EC 545 - Cyber Physical Systems
Final Project Report

Spontaneous Obstacle Avoidance and Following Robots

Boston University

Authors: Guillermo Ao, Piyusha Dongre,
Xiaopeng Huang, Zhaozhong Qi

GitHub Link -
[Dnisde/Spontaneous Obstacle Avoidance and Following Robots]

Introduction

Over the last few decades an increasing interest in utilizing the emerging technologies based in wireless communication, AI, and robotics has taken place in the development of autonomous security systems. Global markets for security robots are said to be projected to keep increasing exponentially over the coming years. The reason is clear, robots can often be reliable for precision tasks and they come with mathis projects that will be focused on developing a system that has the potential to scale inny benefits. Many applications are focused on monitoring and securing people or property. Given this large interest in creating robots for security purposes, to an industry product.

Modern day autonomous mobile robots are capable of navigating through unknown environments by using a combination of complex path planning algorithms, sensors, cameras, and other software. These robots use sensors to not collide with obstacles and have some awareness of their surroundings. The more advanced robots are capable of taking independent decisions while identifying problems and processing large amounts of information. In addition, automated robots come with many benefits starting with human safety and reducing their risk of going to danger zones. Robots can take the place of humans when required to be in dangerous environments and they can be operational for longer periods of time.

Therefore, based on the popularity of robot applications in security and surveillance, this project aims to develop a robotics system that can go through an unknown environment while also avoiding obstacles. The system will be composed of two self-balancing robots chosen for their mobility advantages. One of them (Master robot) will have the function of obstacle avoidance and will lead the other robot to complete the map. Whereas the other robot (Slave) will have the following function mode, this robot is intended to act as support to the Master when used for security purposes. Note that due to time constraints, it will not be possible to implement the many features required in a security/monitoring system such as threat detection, user alert, recording data, among others.

The deliverables of the project are divided into two main functionalities. First, obstacle avoidance where the robot should be able to autonomously go through an environment from a start position to a goal. Second, following mode centered around the Slave robot which should be able to accurately follow the Master at a reasonable speed and distance. The report will expand more on the overall functionalities of the systems, modeling, and analysis.

Design Diagram

The self-balancing robots must be able to go through a restricted closed environment (e.g. a maze) from an entry to an exit point while avoiding obstacles.

To achieve this, a composition of electrical components has to be made and it is shown as a high-level block diagram in Figure 1. The robot has to go through a maze and control the integrated component without touching the barriers, so the adequate device that can accomplish that is the Arduino microcontroller. This is an industry standard controller used for prototyping and building robotic systems. Also, the device needs to be aware of its immediate surroundings which can be measured by different sensors. Other parts include the motor control, battery pack, and the Raspberry PI interfaced with a camera module.

