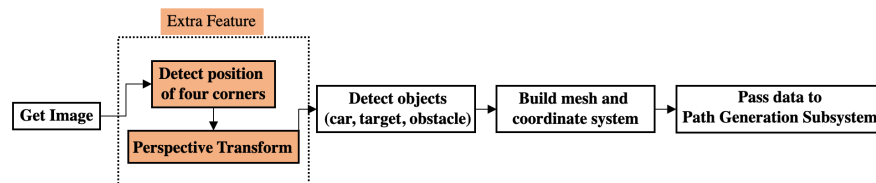


Figure 21: Image Processing Diagram

2.4.3 Path Generation Subsystem

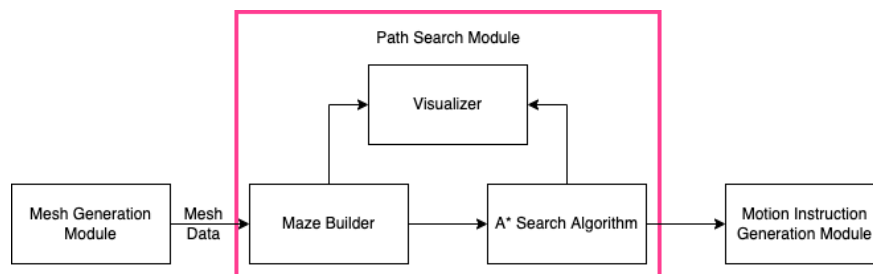


Figure 22: Path Search Module

2.4.4 Vehicle Subsystem

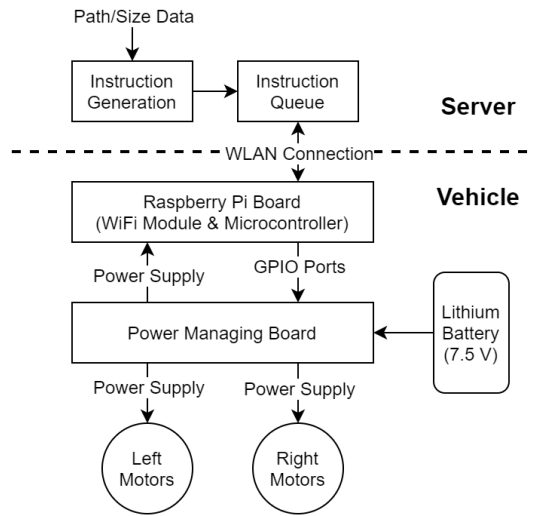


Figure 23: Vehicle Subsystem Diagram

3 Cost & Schedule

3.1 Cost

Our fixed development costs are estimated to be 100RMB/hour, 10hours/week for 4 people. And this project will last for 14 weeks in total. Thus, our labor cost will be:

$$LaborCost = 4 \times \frac{100RMB}{1hour} \times \frac{10hours}{1week} \times 14weeks = 56000RMB$$

Part	Unit Cost (RMB)	Number	Total Cost (RMB)
Car skeleton and power control board	419	2	838
Raspberry Pi 3B+ board	760	2	1520
Labor cost	14000	4	56000
Sum	/	/	58358

3.2 Schedule

Week	Tasks	Member
3/14/22	Learn Computer Vision Library (Opencv) and begin some tutorials.	Xinyu Lian
	Select vehicle and relevant hardware devices.	Nachuan Wang
	learn image transmission techniques and stream images from mobile phone cameras to the server.	Neng Xiong
	finish coding about instruction transmit by network	Zhongbu Zhu
3/21/22	Finish learning opencv and prepare demos	Xinyu Lian.
	Study documents and write motor driver.	Nachuan Wang
	Study A* algorithm and prepare for demos.	Neng Xiong
	Finish video streaming with OpenCV + existing IP camera APP.	Zhongbu Zhu
3/28/22	Use YOLO to improve the performance and write code to add mesh and coordinate system of the photo	Xinyu Lian.
	Improve the accuracy of motor control.	Nachuan Wang
	Implement A* algorithm.	Neng Xiong
	Finish testing remote control on the physical car	Zhongbu Zhu
4/4/22	Write code to get and pass all the information generated by image processing subsystem to scheduling system.	Xinyu Lian
	Write functions for path decomposition and instruction generation.	Nachuan Wang
	Test and optimize the implementation of A* algorithm.	Neng Xiong
	Develop CV algorithm and finish mesh generation.	Zhongbu Zhu
4/11/22	Randomly generate 100 photos and test the proceeding speed of the image processing subsystem.	Xinyu Lian
	Write correction program for motor control	Nachuan Wang
	Construct the programming interface from the mesh generation to the search algorithm.	Neng Xiong
	Write CUDA code to accelerate the mesh generation algorithm	Zhongbu Zhu

