Pioneer Academics Research Program

Swarm and Search Systems

An Application of Swarm Robotics to the Exploration of Fireground Environments

Shiv Kampani

Spring 2021

Submitted to: Carlotta A. Berry, Ph.D.

June 27, 2021

Executive Summary

This paper examines the dangers associated with the exploration of fireground environments by firefighters, such as low visibility, obstructions and high smoke concentration. The solution proposed by this paper makes use of a swarm of small-sized, coordinated robots to explore, map out and gather essential data about large fireground environments. Existing methods of exploring unfamiliar fireground environments as well as the scope and applications of swarm robotics have been discussed. While describing the proposed solution, this paper justifies the inclusion of all components, describes design criteria as well as all hardware and software subsystems. Tests carried out on the proposed system in a simulated fireground environment have also been evaluated. The system was found to be efficient, cost-effective and scalable due to the small size and low weight of individual robots. It was determined that the proposed system is able to collect essential data about unfamiliar fireground environments at a rate of 0.029365 m^2 per robot per second. Lastly, recommendations about the implementation of the proposed system in a real-world situation have been discussed.

Table of Contents

1 Introduction

Fireground search and rescue operations present several challenges to firefighters. Unfamiliar fireground environments may contain obstructions such as locked doors and heavy furniture; high smoke concentration may reduce visibility; a lack of information about the fireground environment may contribute significant delays to the process of locating survivors. The process of manually searching for survivors may lead to overexertion, a leading cause of firefighter fatalities. According to “Firefighter Fatalities in 2019 - US” (Fahy et al., 2019), prolonged exposure to smoke, carcinogens and other contaminants present in fireground environments could lead to cancer, heart disease, cardiac issues and other long-term effects on firefighters’ physical health. Furthermore, search and rescue operations in fireground environments are stressful and traumatic. Such operations have even been linked to long-term impacts on firefighters’ emotional health, as well as anxiety, depression and post-traumatic stress disorder in some firefighters. Fahy et al. also identify “Overexertion stress/medical” as the leading cause of firefighter fatalities (54% of firefighter fatalities). Thirteen percent of firefighter fatalities have also been attributed to “Rapid fire progress/explosion”. Thus, it may be safer and more efficient for automated robotic systems to survey the unfamiliar environment and collect essential information about the surroundings and locate survivors. Information gathered by automated robotic systems can additionally be used to inform routes chosen by firefighters.

The application of robotic systems to emergency situations is a widely researched field. “Robots in Crisis Management: A Survey" (Kostavelis et al., 2017) presents an overview of specialized robots with applications in crisis response. Although this paper describes applications of robots in emergency situations ranging from urban search to bomb disposal, some of the systems described have applications in fireground environments. Legged robots, including hexapods and humanoid robots, are able to traverse uneven terrain and climb staircases. However, these solutions are extremely large and difficult to deploy. More compact snake-like robots have also been described which can also maneuver through obstacles, but these may not be able to capture a stable video feed of the fireground environment due to the motion of the robot’s head during locomotion. Yet another disadvantage of these systems is that they are not robust, in that, if one component or subsystem of these robots fails, then the overall efficiency of the search operation is reduced significantly. Thus, a swarm robotic system which relies on coordination between multiple, small-sized robots may be most efficient in surveying large, unfamiliar fireground environments.

2 Research and Literature Review

Ihsan and Marhoon (2018) discuss the implementation of a Bluetooth-controlled Arduino based robot which uses sensors to detect fire in closed areas. The paper initially describes the design criteria of a mobile robot system: mobility, autonomy and perception. In order to achieve mobility, the author evaluates legged locomotion and wheel-based locomotion. The robot is able to perceive (that is, make observations about its surroundings) using a flame sensor and an RF camera. This system uses a DC water pump to extinguish any fires that it detects, and is controlled over Bluetooth.

“Development of Mobile Robot with Sensor Fusion Fire Detection Unit” describes another mobile robotics system designed for use in a fireground environment, similar to Ihsan and Marhoon (2018). However, this paper also contains a sensor fusion method which describes how the probability that a fire is present can be determined using inputs from 3 sensors: a temperature sensor, a smoke sensor and a flame sensor. Using regression analysis, the coefficients of the readings of the temperature sensor, smoke sensor and flame sensor were determined to be 0.16, 0.07 and 0.77 respectively.

Navarro and Matía (2013) describe a comprehensive taxonomy of swarm robotics systems and identify several fundamental characteristics that every swarm robotics system must possess. Some of these characteristics include homogeneity, autonomy and collective behavior. This paper also describes precursory taxonomies for swarm robotics and mentions several existing swarm robotic systems. Lastly, this paper analyzes several swarm control algorithms (or applications of swarm control systems) such as aggregation, dispersion, and collective mapping of unknown environments. Navarro and Matía (2013) state that individual robots are incapable of or inefficient at performing the overall task of surveying the environment. Thus, for the robots to map an environment efficiently, swarming is required. Individual robots must be able to communicate and collaborate with nearby robots.

“Swarm Robots with Queue Organization Using Infrared Communication” (de Mendonça et al., 2012) describes the use of e-puck (E-Puck.org, 2018) robots to demonstrate autonomous, collective behavior of a swarm during the process of queue organization, or the process in which robots organize themselves into a queue. This paper discusses swarm intelligence and

explains the functions as well as rationale behind including all of the sensors present on the e-puck robot. The paper also includes pseudocode explaining some of the algorithms used for communication as well as object avoidance. Moreover, the paper comments on issues associated with infrared communication in swarm robotic systems. Due to reflections of emitted infrared light from objects in the environment, the infrared readings from receivers contain lots of noise, which might impede communication between robots. Noise due to infrared light present in the environment may be a large concern due to infrared emissions from fire. Furthermore, the elimination of this noise using digital filters may be complicated and unreliable in dangerous fireground environments. Nevertheless, this paper describes the use of a digital filter to reduce noise, which greatly increased the robots' abilities to communicate with one another.

3 Design Criteria

When designing robotic systems to aid fire fighters, the efficiency of the system (or the amount of time required by the system to gather essential data about the surroundings) is the most important factor, as delays in search and rescue operations could result in fatalities. Weight and size are also important constraints because of the fact that larger robots may be difficult to deploy and may not be able to traverse through an environment that contains small spaces or several obstructions. Lastly, cost is an important design criterion as low-cost solutions are significantly easier to implement. Since multiple robots are generally used in a swarm robotics system, the cost for each robot must be low for the entire system to be scalable and cost efficient. The aforementioned design criteria can also be defined quantitatively:

- Efficiency - the swarm robotic system can be considered efficient if it is able to gather essential information at a rate of 1 m^2 per robot per minute.
- Weight and size - each robot in the swarm robotic system must not weigh more than 1.0 kg. Each robot should also be able to fit inside a 30 cm by 30 cm by 30 cm cube.
- Cost - the cost of each robot in the swarm robotic system must be no more than $20.00.

4 Design Specifications

The proposed solution to problems posed by firefighter-led search and rescue missions uses swarm robotics. Swarm robotics is the study of collective behavior that emerges from the interactions between simple robots. The use of several small robots to collect essential data about unknown environments may be an efficient approach of mapping out fireground environments.Individual robots used in a swarm robotic system must be able to move as well as gather essential data about the fireground environment using sensors. Since robots must communicate with one another to achieve collective behavior, each robot must have the ability to both send data to and receive data from robots around it or a centralized receiver. Robots should also be able to determine either their absolute position in the environment (expressed as a distance and angle from the entry point into the environment) or their position relative to other robots in the swarm.

Within the scope of this research paper, essential data about the simulated fireground environment is assumed to be the positions of any and all fires and obstacles in the environment. Thus, the proposed system must be able to measure distance between itself and the nearest obstacles in the environment as well as detect fire.

4.1 Detecting Fire

In order for robots to be able to determine the positions of the nearest fire, they must be able to detect smoke emitted by the fire, infrared light emitted by the fire, or any rises in temperature that occur while approaching the source of fire. The three possible sensor configurations have been outlined in the decision matrix on the next page.

Table 1: Decision Matrix for Fire Detection Sensors

Feature	Weightage	IR-based Flame Sensor	MQ2 Smoke Concentration Sensor	TMP36 Temperature Sensor
Low Cost	30%	5	4	5
Range	20%	3	5	4
Accuracy	50%	5	2	1
Total	100%	4.6	3.2	2.8

In the decision matrix above, each sensor has been given a score between 1 to 5 inclusive, on the basis of three design criteria: low cost, range and accuracy. Range is measured differently for different sensors. For the IR-based Flame Sensor, it should be able to detect a fire that is between 20cm - 150cm away. The MQ2 Smoke Concentration Sensor should be able to detect between 300-1000 ppm of smoke, and the TMP36 temperature sensor should be able to detect temperatures between 20°C-150°C.

Accuracy, within the scope of this paper, refers to the ability of the sensor to determine the position of the fire. Although smoke and temperature sensors can approximate the location of a fire based on changes in the temperature and smoke concentration readings, it is possible that extraneous factors present in the fireground environment are responsible for changes in these readings. The flame sensor, however, would be able to determine the position and direction of the fire relative to the robot as it is able to detect infrared light emitted directly from the fire.

“Development of Mobile Robot with Sensor Fusion Fire Detection Unit” (Sucuoglu, H. S. et al., 2018) describes a sensor fusion method of determining the probability that a fire is present using inputs from three sensors: a temperature sensor, a smoke sensor and a flame sensor (all three sensors are similar to those described in the design-matrix above). Using a linear regression analysis, the coefficients of the readings of the temperature, smoke and flame sensors were determined to be 0.16, 0.07 and 0.77 respectively. Since the coefficient of the

flame sensor (0.77) is significantly greater than that of the temperature sensor (0.16) and the smoke sensor (0.07), the flame sensor would be able to determine the position of a fire in the environment more accurately.

Based on the design matrix, the flame sensor has been selected in order to detect fire.

4.2 Measuring Distance to Obstacles

In order for robots to be able to determine the positions of the nearest obstacles, they must have an IR-based, sonar-based or LIDAR-based distance measuring sensor. As with sensors that detect fire, range, cost and accuracy are important design criteria. Consider the following design matrix:

Table 2: Decision Matrix for Obstacle Detecting Sensors

Feature	**Weightage**	**IR-based distance sensor**	**HC-SR04 Ultrasonic distance sensor**	**LIDAR**
Low Cost	30%	4	5	1
Range	20%	3	5	3
Accuracy	50%	4	3	5
Total	100%	3.8	4.0	3.4

Since the fireground environment will contain objects and fires that are about 4cm-200cm away from the robot, robots should be able to detect the distance to any obstacles within that range. Although LIDARs and IR-based distance sensors are able to detect object distances within this range in a normal environment, in a fireground environment with high smoke concentration, their ranges are reduced. Moreover, the accuracy of the LIDAR and IR-based distance sensors is also reduced in smoke-filled environments. As described by Starr et al. (2013), LIDARs are inaccurate for areas with dense smoke and low visibility (<5m). LIDARs are also significantly more expensive than ultrasonic sensors. As mentioned earlier, low cost of individual robots is an important design criterion of a swarm robotic system.

Based on the design matrix, the HC-SR04 ultrasonic sensor has been selected.

4.3 Communication Mechanisms

In a swarm robotic system, robots must be able to communicate with each other or with a centralized controller. There are two possible approaches to implementing a communication mechanism between robots: Bluetooth-based or infrared-based robot communication. Consider the table below, which outlines the advantages and disadvantages of both these mechanisms.

Table 3: Advantages and Disadvantages of Communication Mechanisms

Method	Advantages	Disadvantages
Infrared communication	● Robots may make use of infrared sensors to share information about relative position and orientation. This would eliminate the need for additional sensors required for localization.	● Smoke present in the environment may limit the range of infrared sensors. ● The use of multiple infrared sensors may be expensive. ● Multiple sensors may take up a lot of space on the robot.
Bluetooth communication using the HC-05 module	● Low cost ● Bluetooth communication may also be used for communicating with a central receiver. ● Bluetooth communication works efficiently even in smoky environments.	● Bluetooth communication does not allow a robot to determine its position relative to other robots in the swarm. Additional sensors would be required for localization in the environment.

Bluetooth communication would be more cost and space efficient to implement given the small size of individual robots in the swarm. Furthermore, as described by Starr et al. (2013), the range of infrared-based communication mechanisms would be reduced in smoke filled areas, where communication between robots is critical. In smoke filled areas robots must be able to coordinate to rendezvous at the site of a fire. Thus, Bluetooth communication is more reliable and efficient than infrared-based communication.

4.4 Localization

While surveying the fireground environment, swarm robots must be able to construct a map of the environment to guide firefighters or other rescue efforts. In order to do so, robots must be able to determine their absolute position in the environment or their position relative to other robots in the swarm. In order to determine absolute position, a combination of optical encoders and a gyroscope can be used. Determining relative position, however, can also be done using the ring of infrared transmitters and receivers. Since infrared sensor range may be limited by the presence of smoke, it is an unreliable mechanism to determine a robot's location in the swarm. Moreover, flames emit infrared light. Proximity to a fire could lead to unreliable measurements due to the infrared sensors on a robot sensing light emitted from the fire itself. Robot localization is critical, especially near fires, so that the exact positions of fires can be determined and communicated to firefighters. Hence, optical encoders and a gyroscope will be used to determine the location of the robot relative to its starting point.

5 Design Method

Each robot in the swarm must be able to navigate through the fireground environment as well as communicate with neighbouring robots; accordingly, it must have a number of subsystems. In the proposed system, each robot has three main subsystems: the drive subsystem, which enables the robot to move around and explore the environment, the sensor subsystem, which enables the robot to determine the positions of any objects, obstructions or fires in the environment, and the communication subsystem, which enables robots to communicate with each other as well as with a centralized controller.

Each of these subsystems is controlled by the Arduino UNO microcontroller. Eight AA batteries have been used to provide power for each robot. When these subsystems are

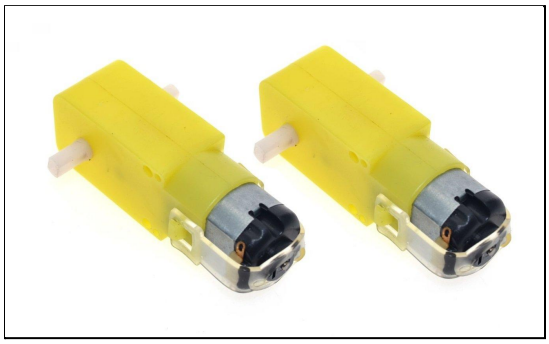
integrated, each robot is able to explore and collect essential data about the environment. With the use of a centralized controller and many such robots, the proposed swarm robotic system can be constructed. The following section of this paper will contain information about the purpose as well as the hardware and software specifics of each of the aforementioned subsystems as well as full system integration.

5.1 Drive Subsystem

5.1.1 Hardware Components

The purpose of this subsystem is to allow each robot in the swarm to move around and explore the fireground environment. The drive subsystem consists of the following electronic components:

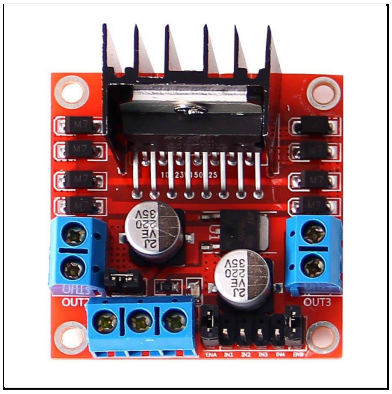
Figure 1: 150 RPM Geared DC Motors



From *150 RPM Dual Shaft Motor*. RoboHaat. (2021, March 19). https://robohaat.com/product/150-rpm-dual-shaft-motor/.

The purpose of including these parts in the subsystem is to drive individual robots. These motors have been selected as they are inexpensive (~$0.67 each) and light-weight (~30 g each) relative to other DC motors; thus, they satisfy the design criteria for the proposed system.