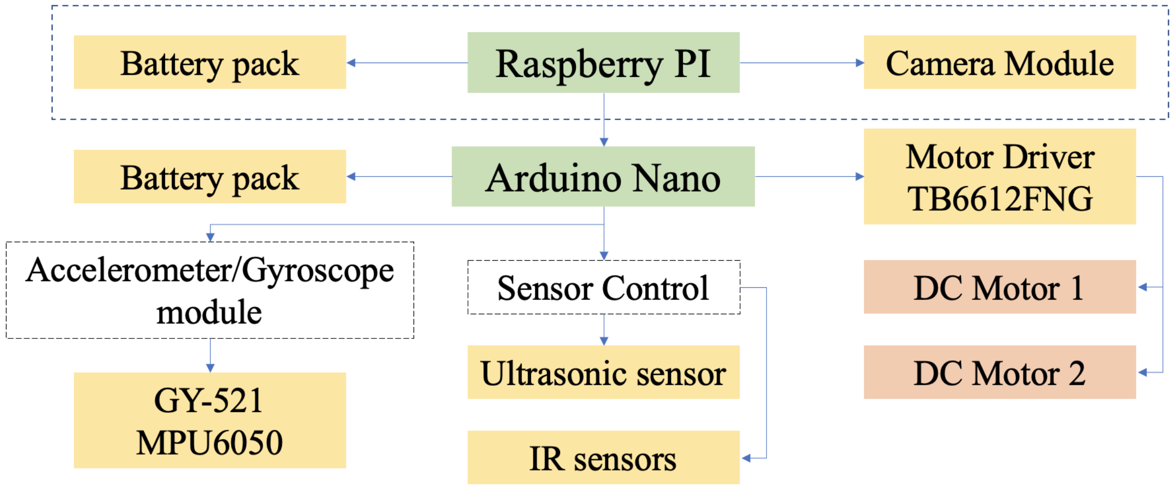

Figure 1: System Overview Block Diagram

Expanding on the hardware used in the system it is necessary to examine specific sections individually. First, the ALU control department is led by the Arduino Nano whose main function is to manage the robot’s components including processing input/output data. And then, the sensor controlling is composed of the ultrasonic and infrared (IR) sensors. Both are used for obstacle detection, avoidance and distance measuring. Third, the robotic system uses two DC motors that consume higher current compared to other components of the system. Therefore, a motor driver should be implemented to interface the controller and motors. Its function is to convert the low-current control signals into high-current that can drive multiple motors. It is not ideal to supply power from the microcontroller directly to the motors. Lastly, the GY-521 is a breakout board with a 3-axis gyroscope and accelerometer that can provide accurate data on the robot’s orientation and proper acceleration. Without this device, the self-balancing feature would not be possible in the robot design.

Additionally, the slave robot requires an extra sub-system in order to perform the following behaviors, and also avoid to touch any barriers. It is realized by using Raspberry PI which possesses a separate process of monitoring. The main function is to load the video input stream which is generated and sourced from the camera module. It uses an input video stream to evaluate the movement logic and transmit it to the Arduino to accomplish a smart following mode. The computer implements OpenCV software (real-time computer vision library) to detect human faces and track them, more details on its application in the robot will be provided in the following sections. The Raspberry PI uses the video data to evaluate the movement logic and transmit it to the Arduino to accomplish a smart following mode.

Modeling

The obstacle avoidance mode largely depends on the inputs from the IR sensors and ultrasonic sensor on the master robot. IR sensors can detect whether there is an obstacle on the left or right side of the car, and generate boolean signals *left_is_obstacle* and *right_is_obstacle*, which will control the turning action. The ultrasonic sensor can get the distance between the master robot and the obstacle in front. If the obstacle is too close or too far to the robot, it will move backward or forward respectively. Also, if the distance is within a certain range, where the master robot is "blocked by surrounding obstacles", it will turn right by default.

- State diagram of the master obstacle avoidance robot:

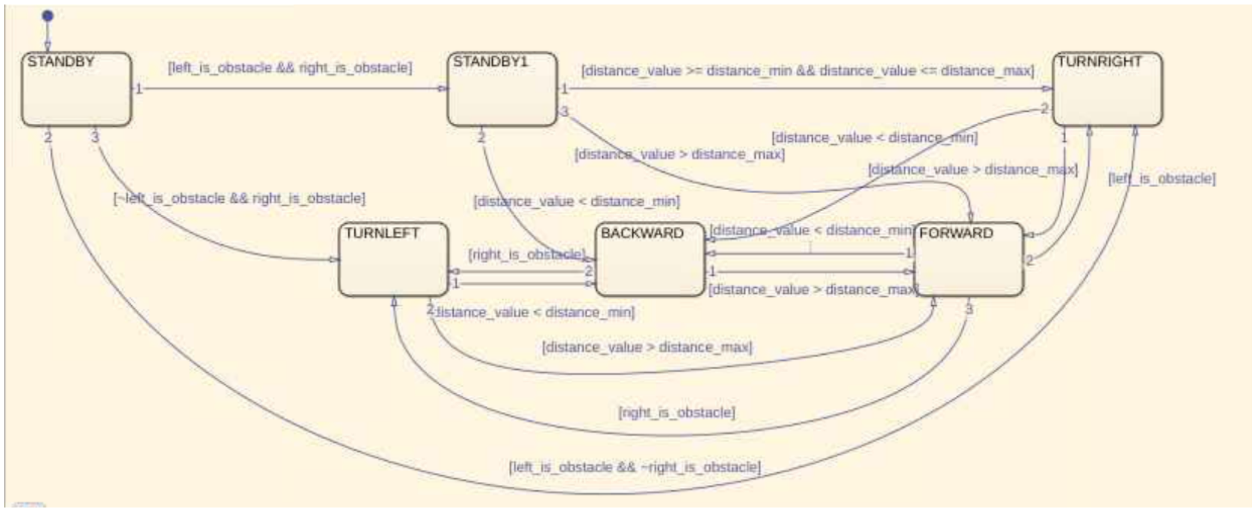
STANDBY
[left_is_obstacle && right_is_obstacle]
STANDBY1
[distance_value >= distance_min && distance_value <= distance_max]
TURNRIGHT
[distance_value < distance_min]
[distance_value > distance_max]
[distance_value > distance_max]
[left_is_obstacle]
[~left_is_obstacle && right_is_obstacle]
[distance_value < distance_min]
TURNLEFT
[right_is_obstacle]
BACKWARD
[distance_value < distance_min]
FORWARD
[distance_value > distance_max]
[distance_value < distance_min]
[distance_value > distance_max]
[right_is_obstacle]
[left_is_obstacle && ~right_is_obstacle]