4/18/22	Develop optimized CV algorithm and adjust the parameters of HSV color space to generate the best result.	Xinyu Lian
	Integrate vehicle subsystem into the whole system.	Nachuan Wang
	Test path generation module with the stable results from the image processing subsystem, expect to see the real time recommended path.	Neng Xiong
	Do basic car navigation in a simple version of maze.	Zhongbu Zhu
4/25/22	Apply rectangle correction to make our system to have a wider range of application scenarios.	Xinyu Lian
	Wrote correction program for the motor control.	Nachuan Wang
	Implement and field-tested the pipeline from mesh generation to real-time path recommendation	Neng Xiong
	Participate in field-testing by debugging with Neng Xiong to get stable real time path generation	Zhongbu Zhu
5/2/22	Build stable environment and prepare the mock demo.	Xinyu Lian
	Do test on read maze and prepare the mock demo.	Nachuan Wang
	Test the car movement and prepare the mock demo.	Neng Xiong
	Merge the rectangle correction into the main branch and tune it to improve the field test	Zhongbu Zhu
5/9/22	Prepare for the presentation and final report.	Xinyu Lian
	Prepare for the presentation and final report.	Nachuan Wang
	Prepare for the presentation and final report.	Neng Xiong
	Prepare for the presentation and final report.	Zhongbu Zhu

4 Requirements & Verification

4.1 Complete Requirements

4.1.1 Camera Subsystem

Based on Figure 20, we came up with the requirements for the wireless video transmission.

1. The sender process should constantly use the camera device to share video frames to a specific IP address. It is required that the receiver process running on our PC should actively capture the frames passed in and then decode the byte stream to

regular image format using RGB color space. With these requirements satisfied, the image processing subsystem can then take this image as an input.

2. As an important component of the architecture, it is required for us to install a network router near the test field to bring a local wireless network. The network bandwidth should be high enough to sustain real-time video transmission, and we estimated that the network bandwidth should be larger than 500kbps.
3. We also need to make sure that the data transmission based on the router will not be blocked by the security firewall of the PC or the cell phone.

4.1.2 Image Processing Subsystem

1. Be able to output the location of the maze's obstacles, and the correctness should be higher than 95%.
2. Be able to detect the car's posture, and the deviation from the actual Angle should be less than 5 degrees.
3. The total time to proceed with one picture need to be less than 100ms to ensure the low latency.

4.1.3 Path Generation Subsystem

1. In terms of space and correctness, in order to generate a feasible path for SUGV to follow, the path generation subsystem needs to output a path represented by a list of coordinates from starting point to the destination. Furthermore, since the SUGV has a certain length and width, all points along the path need to keep a distance from obstacles at least a collision radius away.
2. In terms of time and efficiency, the system is running as fast as ten frames per second, therefore requiring the performance of A* search to be sub-0.1 second.

4.1.4 Vehicle Subsystem

1. The car should react to an instruction within 20ms.
2. The sever should generate proper instructions based on the path data, guiding the car to reach the target with a final error distance less than 10cm.
3. The car should execute instructions (move and rotation) with an average error rate less than 10%.

4.2 Appropriate Verification Procedures

4.2.1 Camera Subsystem

1. Receiver (the PC) has to capture frames passed in by the network, and we can see the passed in images in real-time with a clear GUI. To justify our design, we choose to

create an independent thread on the server side specifically for GUI to fetch video frames from the camera and display it actively. We also cache this frame to the *latestFrame* variable. Whenever the car request a vision analysis (which is an asynchronous event because we want to save computational resources), the system will pick the *latestFrame* to execute the vision analysis algorithm. For each frame update, we update and calculate the average frame rate. Our target is to make it at least 20 FPS if we set the frame rate and resolution on a cell phone to be 24FPS, 1440×1080.