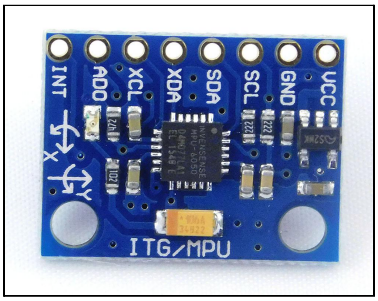
Figure 2: L298N Motor Controller



From *L298N DC Motor Driver Module*. Robocraze. (n. d.). https://www.amazon.in/Robocraze-L298-Motor-Driver-Module/dp/B072NCPM5R.

The purpose of including this part in the subsystem is to control the speed and direction of both motors. In addition to being a dual full-bridge controller, this motor driver was most readily available and is inexpensive (~$2.70), hence its selection.

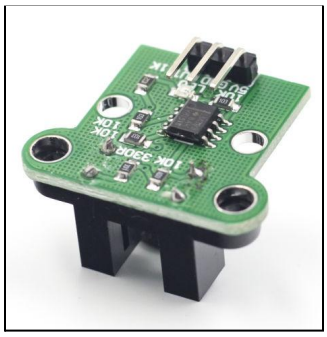
Figure 3: MPU-6050 Six-Axis Gyroscope and Accelerometer



From *MPU6050 - 3 axis gyroscope and 3 axis accelerometer*. Nettigo. (n. d.). https://nettigo.eu/products/mpu6050-3-axis-gyroscope-and-3-axis-acclerometer.

The purpose of including this part in the subsystem is to measure the rate of rotation of the robot in the yaw-axis. The rate of rotation in the yaw-axis can be integrated, with respect to time, to calculate the yaw-angle, tilt, or the heading, of the robot relative to its starting orientation. This value can be used to calculate the position or path travelled by the robot along with measurements of distance travelled by the robot.

Figure 4: HC-020K Optical Encoders

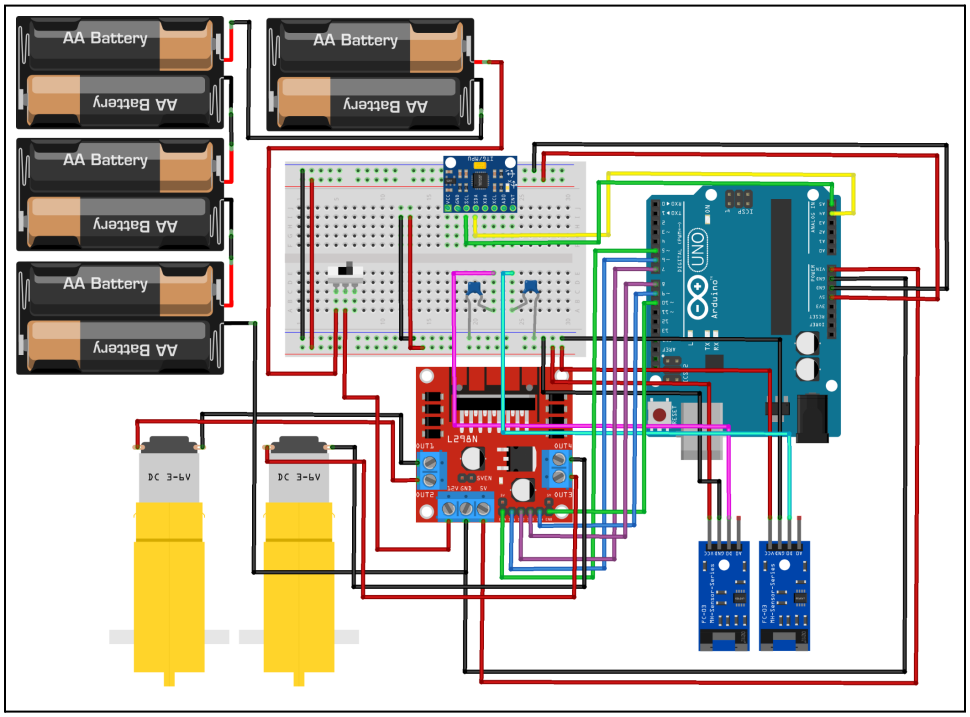


From *Hc-020K Double Speed Measuring Module.* Kuongshun Electronic Limited. (n. d.). https://kuongshun.en.made-in-china.com/product/CypQbJRGsMcm/China-hc-020k-Double-Speed-Measuring-Module-with-Speed-Encoder-Kit.html.

The purpose of including this part in the subsystem was to measure the RPM (revolutions per minute) of each motor of the robot. Using these measurements and the diameter of the wheel (in cm), the speed of the robot can be determined (in cm/s). By integrating the speed of the robot, with respect to time, the distance travelled by the robot can be computed. Along with measurements of yaw-angle from the MPU-6050 gyroscope, the robot's absolute location and path travelled can both be determined, which satisfies the localization criterion of the previously outlined design specifications.

5.1.2 Circuit Diagram

Figure 5: Circuit Diagram of the Drive Subsystem



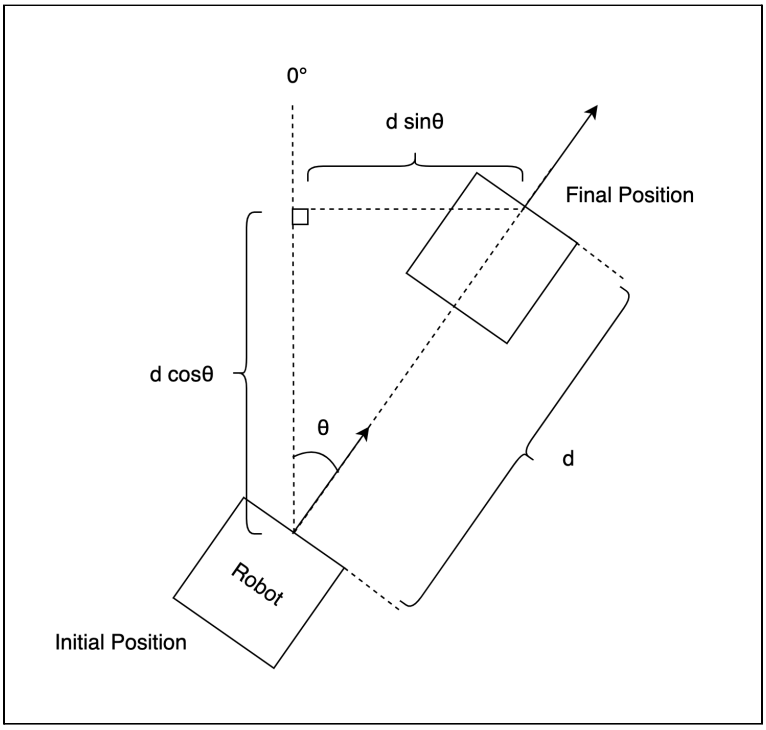
The circuit diagram above shows the breadboard-layout of the drive subsystem, with all of the aforementioned electrical components as well as an Arduino UNO microcontroller, 8 AA Batteries, a slide switch and 2 100 nF capacitors.

5.1.3 Software

Localization

In order to map out the path taken by the robot, measurements are taken from the optical encoders and the gyroscope respectively. This allows us to determine the heading or yaw-angle of the robot relative to the starting yaw-angle (0°) as well as the total distance travelled by the left-motor and the right-motor at any time t_0. Since measurements are taken at regular intervals of time T, where T is the time between loop function calls in an Arduino program, measurements are also taken at time t_1, where $t_1 - t_0 = T$. Consider the diagram below, showing the motion of the robot between two successive measurements, separated by time interval T.

Figure 6: Diagram of Robot Displacement

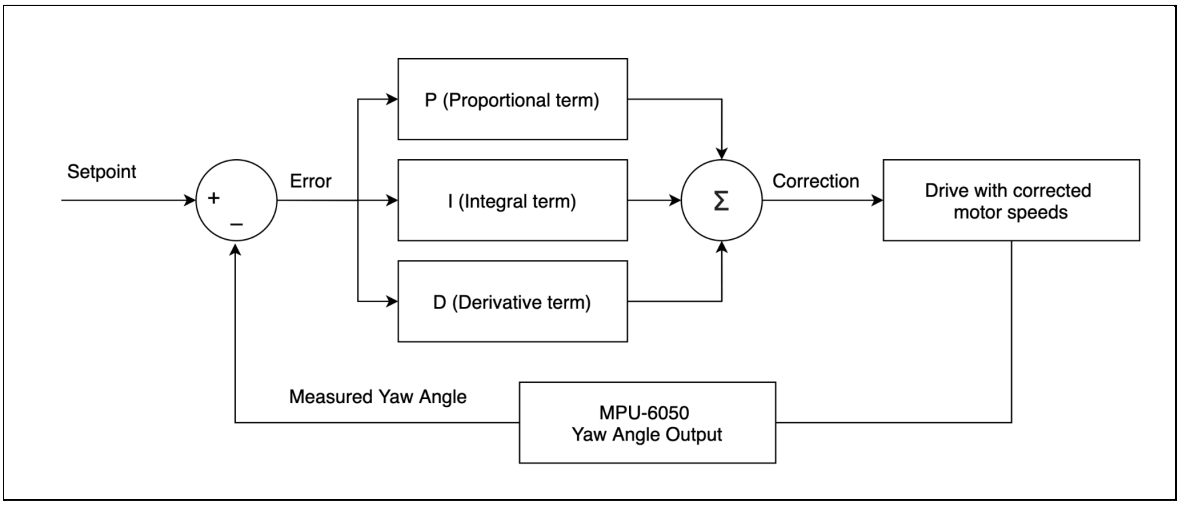


In the given time interval T, the robot has moved a distance of d (in cm) at a yaw-angle of θ°, relative to 0°, or the orientation of the robot at the start of the program. Since T is small (in milliseconds), it can reasonably be assumed that the robot is moving in a straight-line or not changing direction during the time interval T. In the diagram, “Initial Position” represents the position of the robot at time t_0, and “Final Position” represents the position of a robot at time t_1, after an interval of time T. Using trigonometry, the vertical and horizontal displacement of the robot every T milliseconds can be calculated. Hence, given that the unfamiliar environment can be approximated by a 2D x-y plane, the (x,y) coordinate of each robot at any given point in time is known as the starting location of each robot is known and the displacement of the robot in the vertical and horizontal sections can be calculated every T milliseconds. Thus, the absolute position of the robot at all times is known. With these designations, the localization criterion of the design specifications can be met.

PID Control

During initial tests of driving an individual robot, it was observed that even when both the motors were assigned the same speed by the software program, the actual RPM speeds of the motors were different. This subsequently caused the robot to veer off instead of travelling in a straight line. This is because the motors that are being used (150 RPM Geared DC motors) are relatively inexpensive, higher RPM and may be of lower quality than more expensive motors. Thus, even when they are assigned the same speed, differences in the gearboxes of each of the motors would result in them travelling at different RPM speeds. A hardware solution to this problem would not be feasible as low cost is one of the design criteria of this solution and using higher quality motors would increase the cost of each individual robot. Hence, a software solution was implemented to solve the problem. PID (proportional-integral-derivative) control is a method of controlling systems by calculating the error in the system and applying a correction based on that error.

Figure 7: Diagram of Robot Displacement



The diagram above shows a PID feedback loop or PID controller. PID controllers consist of three components: a proportional term, an integral term and a derivative term. Before these components are calculated, the error of the system is determined, which is the difference between the system's current state and the desired setpoint state. The proportional term is equal to the error multiplied by a constant K_P. The integral term is equal to the integration of the error term, with respect to time, multiplied by a constant K_I. The derivative term is equal to the first derivative of the error term, with respect to time, multiplied by a constant K_D. The correction term is the summation of the three PID

components. The correction term is then applied to modify the behaviour of the system. The state of the system is then measured by a sensor and the PID loop repeats itself. Over time, the system corrects itself and performs as originally intended by reducing the error term to approximately 0.

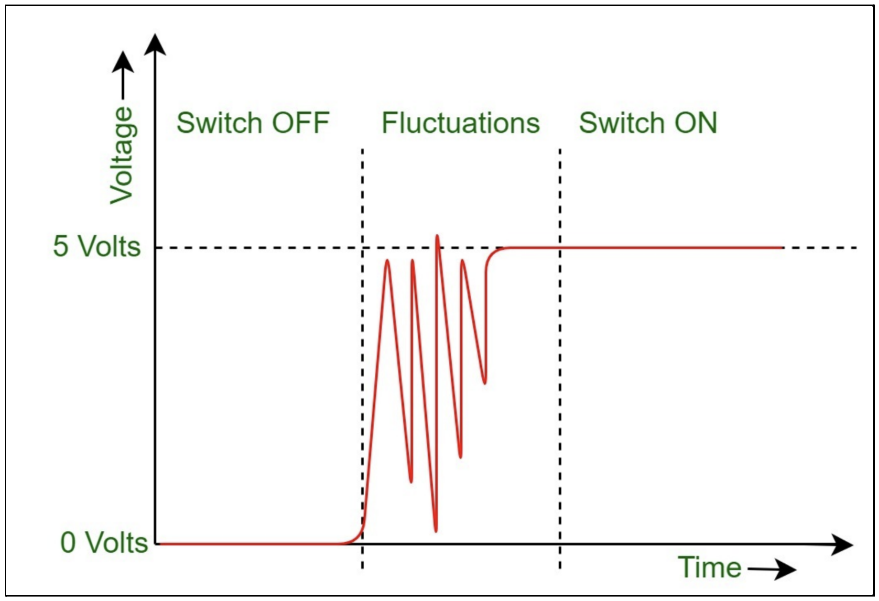
In the drive subsystem, PID control was applied to two functions. The purpose of the first function was to enable each robot to drive in a straight line and to correct any deviations from its path. The error, in this function, was the difference between the setpoint (0°), or the heading that the robot was supposed to maintain, and the actual yaw-angle of the robot as measured by the MPU-6050 gyroscope. The correction was applied in order to assign a higher speed to the motor with the lower RPM value and assign a lower speed to the motor with a higher RPM value. By minimizing error, the PID control loop was successful in enabling the robot to drive in a straight line path. The purpose of the second function was to enable each robot to point-turn to a certain heading (for example, turn to 30° relative to its starting orientation). The error was calculated by taking the difference between the desired heading and the current yaw-angle of the robot. This function is particularly useful in the environment exploration algorithms discussed in the “Full System Integration” section.

For PID subsystems to work as intended, the constants K_P, K_I, and K_D must be correctly selected. In this system, a trial-and-error based tuning method was selected, to determine values for the three constants. However, rule based methods such as the Ziegler-Nichols method can also be used.

Debouncing Encoder Readings

A challenge encountered while measuring values of RPM from an encoder was “bouncing”, which refers to spurious oscillations of the voltage of the interrupt pin due to minor hardware issues. These oscillations of the input voltage of the interrupt pin connected to encoders result in the Arduino UNO microcontroller overcounting the number of transitions between high and low states of the interrupt pin. This leads to an overestimated value for the RPM of the motor, which can lead to incorrect calculations of the distance travelled by a robot.

Figure 8: Diagram of "Bounces" in Signal



From *Switch Debounce in Digital Circuits.* Geeks for Geeks. (2019, November 20). https://www.geeksforgeeks.org/switch-debounce-in-digital-circuits/.

The "Fluctuations" section of the diagram represents the spurious "bounces" of the voltage signal. Although, in reality, the signal changes once from low (0V) to high (5V), "bounces" cause the interrupt handler program to overcount the number of transitions between low and high states. These "bounces" tend to have small time periods. Both hardware and software solutions to this problem have been identified and implemented.

The hardware solution to this problem involves attaching 100 nF capacitors between the input and ground pins of the encoder. Capacitors tend to resist changes in voltage. Having a capacitor with low capacitance would allow genuine changes in the state of the pin to be interpreted, but would effectively resist any spurious or rapid fluctuations and "bounces". In the circuit layout of the subsystem, the two 100 nF capacitors have been shown.