Figure 2: State Diagram for master decision in the obstacle avoidance mode

- State diagram for the following mode by the slave robot:

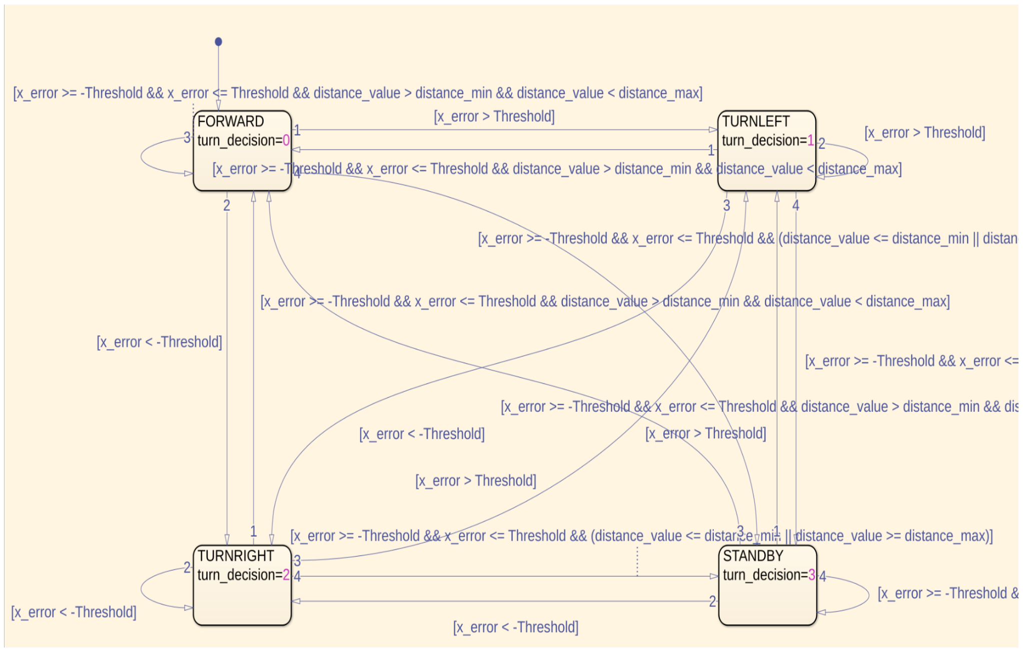
FORWARD
turn_decision=0
TURNLEFT
turn_decision=1
TURNRIGHT
turn_decision=2
STANDBY
turn_decision=3
[x_error >= -Threshold && x_error <= Threshold && distance_value > distance_min && distance_value < distance_max]
[x_error > Threshold]
[x_error >= -Threshold && x_error <= Threshold && distance_value > distance_min && distance_value < distance_max]
[x_error > Threshold]
[x_error >= -Threshold && x_error <= Threshold && (distance_value <= distance_min || distan
[x_error >= -Threshold && x_error <= Threshold && distance_value > distance_min && distance_value < distance_max]
[x_error < -Threshold]
[x_error >= -Threshold && x_error <=
[x_error >= -Threshold && x_error <= Threshold && distance_value > distance_min && dis
[x_error < -Threshold]
[x_error > Threshold]
[x_error > Threshold]
[x_error >= -Threshold && x_error <= Threshold && (distance_value <= distance_min || distance_value >= distance_max)]
[x_error < -Threshold]
[x_error >= -Threshold &
[x_error < -Threshold]