2. Use Windows task manager to monitor the network bandwidth. We need to select the image receiver process and check its network usage report provided by the router and check if it is always larger than 500kbps.
3. We also need to make sure that each asynchronous call to the vision analysis algorithm should be using the latest frame to guarantee the correctness of car control. Therefore, we should compare the image used by the vision analysis algorithm with the real-time video to ensure it is the latest one.

4.2.2 Image Processing Subsystem

1. Use the camera to take 100 photos of the different mazes as the input of the image processing subsystem. After execution, we manually compare the real-world positions of the maze's obstacles and the result we get from our system.
2. Use the camera to take 100 photos of mazes with the random position of SUGVs. After execution, we compare the real angle and position of the SUGV with the output of our system.
3. We add a timer (python time package) before and after the image processing procedure. And the difference is the total execution time. If the time is larger than 100ms, we need to work out an optimized solution.

4.2.3 Path Generation Subsystem

1. To verify whether the algorithm is generating the optimal path as we desire, we implemented a visualizer to see the structure of the maze and the path on it.

The visualizer is built with the PyGame library that plot the whole maze with information in the maze structure. It plots out human-friendly images to check whether the maze-building process is correct and whether the A* search algorithm correctly generates an optimal path from the starting point to the destination with a given distance from the vehicle to objectives.

To test the maze builder's functionality, we use the visualizer to plot the maze object. The result is Figure 24

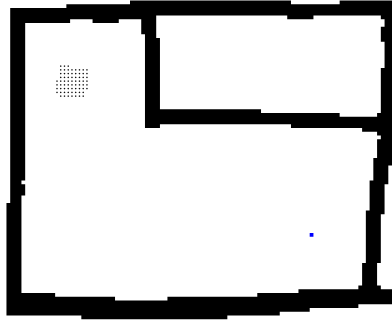


Figure 24: Maze

The black & blue block and black dots correspond correctly to the wall, starting point, and destination points in the maze structure. From this, we can confirm that the maze builder functions correctly.

Then we plot the path generated by A* search together with the maze. Figure 25 is the path with distance set to 0, and Figure 26 is the path with distance set to 10.

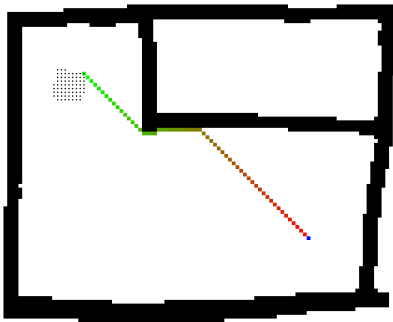


Figure 25: Path with Distance = 0

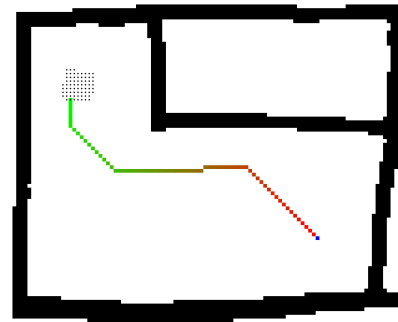


Figure 26: Path with Distance = 10

We can see from the figures that the A* search algorithm finds the optimal path from starting point to the destination correctly, and when we set a particular distance, it generates a path that stays away from the obstacles. In the actual system, this value will be set to the collision radius of the vehicle, so it is promised that the vehicle will not crash into the wall.

2. As for the time cost of each execution of the path generation algorithm, we add timers at the beginning and end of the algorithm and test multiple times to get an average time cost.

4.2.4 Vehicle Subsystem

1. Use "ping" instruction at car's command line to test the connection delay with a server.

2. Run the *autoControl* program 2 and measure the final error distance after it ends.
3. Run the *autoControl* program 2 and measure the error rate of execution for each instruction using *getCarState* function 1.

4.3 Quantitative Results

4.3.1 Camera Subsystem

1. Figure 27 shows that we can see real-time video frames on the server. The top left part is the raw image, which a frame rate of 23FPS can update. The other three images show the intermediate analysis of the vision subsystem, and they only get updated when the car has requested a video analysis. Since the cell phone is sending a video by 24FPS, the 23FPS performance on the server GUI window is nearly ideal. The FPS loss here is less than 5%.
2. The network usage report shown on the task manager shows that the network bandwidth for the receiver process is larger than 500kbps.
3. In Figure 27, we can also see that the top right, down left, and down right images all have the same car position as the top left image, that's because each asynchronous call of the vision analysis, the camera subsystem can always provide the latest frame for future analysis.