The software solution to this problem involves a modification to the interrupt handler function. The interrupt handler function is supposed to record a pulse every time a transition of state of the signal pin is detected. By calculating the number of pulses per unit time, the RPM of the motor can be determined. However, bounces lead to an

overcount of pulses and, consequently, an overestimation of the RPM. The updated interrupt handler is shown on the next page.

```
// Variable to store number of pulses recorded
volatile int pulses = 0;
// Minimum time between genuine state transitions
unsigned long debounceInterval = 10;

void interruptHandler() {
  // Time when interrupt was recorded is stored
  unsigned long interruptTime = millis();
  // If interrupts have been received more than 10ms apart
  if (interruptTime - prevInterruptTime > debounceInterval) {
    // Increment the pulse counter
    pulses++;
  }
  // Set current interrupt time to previous interrupt time
  prevInterruptTime = interruptTime;
}
```

In the code snippet above, the interrupt handler only counts a transition between low and high states if it occurs more than 10ms after the previous transition. Since “bounces” occur small time intervals apart from each other, they will not be recorded.

5.2 Sensor Subsystem

5.2.1 Hardware Components

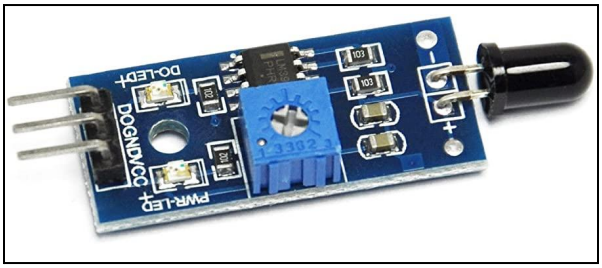
The purpose of this subsystem is to allow each robot in the swarm to measure distance to nearby obstacles or obstructions in the fireground environment. This subsystem also allows the robot to detect flames from a fire. The sensor subsystem consists of the following electronic components:

Figure 9: HC-SR04 Ultrasonic Sensor

From *HC-SR04-Ultrasonic Range Finder.* Robu.in. (n. d.). https://robu.in/product/hc-sr04-ultrasonic-range-finder/.

The purpose of including this part in the subsystem is to measure the distance to obstacles directly in-front of the robot. Possible alternatives for this sensor as well as justification for why this sensor was chosen are outlined in the “Design Specifications” section of this paper.

Figure 10: IR-Based Flame Sensor

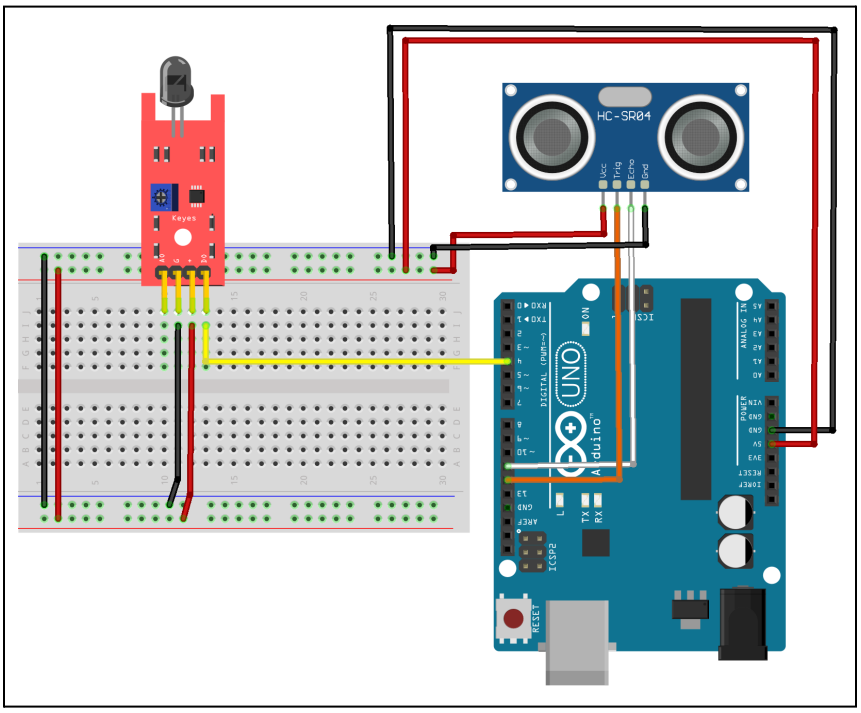


From *IR Flame Sensor Module Detector*. Atomic Market. (n. d.). https://www.amazon.com/Smartsense-Temperature-Compatible-Atomic-Market/dp/B00TNOHTV2.

The purpose of including this part in the subsystem is to detect flames from fires that may be present in the environment. Possible alternatives for this sensor as well as justification why this sensor was chosen are outlined in the “Design Specifications” section of this paper.

5.2.2 Circuit Diagram

Figure 11: Circuit Diagram of the Sensor Subsystem



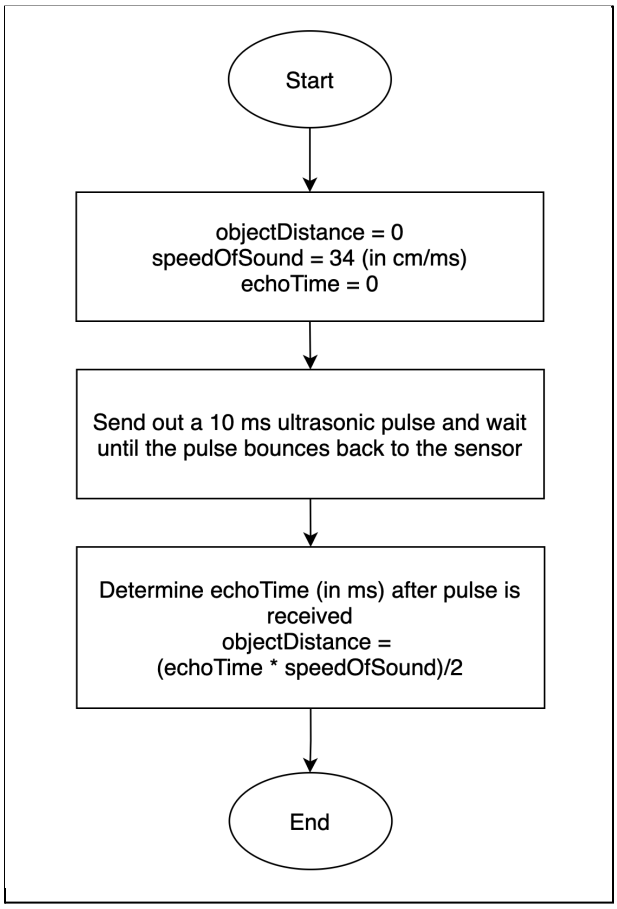
The circuit diagram above shows the breadboard-layout of the sensor subsystem, with all of the aforementioned electrical components as well as an Arduino UNO microcontroller.

5.2.3 Software

Calculating Distance to Obstacles

In order to determine the distance to obstacles in-front of the robot, an ultrasonic sensor was used. The sensor sends out a 10-microsecond long pulse of sound and measures the time taken for the reflected wave to be detected. From this time and the known value for the speed of sound in air (~340 m/s), the distance between the robot and the obstacle can be ascertained. The following flowchart describes the algorithm used to calculate obstacle distance. Values of object distance were also scaled by a factor of 1.026 after it was statistically determined that the ultrasonic sensor tends to underestimate the actual distance of the obstacle from the robot.

Figure 12: Flowchart for Obstacle Distance Measurement



In order to determine the absolute location of the obstacle in the environment, as an (x,y) coordinate, a similar approach was used as described earlier, involving resolution of the object distance into vertical and horizontal distances from the robot's (x, y) location, based on the robot's heading.

Detecting Fire

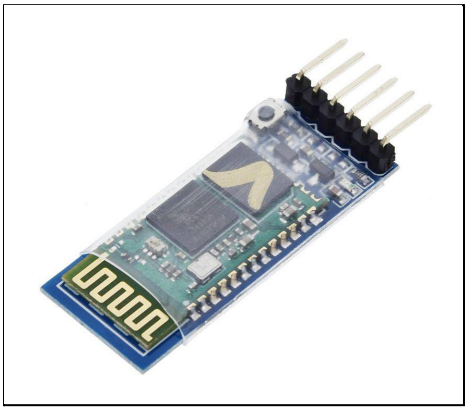
Since the flame sensor is connected to the Arduino UNO by a digital pin, when a flame is detected, the pin is set to a high voltage state and is processed by the software program. The flame sensor also contains a trimmer potentiometer (trimpot), which allows the sensitivity of the sensor to be adjusted.

5.3 Communication Subsystem

5.3.1 Hardware Component

The purpose of this subsystem is to allow each robot in the swarm to communicate with neighbouring robots as well as with a centralized controller. The communication subsystem consists of a single electronic component:

Figure 13: HC-05 Bluetooth Module

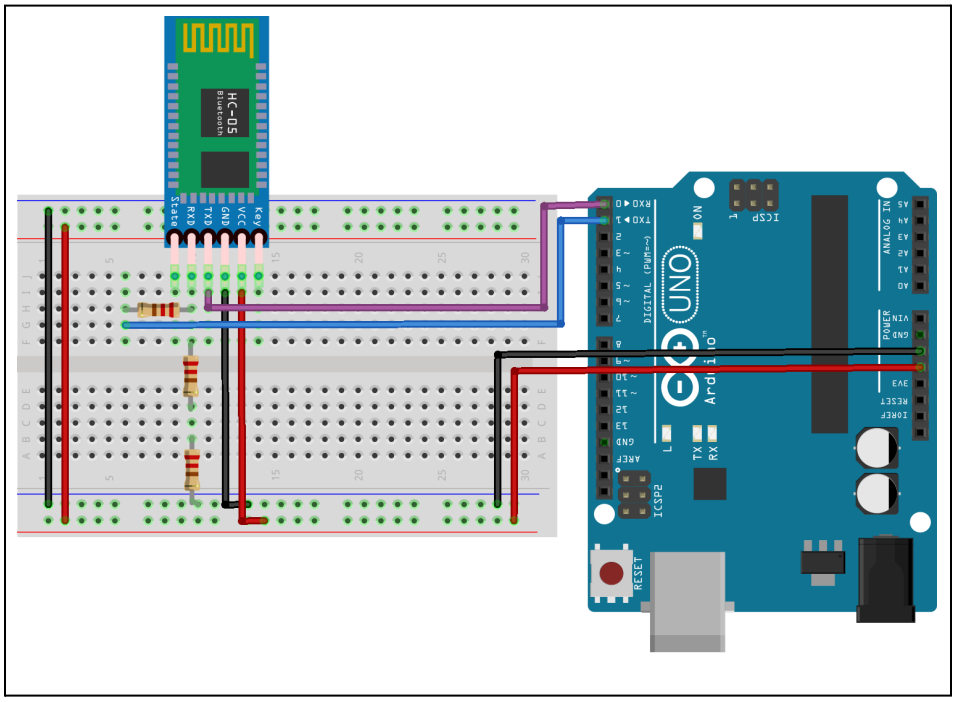


From *Module bluetooth HC-05.* GM Electronic. (n. d.). https://www.gmelectronic.com/bluetooth-modul-hc-05.

The HC-05 Bluetooth module allows each robot to establish a connection with the centralized controlling laptop as well as send and receive data. Possible alternatives for this component as well as justification for why this component was chosen are outlined in the “Design Specifications” section of this paper.

5.3.2 Circuit Diagram

Figure 14: Circuit Diagram of the Communication Subsystem



The circuit diagram above shows the breadboard-layout of the communication subsystem, with the HC-05 Bluetooth module, an Arduino UNO microcontroller and three 1kΩ resistors.

5.3.3 Software

The HC-05 Bluetooth module is connected to the RX and TX pins of the Arduino UNO and uses Serial communication to send and receive data. On the centralized controlling laptop, a Processing sketch connects to the HC-05 modules of individual robots and communicates with them using the Serial interface.

5.4 Full Subsystem Integration

5.4.1 Circuit Diagram

The circuit diagram for the full subsystem integration of a single swarm robot can be found in the Appendix. The circuit diagram shows the full-subsystem integration of all

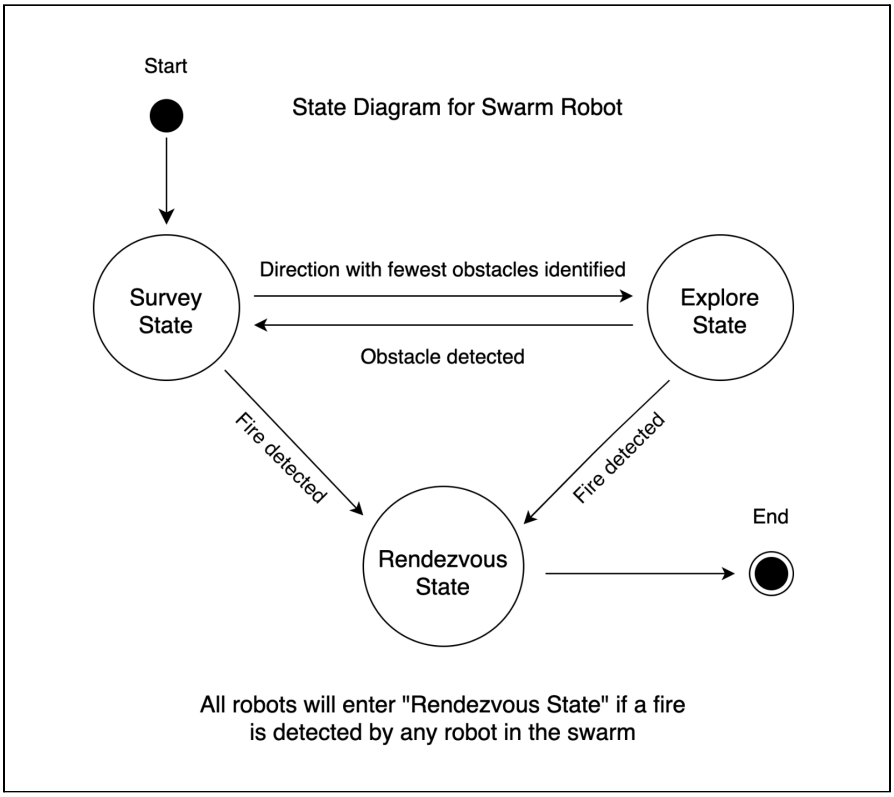
three aforementioned subsystems. Each robot is controlled by an Arduino UNO microcontroller and is powered by eight AA batteries (1.5V each) connected in series.

5.4.2 Software

State Machine

According to the comprehensive taxonomy for swarm robotic systems as proposed by Navarro and Matía (2013), individual robots must be homogenous. Thus, the software and hardware (circuit layout) must remain the same for all robots in the system. In order to control each robot in the swarm robotic system, a state machine approach was employed.

Figure 15: Proposed State Machine for a Swarm Robot



The purpose of the swarm robotic system is to survey the unfamiliar fireground environment and locate the positions of any obstructions and fires in the environment. Hence, each robot must survey and explore the environment. This is accomplished by

the “Survey State” and “Explore State”. When a fire is detected, all robots should gather at the site of the fire. This is accomplished by the “Rendezvous State”.

Survey State

The flowchart for the “Survey State” can be found in the Appendix.

The “Survey State” involves the robot turning 45° in a clockwise direction repeatedly, until it reaches its original heading. Every time the robot turns, it takes a measurement of the nearest obstacle using the ultrasonic sensor and uses the flame sensor to determine whether a fire is detected. If a fire is detected at any point, the program terminates and the state is changed to the “Rendezvous State”. If the robot does not detect any fires, the angle with the furthest obstacle is chosen and the robot turns to that angle. The robot then transitions to the “Explore State”.

Explore State

The flowchart for the “Explore State” can be found in the Appendix.

The “Explore State” involves the robot driving forwards, in a straight line, until it detects an obstacle or fire. If it detects an obstacle, the robot will transition back to the “Survey State” in order to pick a direction to turn to and continue exploring. The robot will transition to the “Survey State” if and only if the obstacle is within 30 cm of the robot (threshold distance). If a fire is detected, the robot transitions to the “Rendezvous State”.

Rendezvous State

The flowchart for the “Rendezvous State” can be found in the Appendix.

