Firefighting Robot

The Caveman

Bryana Baird
Evan Elizondo
Collin Smith

New Mexico Tech

Junior Design
5/6/14

Table Of Contents

List of Figures

List of Tables

1.0 Abstract

A robot was designed to autonomously navigate an 8’x12’ field circumscribed by a white line in search of four candles. The objective was to extinguish all of the candles within three minutes. In addition, the robot had to be able to navigate around obstacles within the course, extinguish the candle within 8” of its base, and deal with sunlight. To accomplish this, the robot had several subsystems: locomotion, obstacle detection, fire detection, fire extinguishing, and line detection. Each of these subsystems functioned well, but still require improvement. The robot was able to successfully extinguish all four candles on multiple runs, but suffered problems with obstacle avoidance that caused it to not be successful all of the time.

2.0 Introduction

The implemented robot known as “The Caveman” was created to perform set tasks for the RoboRave International Firefighting competition. For the competition, several constraints had to be met. The robot had to be capable of autonomously navigating an 8’x12’ field that was marked by white and black reflective tape. The robot needed to be able to locate 4 lit candles within this area and extinguish them. However, in order to extinguish them, the robot had to be within an 8” radius of the candle that was marked by a white line. While trying to locate these candles, the robot, also, had to navigate a set of obstacles. Lastly, all the subsystems needed to be capable of dealing with sporadic sunlight that would be present on the field.

To fulfill these design requirements, several different subsystems were generated. First, the robot needed to be capable of moving around. To fulfill this task, two direct current motors were controlled by a microcontroller through a dual H-bridge motor controller. As the robot moved around the field it, also, needed to avoid obstacles. This was extremely important, therefore, five ultrasonic distance detectors were utilized for this task. The ultrasonics sent pulses to the microcontroller that were proportional to the distance the object was from the sensor. Next, the robot needed to be capable of locating fire as it moved and dodged the obstacles. To do this, an infrared sensing Wii remote was used. This remote was integrated with a microcontroller using an analog circuit. In order to extinguish the fire, once detected, a small propeller was attached to a remote control car motor. The fan was then controlled by the microcontroller through a power MOSFET circuit. The next subsystem needed the ability to detect the lines around the candles and the border of the field. To do this, a reflection detector circuit was created from analog components. The detector gave an analog voltage that showed when it was over a dark object or a white line.

3.0 Design Overview

3.1 Goals

The robot was constructed to compete in the firefighting challenge in RoboRave International. The robot had to autonomously navigate a field to extinguish four candles.The field was 8’x12’ and surrounded by a white line with the candles randomly spaced throughout. All of the candles had to be extinguished within 3 minutes. When these candles were extinguished, the robot had to be within 8” of the candle’s base, the robot could not be moving when the extinguishing device was active, and it could not bump the candle. In addition to this, the candles were obscured by walls scattered throughout the course.

3.2 Constraints

Competing in RoboRave placed a few constraints on the way the way the robot could be built. First, the robot base had to fit within a square base of 144 sq. in. In addition to this, all components on the robot, purchased or donated, had to be within a total of $1500. The final constraint was that all purchased components had to be within a total of $325.

3.3 System Overview

3.3.1 Summary

The robot would randomly choose a direction of travel, and scan for a flame. It would continue until it detected an object or a white line, whereafter it would then choose a new direction to search. Once a candle was detected, the robot would orient itself in the direction of a candle and travel toward it, then extinguish it. It would then choose a new direction of travel to look for more flames. Ultrasonic sensors were used to gauge distances, a Wii remote was used to locate flames, and encoders were used to determine distances traveled. A flowchart of the decision making process can be seen in Fig 3.3.1.

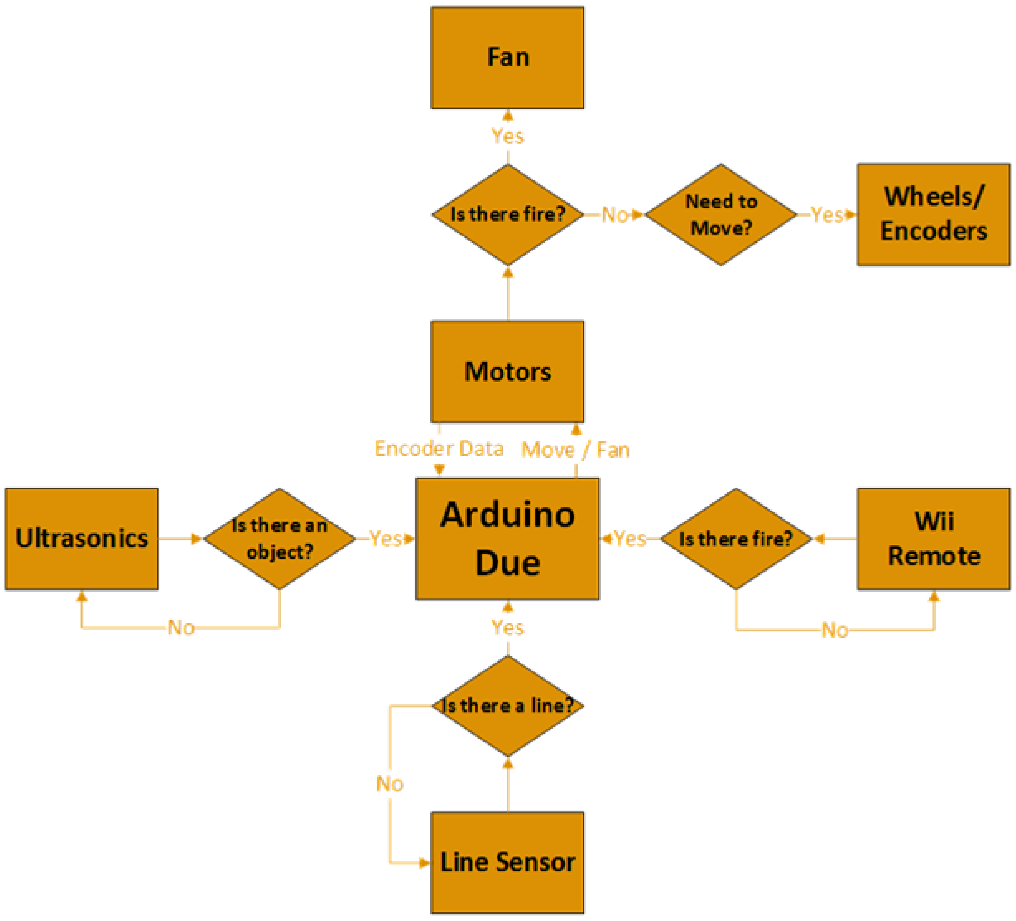


Fig 3.3.1
Flowchart of Decision Making Processes

3.3.2 Search Algorithm

To navigate the field, a Monte Carlo approach was taken. A random direction was chosen and traversed until the robot encountered an object or a white line. If it detected an object, it would then chose a new direction based on the object avoidance algorithm discussed in Sec. 3.3.3. If it encountered a white line, it would turn left randomly between 0 and 90 degrees. This limit ensured that it would attempt to traverse the map in a counter-clockwise fashion while limiting its ability to double back on itself.

In addition to choosing random directions, it would stop traversing the field and spin 360 degrees searching for a candle at set intervals. This helped dramatically increase the likelihood that a candle was detected. Through trial and error, the ideal interval was estimated to be approximately 12 seconds based on our robot's nominal speed. This was used to help compensate for the Wii remote's ability to only scan in one direction.

For future improvements, the authors would recommend traversing the border lines in a counter-clockwise search. This deterministic approach would be an improvement over a random search in several areas: decreased search time, increase probability of finding all flames, and less problems with object avoidance. Occasionally, the robot would accidentally navigate into a corner that also contained a candle, and lost a significant amount of time trying random directions trying to escape.

Deterministic approaches also have drawbacks, however. If the robot loses sight of the line, or wheel slippage occurs, it can be difficult to realign the robot in the correct direction. Implementations of PID control for the motors, initial alignment of wheels when building the base, and mapping algorithms become significantly more important when trying to search deterministically.

3.3.3 Object Avoidance Algorithm

In order to avoid the objects, ultrasonic sensors were used to detect the distance to the nearest obstacle. Five of these sensors were installed on the robot, each pointing a different direction. This way, the direction of the object could be detected. When an object was within a threshold distance for the robot, say 10 cm., it would turn away from the object based on which ultrasonic detected the object.

If the front sensor detected the object, it would choose a new direction of travel. If the sensors on the sides went off, it would rotate 10 degrees away from the object and continue in the new direction of travel. If the middle sensors detected an object, it would turn away 90 degrees from the object, travel forward a little ways, then reorient itself in the original direction. In this way, the robot is able to continually search in a direction without becoming trapped in a corner due to tight spacing of objects. Fig. 3.3.2 Illustrates the different methods of avoiding objects.

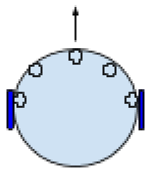


Fig 3.3.3a
Plus - Ultrasonic Detectors
Green Plus - Object Detected
Black Arrow - Original Direction
Red Arrow - New Direction

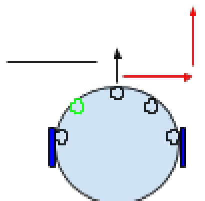


Fig 3.3.3b Middle Sensor Detection

Translates 8" and continues in original direction

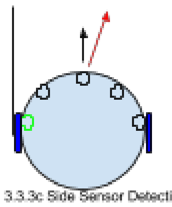


Fig 3.3.3c Side Sensor Detection

Rotates 10 degrees and continues traveling

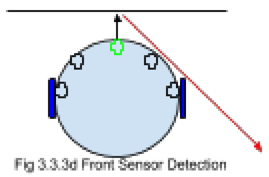


Fig 3.3.3d Front Sensor Detection

Chooses random direction of travel

3.3.4 Flame Search Algorithm

A Wii remote was used to detect the flame. It outputs the coordinates of the flame when it is detected. See Sec. 4.3.2 for further details of operation. While the robot was moving about the field, it was continually searching for a flame by positioning the Wii remote to the left. When one was detected, the robot would turn toward the flame, orient the Wii remote towards the front, then use the coordinates to correct its direction as it drove toward the flame. See Fig 3.3.3a for an illustration of the scanning process.

This was coupled with the object avoidance algorithm. If the robot was driving toward a flame, but encountered an object along the way, it would attempt to avoid the object, and then rescan to locate the candle. See Sec. 3.3.2 for further details about obstacle avoidance.