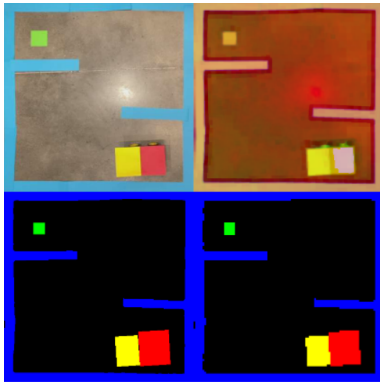


Figure 27: Server Side GUI

```
alexl@alexlian MINGW64 ~/OneDrive/Desktop/Polaris/tests/hsv_vision (main)
$ python image_filter.py
[[155.38275 346.50693]
 [246.83105 341.12766]
 [250.70001 406.90005]
 [159.25171 412.27933]]
[203.04138 376.7035 ]
Process time is: 0.009999990463256836
```

Figure 28: Image Processing Time

4.3.2 Image Processing Subsystem

1. The 27 is one example of one hundred mazes, the position of the obstacles is accurate and the angle error of SUGV is in 3%.
2. The processing time is within 10ms, 28 is one random chose result to prove it.

4.3.3 Path Generation Subsystem

1. As shown in the visualized graph, the path generation subsystem provides the optimal path.

2. We ran the A* search on the various mazes input 100 times with different distance settings from 0 to 10. It turns out that the average execution time is around 0.03 seconds, which meets our requirements.

4.3.4 Vehicle Subsystem

1. The print-out message in command-line 29 shows that the connection delay is around 2-4ms with 0% loss, much less than the required 20ms.
2. The car's states before and after running *autoControl* program 2 are shown in 31 and 32, at final state the car perfectly cover the target point, the distance error is only 1.2 cm.
3. The error rates of all executions during one complete cycle of *autoControl* program 2 are shown in 30, sometimes "rotate" and "move" instructions repeat because the program is correcting car's motion. As shown by the "err" value, the error rate after correction is always less than 0.1.

```
$ ping 192.168.101.17

Pinging 192.168.101.17 with 32 bytes of data:
Reply from 192.168.101.17: bytes=32 time=2ms TTL=64
Reply from 192.168.101.17: bytes=32 time=4ms TTL=64
Reply from 192.168.101.17: bytes=32 time=2ms TTL=64
Reply from 192.168.101.17: bytes=32 time=2ms TTL=64

Ping statistics for 192.168.101.17:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
    Approximate round trip times in milli-seconds:
        Minimum = 2ms, Maximum = 4ms, Average = 2ms
```

```
Auto-running program started
[Rotate] targ: -0.32, real: -0.20116343953220853, err: 0.18568212572092419
[Rotate] targ: -0.32, real: -0.2383283151040273, err: 0.12761260764895734
[Rotate] targ: -0.32, real: -0.2561364587516747, err: 0.09978678320866832
[Move] targ: 21.34, real: 19.665789100262714, err: 0.0392270959649685
[Move] targ: 21.34, real: 20.438864426468378, err: 0.021113766952474744
[Rotate] targ: -0.48, real: -0.4323188313388058, err: 0.04966788402207728
[Rotate] targ: -0.48, real: -0.37703416884412416, err: 0.10725607412070398
[Rotate] targ: -0.48, real: -0.4679271834447722, err: 0.012575850578362302
[Move] targ: 25.4, real: 27.463872365657917, err: -0.040627408772793694
[Move] targ: 25.4, real: 23.50945810959055, err: 0.03721539154349307
[Move] targ: 25.4, real: 26.48527645550511, err: -0.021363709754037674
[Rotate] targ: -0.9, real: -0.8642909422613121, err: -0.019838365410382197
[Move] targ: 27.61, real: 32.064480667569036, err: -0.08066788604797237
[Move] targ: 27.61, real: 24.85554968531526, err: 0.049881389255428066
[Move] targ: 27.61, real: 28.924804395559622, err: -0.023810293291554152
[Rotate] targ: 1.3, real: 1.295886485314224, err: 0.0015821210329907687
[Move] targ: 31.48, real: 27.836383049827248, err: 0.05787193377021527
[Move] targ: 31.48, real: 32.63261623719019, err: -0.018307119396286375
Target reached
Auto-running program ended
```