Figure 3: State Diagram for slave decision in the following mode

The following function involves the action responding to the bias input provided by the camera and the distance input provided by the ultrasonic sensor. The state space contains FORWARD, LEFTTURN, RIGHTTURN, and STANDBY for the slave robot. Each state has a corresponding output called turn_decision, which is used to control the physical behavior of the wheels.

When the camera physically detects the master robot through the range of the camera, the screen will be divided into three regions horizontally. We set up a constant threshold and it will determine the central region of the screen. It suggests where actually the following robot should go in order to get closer to the master robot by the shortest way. Based on the location of the master, whether it is in the left, right or central region, the Raspberry Pi will send an horizontal error rate to Arduino Nano through serial communication. Also, when the follower detects the master that is in the center region, the ultrasonic sensor will also determine the distance between the two robots. Whenever the two robots are getting too close to each other, the follower will stop and enter the STANDBY mode to prevent further collision.

Simulation

For the purposes of developing a path planner algorithm that could allow the robot to go through a maze autonomously, a combination of MATLAB and Simulink was used. Note that the functions come from the ROS, robotics system, and navigation toolboxes. The simulation model is basically a simulated robot going through an environment implementing a simple path planner and obstacle avoidance. It is implemented using 4 sub-systems.

First, process inputs which take in the (x,y) location and angle of the robot. The path is defined as a set of waypoints. The model has start and goal coordinates as inputs and the robot has information about its current location. So, even though the robot does not have knowledge of the map, it can move through it and reach the destination. Second, this subsystem is in charge of computing the linear & angular velocity and the moving direction. This is all computed using the 'Pure pursuit' controller, which is a path tracking algorithm that basically moves the robot from its current position to a look-ahead point of the robot. This is repeated until the robot reaches the goal coordinates which would be the final point of the path. Third, this subsystem is built to avoid obstacles and go through a free path. In other words, when the robot faces an obstacle it will then start to steer around until an obstacle free path is found and allows the robot to continue. This subsystem will check if the direction towards the path computed by the 'Pure Pursuit' controller is blocked by obstacles or not based on sensor data. Last, it is the output subsystem that executes the velocities to move the robot and drive it. This subsystem takes into account the data from 3 and 4. To prevent the simulated robot from collision in case of delays, only when new sensor data is available will it then send velocity commands.

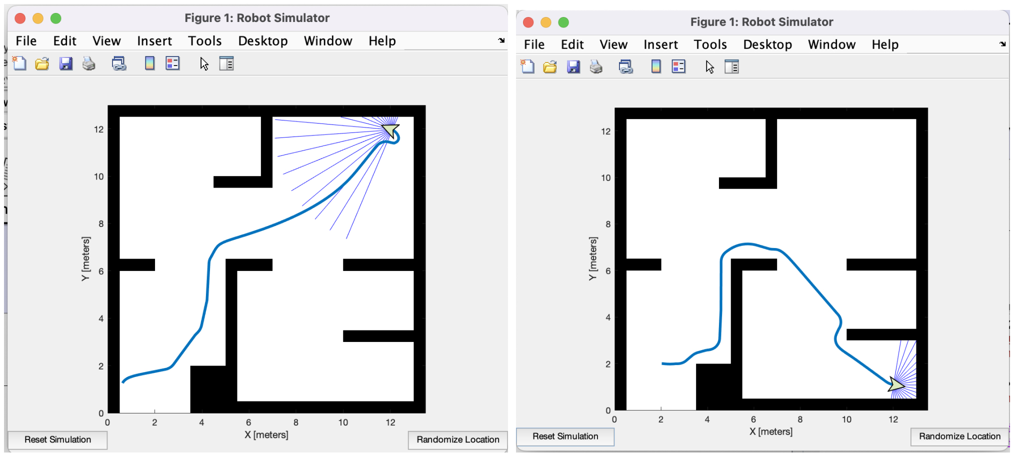
Figure 1: Robot Simulator
File Edit View Insert Tools Desktop Window Help
Y [meters]
X [meters]
Reset Simulation
Randomize Location