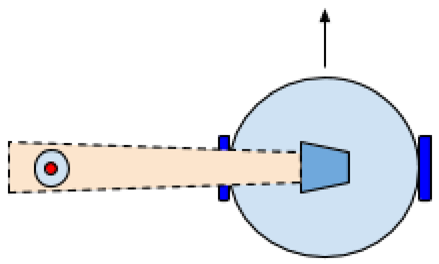


Fig 3.3.4a.1 Initial Detection of Flame

Red Circle - Candle
Trapezoid - Wii Remote
Black Arrow - Direction of Travel

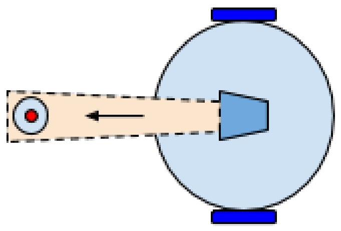


Fig 3.3.4a.2 Driving Detection of Flame

Wii remote is oriented toward the front, robot turns toward candle and drives forward

While the robot was trying to drive toward the flame, it would continually monitor the position of the flame. If the flame was off-center, it would rotate until the flame was back in the center of the field of view. This allowed the robot to correct for factors that caused the robot to not drive straight, such as unbalanced motors or wheel slippage. A window of allowable offset was used to determine when the robot should pause for reorientation. Fig 3.3.3b demonstrates its ability to correct its direction of travel.

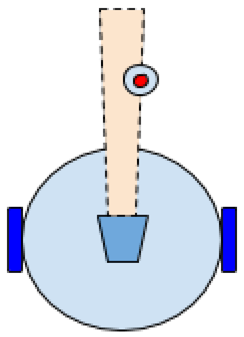


Fig 3.3.4b.1 Top View

Trapezoid - Wii Remote
Red Circle - Candle

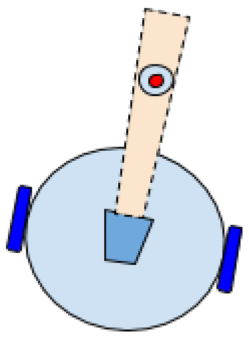


Fig 3.3.4b.3 Top View

Robot reorients toward candle

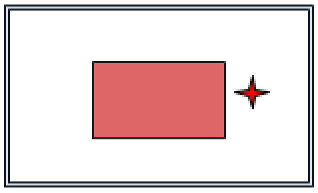


Fig 3.3.4b.2 Wii Remote View

Red Star - Off-Center Detected Flame
Box - Center Region

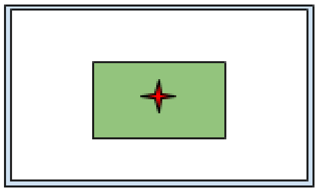


Fig 3.3.4b.4 Wii Remote View

Robot is rotated until flame is back inside center region

3.3.5 Flame Extinguishing

To extinguish the flame, three conditions must be met. The Wii remote must detect a flame, line sensor must detect a white line, and the front ultrasonic sensor must detect an object less than 8 inches away. Including both distance and line detection into the criteria to execute the flame termination algorithm ensured that a fluctuation in either the line sensor or the distance sensor did not prematurely exit the search routine. If the termination algorithm was entered prematurely, the candle risked being extinguished outside of the 8" minimum distance.

When all three of these conditions were met, the robot stopped, then, turned on its fan for 0.75 seconds. If the Wii remote still detected a flame, it would rotate slightly in the direction of the flame, and attempt to extinguish the flame again in a process similar to the process show in Fig. 3.3.4b.

Due to the Wii remote's range and capabilities, it is possible for two flames to be detected simultaneously. See Sec. 4.3 for further details about the Wii remote's abilities. This potential event must be taken into consideration when extinguishing the candle. If one of the candles is extinguished, the brightest IR spot will still exist. Therefore, when monitoring when a candle is extinguished or not, the number of IR spots detected by the Wii remote is monitored rather than the brightest IR spot.

4.0 Subsystems

4.1 Locomotion

4.1.1 Design

The movement of this robot is dependent on these three elements: the H-bridge, two motors, and two ball caster wheels. The robot had a circular base, therefore, having four motors would be inefficient. Both Fig 4.1.1 a. and 4.1.1 b. show how the motors and the ball casters were attached to the bottom of the tier 1 base. The purpose of these components, was to give this robot the ability to rotate 360 degrees in place.

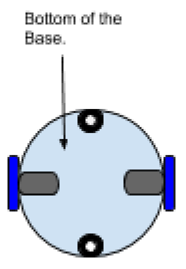


Fig 4.1.1 a. Bottom View

Black circle - Caster Ball Case
White Circle - Caster Ball
Rounded Rectangle - DC Motors

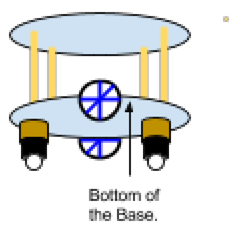


Fig 4.1.1 b. Angled View

Four rods support dual bases. Ball casters balance robot while motorized wheels drive it

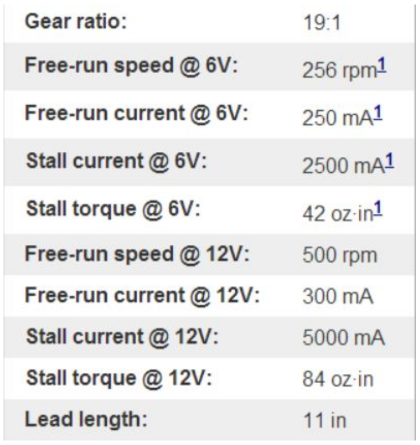
4.1.2 Components

The robots movements are determined by the Locomotion subsystem. Therefore, the first component of choice would be the motors, because they are the foundation for any other elements built around them. There are many different types of motors: alternating current (AC), direct current (DC), stepper, servos…. etc. The DC motor was the best option for the robot because it optimized movement. The alternative stepper motor would not run at a constant low voltage like the DC motor. While, servos would only rotate at 180 degrees and the AC motors were too complicated. However, despite being the best to implement, use of the DC motors still required a balance between torque and speed. The equation in Eq 4.1.1 allowed for a decision on how much torque was required to move the weight of the robot. The equation in Eq 4.1.2 helped determine the rotations per minute (rpm) or speed the robot should move.

$$Torque = m \cdot g \cdot sin(\theta) \cdot wheel\ radius \qquad \text{Eq 4.1.1}$$

$$Rpm = Linear\ speed \div wheel\ radius \qquad \text{Eq 4.1.2}$$

Table 4.1.1
Motor specifications



Gear ratio:	19:1
Free-run speed @ 6V:	256 rpm[1]
Free-run current @ 6V:	250 mA[1]
Stall current @ 6V:	2500 mA[1]
Stall torque @ 6V:	42 oz·in[1]
Free-run speed @ 12V:	500 rpm
Free-run current @ 12V:	300 mA
Stall current @ 12V:	5000 mA
Stall torque @ 12V:	84 oz·in
Lead length:	11 in

The gearhead motors of choice were the Pololu 37D x 64L mm as they allowed for both enough torque and speed based off the specifications in Table 4.1.1. These motors were convenient in other ways as well, since they came with encoders attached. The encoders have two-channel Hall effect encoders used to sense the rotation of a magnetic disk on a rear protrusion of the motor shaft. The quadrature encoder provides a resolution of 64 counts per revolution of the motor shaft when counting both edges of both channels. To compute the counts per revolution of the gearbox output, one would simply multiply the gear ratio by 64. This results in 928 counts per revolution. The motor/encoder has six color-coded, 11" (28 cm) leads terminated by a 1×6 female header with a 0.1″ pitch. Pololu, also, offered brackets, hubs, and wheels that were compatible with the motors. These aluminum mounting hubs allowed for the mounting of custom wheels to 6 mm diameter motor shafts.

The blue 80x10mm pair of wheels decreased the amount of torque provided by the robot, but was still sufficient to move the robot. The hubs and wheel connected together with six 4-40 screws. The gearmotor brackets allowed the motors to be mounted on the robot easily. Each bracket included six M3 screws for securing the motors to the brackets. They, also, feature fourteen mounting holes giving a variety of mounting options to the bottom of the base. The ball caster kit used included a black ABS housing, a 1" diameter plastic ball, three dowel pin rollers, two spacers 1/16" and 1/8" thick, and three #2 screw sets. The spacers helped with adjustments, these will be discussed in Sec 5.2. The L298N dual H-bridge motor driver helped determine the forward and reverse directionality of the motors rapidly while protecting the motor and microcontroller.

4.1.3 Implementation

In order to move, the motors had to be powered directly with 6 volts. As the microcontroller could only put out 3.3 volts, this had to be done through the H-bridge controller. Using the H-bridge as a switch, the microcontroller could put out a low voltage pulse, and still fully power the motors off of the 6 volt battery. The dual directionality of the H-bridge allowed the motors to go in forward and reverse. Which, due to our design, allowed the robot to go forward, reverse, or turn.

The encoders attached to the motors told the microcontroller how far the robot had travelled or how much it had turned. There were problems with the encoders that will be discussed in the section 4.1.4.

4.1.4 Problems

Very few problems were encountered with the motor system. It functioned as intended when it was connected to the H-bridge. However, there were problems with the encoders. They worked well, but the 928 counts per revolution occurred too quickly for the microcontroller to catch regularly. In order to accurately measure distance, each of these edges are needed.

4.1.5 Solutions

To solve the problem with the counts, a separate 74HC4040E ripple counter circuit was designed to lower the resolution. This circuit dropped the number of counts to 1/4th its original value: 232 counts per revolution. The outputs of the ripple counter were read into the microcontroller as an 8 bit number. The ripple counter was then reset, and new distance data was accumulated. The circuit can be seen in Fig 4.1.2.

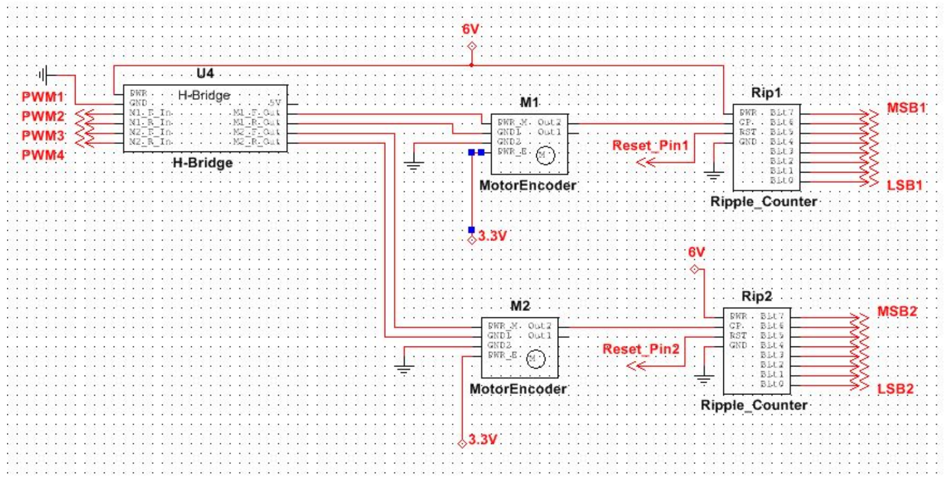


Fig 4.1.2
Locomotion and Distance Recording Circuitry

Once the communication between the microcontroller and the encoders was working, data could be collected. The revolutions per second according to the voltage sent to the motors can be seen in Table 4.1.2. The duty cycle (time spent high per cycle period) was at 100% for all tests

Table 4.1.2
Encoder data according to voltage input

Voltage (100% Duty Cycle)	Rotations per Second
6	3.52
7	4.12
8	5.27

4.2 Obstacle Detection

4.2.1 Design

Obstacle detection was accomplished through ultrasonic range finders. Ultrasonic sensors work through echolocation. They send pulses of sound and measure the time of return. It then generates a pulse to send back to the microcontroller, the length of which corresponds to the distance from the object. These pulses are not within the audible range; the sensors, therefore, are largely immune to ambient background noise.