Robots transition to the “Rendezvous State” when a fire has been detected. When a robot detects fire, it communicates the approximate (x, y) coordinates of the fire via Bluetooth communication to the centralized controller. When a single robot transitions to the “Rendezvous State”, all other robots receive a signal that a fire has been detected from the centralized controller. Hence, all robots transition to the “Rendezvous State” in

order to rendezvous at the location of the fire. Once this has been accomplished, the program terminates; the system has achieved its purpose (as described in the “Design Specifications” section).

Greedy Survey

While testing the system, it was found that the “Survey State” is rather time consuming. The survey state involves point-turning clockwise by 45° and measuring the distance to the nearest obstacle in that direction. The direction with the furthest obstacle present is selected and explored. However, surveying all 8 directions is inefficient. Thus, an alternative version of the survey algorithm, “Greedy Survey”, is proposed. Similar to the previously employed survey algorithm, “Greedy Survey” involves the robot point-turning clockwise by 45° and measuring the distance to the nearest obstacle in that direction. However, “Greedy Survey” stops as soon as it finds a direction in which the nearest obstacle is at least 30 cm (threshold distance) away. For example, if the robot measures the distance to the nearest obstacle at a 135° heading as 35cm, the robot will remain at this heading and transition to the “Explore State” without checking distances to the nearest obstacles at other headings.

By finding a satisfactory solution (an object that is **far enough**) instead of an optimal solution (the furthest object), this algorithm saves time and makes the system more efficient but does not compromise on its functionality. Since efficiency is an important design criterion, the “Greedy Survey” algorithm was chosen to replace the previously described survey algorithm.

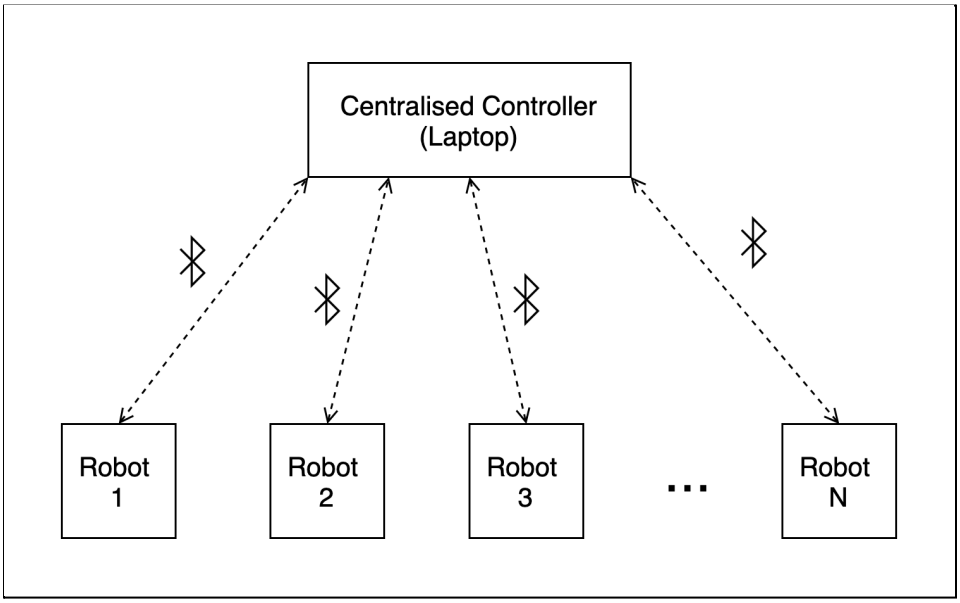
Mapping Robot Motion

While each robot follows the described state machine, it sends data about its heading, distance travelled, and distance to the nearest obstacle to the centralized controller during every loop function call in the program. Thus, the centralized controller is able to calculate the absolute position of the robot and of any obstacles detected in the environment. From these readings, a map of the environment can be constructed using a program written in the Processing language. The centralized controller, in this

study, is a laptop computer paired with the HC-05 Bluetooth modules of each of the robots.

5.4.3 Bluetooth System Setup

Figure 16: Diagram showing Bluetooth Communication



The diagram above shows Bluetooth communication for the proposed system with N robots. Each robot will be connected to the centralized controller (in this case, a laptop computer) and will be able to send data to and receive data from it. Robots constantly send data about their position in the environment to the centralized controller. This data is used to construct the map of the environment. When a single robot detects fire, it communicates this information to the centralized controller. The centralized controller then communicates that information to all robots in the swarm; consequently, all robots transition to the “Rendezvous State”.

5.4.4 Images of the Final Robot System

Images of the final robot system and full subsystem integration can be found in the Appendix.

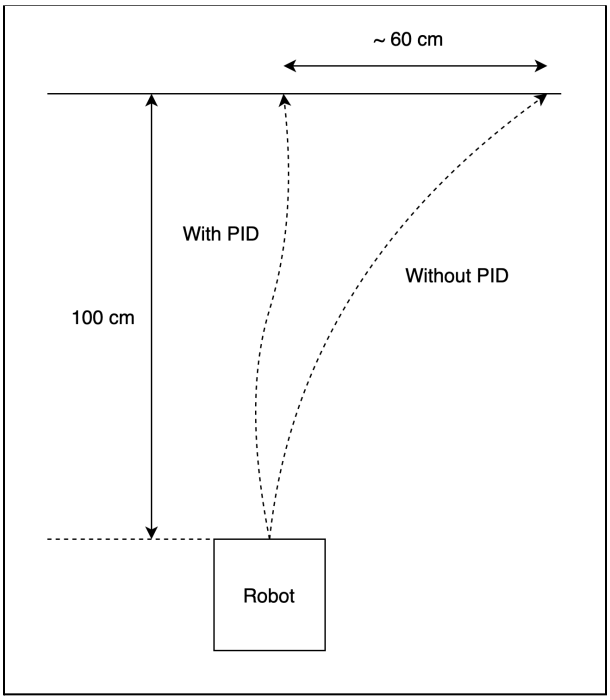
6 Testing and Verification

In order to determine whether or not the proposed system meets the aforementioned design criteria, a series of tests were carried out. The following section describes these tests and analyses the results obtained from them. Testing was conducted on individual subsystems as well as on the fully integrated system.

6.1 Individual Subsystem Tests

6.1.1 Drive Subsystem

Figure 17: Diagram of Approximate Robot Trajectory



The PID function which enabled the robot to move forward in a straight line was compared with a function that simply assigned both motors the same speed. Each time, the robot was driven a distance of 100 cm in the forward direction and its horizontal deviation from the straight line path was measured. The horizontal deviation from the straight path without using PID was ~60 cm more than while using the PID function. This test was repeated and a difference of ~60 cm was obtained each time. Thus, the PID

function was successful in preventing the robot from veering off from its straight line path.

Similar tests were repeated on the PID function which enabled the robot to point turn by a specified angle. The function was allowed a margin of error of ±5°. This was done in order to reduce the duration of time spent on each turn and to prevent oscillation, due to overcorrection by the controller, of the robot when it was less than 5° away from the setpoint heading. This function also worked as intended across several trials, within the margin of error of ±5°.

6.1.2 Sensor Subsystem

Ultrasonic Sensor

In preliminary tests of the ultrasonic sensor, it was noticed that the ultrasonic sensor tends to underestimate the actual distance between itself and the object. It was observed that the distance calculated by the ultrasonic sensor was often less than the actual distance of the object from the robot. For a graph of actual distance (y-axis) versus calculated distance (x-axis), if the gradient of the least-squares regression line is greater than one, we can conclude that the ultrasonic sensor does indeed underestimate the actual distance between itself and the object. Hence, a hypothesis test can be conducted.

- Null Hypothesis (H_0): $\beta = 1$
- Alternate Hypothesis (H_A): $\beta > 1$, where β is the gradient of the least squares regression line of a plot of actual distance versus calculated distance.

A t-test for the slope of a regression line was conducted to test the hypotheses, at a significance level of $\alpha = 0.05$. A sample of 15 random distances between 1-100 cm was generated using a random number generator, and both the actual and calculated distances were recorded. Using this sample, a least squares regression line was calculated.

Tables 4 and 5: Linear Regression Output Summary

Regression Statistics	
Multiple R	0.999816889
R Square	0.999633811
Adjusted R Square	0.999605643
Standard Error	0.472458906
Observations	15

	Coefficients	*Standard Error*	*t Stat*	*P-value*
Intercept	-0.130894438	0.31314583	-0.4179983	0.68277035
Calculated Distance	1.02604046	0.00544659	188.382095	1.0018E-23

As shown above, there is a strong positive linear relation between calculated distance and actual distance, with a r-squared value of ~99.96%.

Figure 19: Scatter Plot of Actual Distance versus Calculated Distance

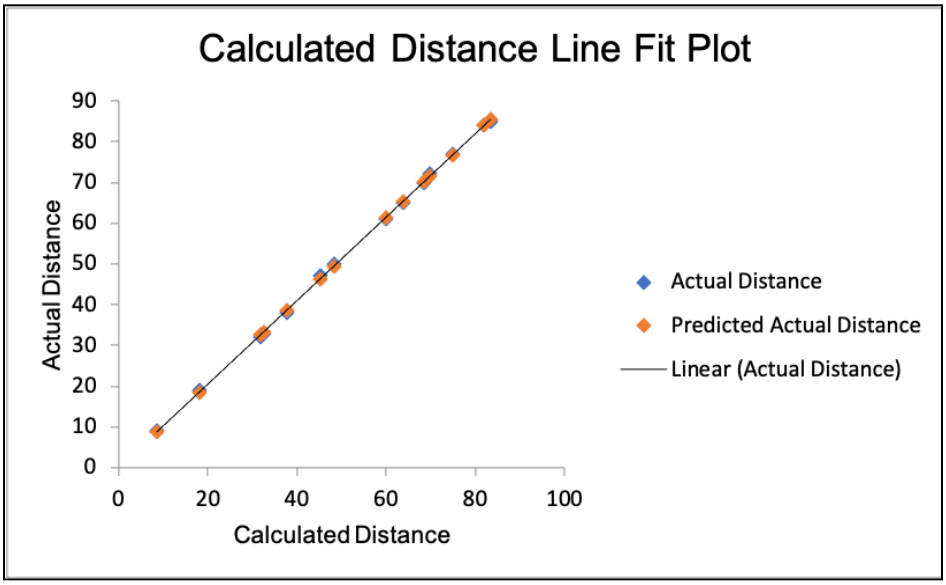
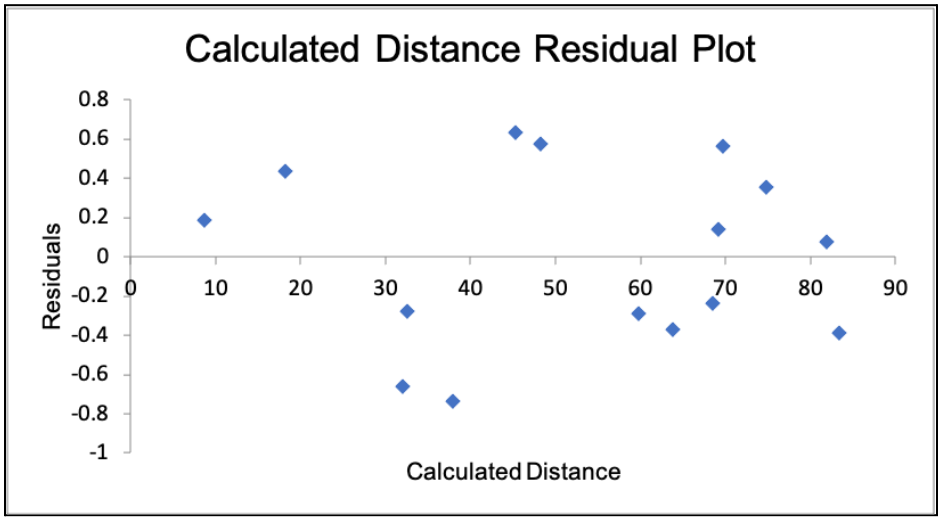


Figure 20: Residual Plot of Calculated Distance



Since calculated distance and actual distance have a strong linear relation, residuals are evenly spaced and the sample of 15 actual distances were randomly selected from the range 1-100 cm, the conditions for conducting a t-test for the slope of a regression line have been satisfied.

The value of the test-statistic is t = 4.78106 and the corresponding p-value is p = 1.7942×10^{-4}. Since the p-value is less than the significance level, there is sufficient statistical evidence to reject the null hypothesis that $\beta = 1$. Hence, we can conclude that the ultrasonic sensor does indeed underestimate the actual distance of an object. Since the slope of the regression line is ~1.026, calculated distance values will be scaled by this factor to obtain actual distance. Nevertheless, the size of the underestimation of actual distance is small and it will have a negligible impact on the performance of the system.

Flame Sensor

Since the flame sensor was connected to a digital pin, its sensitivity was adjusted using a trial-and-error method. Since the proposed system should be able to sense a fire that is at most 20 cm away, the sensitivity of the flame sensor was adjusted using its trimpot until it was able to consistently sense fires that are at most 20±2 cm away. The

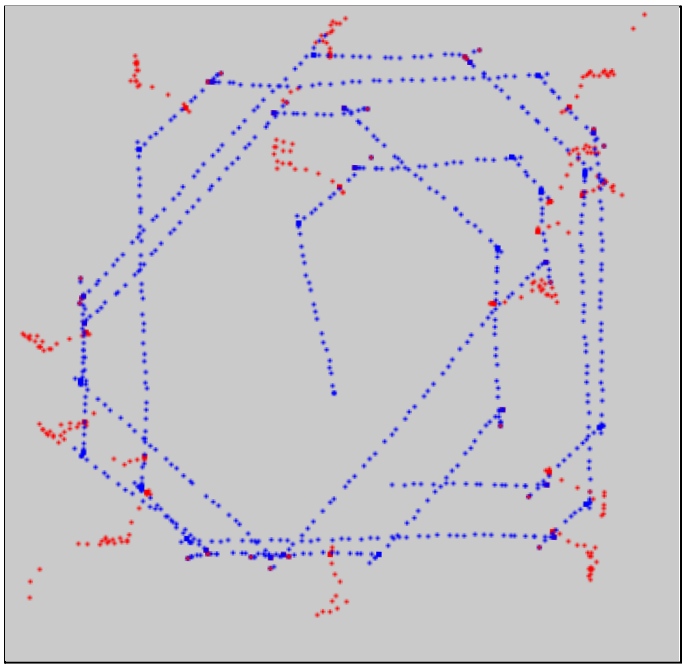
flame sensor is also able to detect fire when it is angled less than 10° away from the flame. Another observation that was made was that when the flame sensor is blocked by an opaque obstacle, it was not able to sense the flames. This is reasonable, since the rendezvous algorithm has been designed to avoid obstacles in the path of the robot.

6.2 Full Subsystem Integration Tests

6.2.1 Single Robot

In order to test the full-subsystem integration of a single robot, a robot was tasked to explore a simulated environment and create a map of it. Since only a single robot was tested, a fire was **not** placed in the environment as the rendezvous state can only be tested with multiple robots. Five trials were conducted, wherein the robot was placed at the center of the unfamiliar environment and switched on. The robot was also connected to a centralized controller that plotted the robot's output in the form of a map. The size of the simulated environment was a 150cm-by-150cm square shaped wooden enclosure with 15 cm high walls. The robot was given two minutes to construct a map of the environment.

Figure 21: Output of Processing sketch for Trial 3



Note: Blue points represent the robot's trajectory and red points represent obstacles detected in the environment.