Figure 4: Path planner simulations

From Figure 4, it can be seen that effectively the simulated player is able to go from a starting location to a specified goal destination while avoiding the walls in the map. This scenario was tested with the Master robot built in real-life and it followed the path taken as the simulation on the right in figure 4. However, it is important to note that currently the simulation does not entirely reflect the algorithm implemented in the Master robot. The reason is because the Master does not have a component that can provide an accurate (x,y) location of itself in an unknown map. So, this simulation was built as a proof of concept that a non-complex path planning algorithm was a possibility to implement in the project to avoid having to use complex and computational heavy algorithms used in industry robots. Due to time constraints the path planner from the algorithm was not implemented in the project.

Detailed Specification and Working

Master, Obstacle Avoidance

The master robot has Arduino Nano, infrared sensors (IR), ultrasonic sensors, bluetooth module (BT-16), motors and wheels as its main components. The ultrasonic is placed in the front and IR sensors are each placed on the left and right of the robot at angles that ensure optimum functionality and help the master detect obstacles both in front and to the sides. The slave robot is equipped with the same components as the master along with additional components of Raspberry Pi 4B and a Raspberry Pi Camera Module.

The code for obstacle detection was fine-tuned using Arduino IDE. The logic is as follows:

- If the robot detects an obstacle to the right and left it keeps going straight:
    - If the distance between the robot and the obstacle in front of it is smaller or bigger than the threshold value, it moves backward or forward respectively.
- If the robot detects an obstacle to the left, right and in front of it and if the distance between the robot and the obstacle is within the threshold range, the robot is considered blocked. The program by default will make it turn right in this case.
- If the robot detects an obstacle only to the left, it makes a slight right turn and then senses its environment before moving again.
- If the robot detects an obstacle only to the right, it makes a slight left turn and then senses its environment before moving again.

In the current working prototype for the project we have achieved robot avoiding obstacles based on the data from the sensors. One of goals was to implement a specific path planning algorithm that will allow the robot to go from a start point to a certain destination. However, due to time constraints and more time spent on debugging than estimated, we were not able to achieve this specific goal for this project. A future extension of this project is making both the robots communicate with each other to inspect a specific designed maze.

Slave, Following and Obstacle Avoidance

The Pi Camera v2 module, which is mounted as an additional component on the slave robot to enables capturing the location of the master robot. In our prototype, we are using a face detection method to track a face stuck on the back of the master, rather than tracking the master robot entirely. A future extension to this and a more powerful solution, will be to train the system to be able to recognize the front object (Master Robot) as the target.

We use the model of the Haar cascade classifier to recognize the face and get the horizontal coordinate with the help of OpenCV library. By setting a customized threshold, the captured screen is divided into Left/Central/Right regions as introduced in the *Modeling* section. The horizontal coordinate difference (x_error) between the screen center and face is used to determine which region the master is currently in. As well, it is the final parameter that is passed in the transmission between the Raspberry Pi and Arduino.

The entire localization logic is calculated by the Raspberry Pi, and finally sends the motion error_rate in X that has been smoothed by the PID logic as the output to Arduino. This is used to determine the next step for the slave.

- IF The master is in the left region of the screen if x_error > 0, and Raspberry Pi sends an error_rate data into Arduino. While the Arduino is able to monitor and receive an error_rate of X. It will then be plugged into the Ultrasonic::FineTunning() function. And then to finally calculate how much power that the control unit should give for two separated motors to allow the Robotics to turn at a specific angle.
    - The way of percentage error based on the Angle Threshold and Power Of Motor Threshold could be suggested as:
        - *percentage = (Ultrasonic::error_rate - threshold) / threshold*
        - *power = motor_threshold * percentage*
- IF The master is in the right region of the screen if x_error < 0, and Raspberry Pi will also send an error_rate data into Arduino. While the Arduino is able to monitor and receive an error_rate of X. It will then be plugged into the FineTunning() function to calculate the percentage error and power for each motor, the same as if the front object is in the left region.
- IF The master is in the central region of the screen, that is determined by the calculation within the ALU of Arduino Nano, if x_error <= threshold && x_error >= -threshold, it will be treated as the object is placed at the central region of the vision, and being "safety" without any action and fine tuning based on horizontally.