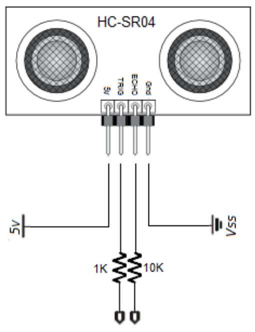


Fig. 4.2.1
HC-SR04 Diagram

4.2.2 Components

The HC-SR04 ultrasonic was chosen because it is generally robust, and very easy to integrate. Pulses are triggered by the “Trig Pin”, and the return pulses are received on the “Echo Pin”. This makes it an ideal component to work with. Distances were able to be accurately measured within 1 cm. Table 4.2.1 shows the data taken at different distances. A picture of the ultrasonic sensor can be seen in Fig 4.2.1.

Table 4.2.1
Distance as determined by HC-SR04

Actual distance (cm)	Sensor Distance (cm)
24	24
30	30
35	34
40	40
45	45
50	50
60	61
65	66
70	71

4.2.3 Implementation

Five ultrasonic sensors were placed around the front of the base. These each pointed in a different direction to attempt to remove blind spots. Each sensor would cause the robot to react differently (see Sec 3.3.2). The ultrasonics were mounted in a plexiglass case to ensure that they were immobile, as well as to make them look nice. Fig 4.2.2 shows a diagram of ultrasonic sensors.

4.2.4 Problems

The ultrasonic sensors had three main problems: timing, viewing angles, and spikes of data. The ultrasonic sensors would be fairly time consuming to read because they had to wait for a sonic burst to be returned to the sensor, and then the pulse itself can be relatively lengthy for objects far away. Since this process had to be repeated five times to read each ultrasonic, it caused slight delays in executing the loop. These delays were not terribly long, but were the slowest part of the program.

The viewing angle was another major issue. Due to their narrow viewing angle, the ultrasonic sensors were not able to detect objects slightly offset to one side. This caused some unanticipated blind spots to occur. This became a problem mostly when pursuing a candle and being unable to determine when no forward progress was made toward the candle.

Finally, sometimes the ultrasonic sensors would glitch and not detect an object. These spikes in data were potentially dangerous, and could cause problems in the avoidance protocol. Usually, these spikes in data indicated that there was not an object in range. This would only be problematic when trying to be within 8” of the candle, so it did matter as much. When the sensor indicated that there was an object when there wasn’t one, this would cause the search algorithm to potentially miss a candle.

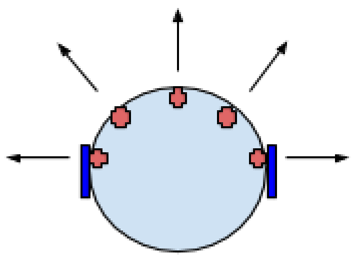


Fig 4.2.2a Top View

Plus - Ultrasonic Sensors
Orange Rectangles- Plexiglass case
Gray Circles - Ultrasonic Sensors
Black Arrows - Direction of sensor

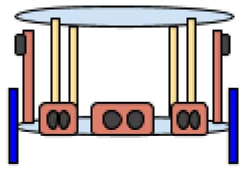


Fig 4.2.2b Front View

Ultrasonic sensors detect objects in the direction of travel and are encased in protective shielding

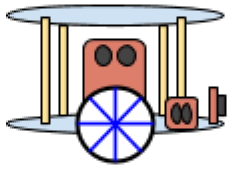


Fig 4.2.2c Side View

Right and Left sensors were mounted above wheels to maintain 180 degree visibility

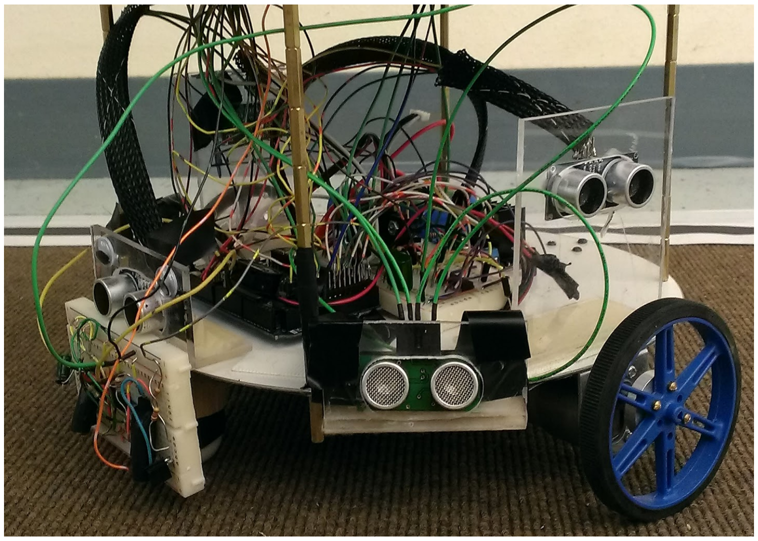


Fig. 4.2.3 Actual placement of ultrasonics

4.2.5 Solutions

For the timing issues, two solutions were used. First, the robot was run at a decreased speed to make sure readings from other sensors were not missed. Second, a timeout for reading the pulse was implemented. This way, if no objects were detected or if the object was a significant distance away, time would not be wasted reading in the pulse.

The viewing angle was more problematic. A timeout was, also, used in conjunction with the encoders to determine how much distance had been travelled within a set time. If a minimum distance had not been met, it would be safe to assume that the robot was stuck on some sort of obstacle. It would then back up, travel to the right a set distance, then rescan for candles. This approach worked fairly well, but was not thoroughly tested. It was implemented during competition with mediocre success. Future solutions to this problem would be to include bump sensors on the robot as a redundancy to the ultrasonics.

Finally, the spikes in data were dealt with by using a median filter. This type of filter removes spikes in data. An averaging filter, for example, would still take the spikes into account, thus skewing the final result. A median filter completely ignores the spikes as long as accurate data makes up more than half of the filter window. An example can be seen in Fig 4.2.4. This filter did slow down actual data processing, but not significantly so, and helped to remove unwanted data.

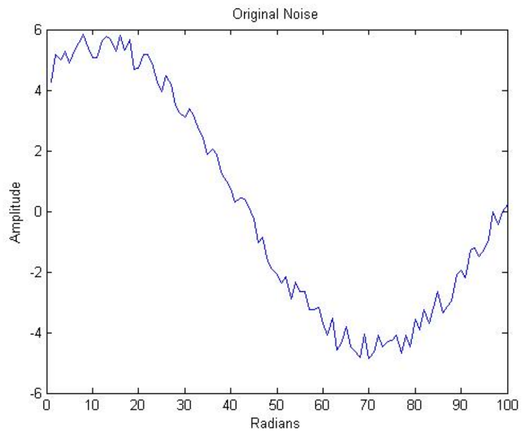


Fig 4.2.4a
Original Noisy Signal

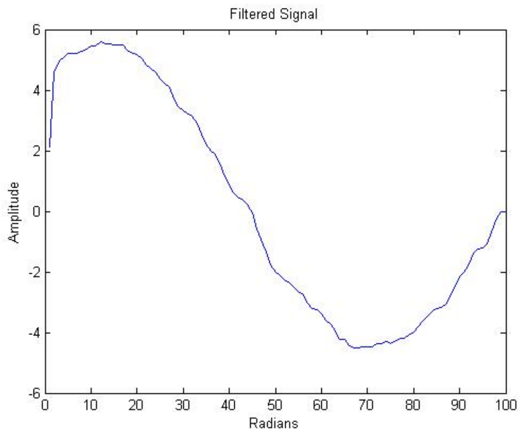


Fig 4.2.4b
Signal after Median Filter

4.3 Fire Detection

4.3.1 Design

Flame detection was achieved by hacking a Wii remote. A Wii remote is the controller for a popular gaming system. On this remote is an IR camera that is capable of on-board processing. In addition to this, threshold levels, gain, and other features are able to be calibrated upon setup. Finally, it comes with IR shielding that prevents ambient light from entering, as well as a plastic case to protect the internal circuitry. It is capable of detecting a flame at a distance of 5 ft. Overall, the Wii remote is a great piece of robust hardware.

4.3.2 Components

The Wii remote works by detecting the four brightest IR hotspots that exceed the programmed threshold in the area, and transmitting these coordinates over I2C protocol to the microcontroller. It has 1024x1024 resolution and runs on a 25 MHz clock. This is very useful when encoding directional algorithms, and it makes visualizing data much easier. Fig 4.3.1 gives a visualization of the coordinates outputted from the Wii remote when the candle was positioned in the lower left hand corner of the viewing range. It can be seen that stable data can be acquired even at the edge of this screen. Fig 4.3.2 shows the candle when positioned in the middle of the viewing screen. Data was acquired over a period of 5 seconds and overlayed on top of each other.