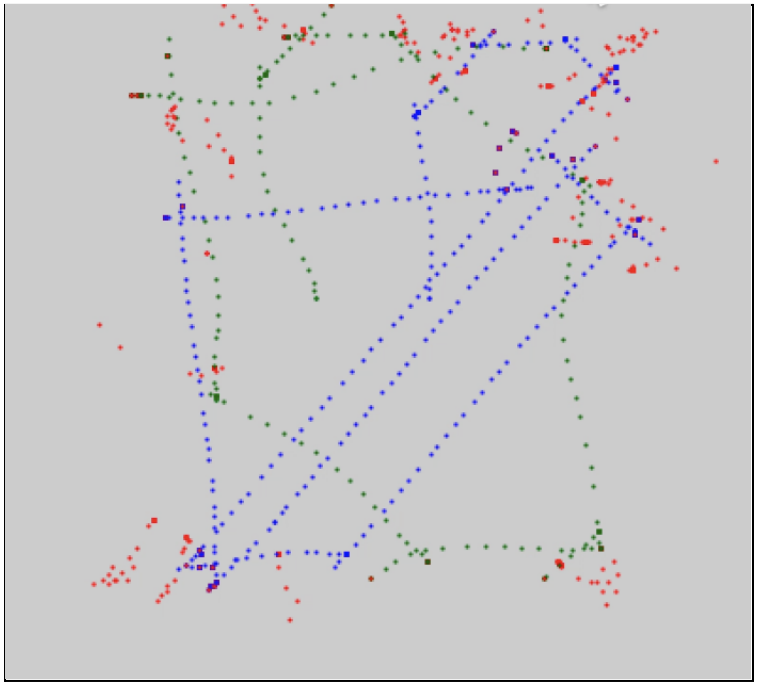
Above is an example of a map generated by the Processing sketch run on the centralized controller, during Trial 3. While the robot was successful in avoiding obstacles and exploring the simulated environment, there are a few inaccuracies in the constructed map. For example, noise in the readings of the ultrasonic sensor caused the robot to change state even when there weren't any obstacles in the robot's path. Another observation made across all trials was that the robot tends to explore the boundaries or edges of the environment instead of the center of the environment. This is because of the "Greedy Survey State" algorithm, which takes the first satisfactory path found (i.e. along the wall of the enclosure) as opposed to the original "Survey State", which searches for the path with the furthest obstacle, which would have been towards the center of the simulated environment. Maps constructed by the robot for Trials 1 through 5 can be found in the Appendix.

6.2.2 Multiple Robots

Mapping Task

The mapping task described previously was repeated, but with two robots instead of using a single robot. The same environment was used as previously described. The robots were given two minutes to map out the simulated environment. 5 Trials were conducted. Since only the mapping abilities of the robots were being tested, a flame was **not** placed in the simulated environment. Both robots were connected to the centralized controller and they subsequently transmitted data to it. Robots were placed with a 30 cm gap between them.

Figure 22: Output of Processing sketch for Trial 1



Note: Green points represent robot 1's trajectory, blue points represent robot 2's trajectory and red points represent obstacles detected in the environment.

Above is an example of a map generated by the Processing sketch run on the centralized controller, during Trial 1. Robots were successful in avoiding obstacles and following the state machine. However, due to noise from the ultrasonic sensor and small mechanical differences in the encoders used by both robots, the map contains slight inaccuracies. For example, red dots can be observed in certain areas in the map in which there were no obstacles placed. Moreover, due to the robots detecting each other

as obstacles when they crossed paths, red dots can be observed at intersections of their paths.

Fire Detection and Rendezvous State

In order to test the “Rendezvous State” of the system, I placed a simulated fire in the 150 cm by 150 cm environment used previously. The fire was simulated by a lit candle. The robots were placed at the left-wall of the enclosure and the fire was placed at a random location in the environment. The time taken for the robots to detect and rendezvous at the fire was recorded. 10 such trials were conducted. A t-statistic was used to estimate the mean time taken for the robots to detect and rendezvous at the fire. A 95% level of confidence was used. The raw data for trials 1 through 10 can be found in the Appendix.

The mean time taken by 2 robots to detect and rendezvous at the site of the fire for the sample of 10 trials was $\bar{x}$ = 33.338 seconds. This is a point estimate of μ, or the mean time taken by 2 robots to detect and rendezvous at the site of a fire in an area of 2.25 m^2. The standard deviation in the time taken by the two robots to detect and rendezvous at the site of the fire for the sample of 10 trials was s = 7.0525 seconds. In order to determine a 95% confidence interval for μ, we must use the critical value of the t-statistic with 9 degrees of freedom (t-crit = 2.26216).

Sample Size (n) = 10

Standard Error (SE) = $\frac{s}{\sqrt{n}} = \frac{7.0525}{\sqrt{10}}$ = 2.230

Margin of Error (MOE) = t-crit × SE = 2.26216 × 2.230 = 4.9733 seconds

95% Confidence Interval of μ = $\bar{x}$ ± MOE = 33.338 ± 4.973 seconds

Therefore, we are 95% confident that the mean time taken by the two robots to detect and rendezvous at the site of a fire in an area of 2.25 m^2 lies in the range (28.365 seconds, 38.311 seconds).

6.2.3 Verification of Design Criteria

In order to determine whether the proposed system meets the design criteria proposed previously, the results of the aforementioned tests carried out on the system were analyzed.

Efficiency

The design criteria stated that, for the proposed system to be called efficient, it would have to collect essential data at a rate of 1 m^2 per robot per minute or ~0.0167 m^2 per robot per second. In the “Fire Detection and Rendezvous State” trials, it was determined that essential data (that is, the position of fire and obstacles in the environment) can be collected from an area of 2.25 m^2 using 2 robots in at most 38.311 seconds. This is a rate of ~0.029365 m^2 per robot per second, which exceeds 0.0167 m^2 per robot per second. Hence, the proposed system is efficient. Moreover, the efficiency of the system can be increased further by modifying the robot state machine such that it avoids other robots in the swarm. If robots survey different areas of the unfamiliar environment, there will be fewer robot-robot collisions and thus higher efficiency.

Size and Weight

Each robot has a size of approximately 15 cm by 17 cm by 20 cm. Hence, each robot is able to fit within a 30 cm by 30 cm by 30 cm cube. Thus, it has met the size criterion. Robots weigh 604g and 588g. Since both robots weigh less than 1000g (1.0 kg), they meet the weight criterion. Robots are small and portable. This would allow the system to be extended to more than two robots without significant logistical challenges.

Cost

The cost of each swarm robot was determined to be ~$32.05, which exceeds the $20 budget mentioned in the design criteria. However, $32.05 is a reasonable amount and costs of individual robots would decrease if parts are purchased in bulk for multiple swarm robots.

7 Conclusion

In conclusion, this paper proposes a swarm robotic system that is able to explore an unfamiliar fireground environment and gather essential information. The robotic system is able to determine the positions of fires and obstructions as well as construct an approximate map of the unfamiliar environment. In this paper, the design of a single swarm robot has been described and the swarm system has been demonstrated using two identical robots. Design criteria for the system have been outlined and have been verified using statistical tests. The proposed system is able to explore, map out and collect essential data at a rate of at least 0.029365 m^2 per robot per second. Thus, it is both efficient and scalable, due to its low cost and small size per robot in the swarm. Not only does this paper propose a novel application of swarm robotics but it also contributes to ongoing research on robotic systems for the exploration and mapping of fireground environments. If implemented, the proposed system would be able to help firefighters gain valuable information about dangerous fireground environments, which could then be used to inform the process of planning escape routes and locating survivors.

8 Recommendations

In order for the proposed system to be implemented in a real-life fireground environment, the following improvements can be made to the system.

The swarm robotic system was demonstrated using only two robots, due to logistical constraints created by the COVID-19 pandemic. However, this system can be extended by including more identical robots that follow the state machine described previously. With more robots, large fireground environments can be surveyed quickly and easily. Furthermore, if the state machine is modified, such that robots avoid each other while exploring the environment, each robot will be able to explore a larger area and there will be fewer robot collisions. Thus, the efficiency of the system would increase further.

Although the maps plotted by the two robots in this paper depict all of the important features present in the simulated environments, there is a lot of noise in these maps. This is primarily due to noise from the ultrasonic sensor. If the budget for individual swarm robots is increased, a more expensive but more accurate sensor, such as a LIDAR, can be included instead of the ultrasonic sensor. Another observation that was made during the testing phase

was that the ultrasonic sensor was only able to accurately measure object distances when the surface of the object was normal to the waves of ultrasound emitted by the sensor. If the obstruction was placed at an angle, it would not be detected by the sensor and the robot would collide with it. In order to solve this problem, the sensor could have been mounted on a servo-motor, which would enable the sensor to rotate to and measure object distance at various angles instead of just directly ahead of the robot. An Arduino Mega, or a microcontroller with a greater number of PWM pins, can be used instead of the Arduino UNO, to accommodate the servo motor.

Lastly, modifications can be made to swarm robots in order to achieve different functions. If survivors need to be located instead of fires, flame sensors can be replaced by cameras. A vision processing algorithm could be implemented to detect survivors. Nevertheless, the state machine for this system would remain the same. If the swarm must extinguish fires, a fire-extinguishing mechanism can be added to each robot, which could be operated after all robots have rendezvoused at the site of the fire.

9 References

Athira A., Agnal, M., & Kumar, S. (2019). Fire and Rescue Robot. International Journal of Computer Applications, 182(45), 975–8887. https://www.ijcaonline.org/archives/volume182/number45/kumar-2019-ijca-918577.pdf

Bakhshipour, M., Jabbari Ghadi, M., & Namdari, F. (2017). Swarm robotics search & rescue: A novel artificial intelligence-inspired optimization approach. Applied Soft Computing, 57, 708-726. doi:10.1016/J.ASOC.2017.02.028

Becker, A., Fekete, S. P., Kröller, A., Lee, S. K., McLurkin, J., & Schmidt, C. (2013). Triangulating unknown environments using robot swarms. Proceedings of the 29th Annual Symposium on Symposuim on Computational Geometry - SoCG '13. https://doi.org/10.1145/2462356.2462360

de Mendonça, R. M., Nedjah, N., & de Macedo Mourelle, L. (2012). Swarm Robots with Queue Organization Using Infrared Communication (B. Murgante, O. Gervasi, S. Misra, N. Nedjah, A. M. A. C. Rocha, D. Taniar, & B. O. Apduhan, Eds.). Springer Link; Springer. https://doi.org/10.1007/978-3-642-31125-3_11

E-puck education robot. (2018). E-Puck.org. http://www.e-puck.org/

Hamann, H. (2018). Swarm Robotics: A Formal Approach. Cham: Springer International Publishing. Retrieved April 3, 2021, from https://ebooks.ohiolink.edu

Kostavelis I., Gasteratos A. (2017) Robots in Crisis Management: A Survey. In: Dokas I., Bellamine-Ben Saoud N., Dugdale J., Díaz P. (eds) Information Systems for Crisis Response and Management in Mediterranean Countries. ISCRAM-med 2017. Lecture Notes in Business Information Processing, vol 301. Springer, Cham. https://doi.org/10.1007/978-3-319-67633-3_4

Navarro, I., & Matía, F. (2013). An Introduction to Swarm Robotics. ISRN Robotics, 2013, 1–10. https://doi.org/10.5402/2013/608164

Penders, J., Alboul, L., Witkowski, U., Naghsh, A., Saez-Pons, J., Herbrechtsmeier, S., & El Habbal, M. (2011). A Robot Swarm Assisting a Human Fire-Fighter. Advanced Robotics, 25(1-2), 93–117. https://doi.org/10.1163/016918610x538507

Starr, J.W., Lattimer, B.Y. Evaluation of Navigation Sensors in Fire Smoke Environments. Fire Technol 50, 1459–1481 (2014). https://doi.org/10.1007/s10694-013-0356-3

Sucuoglu, H. S., Bogrekci, I., & Demircioglu, P. (2018). Development of Mobile Robot with Sensor Fusion Fire Detection Unit. IFAC-PapersOnLine, 51(30), 430–435. https://doi.org/10.1016/j.ifacol.2018.11.324

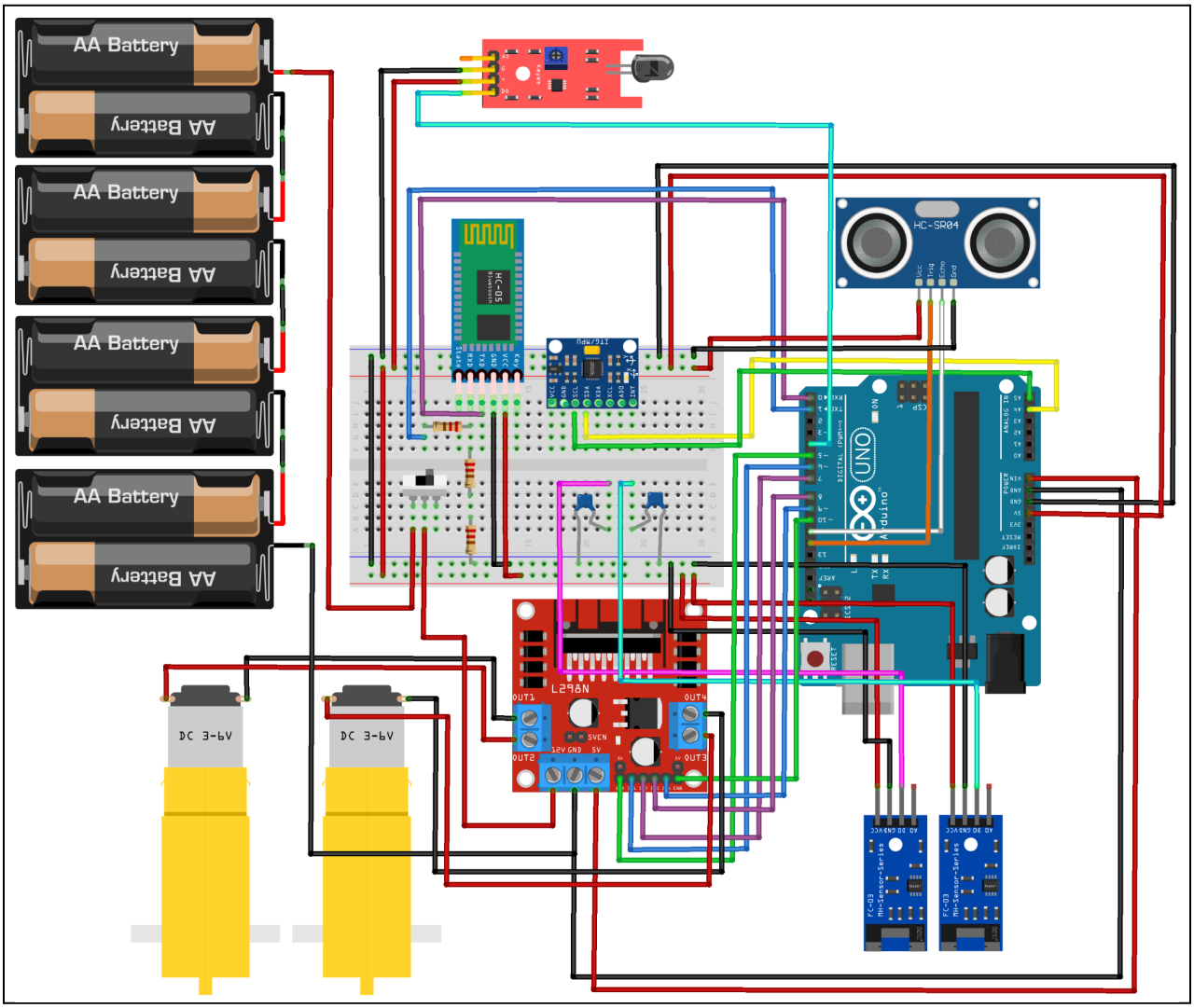
Taha, Ihsan A., and Hamzah M. Marhoon. “Implementation of Controlled Robot for Fire Detection and Extinguish to Closed Areas Based on Arduino.” TELKOMNIKA (Telecommunication Computing Electronics and Control) 16, no. 2 (April 1, 2018): 654. https://doi.org/10.12928/telkomnika.v16i2.8197.