The direction of where the Robots should turn, could be determined by the sign of error_Rate that has been transmitted from Raspberry Pi. Just like the initial if statement that we are using to judge whether the front object is on the right or left side of the robot.

The two photoelectric sensors have been applied and used within the Following mode, too. The major difference of the application of those two sensors has been in obstacle avoidance, which is they have been used for checking surroundings as well, but only "Check". In case, once the 39 square waves continuously emit infrared light and receive the "feedback", the interrupt of the IR-sensor will be raised. But the difference is, it will let the Robots immediately "STANDBY", instead of making an action of tuning the angle of the robots in Obstacle avoidance.

- The interrupt of the IR-sensor and Ultrasonic are mutually affecting each other, which intend to prevent the robots from colliding with any side of the obstacles. Camera detection and tune has been involved and wrapped into Ultrasonic:check() function. It integrated with the Ultrasonic sensor interrupt and adjusting the angle of the robots in real-time, and also keeping the robot to remain in an appropriate distance from the tracking object (In our demo, it is the Master Robot).

In another word as conclusion, the Slave robot will always try to keep tunning the point of view of itself to direct facing the object that it is tracking; And meanwhile, keep closer enough to the tracking object without crash.

The communication between Raspberry Pi and Arduino for the slave robot is through the serial port. Since we just need the robot to receive the package of data from the Raspberry Pi instead of the bidirectional transmission. The simplest way to build the connection between Raspberry Pi and Arduino, is hardware connection by the USB connector of two boards. This also allows us to connect through USB for our Arduino Nano.

- At the Arduino side, we need to start a Serial and choose a baud rate, and then listen. We set up the most commonly used baud rate between our Raspberry Pi and Arduino of 9600. We keep it always monitoring the serial port when it is available, then receive the package of data that is sent from the Raspberry Pi for every 1/100 seconds.
- On the Raspberry side, we design an asynchronous structure to package and send the data in a specific frequency since the Sending And Awaiting feedback() function and Object Tracking() function might interact with each other and because those two functions cost different time to execute and finish. If we make them implement in a synchronous loop, it will potentially cause a certain amount of delay to await each other to finish. It will be okay without applying any transmission of the data itself, but would be fatal when that data is transmitted to another device.
    - Thus, we design an start_regonize() function to only calculate the error_rate that we described as above. Meanwhile, we also applied a function to convert and transmit the data through the serial port only.
    - We make the Raspberry PI enable to send an “error_rate” string object by applying the line of code `Serial.write()` appends with a newline character `\n` at the end of the string, followed by a certain rate in function sendTO_Arduino(). The default value of the sending has been set up by 100 times/ per-second.

At this time, since the sendTO_Arduino() and start_regonize() functions are not affected and wait for each other. The `error_rate` would be checked and calculated in the most current and efficient way, and sent the `error_rate` by an asynchronous thread. The program would be able to run in best efficiency to improve performance and reduce the delay among the two device serial communication.

Software and Hardware components used

Component	*Price (USD)*
Elegoo Tumbller Self-Balancing Robot Car V1.1 (x2)	$160
Raspberry Pi Noir Camera Module V2	$20
Raspberry Pi Model 4B	$80
Micro HDMI & cables	$25
WENSILING 2 Pack 3.7V 20A 3200mAh Flat Top Rechargeable Battery (For Raspberry pi)	$25
Arduino IDE	$20
SunFounder Raspberry Pi UPS Power Supply Module V2.0	$20
Python (to write the code for Raspberry Pi)	
C++ (to write the code for Arduino)	
Total	**$350**