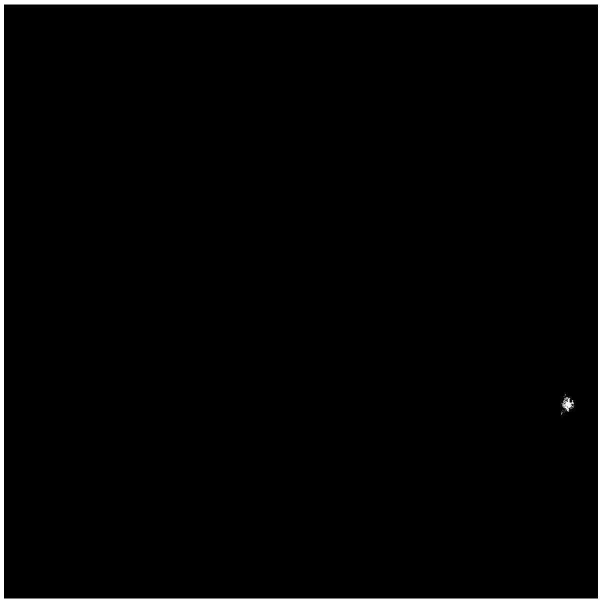


Fig 4.3.1 Wii Remote View
Candle Position: Lower-Left
Accumulation Time: 5 seconds

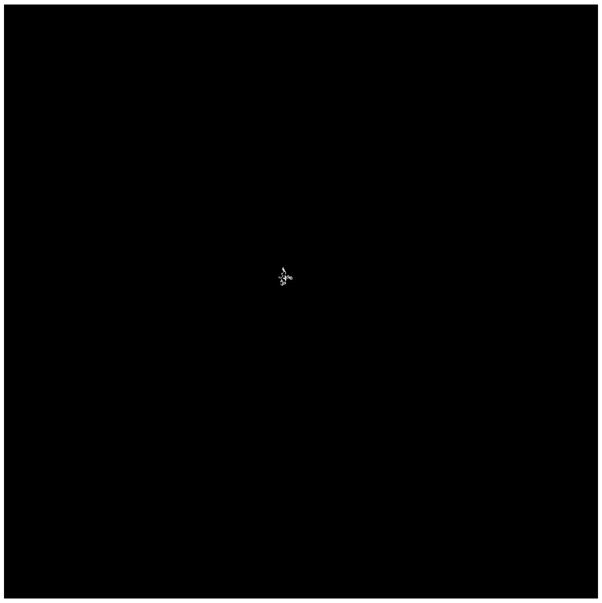


Fig 4.3.2 Wii Remote View
Candle Position: Center
Accumulation Time: 5 seconds

The Wii remote required some external circuitry to function. A 25 MHz clock had to be built to drive the Wii remote. Additionally, if choosing to interface with a 5V microcontroller, an IC interface is required. Both of these circuits can be seen in Fig 4.3.3. Since the Due was used, the SCL and SDA lines were placed directly into the microcontroller rather than putting them through a level shifter.

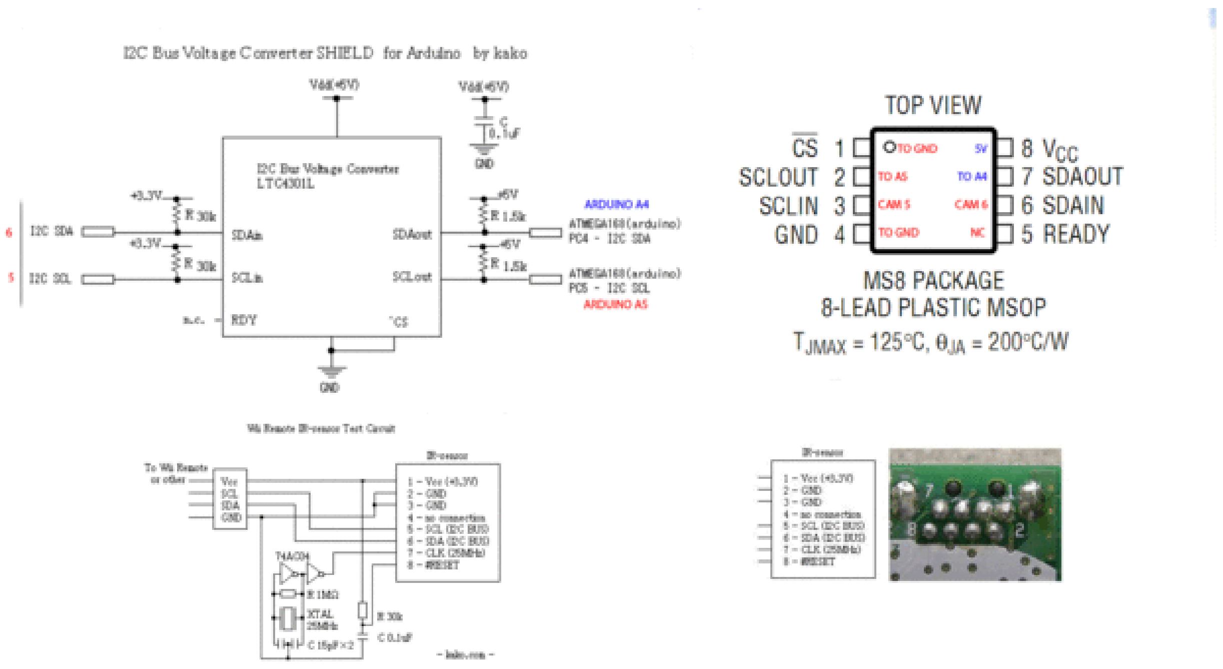


Fig 4.3.3
Wii Remote External Circuitry

4.3.3 Implementation

The Wii remote was mounted on a 180 degree servo. This rotation allowed the robot greater range of freedom while locating flame. The robot could position it to either side while it scanned or move it to the front while moving toward a detected flame. The servo also added a boost in height such that it was positioned above the fan. This allowed the Wii remote a larger, unobscured range of viewing.

4.3.4 Problems

A couple of problems come with working with the Wii remote. It runs on 3.3V, which can be problematic if interfacing with a 5V microcontroller. There are chips that are designed to help interface instruments of different voltages. The one recommended by the Instructables in the references was a little problematic due to its size. It is recommended to find one that is able to be used on a breadboard.

4.3.5 Solutions

The microcontroller used was an Arduino Due, which runs on 3.3V, so this was not an issue in the final design. Prior to choosing the Arduino Due, a level shifter was used to interface between the Wii remote and the Arduino Uno. It turned out to be slightly more impractical than anticipated.

4.4 Fire Extinguisher

4.4.1 Design

To extinguish the candle, a fan was used. A propeller was fashioned to a remote control (RC) car motor. This fan was exceedingly effective, extinguishing flames at distances over 2 ft. We, also, blew more than one fuse when prototyping the fan circuitry. Since the fan was such a power drain, we could only turn it on for a limited amount of time. A circuit was developed to interface the Arduino Due with the battery voltage, and can be seen in Fig 4.4.1. The first two transistors act as a switch. When 3.3V is applied to the input, a 6V signal is sent to the power MOSFET. This causes the high-amperage signal to be sent through the MOSFET to the fan. When the input goes low, the current through the MOSFET is shut off.

4.4.2 Components

The power MOSFET used was the IRF510. It is a common, well documented transistor that is capable of handling the high current needed to drive the fan. A heat sink was attached to help deal with heating issues due to the power dissipated across the transistor. The transistor comes with a screw hole for easy attachment.

The initial switch was designed using two common NPN transistors: 2N3904. These common BJTs make the interface between the MOSFET and the Arduino. Since there is not a lot of current running through these transistors, no power components or heat sinks were necessary.

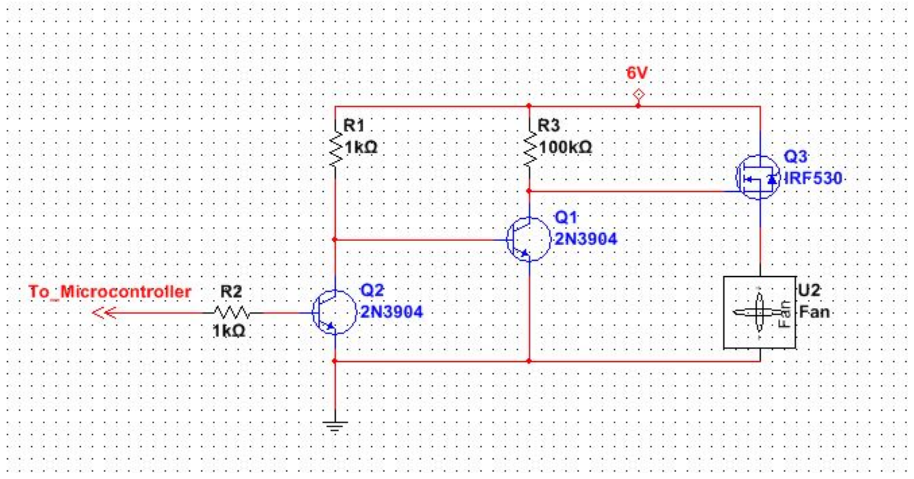


Fig 4.4.1
Interface Circuitry for Fan

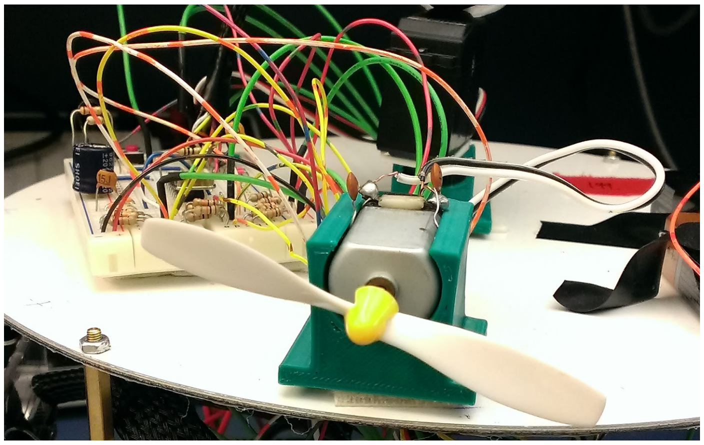


Fig 4.4.2
Fan with Custom 3D Printed Case

4.4.3 Implementation

The fan was mounted using a custom 3D printed case. This allowed the fan to remain stationary. This can be seen in Fig 4.4.2. This was necessary due to the torque generated by the fan turning at such high velocity. It was positioned below the Wii remote such that the propellor did not block the view of the camera. Unfortunately, this slight offset in positioning did not place the fan in direct alignment with the candle itself. This was not a problem, however, since the massive influx of air was more than adequate to terminate the flame.

4.4.4 Problems

The two biggest problem when designing the fan was dealing with the power dissipation across the fan and interfacing it with the 3.3V output of the microcontroller. The fan was essentially a motor, one that required a lot of power to operate. This means that anything used to control its operation would have to be able to dissipate the power safely. It would also have to be able to be turned on by the 3.3V output. Finally, it must not require a lot of power to operate in order to prevent damage to the port.

Another problem was the kick from the motor. When spinning at high speeds, it caused the base to want to twist away. Its odd shape made it hard to find standardized methods to attach it to the base.

4.4.5 Solutions

A power MOSFET was used as the switch for the fan, and a heat sink was attached to it to help dissipate the heat. Two BJTs were used to interface with the microcontroller as described in the Design section above. This switch took little current to operate, which helped protect the ports on the microcontroller.

A 3D printed case was used to fix the location of the fan. This case fit securely over the fan, which helped with the torque generated by the propellor. The base was secured to the robot using industrial strength velcro. This caused the fan to be totally immobile until it needed to be moved.