Figure 29: Ping Message

Figure 30: Error Rate of Executions

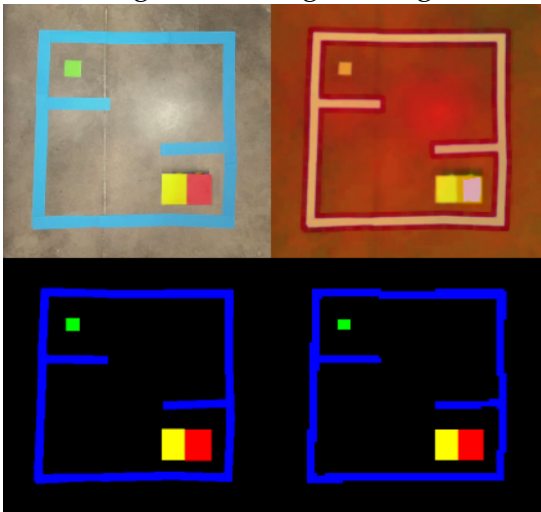


Figure 31: Start State

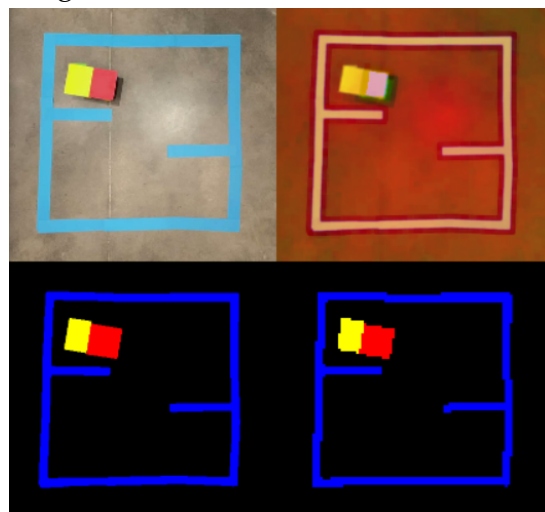


Figure 32: End State

5 Conclusion

5.1 Accomplishments

- The system in a whole is able to get real-time image information from the camera subsystem and allows the SUGV to move from starting point to destination smoothly.
- After receiving the image information from the camera subsystem, the image processing subsystem is capable of transform the images into a four-point corrected, stable mesh image of the maze containing obstacles, paths, head and tail of the SUGV, and destination.
- According to the mesh information from image processing subsystem, the path generation subsystem could provide an optimal path from the starting point to the destination that keeps the SUGV from crushing into obstacles.
- With the path provided by the path generation subsystem, the vehicle subsystem could follow the instruction to the destination within few effective motion corrections.

5.2 Uncertainties

Perspective Transform Drawback The perspective transform algorithm we illustrated above measures the perspective transform matrix based on the contour of our maze and applies it to the entire image. However, the car has its height, and the plane of the car is closer to the camera. Therefore, if we do not put the camera high enough, the car's height can be ignored, and we might see a distortion of the car's shape.

5.3 Future Work / Alternatives

There are the following concerns towards the performance of our system that can be further optimized in the future.

- We are only considering single vehicle path findings; therefore, we are not calculating paths of multiple cars simultaneously. Such scheduling task involves scheduling the common segments in different vehicles' optimal paths.
- When the collision radius is set to be larger, the time consumption of the path generation algorithm increases dramatically. If the actual size of the car is larger, we may need to use a mesh of bigger sizes to reduce massive calculations. This requires an algorithm that weighs terrain, vehicle, and obstacle sizes.
- Constrained by limited time and funding, and the impact of the COVID-19 epidemic, we replaced the drone camera in motion with a cell phone camera fixed above the field. Although our system integrates four-point correction to correct the deviation of the shooting angle, the images obtained from the actual drone shots will be more complex, which may require better image correction algorithms.

5.4 Ethical Considerations

- According to the first line of IEEE Code of Ethics, we need to "protect the privacy of others". To prevent the invasion of privacy, we will ensure that the data is closed loop transmission. And We will use public keys and private keys that we use in network to ensure the privacy. Moreover, we will try to keep drones out of residential areas.
- According to the ninth point of IEEE Code of Ethics, it is our responsibility to "avoid injuring others"[4]. Inappropriate use of drones can lead to injury. So we will test and experiment in an open field with no other people. Also, the improper arrangement of the maze terrain and the vehicle movement may cause the crossovers to fall and get injured. So we would only conduct the experiment in authorized areas and put on prominent notice to avoid intrusion of uninvolved personals.