Methodology and Technical Challenges

1. First the self-balancing robots were built following the instructions from the ELEGOO Tumbller Manual and tested to confirm the assembly was correct.
2. The control flow for the code (written in C++ using Arduino IDE) for self-balancing and obstacle avoidance robot was first understood and then referenced from the tutorials provided by Elegoo Tumbller. It was fine tuned to construct an optimal obstacle avoidance function. PID calculations were kept as is because self-balancing was not the primary focus of this project.
3. Achieving optimal speed for obstacle detection was a challenge as the master robot crashed into obstacles because its speed was too fast to process the data from the sensors. Different speeds were tried for moving the robot FORWARD, TURN_LEFT and TURN_RIGHT in isolation. The robot was tested on a variety of mazes. It was found that the robot's optimum speed was achieved when the moving FORWARD speed was decreased and the speed for turning both TURN_RIGHT and TURN_LEFT was kept equal.
4. Once, the obstacle avoidance for the master robot was achieved, we started implementing the slave following functionality. To build the follower functionality our initial idea was to use bluetooth module to communicate between the master and the slave robot. Firstly we tested if the robot was able to communicate with the app on the phone by referencing the datasheet for the BT-16 bluetooth module and the Elegoo Tumbller Manual to write a basic code. It was found that both the robots were in the slave mode. To change one of the robots to act as a master, AT Commands were referenced from BT-16's datasheet and entered from the Arduino IDE Serial Monitor. However, there was no response from the device despite giving any command (when the expected response was an 'OK' message).
   Various measures such as trying out different BT-16 chips, using Arduino Uno instead of Arduino Nano, building an intermediate circuit using breadboard and resistors were used to debug the issue, but we were not able to generate a response from the device. We then contacted the customer service (using the contact number provided in the BT-16's datasheet), where we understood that mode is hardwired and we as customers cannot change it.
   As an alternative solution, Raspberry Pi camera along with serial communication between Raspberry Pi and Arduino was used to achieve the following functionality.
5. We decided to use an alternative solution where Raspbery Pi camera on the slave captured the coordinates of a face stuck on the master and communicated this data with Arduino. Movement decision was done on the Raspberry Pi at each clock cycle. Inside Arduino, the turning speed is set for PID control to ensure the balance of the car.
6. Initially, a dummy serial communication between Arduino and Raspberry Pi was established where the Raspberry Pi sent a simple string message which was read by the Arduino. Later, the required turning decision was sent from the Raspberry Pi to Arduino which allowed it to follow the master correctly.

7. In this project, Picamera2 and cv2 libraries were used for master detection. The Pi Camera keeps capturing frames in a permanent loop and OpenCV will flip and graying the frame. The face attached to the master is detected by a CascadeClassifier and then OpenCV will get the pixel coordinates.

Final Demos

1. Master Obstacle Avoidance mode in the maze
   This video shows maneuvering of the master robot based on the logic explained under the *Detailed Specification and Working* section. Here we built the same maze simulated in the MATLAB Simulink, and the master robot had a good performance on moving around the maze without colliding with any obstacle.
2. Slave Following mode in the maze
   This video shows the slave following the master using the Raspbery Pi Camera and the Raspbery Pi - Arduino serial communication (as explained under the *Detailed Specification and Working* section). Here the master is moved around manually, and the slave robot successfully followed and repeated the actions of the master.
3. Framestream from Pi Camera on the slave robot
   This video demonstrates the slave real-time detection while doing following actions. The framestream captured by Pi Camera 2 was being displayed on the VNC viewer, where the master (face) is being detected at the same time with a blue rectangle box as the indicator.
4. Master leads slave in real-time
   This is the final demo of everything working together. Here the master is moved around using bluetooth communication between itself and the Elegoo App on one of our phones. We controlled the master to perform random movements and the slave was able to follow it most of the time. Due to the delay issue of serial communication, the slave robot is possible to miss the master if it makes a big turn in a short period. In that case, the slave can easily fail to detect the master. Future work is expected to improve the performance.

Analysis

Even though we have reduced the timing delay of the slave responding to the master when turning at angles, there is still a slight delay of 0.4s - 0.6s. The slave successfully follows the master, when the latter moves straight or turns at slight angles. However, when the master makes abrupt sharp turns the slave stalls and does not know how to respond. We think that the time taken to process calculations on the boards and the serial communication between the two boards is the main cause of the time delay in our system. Due to time constraints, we were not able to completely solve this problem.

An effective solution to calculate the timing delay will be to insert a timer in the code for the slave robot. This will help capture the time span spent between the detection of the face on the master robot by the slave robot and its reaction by physically moving to follow the master robot. This will give a more accurate representation of the timing delay involved in our prototype.