4.5 Line Sensing

4.5.1 Design

The line sensor was chosen as the analog circuit that would be designed and constructed from scratch. To design this circuit, several important constraints were taken into account. The first requirement was that the line detector had to be capable of telling the difference between the carpeted arena and the white lines. Next, it was important that the circuit be capable of dealing with sunlight that would be lighting up the final fields at the RoboRave competition. Lastly, it was ideal for the line sensor to be capable of telling the differences between the black line and the carpet.

The design was created so as to fulfill each of these tasks according to the level of importance set for each one. To tell the difference between a white line and dark carpet, it was determined that a reflection detector would work well. The reflection detector would consist of a light emitting diode and a photodiode behaving as emitter and receiver, respectively. The light emitting diode would send light down towards the ground, and then the light would be reflected back towards the receiver. This behavior is shown in Fig 4.5.1.

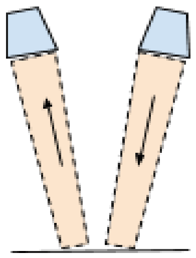


Fig 4.5.1
Light emitted and received

When the emitter sends light to the dark carpet or black line, the light is absorbed. This leaves less light to reflect and be seen by the receiver. When the emitter sends light to the white line, very little is absorbed and so more reflects into the receiver. The amount of current the photodiode lets through it is proportional to the amount of light reflected into it. Therefore, this design allowed the line detector to output a higher voltage when a white line was seen, versus when a black line or carpet was seen.

The next important constraint for designing the line detector was that it needed to be able to deal with changes in ambient light. To do this, the line sensor needed to be capable of isolating the signal being created by the light emitting diode. The chosen method for this was a modulated signal. A frequency would be pulsed on the emitter, and a circuit would be created after the diode to isolate this frequency against all outside light. The frequency chosen was 1 kHz, and it was isolated using a bandpass filter. It was found, that by isolating the emitter and detector from ambient light, the circuit was capable of telling the difference between a black line and the carpet.

4.5.2 Components

The components for the design were chosen specifically to fulfill these tasks as well as possible. For the emitter/receiver pair, the light emitting diode and photodiode were chosen in order to match up their intensities. The chosen photodiode was a 475-2649-ND. This diode has a peak intensity at 840nm but the second strongest intensity was near the 600 nm range. For this reason, an emitter was chosen that had an emission wavelength of 640 nm in order to overlap with the peak intensities of the photodiode, this was a 754-1299-ND.

Next, the components for the filter needed to be determined. For this filter, it was determined that the task could be completed using a basic resistor and capacitor set up as long as enough iterations were used. The components for this filter were determined using the cutoff equation for a filter as shown below in equation 4.5.1.

$$f = \frac{1}{2\pi RC} \qquad \text{eq 4.5.1}$$

For the bandpass filter, a low pass filter and a high pass filter were implemented. Two iterations of each, when done creatively, were enough to isolate the frequency and remove nearly all noise. The final components needed were operational amplifiers. In the case of this project, the operational amplifiers chosen were LF411 due to their availability and good slew rate. The operational amplifiers were used to make the small signal sent from the photodiode into something that could be filtered and used.

4.5.3 Implementation

In order to implement this circuit, many details had to be addressed. When working with the tiny signals given by a photodiode, the circuit must be made to control losses and maintain the integrity of the signal. The first thing that had to be done to ensure this was to use a transimpedance amplifier as the amplifier immediately after the photodiode. If a typical inverting amplifier circuit had been used, the voltage drop across the initial input resistor would have had negative effects on the circuit. After the transimpedance amplifier came the low-pass filter. The resistor values chosen for this sent the signal through a 212Hz filter and then through a 1591Hz filter. The resistor values were chosen to be low (75Ω and 10Ω respectively) in order to minimize the amount of voltage loss across them.

The signal was then sent through another operational amplifier in order to boost it before the next filter. This was done, because when the whole bandpass filter was implemented at this stage initially, the losses were extreme, and the signal disappeared. So, by boosting the signal before the second half of the filter, the signal survived through the whole filter. The high pass filter was implemented at this stage. The first cutoff frequency it went through was a 24Hz cutoff followed by a 34Hz cutoff. Despite these values seeming low for a 1kHz filter, the result was a perfectly isolated signal. The final stage the signal went through was two amplifiers used to get the signal near the 3.3 volts that the Arduino Due can take in. The final results of the circuit were that the signal could see differences between carpet, a white line, and a black line. The circuit diagram can be seen in Fig 4.5.2. The results for each surface (black line, carpet, and white line) can be seen below in Fig 4.5.3a-4.5.3c.

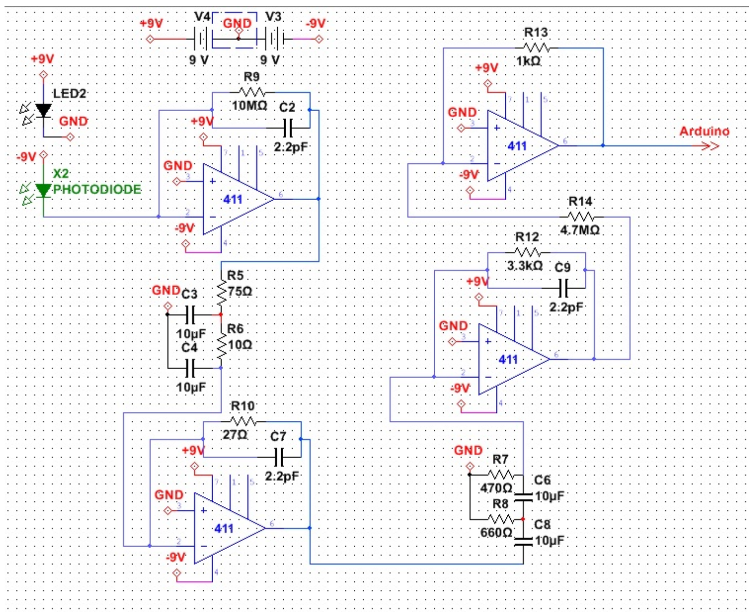


Figure 4.5.2
Circuitry for modulated line detector

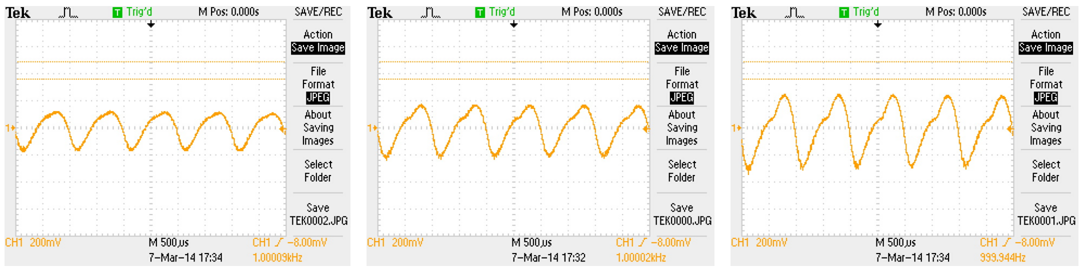


Fig 4.5.3 (a,b,c)
Black line, carpet, and white line responses

To use this circuit effectively, however, a peak detector was added at the final output so that the Arduino Due could sample at its convenience rather than at the nyquist frequency. Despite full functionality of the individual circuit, however, the integration step proved to be too complicated. Therefore, a smaller, less exact, circuit had to be constructed. The specific problems encountered can be found in the following section.

The circuit that was integrated into the robot was the first portion of the original circuit. To remove complexity, the modulation was removed, and therefore, the bandpass filter was removed. This left a basic line detector as described above. However, this detector was capable of seeing white lines very well against the black lines and carpet.The behavior of the circuit can be seen in Fig 4.5.5. This figure shows a low value as the robot moves across the carpet and a high spike upon seeing the white line. To deal with the concerns about sunlight, a shield was built for the circuit to remove ambient light. The circuit itself can be seen below in Fig 4.5.4.

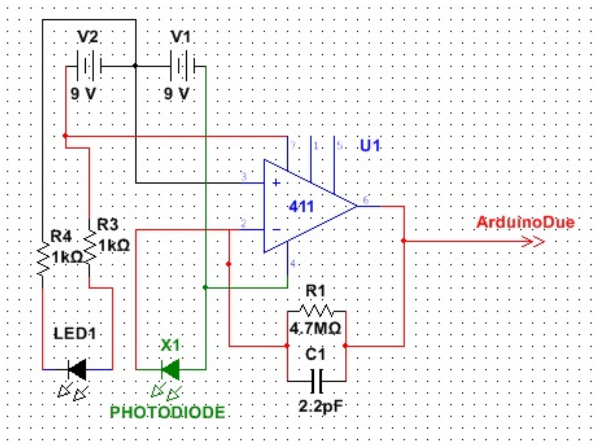


Fig 4.5.4
Integrated reflection detector circuit

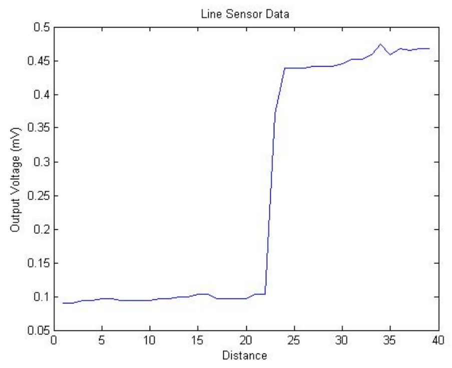


Fig 4.5.5
Difference between carpet and white line

4.5.4 Problems

Several problems were encountered during the making of this circuit. The first, as mentioned above, were the issues of signal losses. The next problem encountered was slightly less intuitive. During the implementation of the band pass filter, an unexpected, and strong, signal of 2Hz was appearing in the system after it was sent through the filter. It was determined that the integration that the filters implement in order to filter the signal was causing this problem.

The next problem encountered was when several operational amplifiers and other circuits began to be attached to the sensor. The circuit began having trouble with proximity to humans and other objects. When in proximity to these objects, the operational amplifiers would rail.

Occurring with the above problem, was a similar response to the removal of ambient light. In the case of the circuit being shaded or faced downward, the operational amplifiers would, again, rail.

The final problem was that, on occasion, the signal would make it safely through the circuit, but would not respond to changes in the surface it was looking at. This was the problem that prevented integration of the system with the remainder of the robot.

The last problem was with the circuit that was utilized on the robot. For this circuit only one operational amplifier was being used, but the Arduino Due could not take an input of negative voltages.Therefore, the circuit needed to output a positive voltage.

4.5.5 Solutions

As the problems came up during the design of the circuit, each one needed to be solved. First, the problem of losing the signal was solved by lowering the value of the resistors used in the filters and including operational amplifiers between each filter

The extra 2Hz signal was caused by the use of a square wave to modulate the signal. By using a sine wave that problem was solved.