World Fire Statistics. (2021, October 15). World Fire Statistics | CTIF - International Association of Fire Services for Safer Citizens through Skilled Firefighters. https://www.ctif.org/world-fire-statistics

10 Appendices

10.1 Circuit Diagram

Figure 23: Circuit Diagram of Full Subsystem Integration of a Swarm Robot



10.2 Flowcharts of States

Figure 24: Flowchart of Survey State

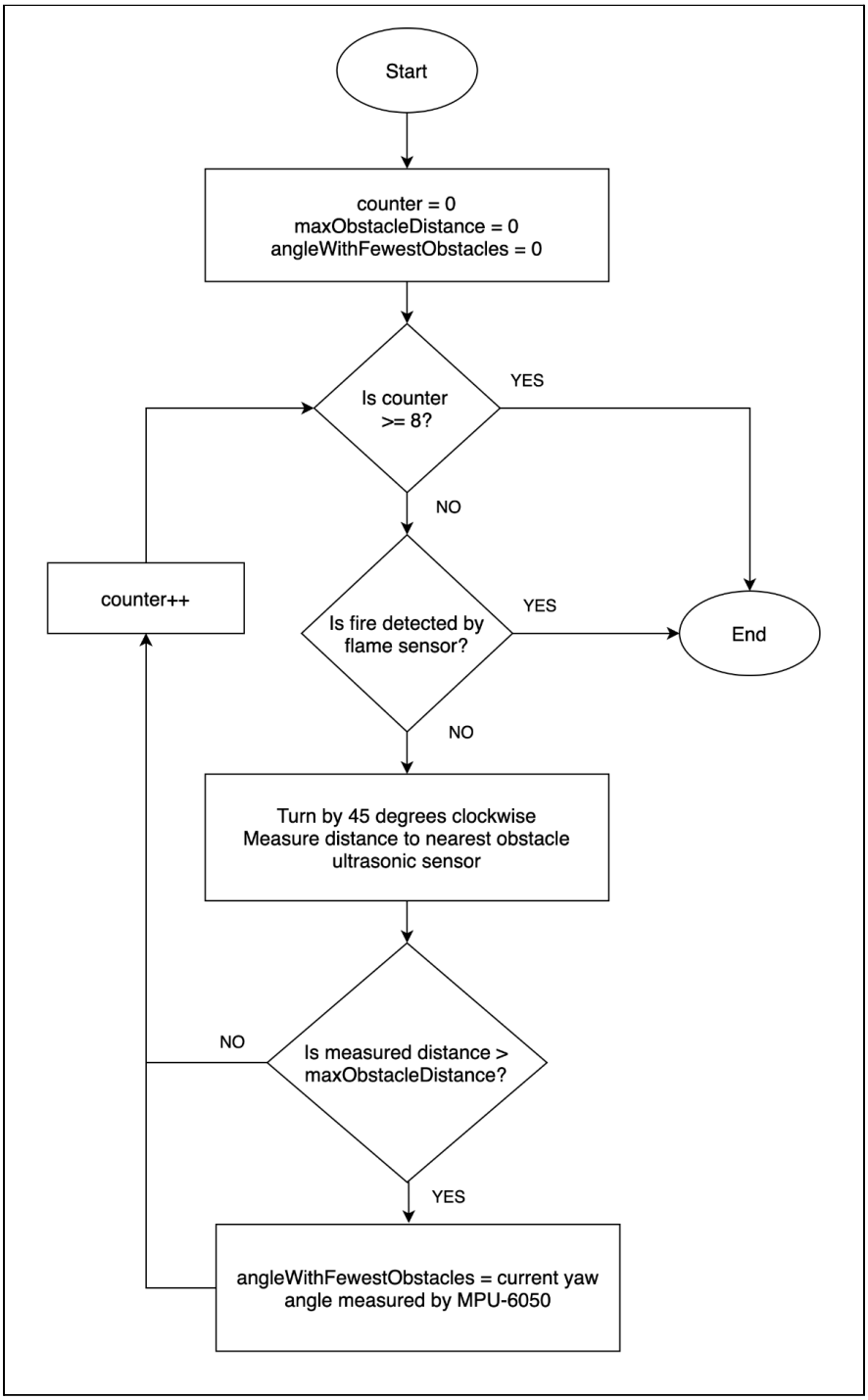


Figure 25: Flowchart of Explore State

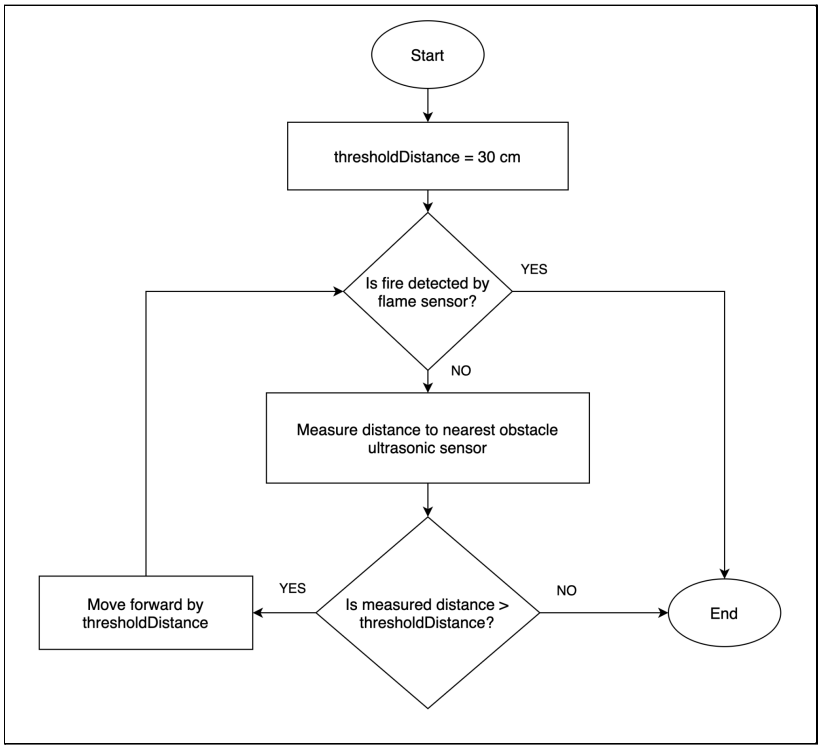
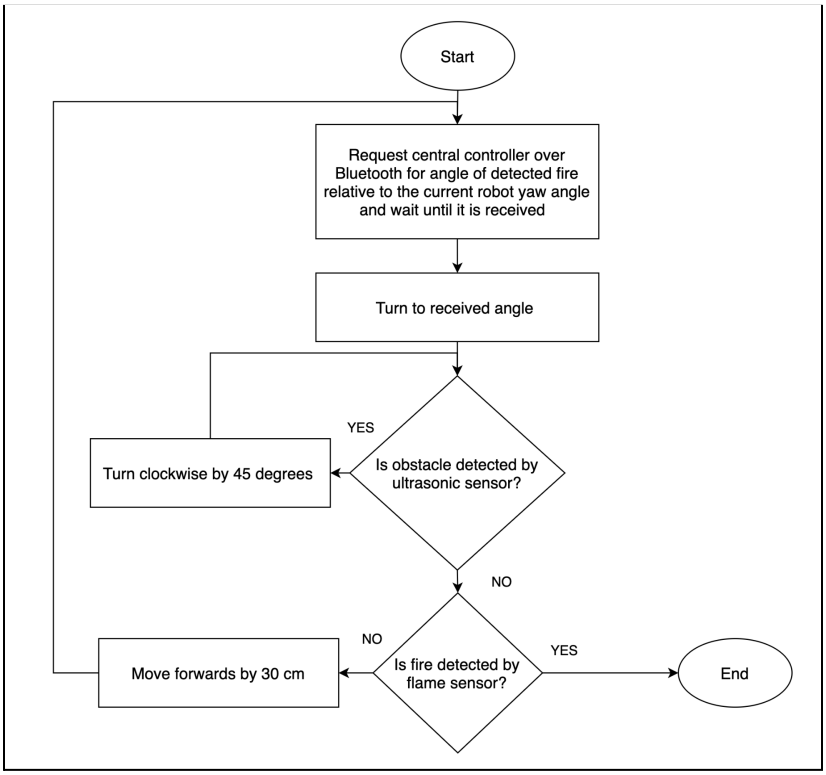
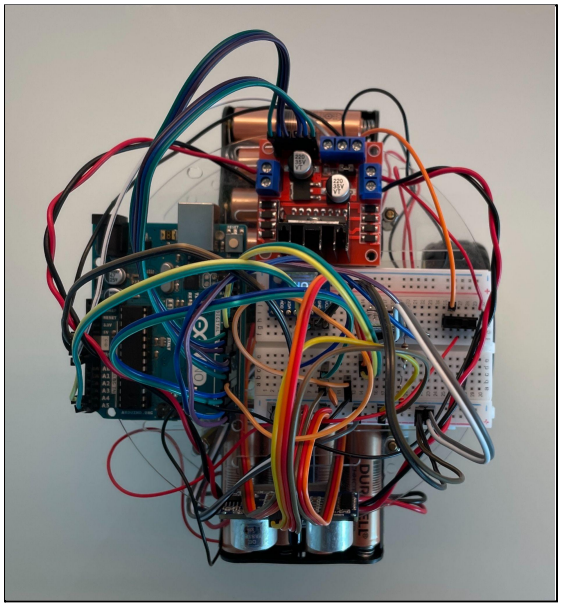


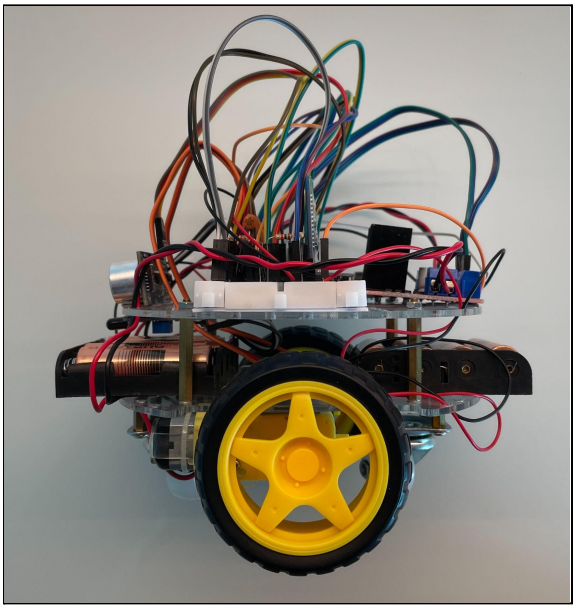
Figure 26: Flowchart of Rendezvous State



10.3 Images of Proposed System

Figures 27, 28, 29: Top, Side and Front Views of a single Swarm Robot





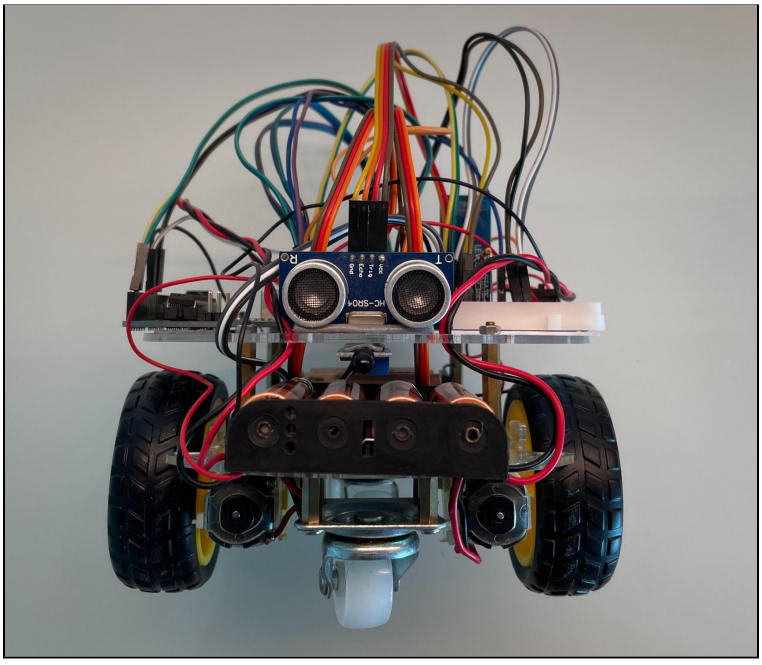


Figure 30: Two Swarm Robots

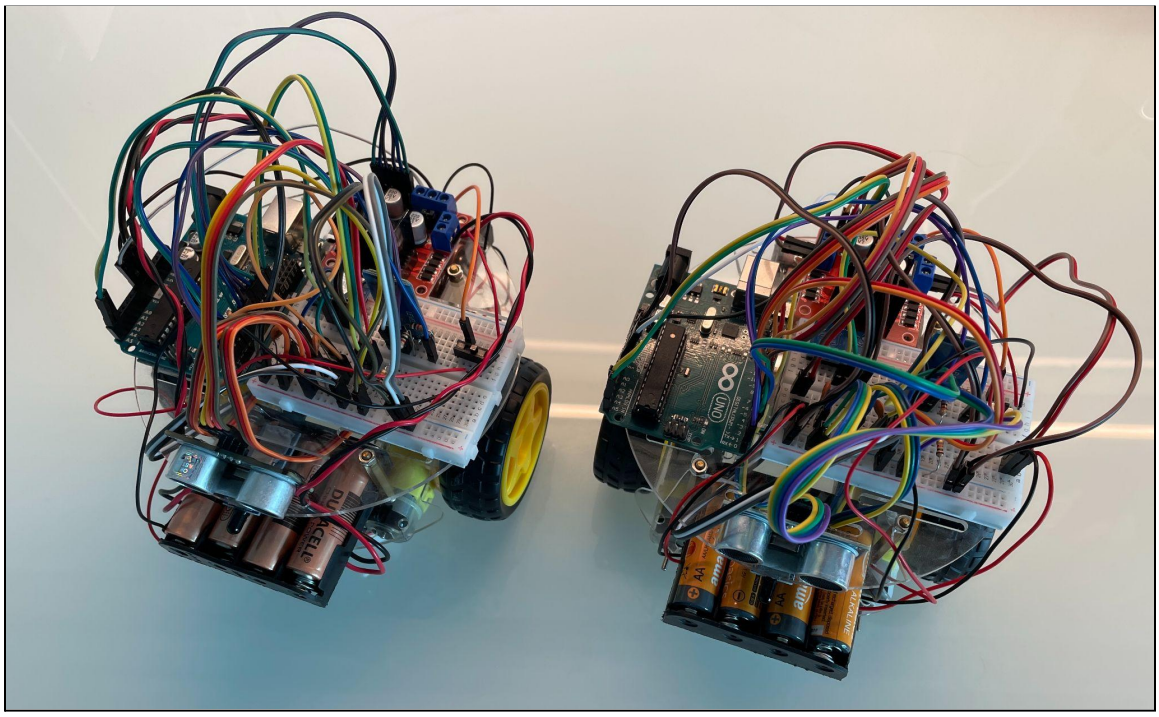


Figure 31: Two Swarm Robots Rendezvousing at Fire in Simulated Environment

10.4 Raw Data from Testing

10.3.1 Ultrasonic Sensor Tests

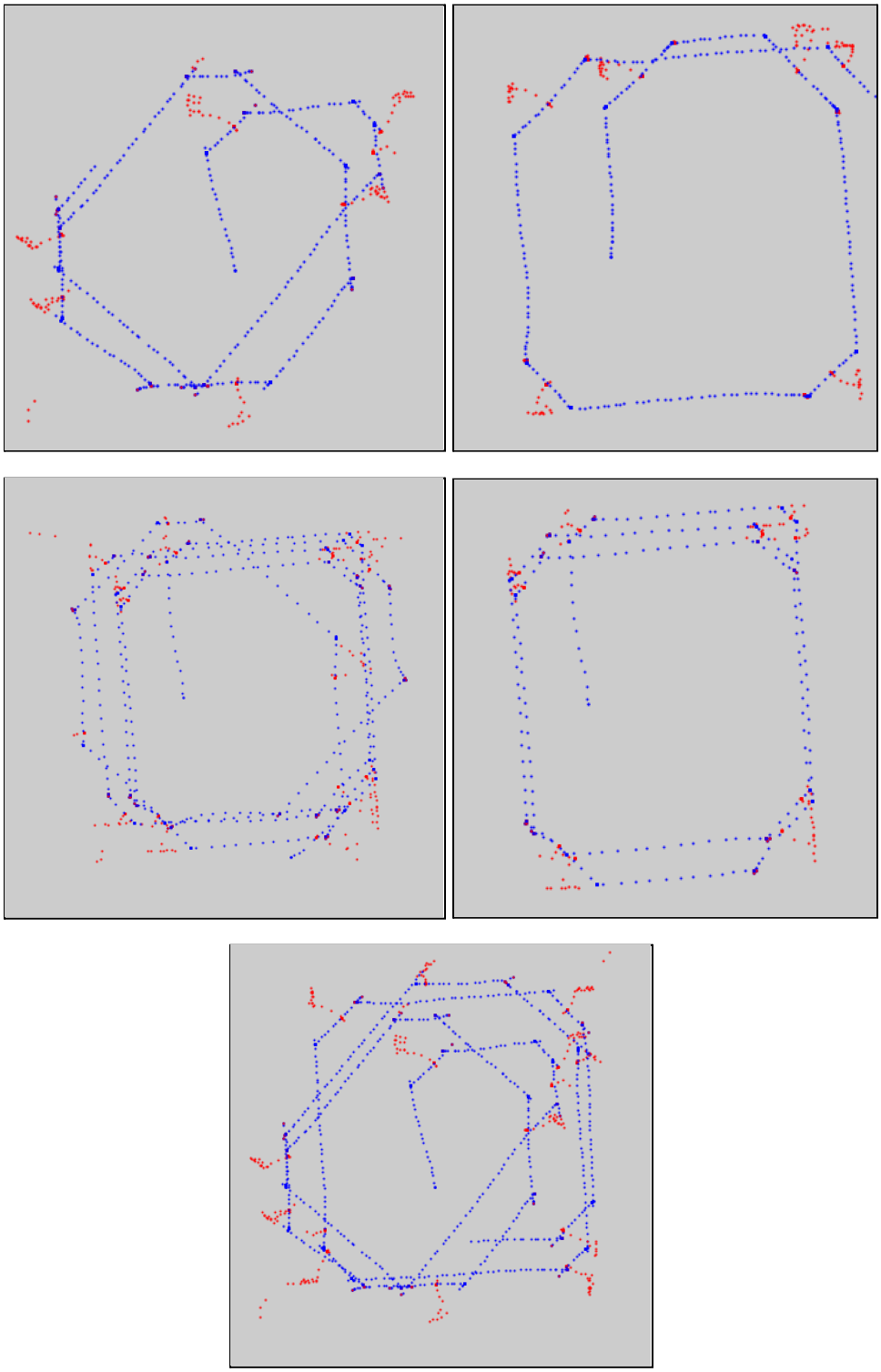
Table 3: Raw Data from Tests of Ultrasonic Sensor