References

- [1] Wikipedia. "HSL and HSV." (2022), [Online]. Available: https://en.wikipedia.org/wiki/HSL_and_HSV (visited on 05/14/2022).
- [2] GeeksforGeeks. "Erosion and Dilation of images using OpenCV in python." (2021), [Online]. Available: <https://www.geeksforgeeks.org/erosion-dilation-images-using-opencv-python/> (visited on 05/14/2022).
- [3] G. for Geeks, *A* serch*, 2018. [Online]. Available: <https://www.geeksforgeeks.org/a-search-algorithm/> (visited on 2022).
- [4] IEEE. "'IEEE Code of Ethics'." (2016), [Online]. Available: <https://www.ieee.org/about/corporate/governance/p7-8.html> (visited on 02/08/2020).

Appendix A

Algorithm 3 Robust Image Filtering

Require: img

Ensure: canvas

```
1: function ROBUSTIMAGEFILTER(img)
2:   canvas = emptyImg(img.shape)
3:   HSV = convertColor(img, RGB2HSV)
4:   erodeHSV = erode(HSV)
5:   wallRegion = inRange(erodeHSV, wallColorRange)
6:   drawColor(canvas, wallRegion, wallColor)
7:   carFrontRegion = inRange(erodeHSV, carFrontColorRange)
8:   carEndRegion = inRange(erodeHSV, carEndColorRange)
9:   destRegion = inRange(erodeHSV, destColorRange)
10:  drawColor(canvas, carFrontRegion, carFrontColor)
11:  drawColor(canvas, carEndRegion, carEndColor)
12:  drawColor(canvas, destRegion, destColor)
13:  return canvas
14: end function
```

Algorithm 4 Segment Instruction Generation (*segmentInst*)

Require: end points of segment $start_{seg}$, end_{seg} ; vehicle's direction $\vec{d}_{pre}$, $\vec{d}_{cur}$, mesh scale c

Ensure: time to perform rotation and forward movement t_{irot} , t_{iforw}

```
1:  $\theta \leftarrow \cos^{-1}\left(\frac{\vec{d}_{pre} \cdot \vec{d}_{cur}}{||\vec{d}_{pre}|| ||\vec{d}_{cur}||}\right)$ 
2:  $t_{irot} \leftarrow rotateEqu^{-1}(\theta)$  ▷ Equation that we get in section 2.1.1
3:  $dist \leftarrow ||end_{seg} - start_{seg}|| \cdot c$ 
4:  $t_{iforw} \leftarrow forwardEqu^{-1}(dist)$  ▷ Equation that we get in section 2.1.1
```

Algorithm 5 Maze Builder

Require: input parameters: mesh_matrix, starting_point;

Ensure: output: build the maze object

```
1: self.wallChar = 1
2: self.startChar = [2, 3]
3: self.objectiveChar = 4
4: self.objective = []
5: self.dimension = mesh_matrix.dimension
6: self.starting_point = starting_point
7: self.maze = mesh_matrix
8: for mesh block in self.maze do
9:     self.objectives.Append(block)
10: end for
11: function ISWALL(self, col, row)
12:     Return self.maze[col][row] == self.wallChar
13: end function
14: ...
```

Algorithm 6 A* Search

Require: input parameters: *maze*, *distance*;

Ensure: output: path

```
1: openList = []
2: closedList = []
3: openlist.put(maze.starting_point)
4: while openList not empty do
5:   Find the node with the least f on the open list, call it q
6:   current_node = openList.pop()
7:   generate q's validate successors, check if their surrounding of given distance does
   not collide with obstacles, set their parent to q if yes
8:   for each validate successor do
9:     if successor is objective then stop search
10:    else
11:      compute both g and h for successor
12:      successor.g = q.g + distance between successor and q
13:      successor.h = distance from goal to successor
14:      successor.f = successor.g + successor.h
15:    end if
16:    if a node with the same position as successor is in the OPEN list which has a
    lower f than successor then skip the successor
17:    end if
18:    if a node with the same position as successor is in the closedList list which has
    a lower f than successor then skip this successor
19:    else
20:      add the node to the openList
21:    end if
22:  end for
23:  push q on the closed list end (while loop)
24: end while
```