Next, were the proximity, ambient light, and lack of response problems. After much research, failed solutions, and consulting with an expert in the field, it was found that capacitance was the problem. The photodiode requires matched capacitances within its circuit. Therefore, as a new item was added, the capacitance changed and so the circuit had to be altered. The key alteration was capacitors across the feedback loop of the amplifiers that were used to match the circuit’s capacitance. However, as anything new happened (a component broke and had to be replaced, a new circuit was added, a battery changed..etc) the capacitors across the feedback had to change. Once the correct capacitors were used, the circuit was perfect for that situation from then on, no matter what, but finding the capacitors proved time consuming.

To deal with the desire for one operational amplifier and a positive output voltage, the solution was to simply power the photodiode with a negative voltage. By doing this, the negative voltage would then be inverted and go into the Arduino as a positive voltage.

4.6 Microcontroller

4.6.1 Design

The Arduino Due was used as the processing unit of the robot. This was chosen because of its large memory, plethora of IO ports, as well as its being well documented. To interface with the circuitry, several connectors were developed to help reduce the number of wires used to interface with the board.

4.6.2 Components

The Due operates at 3.3V, can output PWM signals with 12 bits of precision to control the H-bridge, comes with ADC converters for the line sensor, and digital ports for the ultrasonic sensors. The Due also comes with a myriad of open source libraries, such as digital filters, wrappers for the Wii remote, I2C communication, servo controls, etc. These external libraries make programming much easier.

4.6.3 Implementation

The microcontroller was placed at the bottom of the robot by the H-bridge. It was centrally located to easily interface with the motors, Wii remote, and ultrasonics. It was secured to the base by three screws, slightly elevated from the base to avoid shorting any of the circuitry on the back. This system allowed components to be moved around while keeping the microcontroller centrally located. The microcontroller could also be easily removed when needed while staying securely fashioned otherwise.

4.6.4 Problems

The Due operates at 3.3V, which allows it to interface easily with the Wii remote. The ultrasonic sensors, however, operate at 5V. If not careful, these ultrasonic sensors could potentially destroy a port on the Due. The trigger pin of the ultrasonics could be activated by a high from the microcontroller, but the return pulse had to be stepped down from 5V to 3.3V. Other chips that operate nominally at 5V were operated outside of the recommended voltage range as well. See Sec. 4.2 for further details about ultrasonic operations.

4.6.5 Solutions

The return pulse from the ultrasonic sensors were stepped down to 3.3V through a voltage divider. Since the input pins to the arduino are high impedance, this was a feasible solution. Other chips that fed information into the microcontroller, were operated at 3.3V instead of the recommended 5V. Though it would have been possible to send each through a voltage divider or level shifter, this would have required much more board space to properly implement. The ripple counter, for example, was operated at 3.3V both in the inputs to and the outputs from the microcontroller, and there was never a problem with it operating correctly.

5.0 Structure

5.1 Design

The robot design was founded on a few requirements the authors deemed important. The first, was a need for mobility and ease of turning. The second requirement, was for space to place the various components and power sources. Finally, all of the components and power sources needed to be securely mounted to avoid losing parts as the robot moved.

The need for mobility and ease of turning lead to the creation of a round chassis. This chassis, with two motors, would allow the robot to turn itself in place. Considering the need to avoid obstacles that may be close together, this was considered to be extremely important.

To deal with the need for space, the robot was created with two tiers. This allowed three surfaces for mounting (top of the base, bottom of second tier, and top of second tier) and raised the IR detector and fire extinguisher to the level of the candles.

Finally, to secure all of the components and power sources, a combination of industrial strength velcro, plexiglass mounts, and 3D printed mounts were used.

5.2 Implementation

The robot's chassis was built using two circular aluminum signs with a 10" diameter. These signs kept the base within the 144 square inch requirement for the competition. The two tiers were obtained using standoffs that allowed the height to be adjustable within one inch.

A ruler was used to line up the motor brackets to make sure that they were parallel. Tape held the brackets down, while two holes were drilled through the center of them. They were each secured in the middle with two 4-40 bolts and nuts. After that was complete, the drilling continued on the two inside holes and on the two outside for a total of six nuts and bolts. This secured the motors and brackets onto the bottom of the base.

The ball casters were attached by taking one of the spacers and sizing a drill bit to fit the size of the holes. Then three #2 holes were drilled into the base to secure the caster case. A 90 degree angle protractor was used to make sure the ball casters were perpendicular to the DC motors.

The ultrasonic sensors were mounted in plexiglass shields that allowed for differences in height for different sensors, and prevented shorting across the mount. The line sensor, power battery, ripple counter circuitry, and Wii remote were all mounted using industrial strength velcro. This velcro allowed the components to be removed when necessary but held securely in place during movement.

The fire extinguisher was held in place with the combination of a 3D printed mount, and velcro. This was necessary, because the fan would rotate the robot if not allowed some amount of movement (the mount) but still needed to be securely placed on the robot (the velcro).

Lastly, the Arduino Due and the H-bridge motor controller were mounted using bolts and custom drilled holes. These bolts had rubber boots underneath them to prevent connections between the components and the base. Holes were drilled through the base so the wires from the H-bridge could be run to the motors.

The orientation of the components on the top tier and bottom tier can be seen in Fig 5.2.1 and Fig 5.2.2.

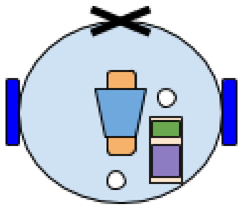


Fig 5.2.1 Top View - Top Level

Trapezoid - Wii Remote
White Circle - Holes
Green Square - Voltage Dividers
Purple Square - Wii External Circuitry
Orange Rectangle - Servo
Black Cross - Fan

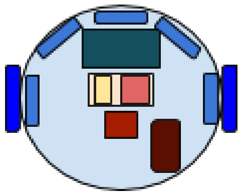


Fig 5.2.2 Top View - Bottom Level

Blue Rectangle - Ultrasonic Sensors
Cyan Rectangle - Arduino Due
Pink Rectangle - Motor Circuitry
Yellow Rectangle - Fan Circuitry
Red Rectangle - H-Bridge
Brown Rectangle - Battery

5.3 Problems

The first problem encountered with the structure of the robot was that the ball caster wheels were not large enough to reach the floor from the base of the robot. The second problem was that the robot was high centering.

5.4 Solutions

To solve the problem of the ball caster wheels not reaching, a simple wooden dowel was used to make up the difference. To fix the high centering, it was found that the caster wheels needed to be sanded down slightly as they were lifting the base up too high.

6.0 Performance

6.1 Individual Subsystems

All the subsystems of the robot were fully functional individually as was necessary for an effective integrated system. The locomotion allowed the robot to move efficiently, rotate in place, and, after modifications, the high centering problem was solved. The ultrasonic range finders worked well as object detection despite delays and small blind spots. After a few adjustments to the navigation algorithm, the object detection was fully functional. The fire detection system, though problematic at first, proved extremely effective when implemented with an Arduino Due. It was capable of finding the candles from a fair distance away with a lot of accuracy.

The fire extinguisher subsystem was more effective than we could have hoped. The propeller attached to the remote control car motor put out candles in under a second. This gave more time for the robot to navigate the field. The initial line sensor subsystem worked perfectly, but was not implemented on the robot. Instead, a simpler system was implemented, which fulfilled the require role. The system could quickly sense white lines, and an external shield was implemented to deal with sunlight. This shield was used only once, and so the effectiveness was difficult to determine.

The decision making subsystem, or navigation algorithm, was extremely effective. Despite the use of a Monte Carlo search method, the robot was still capable of finding and extinguishing all four candles in three minutes. The algorithm, also, helped deal with problems of the other sensors such as the blind spots in the ultrasonic sensors.

6.2 Integrated System

When integrated, each subsystem continued to perform its needed task. The line sensor was still capable of finding where a white line was. Once the line was found, the algorithm would then tell the robot to turn away from the line until the line sensor was no longer showing that the line was present.

The ultrasonic sensors were capable of sensing objects. They would then alert the microcontroller when objects were in front of it, caddy corner, or to its side. The information would then be sent from the microcontroller that told the robot to avoid said object.

The fire detection was capable of finding fires, and informing the microcontroller where that fire was located. The algorithm would then determine how to alter the robot’s course to send it towards the flame. As this was happening, the robot would check to see if a white line and an object were detected. When these two other sensors were activated, the microcontroller would activate the fan, as desired, and remove the flame. If the fan was turned on and the flame sensor said fire was still there, the robot would correct, and attempt the extinguish, again.

In the case of the robot getting stuck on an object, the timer would begin counting and trigger if the microcontroller did not see movement in the wheels. When the timer triggered, the robot would reverse and attempt to circumvent the object.

All of these behaviors were working as intended and, therefore, it can be concluded that the robot was fully functional when integrated.

6.3 At RoboRave International

At the RoboRave International competition, the robot performed extremely well. The scores counted by the judges (the top three) consisted of one perfect run where all 4 candles were extinguished in the 3 minute restriction and two runs where three candles were extinguished.

7.0 Budget

Though the budget did not exactly match our predicted, it was extremely close. Also, the cost of parts was considerably less than the allocated budget.

7.1 Predicted

Table 7.1
Predicted budget

Component	Number Needed	Allocated Amount ($)
Wheels/Mounts/Hubs	4	90
Direct Current/Step Motors	2	40
Heat Sensor	1	25
Reflective Sensor	3	15
Object Sensors	~3	40
Integration Circuitry	TBD	20
Replacement Parts	TBD	95
	Total Cost	**325**

7.2 True

Table 7.2
True budget

Component	Number Needed	Amount Spent ($)
Wheels/Mounts/Hubs	4	30.52
Direct Current/Step Motors	2	57.98
Heat Sensor	1	25
Reflective Sensor	2	25.36
Object Sensors	5	40
Integration Circuitry	3	23
Replacement Parts	1	50
	Total Cost	**251.68**

8.0 Schedule

8.1 Predicted

Table 8.1
Predicted schedule

Tasks	IR Algorithm	Mapping Algorithm	Object Sensor	Line Sensor	Condition Testing	Base Integration	Motor Integration	Testing
2/3-2/9			▨	▨				
2/10-2/16	▨		▨	▨		▨		
2/17-2/23	▨		▨	▨		▨		
2/24-3/2	▨		▨	▨		▨		
3/3-3/9	▨	▨	▨	▨		▨		
3/10-3/16	▨	▨	▨	▨		▨		
3/17-3/23	▨	▨		▨	▨	▨		
3/24-3/30		▨			▨		▨	
3/31-4/6							▨	
4/7-4/13							▨	
4/14-4/20								▨
4/21-4/27								▨

8.2 True

Table 8.2
True schedule

Tasks	IR Algorithm	Mapping Algorithm	Object Sensor	Line Sensor	Base Integration	Motor Integration	Testing
2/3-2/9				▨			
2/10-2/16			▨	▨	▨		
2/17-2/23			▨	▨	▨		
2/24-3/2			▨	▨	▨		
3/3-3/9			▨	▨	▨		
3/10-3/16			▨	▨	▨	▨	
3/17-3/23	▨			▨	▨	▨	
3/24-3/30	▨			▨		▨	
3/31-4/6	▨	▨		▨		▨	
4/7-4/13	▨	▨		▨			
4/14-4/20		▨		▨			
4/21-4/27		▨					▨
4/28-5/3		▨					▨

The category for condition testing was removed due to the fact that all condition testing was done on the individual subsystems and is, therefore, incorporated into each respective subsystem category.