Trial Number	Actual Distance (cm)	Calculated Distance (cm)
1	84	81.92
2	47	45.32
3	19	18.22
4	65	63.84
5	72	69.75
6	70	68.58
7	77	74.83
8	9	8.72
9	71	69.19
10	38	37.88
11	33	32.56
12	50	48.30
13	61	59.86
14	85	83.35
15	32	31.96

Note: For each trial, values of “Actual Distance” were generated using a random number generator. The corresponding “Calculated Distance” was recorded.

10.3.2 Single Robot Tests

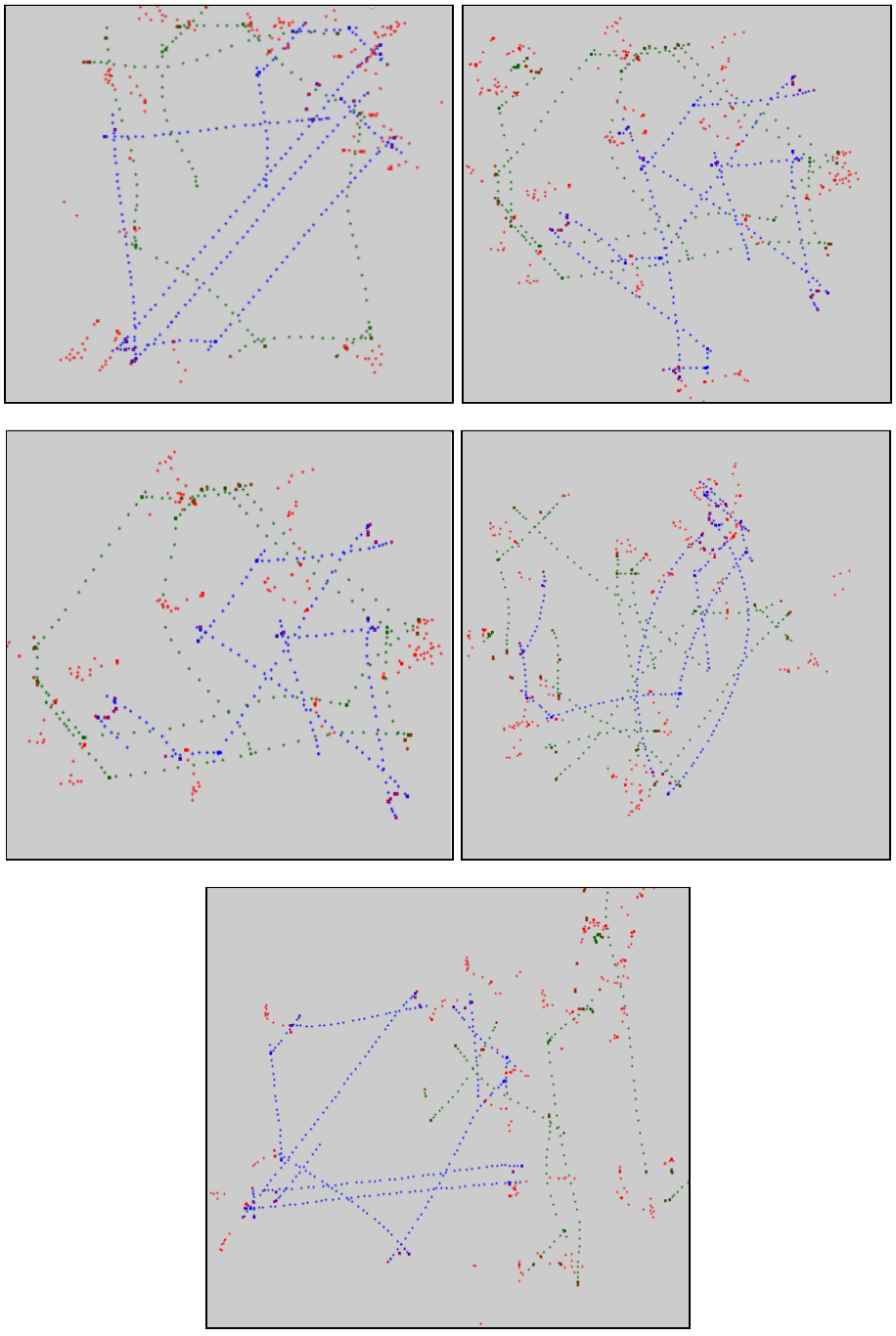
Figures 32, 33, 34, 35, 36: Trials 1-5 for Single Robot Mapping



Note: Blue points represent the robot's trajectory and red points represent obstacles detected in the environment.

10.3.2 Multiple Robot Tests

Figures 37, 38, 39, 40, 41: Trials 1-5 for Multiple Robot Mapping



Note: Green points represent robot 1's trajectory, blue points represent robot 2's trajectory and red points represent obstacles detected in the environment.

Table 5: Raw Data from Tests of Rendezvous State

Trial Number	Time Taken (s)
1	25.36
2	30.02
3	43.02
4	32.45
5	40.19
6	37.87
7	22.76
8	27.76
9	41.53
10	32.42

10.5 Video Demonstration

A video demonstration of the proposed system can be found on YouTube at https://youtu.be/uKg5bWZSw3I.

10.6 Bill of Materials

Table 6: Bill of Materials for a Single Robot

Part Name	Quantity (units)	Cost ($/unit)	Total Cost ($)
150 RPM Geared DC Motors	2	$ 0.67	$ 1.34
L298N Motor Controller	1	$ 2.70	$ 2.70
MPU-6050	1	$ 3.25	$ 3.25
HC-020K Encoder	2	$ 5.20	$ 10.40
HC-SR04 Ultrasonic Sensor	1	$ 1.30	$ 1.30
IR-Based Flame Sensor	1	$ 0.65	$ 0.65
HC-05 Bluetooth Module	1	$ 2.86	$ 2.86
Robot Chassis	1	$ 4.55	$ 4.55
Other Expenses	-	$ 5.00	$ 5.00
Total	-	-	$ 32.05