Division of work on team members

	Build Robots	Stateflow modeling + Simulation	Obstacle Detection	Bluetooth Module testing	Camera and OpenCv detection	Arduino Raspberry Pi Communication	Testing	Report
Guillermo	X	X		X			X	X
Piyusha			X	X		X	X	X
Xiaopeng	X	X	X		X	X	X	X
Zhaozhong	X		X	X	X	X	X	X

Conclusion

In sum, this project was able to achieve the development of a robotics system that could go through an unknown environment while avoiding obstacles. The overall system is composed of two self-balancing robots that are distinguished as Master and Slave relationship. Each robot has a unique main function. The master has the obstacle avoidance feature that prevents the system from collisions and going through the maze. Even though our initial plan was to use a Bluetooth module to communicate between the two robots, there was a technical limitation of hardwired robot modes that could not be changed. Hence, we chose an alternative solution where the slave has the following mode that tracks the master robot through a camera by identifying a face using OpenCV software. In terms of the deliverables, we were able to complete the obstacle avoidance function and the master robot was able to complete the maze. A simple planner was experimented through simulations but time constraints prevented us from fully implementing this in the actual robots. However, we noted that for more complex maps a path planner is necessary to be implemented into the system. Possible methods include probabilistic roadmaps or A* algorithm. These would require extra hardware since they need to know the robot's coordinates in a map in order to compute a possible path. The following mode was almost complete, the reason is because some parameters still needed to be properly tuned. The slave robot was capable of following the master robot, but if sharp turms were made then the slave would lose track and stop. Also, there is a slight timing delay of a few seconds that the slave takes to react when following the master.

Further improvement to our current design is to implement a path planning algorithm and a mechanism where both the robots communicate with each other to inspect a specific maze. Both these robots can be equipped with IR temperature sensors for human detection and sound sensors to alert incase of intruders. This will help achieve surveillance and security inspection of a certain area with more precision than watchmen can achieve by ruling out the problem of human bias.

References

1. 2-Way Communication between Raspberry Pi and Arduino – Automatic Addison. https://automaticaddison.com/2-way-communication-between-raspberry-pi-and-arduino/
2. Dejan. "How to Configure and Pair Two HC-05 Bluetooth Modules as Master and Slave | at Commands." *HowToMechatronics*, 15 Apr. 2016, https://howtomechatronics.com/tutorials/arduino/how-to-configure-pair-two-hc-05-bluetooth-module-master-slave-commands/
3. Dokania, Harsh. "Object Tracking Camera Using Raspberry Pi and OpenCV." *IoTEDU*, 20 June 2020, https://iot4beginners.com/object-tracking-camera-using-raspberry-pi-and-opencv/
4. *DX-BT16 Bluetooth module Menu* https://epow0.org/~amki/car_kit/Datasheet/ELEGOO%20BT16%20Bluetooth%20UART%20Module.pdf
5. "ELEGOO Tumbller Self-Balancing Robot Car Kit Compatible with Arduino IDE." *ELEGOO Official*, https://www.elegoo.com/products/elegoo-tumbller-self-balancing-robot-car
6. "ELEGOO Tumbller Self-Balancing Robot Car V1.1/V1.0 Tutorial." *ELEGOO Official*, https://www.elegoo.com/blogs/arduino-projects/elegoo-tumbller-self-balancing-robot-car-tutorial
7. "Face Tracking Robot with an ESP32-CAM – Robot Zero One." *Robotzero.one*, 10 Apr. 2020, https://robotzero.one/face-tracking-robot/
8. Gopal, Venu. "Testing Bluetooth at Commands." *Engineers Garage*, www.engineersgarage.com/testing-bluetooth-at-commands/. https://www.engineersgarage.com/testing-bluetooth-at-commands/
9. "Path Following with Obstacle Avoidance in Simulink - MATLAB & Simulink." *www.mathworks.com*, https://www.mathworks.com/help/nav/ug/path-following-with-obstacle-avoidance-in-simulink.html
10. "Raspberry Pi Arduino Serial Communication - Everything You Need to Know." *The Robotics Back-End*, 11 Nov. 2019, https://roboticsbackend.com/raspberry-pi-arduino-serial-communication/