9.0 Conclusion

Through the course of this project a robot capable of putting out 4 lit candles in 3 minutes was designed. This robot could navigate a 8'x12' field through the use of two direct current motors. It did this task autonomously due to the navigation algorithm coded onto an Arduino Due microcontroller. The microcontroller took data from ultrasonic sensors in order to avoid obstacles placed on the field and data from a reflection detector to determine the location of the edge of the field and candles. It could eliminate the fire through the use of a high powered fan, and tell when that fire was removed.

This robot was constructed under a budget of $251 making it sufficiently under both the RoboRave and purchasing budgets allotted. The robot's functionality was also tested at RoboRave International 2014 where it successfully put out all 4 lit candles in three minutes.

10. Reference

Arduino. “Arduino Due.” *Arduino*. Arduino. June 2013. Web. http://arduino.cc/en/Main/arduinoBoardDue

BigRedRocket. “Wii Remote IR Camera Hack....” *Instructables*. Autodesk, Inc. 2000. Web. http://www.instructables.com/id/Wii-Remote-IR-Camera-Hack/

CTS. “ATS/ATS-SM Series Quartz Crystal.” *CTS*. CTS Corp. 2014. Web. http://www.ctscorp.com/components/Datasheets/008-0309-0.pdf

Fairchild. “NPN General Purpose Amplifier.” *Fairchildsemi*. Fairchild. Oct 2011. Web. http://www.fairchildsemi.com/ds/MM/MMBT3904.pdf

Harris Semiconductor. “High Speed CMOS Logic 12-Stage Binary Counter.” *TI*. Texas Instruments. Oct 2003. Web. http://www.ti.com/lit/ds/symlink/cd74hc4040.pdf

Kako, E. “Small Sotry 2008-2009.” *Kako*. Kako. 2008. Web. http://translate.google.com/translate?u=http%3A%2F%2Fwww.kako.com%2Fneta%2F2008-009%2F2008-009.html&hl=en&ie=UTF-8&sl=ja&tl=en

Pololu. “Motor with 64 CPR Enconder for 3rD mm Metal Gearmotors.” *Pololu*. Pololu. 2014. Web. http://www.pololu.com/product/1440

STMicroelectronics. “Dual Full-Bridge Driver.” *ST*. STMicroelectronics. Jan 2000. Web. https://www.sparkfun.com/datasheets/Robotics/L298_H_Bridge.pdf

Storr, Wayne. “Passive Low Pass Filter.” *Electronics Tutorials*. Basic Electronic Tutorials. 3 May 2014. Web. http://www.electronics-tutorials.ws/filter/filter_2.html

Texas Instruments. “Hex Inverters.” *TI*. Texas Instruments. July 2003. Web. http://www.ti.com/lit/ds/symlink/sn74hct04.pdf

Tillaart, Rob. “A runningMedian Class for Arduino.” *Playground.Arduino*. Arduino. 30 Nov 2013. Web. http://playground.arduino.cc/Main/RunningMedian

Vishay. “Power MOSFET.” *Vishay*. Vishay. 21 Mar 2011. Web. http://www.vishay.com/docs/91015/sihf510.pdf