10.7 Full System Code

10.7.1 Arduino Code

```
// This library is needed to read data from the gyroscope
#include <Wire.h>

// Right motor pins
int pwmR = 10;
int in1 = 9;
int in2 = 8;

// Left motor pins
int pwmL = 5;
int in3 = 7;
```

```
int in4 = 6;

// Encoders pins
int encL = 2;
int encR = 3;

// IMU (MPU-6050)
int MPU = 0x68;

// Ultrasonic sensor pins
int trig = 12;
int echo = 11;

// Flame Sensor pin
int flame = 4;

// Constants associated with the drive function
int maxSpeed = 120;
int minSpeed = 80;
int driveSpeed = 100;
int speedRange = (maxSpeed - minSpeed) / 2;
int assignedLeftSpeed = 0;
int assignedRightSpeed = 0;

// Constants associated with PID functions
unsigned long prevPIDTime = millis();
float kP = 1.2;
float kI = 0.1;
float kD = 0.4;
float prevError = 0;
float integral = 0;
float derivative = 0;

// Constants associated with distance calculations from encoder
unsigned long rpmL = 0, rpmR = 0;
volatile int pulsesL = 0, pulsesR = 0;
unsigned long prevRPMTime = 0;
unsigned long readingDelay = 1000;    // Readings are adjusted (rpmL and
rpmR) every 1000 ms
unsigned int pulsesPerTurn = 20;      // The encoder's optical disc has 20
etched holes
unsigned long debounceInterval = 10; // If a pulse has time period > 10 ms,
it is genuine
```

```
unsigned long prevInterruptTimeL = millis();
unsigned long prevInterruptTimeR = millis();
float wheelDiameter = 6.5;  // Wheel diameter in cm
float wheelCircumference = 3.1415 * wheelDiameter;
float distancePerPulse = wheelCircumference / (float) pulsesPerTurn;
float totalDistanceLeft = 0;
float totalDistanceRight = 0;
float previousDistanceRight = 0;
float previousDistanceLeft = 0;
float averageDistance = 0;


// Constants related to calculations of yaw angle from gyroscope
float rateX, rateY, rateZ;
float roll = 0, pitch = 0, yaw = 0;
float errorX = 0, errorY = 0, errorZ = 0;
unsigned long previousIMUTime = millis();

// Constants related to ultrasonic sensor
float objectDistance = 150.0;
float thresholdDistance = 30.0;

// Constants related to Bluetooth communication subsystems
unsigned long dataDelay = 200;
unsigned long previousSendTime = millis();
float data = 0;
int bluetoothPort = 1;  // Port number is 1,2...N for each robot in the
swarm.

// Constants related to the state machine programming
int state = 0;
float heading = 0;   // The heading that the robot is supposed to be at
int counter = 0;
long completionTimer = millis();
int maxDistance = 0;
float bestAngle = 0;
long ultrasonicTimer = millis();
long backupTime = 200;  // The robot backs up for 200 ms when an obstacle
is detected
bool stopped = false;   // This boolean variable stores whether or not the
robot is stuck/stopped

// Constants related to flame sensor
```

```
bool fire = false;
int flameValue = 1;

// Gives the sign of the number as +1 (positive), -1 (negative) or 0
(zero).
int sign(float number) {
  if (number < 0) return -1;
  else if (number > 0) return 1;
  else return 0;
}

// Drive Subsystem Methods

// Assign the right motor a supplied speed using PWM
// If the supplied speed is negative, the motor must be driven in the
reverse direction
void right(int speed) {
  if (speed > 0) {
    digitalWrite(in1, HIGH);
    digitalWrite(in2, LOW);
  } else if (speed < 0) {
    digitalWrite(in1, LOW);
    digitalWrite(in2, HIGH);
  } else {
    digitalWrite(in1, LOW);
    digitalWrite(in2, LOW);
  }
  analogWrite(pwmR, abs(speed));
}

// Assign the left motor a supplied speed using PWM
// If the supplied speed is negative, the motor must be driven in the
reverse direction
void left(int speed) {
  if (speed < 0) {
    digitalWrite(in3, HIGH);
    digitalWrite(in4, LOW);
  } else if (speed > 0) {
    digitalWrite(in3, LOW);
    digitalWrite(in4, HIGH);
  } else {
    digitalWrite(in3, LOW);
    digitalWrite(in4, LOW);
```

```
  }
  analogWrite(pwmL, abs(speed));
}


// Drive by assigning the right and left motors supplied speeds
void drive(int l, int r) {
  assignedLeftSpeed = l;
  left(l);
  assignedRightSpeed = r;
  right(r);
}

// Reset the correction components of the PID function
void resetTerms() {
  prevPIDTime = millis();
  prevError = 0;
  integral = 0;
  derivative = 0;
}

// PID function to drive forwards at a specified heading
// For e.g. forward(0.0) would make the robot drive forwards at a 0.0 deg
angle, or simply, straight forwards
// For e.g. forward(30.0) would make the robot drive forwards at a 30.0 deg
heading
void forward(float setpoint) {

  // Calculate error and the components of correction
  float error = setpoint - yaw;
  float proportional = kP * error;
  derivative = kD * (error - prevError) * ((float) (millis() - prevPIDTime)
/ 1000);
  integral += kI * error * ((float) (millis() - prevPIDTime) / 1000);

  // Reset the integral term when the robot has stopped
  if (rpmL == 0 && rpmR == 0) integral = 0;

  // Calculate the correction term
  int pid = proportional + integral + derivative;
  int correction = (int) sign(pid) * constrain(abs(pid), 0, 50);

  // Apply correction to the drive subsystem
```

```
  drive(driveSpeed + correction, driveSpeed - correction);

  prevError = error;
  prevPIDTime = millis();

}

// PID function to move to a specified heading
// For e.g. moveToAngle(30.0) would make the robot turn to 30.0 deg
heading, relative to its starting orientation
void moveToAngle(float setpoint) {

  // Since the robot is not driving forwards and is simply point turning,
do not increment distance travelled variables
  resetDistances();

  // Calculate error and the components of correction
  float error = setpoint - yaw;
  float proportional = kP * error;
  derivative = kD * (error - prevError) * ((float) (millis() - prevPIDTime)
/ 1000);
  integral += kI * error * ((float) (millis() - prevPIDTime) / 1000);

  // Reset the integral term when the robot has stopped
  if (rpmL == 0 && rpmR == 0) integral = 0;

  // Calculate the correction term
  int pid = proportional + integral + derivative;
  int correction = (sign(pid) * (int) constrain(abs(pid), minSpeed,
maxSpeed));

  // Apply correction to the drive subsystem. If the robot is within 5° of
its setpoint, it can stop
  if (abs(error) < 5.0) drive(0, 0);
  else drive(correction, -correction);

  prevError = error;
  prevPIDTime = millis();

}

// Interrupt handler for left encoder
void debounceL() {
```

```
  unsigned long interruptTimeL = millis();
  if (interruptTimeL - prevInterruptTimeL > debounceInterval) {
    pulsesL++;
    totalDistanceLeft += distancePerPulse;
  }
  prevInterruptTimeL = interruptTimeL;
}

// Interrupt handler for right encoder
void debounceR() {
  unsigned long interruptTimeR = millis();
  if (interruptTimeR - prevInterruptTimeR > debounceInterval) {
    pulsesR++;
    totalDistanceRight += distancePerPulse;
  }
  prevInterruptTimeR = interruptTimeR;
}

// Resets calculated distances for left and right side
void resetDistances() {
  totalDistanceLeft = 0;
  totalDistanceRight = 0;
  previousDistanceLeft = 0;
  previousDistanceRight = 0;
}

// Processes inputs of encoders and adjusts RPM values
void processEncoders() {

  if (millis() - prevRPMTime >= readingDelay) {
    // Detach interrupts
    detachInterrupt(digitalPinToInterrupt(encL));
    detachInterrupt(digitalPinToInterrupt(encR));
    // Calculate RPM for left and right side
    unsigned long currentTime = millis();
    rpmL = (60 * 1000 / pulsesPerTurn) / (currentTime - prevRPMTime) *
pulsesL;
    rpmR = (60 * 1000 / pulsesPerTurn) / (currentTime - prevRPMTime) *
pulsesR;
    prevRPMTime = millis();
    pulsesL = 0;
    pulsesR = 0;
    // Attach interrupts again
```

```
    attachInterrupt(digitalPinToInterrupt(encL), debounceL, FALLING);
    attachInterrupt(digitalPinToInterrupt(encR), debounceR, FALLING);
  }
}

// Adjust yaw angle based on rate of rotation measured
void processIMU() {
  Wire.beginTransmission(MPU);
  Wire.write(0x43);
  Wire.endTransmission(false);
  Wire.requestFrom(MPU, 6, true);
  rateX = (Wire.read() << 8 | Wire.read()) / 131.0;
  rateY = (Wire.read() << 8 | Wire.read()) / 131.0;
  rateZ = (Wire.read() << 8 | Wire.read()) / 131.0;
  rateX -= errorX;
  rateY -= errorY;
  rateZ -= errorZ;
  roll += rateX * ((float) (millis() - previousIMUTime) / 1000);
  pitch += rateY * ((float) (millis() - previousIMUTime) / 1000);
  yaw += rateZ * ((float) (millis() - previousIMUTime) / 1000);
  previousIMUTime = millis();
}

// Calculate bias in rate of rotation measurements by gyroscope
void calculateIMUError(int caliberate) {
  int counter = 0;
  while (counter < caliberate) {
    Wire.beginTransmission(MPU);
    Wire.write(0x43);
    Wire.endTransmission(false);
    Wire.requestFrom(MPU, 6, true);
    rateX = Wire.read() << 8 | Wire.read();
    rateY = Wire.read() << 8 | Wire.read();
    rateZ = Wire.read() << 8 | Wire.read();
    errorX += (rateX / 131.0);
    errorY += (rateY / 131.0);
    errorZ += (rateZ / 131.0);
    counter++;
    delay(20);
  }
  errorX /= (float) caliberate;
  errorY /= (float) caliberate;
  errorZ /= (float) caliberate;
```

```
}

// Sensor Subsystem Methods

// Calculate object distance from ultrasonic sensor
void processUltrasonic() {
   drive(0, 0);
   digitalWrite(trig, HIGH);
   delayMicroseconds(10);
   digitalWrite(trig, LOW);
   long duration = pulseIn(echo, HIGH);
   objectDistance = ((float) (duration) * ((float) 34 / 2000));
   delay(100);
}


// Process flame sensor reading
void processFlame() {
  flameValue = digitalRead(flame);
  // This robot has detected fire
  if (flameValue == 0 && !fire) {
    fire = true;
    state = 3;
    drive(0, 0);
    delay(100);
    // Send bluetooth data if robot has detected fire
    Serial.print(bluetoothPort);
    Serial.println(" fire");
    delay(dataDelay);
    Serial.print(bluetoothPort);
    Serial.println(" fire");
    delay(dataDelay);
    Serial.print(bluetoothPort);
    Serial.println(" fire");
    delay(dataDelay);
  }
}

// Bluetooth Communication Subsystem

// Send bluetooth data about position and orientation
void sendBluetoothData(int port) {
```

```
  if (millis() - previousSendTime > dataDelay) {
    // Current orientation of robot
    float angle = yaw;
    // Compute average distance travelled since last time data was sent
    averageDistance = ((totalDistanceRight - previousDistanceRight)/2.0) +
((totalDistanceLeft - previousDistanceLeft) / 2.0);
    if (totalDistanceRight == previousDistanceRight) averageDistance = 0;
    if (totalDistanceLeft == previousDistanceLeft) averageDistance = 0;
    previousDistanceRight = totalDistanceRight;
    previousDistanceLeft = totalDistanceLeft;
    previousSendTime = millis();

    // Send port number, orientation, distance travelled, object distance
    Serial.print(port);
    Serial.print(" ");
    Serial.print(angle);
    Serial.print(" ");
    Serial.print(averageDistance);
    Serial.print(" ");
    Serial.println(objectDistance);
  }

}

// Read and process bluetooth data
void readBluetoothData() {

  if (Serial.available() > 0) {
    // Other robot has detected fire
    if (!fire) {
      while (true) {
        // Adjust yaw angle
        if (yaw <= 180) {
          if (yaw > -180) break;
          else yaw -= 360.0;
        }
        else yaw += 360.0;
      }
      // Get relative angle of fire
      data = Serial.parseFloat();
      fire = true;
      // Rendezvous state
      state = 2;
```

```
    }
  }
}

// State Machine

// Handle states
void stateMachine() {
  switch(state) {
    // State 0 - Survey State
    case 0:
      // surveyState();
      greedySurvey();
      break;
    // State 1 - Explore State
    case 1:
      exploreState();
      break;
    // State 2 - Rendezvous State
    case 2:
      rendezvousState();
      break;
    // State 3 - End
    case 3:
      drive(0, 0);
      break;
    default:
      drive(0, 0);
      break;
  }
}


// Survey State
void surveyState() {

  if (counter < 9) {
    float turnAngle = 45.0 * counter;
    if (abs((yaw - heading) - turnAngle) < 5.0) {
      drive(0, 0);
      if (millis() - completionTimer > 2000) {
        processUltrasonic();
        if (objectDistance > maxDistance) {
```

```
          maxDistance = objectDistance;
          bestAngle = turnAngle;
        }
        counter++;
        if (counter == 9) heading += bestAngle;
        completionTimer = millis();
      }
    }
    else {
      moveToAngle(heading + turnAngle);
      completionTimer = millis();
    }
  }
  else {
    // Move to computed best angle
    if (abs(yaw - heading) < 5.0) {
      drive(0, 0);
      if (millis() - completionTimer > 2000) {
        state = 1;
        counter = 0;
        maxDistance = 0;
        bestAngle = 0;
        completionTimer = millis();
      }
    }
    else {
      moveToAngle(heading);
      completionTimer = millis();
    }
  }

}

// Greedy Survey State
void greedySurvey() {
  if (counter < 9) {
    float turnAngle = 45.0 * counter;
    if (abs((yaw - heading) - turnAngle) < 5.0) {
      drive(0, 0);
      if (millis() - completionTimer > 2000) {
        processUltrasonic();
        if (objectDistance > thresholdDistance) {
          // If object is far enough, go to Explore State
```

```
          state = 1;
          counter = 0;
          heading += turnAngle;
          completionTimer = millis();
        }
        counter++;
        completionTimer = millis();
      }
    }
    else {
      moveToAngle(heading + turnAngle);
      completionTimer = millis();
    }
  }
  else {
    state = 1;
    counter = 0;
    completionTimer = millis();
  }
}


// Explore State
void exploreState() {

    int averageAssignedSpeed = (assignedLeftSpeed + assignedRightSpeed) /
2;
    stopped = ((averageDistance == 0) && (averageAssignedSpeed != 0));

    if (millis() - ultrasonicTimer > 250) {
      if (stopped) {
        objectDistance = 0.0;
        // Robot encoders overcount about 10.0 cm in distance when robot is
stuck
        totalDistanceLeft -= 10.0;
        totalDistanceRight -= 10.0;
      } else processUltrasonic();
      ultrasonicTimer = millis();
      // If an obstacle is detected or robot is stopped, back up and go to
Survey State
      if (objectDistance < thresholdDistance) {
        drive(-driveSpeed, -driveSpeed);
        delay(backupTime);
```

```
        drive(0, 0);
        delay(50);
        // Robot backs up about 5.0 cm in backupTime
        totalDistanceLeft -= 5.0;
        totalDistanceRight -= 5.0;
        state = 0;
       }
      }
      else {
       forward(heading);
     }
}

// Rendezvous State
void rendezvousState() {

  // If this robot detects fire, it should stop
  if (flameValue == 0) {
    state = 3;
  } else {
    if (abs(yaw - data) < 5.0) {
      drive(0, 0);
      if (millis() - completionTimer > 2000) {
        // When the robot turns to the heading at which fire is located, it
should move forward
        int averageAssignedSpeed = (assignedLeftSpeed + assignedRightSpeed)
/ 2;
        stopped = ((averageDistance == 0) && (averageAssignedSpeed != 0));
        if (millis() - ultrasonicTimer > 250) {
          processUltrasonic();
          ultrasonicTimer = millis();
          if (stopped || objectDistance < thresholdDistance) {
            drive(-driveSpeed, -driveSpeed);
            delay(backupTime);
            // Robot backs up about 5.0 cm
            totalDistanceLeft -= 5.0;
            totalDistanceRight -= 5.0;
            state = 3;
          }
        }
        else {
          forward(data);
        }
```

```
      }
    }
    // Move to the heading of the fire, as sent by the centralized
controller
    else {
      moveToAngle(data);
      completionTimer = millis();
    }
  }

}

void setup() {

  // Begin serial communication
  Serial.begin(9600);

  // Initialize pins
  pinMode(pwmL, OUTPUT);
  pinMode(pwmR, OUTPUT);
  pinMode(in1, OUTPUT);
  pinMode(in2, OUTPUT);
  pinMode(in3, OUTPUT);
  pinMode(in4, OUTPUT);
  pinMode(encL, INPUT);
  pinMode(encR, INPUT);
  pinMode(trig, OUTPUT);
  pinMode(echo, INPUT);
  pinMode(flame, INPUT);

  Wire.begin();
  Wire.beginTransmission(MPU);
  Wire.write(0x6B);
  Wire.write(0x00);
  Wire.endTransmission(true);

  attachInterrupt(digitalPinToInterrupt(encL), debounceL, FALLING);
  attachInterrupt(digitalPinToInterrupt(encR), debounceR, FALLING);

  calculateIMUError(1000);

  delay(2000);
```

```
}

void loop() {

  processIMU();
  processEncoders();
  sendBluetoothData(bluetoothPort);
  readBluetoothData();
  processFlame();

  // Follow state machine
  stateMachine();

}
```

10.7.2 Processing Sketch

```
import processing.serial.*;
import java.lang.*;
import java.util.*;

// Bluetooth Ports for both robots
String one = "/dev/tty.HC-05-SPPDev";
String two = "/dev/tty.HC-05-SPPDev-1";

// Variables to store data used to track the positions of both robots
Serial port1;
Serial port2;
float[] coordinates1;
float[] coordinates2;
float[] object1;
float[] object2;
boolean state1 = false;
boolean state2 = false;

int size = 150;  // Approximate size of enclosure in cm (used to size
plotting window)
float xoffset = (float) size/2;
float yoffset = (float) size/2;
int scale = 4;  // Scale of plotted map in px per cm
```

```
// Starting position of Robot 1
float x1 = -15;
float y1 = 0;
// Starting position of Robot 2
float x2 = 15;
float y2 = 0;
// Absolute coordinates of detected fire
float xFire = 0;
float yFire = 0;

boolean fire = false;
int portFire = 0; // Stores which port the fire was reported from

void setup() {

  size(600, 600);  // (scale*size, scale*size)
  strokeWeight(2);
  colorMode(RGB);

  // Initialize serial ports
  port1 = new Serial(this, one, 9600);
  port2 = new Serial(this, two, 9600);
  port1.bufferUntil('\n');
  port2.bufferUntil('\n');

}

// Modify angle calculated from atan function based on quadrant
float correctAngle(float angle, float x, float y) {
    if (x < 0 && y < 0) return angle - 180;
    if (x < 0 && y >= 0) return angle;
    if (x >= 0 && y < 0) return angle + 180;
    if (x >= 0 && y >= 0) return angle;
    return angle;
}

// Called when a serial event occurs
void serialEvent(Serial port) {

  String text = port.readStringUntil('\n');

  if (text.contains("fire")) {
```

```
  // If a fire has been detected
  fire = true;
  try {
    String[] data = text.split(" ");
     portFire = Integer.parseInt(data[0]);
  }
  catch(Exception e) {
    System.out.println(text);
  }
  if (portFire == 1) {
    // Calculate coordinates of fire
    float fx = 20.0 * (float) Math.sin(Math.toRadians(coordinates1[0]));
    float fy = 20.0 * (float) Math.cos(Math.toRadians(coordinates1[0]));
    xFire = x1 + fx;
    yFire = y1 - fy;
  } else if (portFire == 2) {
    // Calculate coordinates of fire
    float fx = 20.0 * (float) Math.sin(Math.toRadians(coordinates2[0]));
    float fy = 20.0 * (float) Math.cos(Math.toRadians(coordinates2[0]));
    xFire = x2 + fx;
    yFire = y2 - fy;
  }

}
else {
  // Modify variables storing position, heading of robot 1 and 2
  try {
    String[] data = text.split(" ");
    int portNumber = Integer.parseInt(data[0]);

    if (portNumber == 1) {
      float angle = Float.parseFloat(data[1]);
      float distance = Float.parseFloat(data[2]);
      float object = Float.parseFloat(data[3]);
      coordinates1 = new float[] {angle, distance};
      object1 = new float[] {angle, object};
      state1 = true;
    }
    else if (portNumber == 2) {
      float angle = Float.parseFloat(data[1]);
      float distance = Float.parseFloat(data[2]);
      float object = Float.parseFloat(data[3]);
      coordinates2 = new float[] {angle, distance};
```

```
        object2 = new float[] {angle, object};
        state2 = true;
      }
      else {
        System.out.println(portNumber);
      }

    }
    catch(Exception e) {
      System.out.println(text);
    }
  }


}


// Plot map
void draw() {

  if (fire) {
    // If a fire has been detected, determine angle of fire relative to
second robot
    float angle = 0.0;
    if (portFire == 1) {
      float dx = xFire - x2;
      float dy = y2 - yFire;
      System.out.println(xFire + " " + yFire);
      System.out.println(x1 + " " + y1);
      System.out.println(x2 + " " + y2);
      angle = (float) Math.toDegrees(Math.atan(dx/dy));
      angle = correctAngle(angle, dx, dy);

      // Send the angle to the robot which has not detected fire
      port2.write(Float.toString(angle));
    }
    else if (portFire == 2) {
      float dx = xFire - x1;
      float dy = y1 - yFire;
      System.out.println(xFire + " " + yFire);
      System.out.println(x1 + " " + y1);
      System.out.println(x2 + " " + y2);
      angle = (float) Math.toDegrees(Math.atan(dx/dy));
```

```
      angle = correctAngle(angle, dx, dy);

      // Send the angle to the robot which has not detected fire
      port1.write(Float.toString(angle));

    }
  }

  if (state1) {
    // If new data has been received from robot 1, plot it on the map
    float dx = coordinates1[1] * (float)
Math.sin(Math.toRadians(coordinates1[0]));
    float dy = coordinates1[1] * (float)
Math.cos(Math.toRadians(coordinates1[0]));
    float ox = object1[1] * (float) Math.sin(Math.toRadians(object1[0]));
    float oy = object1[1] * (float) Math.cos(Math.toRadians(object1[0]));
    // Plot obstacle position in red
    stroke(255, 0, 0);
    point((x1 + xoffset + ox) * scale, (y1 + yoffset - oy) * scale);

    x1 += dx;
    y1 -= dy;
    // Plot robot 1 path in green
    stroke(0, 100, 0);
    point((x1 + xoffset) * scale, (y1 + yoffset) * scale);
    state1 = false;
  }

  if (state2) {
    // If new data has been received from robot 2, plot it on the map
    float dx = coordinates2[1] * (float)
Math.sin(Math.toRadians(coordinates2[0]));
    float dy = coordinates2[1] * (float)
Math.cos(Math.toRadians(coordinates2[0]));
    float ox = object2[1] * (float) Math.sin(Math.toRadians(object2[0]));
    float oy = object2[1] * (float) Math.cos(Math.toRadians(object2[0]));
    // Plot obstacle position in red
    stroke(255, 0, 0);
    point((x2 + xoffset + ox) * scale, (y2 + yoffset - oy) * scale);

    x2 += dx;
    y2 -= dy;
    // Plot robot 2 path in blue
```

```
    stroke(0, 0, 255);
    point((x2 + xoffset) * scale, (y2 + yoffset) * scale);
    state2 = false;
  }

}
```