11. Appendix

11.1 Main Code

```
#include <RunningMedian.h>
#include <Wire.h>
#include <Servo.h>

#define varlen 400
#define fieldWidth 96
#define fieldHeight 144
#define NumUS 5
#define LEFT 0
#define UP 1
#define RIGHT 2
#define DOWN 3
#define NOTHING 0
#define DETECTED 1

//Filter Sizes
#define IRCam_FiltSize 5
#define US_FiltSize 4
#define Line_FiltSize 3

//-----------------------------------------------------------------
//Pins
//-----------------------------------------------------------------

//Ultrasonic Sensors
const byte US0_EchoPin = 43;
const byte US0_TrigPin = 45;
const byte US1_EchoPin = 40;
const byte US1_TrigPin = 38;
const byte US2_EchoPin = 48;
const byte US2_TrigPin = 49;
const byte US3_EchoPin = 34;
const byte US3_TrigPin = 32;
const byte US4_EchoPin = 46;
const byte US4_TrigPin = 47;

//Motor Control
const byte lrmotor = 9;          //Left Forward Motor Control
const byte lfmotor = 8;          //Right Forward Motor Control
const byte rrmotor = 7;          //Left Reverse Motor Control
const byte rfmotor = 6;          //Right Reverse Motor Control
const int whitePin = 24;         // white wire on motor
const int yellowPin = 22;        // yellow wire on motor

//Distance from ripple counter
const byte dist_reset = 23;
const byte dist7 = 25; //MSB
const byte dist6 = 27;
const byte dist5 = 29;
const byte dist4 = 31;
const byte dist3 = 33;
const byte dist2 = 35;
const byte dist1 = 37;
const byte dist0 = 39; //LSB
```

```
//Line Sensors
const int line1Pin = A3;
const int line2Pin = A4;

//Miscellaneous Pins
const byte servoPin = 5;        //Servo for Wii Remote
const byte fanPin = 30;         //Control pin for Fan
const byte calibrationPin = 41;  //Button to calibrate turns if wanted
const byte ledPin = 13;         //On Board LED

//Robot dimensions
const float wheelDiam = 3.0;
const float roboDiam = 11.75;

//-----------------------------------------------------------------
//Driving/Motor Variables
//-----------------------------------------------------------------
int curdir = UP;       //Direction of robot
int coord[2] = {0, 0};//Location of robot
int totrot[2] = {0, 0};   //Total distance travled [full rotation, fraction]
int origLoc[2] = { -1, -1}; //Location to return to after putting out candle
int origdir = UP;      //Original direction of travel
int origIRCamdir = LEFT;  //Original direction of Wii remote
int totdist = 0;        //Total distance traveled during forward
int minwind = 0;       //Difference between left and right motor PWM duty cycles
int ldist = 0;          //Distance to turn left 360 degrees
int rdist = 0;          //Distance to tyurn right 360 degrees
int cnt = 0;

//-----------------------------------------------------------------
//Timeout Variables
//-----------------------------------------------------------------
unsigned int time0 = 0;         //Time for lack of movement
unsigned int time1 = 0;         //Current time
unsigned int time2 = 0;         //Time for spin search
unsigned int timedif = 5000;   //Time difference to check for movement
unsigned int timedif2 = 12000; //Time difference for spin search
unsigned int mindist = 2;       //Minimum distance robot must travel within timedif
bool ignoretime = 0;            //If a routine that clears distance is called, this variable is set high to avoid backing
    up afterwards

//-----------------------------------------------------------------
//Flame Sensor Variables
//-----------------------------------------------------------------
int flamex[4];                  //X-coordinates
int flamey[4];                  //Y-coordinates
int IRPosx[4];                  //Single X-coordinates
int IRPosy[4];                  //Single Y-coordinates
int IRAlert[4];        //Detects Flame == 1
int IRsensorAddress = 0xB0;
int slaveAddress;

Servo FlameServo;               //Servo
float servoPos = 90;            //Degrees of servo rotation
byte IRCam_Dir = UP;            //Direction of servo
int IRCam_Window = 120;    //Window to keep candle within
```

```
//Filters for IR cam data
RunningMedian IRx_Filt0 = RunningMedian(IRCam_FiltSize);
RunningMedian IRx_Filt1 = RunningMedian(IRCam_FiltSize);
RunningMedian IRx_Filt2 = RunningMedian(IRCam_FiltSize);
RunningMedian IRx_Filt3 = RunningMedian(IRCam_FiltSize);

RunningMedian IRy_Filt0 = RunningMedian(IRCam_FiltSize);
RunningMedian IRy_Filt1 = RunningMedian(IRCam_FiltSize);
RunningMedian IRy_Filt2 = RunningMedian(IRCam_FiltSize);
RunningMedian IRy_Filt3 = RunningMedian(IRCam_FiltSize);

//-----------------------------------------------------------------
//Ultrasonic Sensor Variables
//-----------------------------------------------------------------
float USDist[NumUS];            //Distance to nearest object
bool USAlert[NumUS];            //Object within threshold
float ObjDistThresh = 20;   //Distance before USAlert goes high

//Filters for US sensors
RunningMedian US_Filt0 = RunningMedian(US_FiltSize);
RunningMedian US_Filt1 = RunningMedian(US_FiltSize);
RunningMedian US_Filt2 = RunningMedian(US_FiltSize);
RunningMedian US_Filt3 = RunningMedian(US_FiltSize);
RunningMedian US_Filt4 = RunningMedian(US_FiltSize);

//-----------------------------------------------------------------
//Line Sensor
//-----------------------------------------------------------------
int Line_1 = 0;         //ADC value Line sensor 1
int Line_2 = 0;         //ADC value Line sensor 2
int Line_1_Thresh = 0;   //Nominal carpet value as read by sensor 1
int Line_2_Thresh = 0;   //Nominal carpet value as read by sensor 2
int LineDif1 = 50;    //Difference between carpet and white line
int LineDif2 = 50;    //Difference between carpet and white line
int LineAlert[2] = {0, 0};  //Alert when white line is detected

//Filters for line sensors
RunningMedian Line_1_Filt = RunningMedian(Line_FiltSize);
RunningMedian Line_2_Filt = RunningMedian(Line_FiltSize);

//-----------------------------------------------------------------
//PROGRAM
//-----------------------------------------------------------------

void setup() {
  Serial.begin(9600);

  //Initialize Pins
  pinMode(whitePin, INPUT);
  pinMode(yellowPin, INPUT);

  pinMode(US0_EchoPin, INPUT);
  pinMode(US0_TrigPin, OUTPUT);
  pinMode(US1_EchoPin, INPUT);
  pinMode(US1_TrigPin, OUTPUT);
  pinMode(US2_EchoPin, INPUT);
  pinMode(US2_TrigPin, OUTPUT);
  pinMode(US3_EchoPin, INPUT);
  pinMode(US3_TrigPin, OUTPUT);
  pinMode(US4_EchoPin, INPUT);
  pinMode(US4_TrigPin, OUTPUT);

  pinMode(dist0, INPUT);
  pinMode(dist1, INPUT);
  pinMode(dist2, INPUT);
  pinMode(dist3, INPUT);
  pinMode(dist4, INPUT);
  pinMode(dist5, INPUT);
  pinMode(dist6, INPUT);
  pinMode(dist7, INPUT);
  pinMode(dist_reset, OUTPUT); ResetDist();

  pinMode(fanPin, OUTPUT); digitalWrite(fanPin, LOW);
  pinMode(calibrationPin, INPUT);
  pinMode(ledPin, OUTPUT); digitalWrite(ledPin, LOW);

  //Initialize Flame Sensor
  slaveAddress = IRsensorAddress >> 1;   // This results in 0x21 as the address to pass to TWI
  IRCam_Init();                    //Initialize IR camera
  FlameServo.attach(servoPin);   //Attach Servo
  FlameServo.write(servoPos);            //Set initial position to forward

  delay(1000);
  CalibrateTurn(140);              //Calibration routine for turning if calibrate button is pressed
  Line_Initialize();               //Calibration routine for line sensors

//Read data to help initialize filters
  for (int i = 0; i < US_FiltSize; i++) {
      Line_ReadSensors();// Serial.println("Done Line");
      US_ReadSensors();// Serial.println("Done Ultrasonics");
  }
}

void loop() {

  //Get data from all sensors
  Line_ReadSensors();
  US_ReadSensors();
  IRCam_GetData();

  IRCam_TurnDirection(LEFT);  //Ensure camera is pointed to the left
  digitalWrite(ledPin, LOW);  //Turn off on-board LED

  //If flame was detected, search for it
  if (IRAlert[0] == DETECTED) {
      Serial.println("Seeking Flame");
      SeekFlame(175);
  }

  //Avoid object within x centimeters
  AvoidObject(5);

  time1 = millis();  //Get current time
  //Timeout Routine for checking for minimun distance travelled
  if (((time1 - time0) > timedif)&&(ignoretime == 0)) {
      time0 = time1;
      ignoretime = 0;
      if (totdist < mindist) {
      Reverse(240, 3, 0, 0);
      RTurn(240, 20);
      }
      totdist = 0; totrot = {0, 0}; ResetDist();
  }

  //Timeout routine for spin searching for flame
  if ((time1 - time2) > timedif2) {
      time2 = time1;
      RSpinSearch(220, 360);
  }

  //Change directions if run forward into object or hit white line
  if ((USAlert[2] == DETECTED) || (LineAlert[0] == DETECTED) || (LineAlert[1] == DETECTED)) {
      Stop(); Reverse(240, 3, 0, 0);  //Back up slightly

      //LED goes high if line sensor goes off for debugging
      if (USAlert[2] == DETECTED) {
      digitalWrite(ledPin, LOW);
      }
      if (LineAlert[0] == DETECTED) {
      digitalWrite(ledPin, HIGH);
      }
      if (LineAlert[1] == DETECTED) {
      digitalWrite(ledPin, HIGH);
      }

      //Turn left between 0 and 90 degreees
      int newDir = random(0, 90);
      LSpinSearch(240, newDir); Stop();

      //Refresh distance data
      for (int i = 0; i < US_FiltSize; i++) {
      US_ReadSensors();
      }
  }

  //If no candles or objects are seen, go forward
  else {
      Forward(175, -1, 0, 0);
  }
}

//Displays data from all sensors
void DisplayData(){
  Serial.print("US0 Dist:  "); Serial.print(USDist[0]);
  Serial.print("   US0 Alert: "); Serial.println(USAlert[0]);
  Serial.print("US1 Dist:  "); Serial.print(USDist[1]);
  Serial.print("   US1 Alert: "); Serial.println(USAlert[1]);
  Serial.print("US2 Dist:  "); Serial.print(USDist[2]);
  Serial.print("   US2 Alert: "); Serial.println(USAlert[2]);
  Serial.print("US3 Dist:  "); Serial.print(USDist[3]);
  Serial.print("   US3 Alert: "); Serial.println(USAlert[3]);
  Serial.print("US4 Dist:  "); Serial.print(USDist[4]);
  Serial.print("   US4 Alert: "); Serial.println(USAlert[4]);
  Serial.print("Flamex: "); Serial.println(flamex[0]);
  Serial.print("IR Alert: "); Serial.println(IRAlert[0]);
  Serial.print("Line 1: "); Serial.print(Line_1); Serial.print("    Line 2: "); Serial.println(Line_2);
  Serial.println(" ");
  delay(100);
}

void SeekFlame(int dutcyc) {
  int loff = 0;
  int roff = 0;
  int maxoffset = 200;
  int lturn = 0;
  int rturn = 0;
  int timeout = 0;
  int timethresh = 500;
  bool breakvar = NOTHING;
  Stop();
  //Record orignal location
  RecordLocation(coord);

  //Keep searching for flames while ultrasonic sensors, line sensors, and IR sensosrs detect nothing
   while ((((LineAlert[1] == NOTHING) && (LineAlert[2] == NOTHING)) || (IRAlert[0] == NOTHING) || (USAlert[2] ==
      NOTHING)) && (breakvar == NOTHING)) {
      rturn = 0;
      lturn = 0;
      breakvar = NOTHING;
      time1 = millis();

      //Timeout for backing up
      if (((time1 - time0) > timedif)&&(ignoreTime == 1)) {
      time0 = time1;
      ignoreTime = 0;
      if (totdist < mindist) {
      Reverse(240, 3, 0, 0);
      LTurn(240, 20);
      }
      totdist = 0; totrot = {0, 0}; ResetDist();
      }
      IRCam_GetData();
      US_ReadSensors();
      Line_ReadSensors();

      //Algorithm for searching for flames based upon direction of camera when it is detected
      switch (IRCam_Dir) {
      //If up, keep the flame in the center of the field. While it is, go forward. When it exits the center, correct by
      turning until it is back in the center
      case (UP):
      if (flamex[0] < -IRCam_Window) {
      Stop();
      while ((flamex[0] < 0) && (breakvar == NOTHING)) {
            IRCam_GetData();
            LTurn(140, 1);
            lturn++;
```

```
        totrot = {2, 2};
        totdist = mindist + 1;
        if (lturn > 400) {
        breakvar = DETECTED;
        time2 = millis();
        time0 = time2;
        }
}
}
else if (flamex[0] > IRCam_Window) {
Stop();
while ((flamex[0] > 0) && (breakvar == NOTHING)) {
        IRCam_GetData();
        RTurn(140, 1);
        rturn++;
        totrot = {2, 2};
        totdist = mindist + 1;
        if (rturn > 400) {
        breakvar = DETECTED;
        time2 = millis();
        time0 = time2;
        }
}
//roff += 3;
}
else {
loff = 0;
roff = 0;
} break;

//If it is detected, go forward until it is in the center of the field of view
case (LEFT):
while (flamex[0] > IRCam_Window) {
IRCam_GetData();
Forward(dutcyc, -1, 0, 0);
}
Stop();
IRCam_Dir = UP;
servoPos = IRCam_Dir * 90;
FlameServo.write(servoPos);
LTurn(200, 90);
break;

//If it is detected, go forward until it is in the center of the field of view
case (RIGHT):
while (flamex[0] < -IRCam_Window) {
IRCam_GetData();
Forward(dutcyc, -1, 0, 0);
}
Stop();
IRCam_Dir = UP;
servoPos = IRCam_Dir * 90;
FlameServo.write(servoPos);
RTurn(200, 90);
break;
}
```

```
      //Ensures Limits are imposed on offsets
      if (loff > maxoffset) {
      loff = maxoffset;
      }
      if (loff < 0) {
      loff = 0;
      }
      if (roff > maxoffset) {
      roff = maxoffset;
      }
      if (roff < 0) {
      roff = 0;
      }

      //Check for objects to avoid
      AvoidFlameObject(5);
      Forward(dutcyc, -1, loff, roff);
  }

  //When all three conditions are met, turn on fan
  Stop();
  PulseFan();
  //RTurn(240,lturn);
  //LTurn(240,rturn);
  //ReturnOrigLoc();

  //Back up and record data
  Reverse(240, 3, 0, 0); Stop(); ResetDist();
  for (int i = 0; i < US_FiltSize; i++) {
      Line_ReadSensors();
      US_ReadSensors();
  }
}
```

11.2 Navigation

```
//Converts wheel rotations and edges to physical distances
//Note, the LSB is the 3rd flip flop on the Ripple counter
//Total edges per wheen rotation: 232
//Input rot[0] - number of wheel rotations
//Input rot[1] - number of edges detected
float GetDistance(int rot[2]) {
  byte bitrot = 0;

  //Reads ripple counter and converts to an 8-bit number
  for (int i = 0; i < 8; i++) {
      bitrot <<= 1;
      bitrot |= digitalRead(dist7 + i * 2);
  }

  rot[1] = bitrot;  //Record number of edges on ripple counter

  //If edges are greater than 232, increment wheel counter and subtract 232 from edge count
  if (rot[1] >= 1856 / (4 * 2)) {
      rot[0]++; rot[1] = bitrot % (1856 / (4 * 2));
      ResetDist();
  }

  //Return physical distance
  return (((float)rot[1] / (1856.0 / (4.0 * 2.0)) + (float)rot[0])) * wheelDiam * 3.14159;
}

//Turns left while searching for a flame
//Input rotspd - speed of turn (max 255)
//Input deg - degree of turn (min 0)
void LSpinSearch(int rotspd, int deg) {
  int SSrot[2] = {0, 0};  //tracks rotation of wheel and edges from encoders
  float dist = 0;         //Distance tracking
  bool breakvar = NOTHING;  //Breaks while loop when high

  IRCam_TurnDirection(UP);  //Turn IRCamera up
  ResetDist();          //Reset distance count on ripple counter

  //Set speed of motors to turn
  analogWrite(lrmotor, rotspd);
  analogWrite(lfmotor, 0);
  analogWrite(rfmotor, rotspd);
  analogWrite(rrmotor, 0);

  //Turn until degree is reached or flame is detected
  while ((dist < (((float)deg) / 360.0 * (rdist))) && (breakvar == NOTHING)) {
      IRCam_GetData();
      dist = GetDistance(SSrot);
      if (IRAlert[0] == DETECTED) {
      Stop();
      SeekFlame(175);
      breakvar = DETECTED;
      }
  }
  Stop();

  //Updates current direction for grid naviagation
  if (curdir + deg / 90 < 0) {
      curdir = (curdir + deg / 90) % 4 + 4;
  }
  else {
      curdir = (curdir + deg / 90) % 4;
  }

  ResetDist();          //Reset ripple counter
  Line_ReadSensors(); //Read line sensors
  ignoretime = 1;     //Ignores immobile timeout
}

//Turns Right while searching for a flame
//Input rotspd - speed of turn (max 255)
//Input deg - degree of turn (min 0)
void RSpinSearch(int rotspd, int deg) {
  int SSrot[2] = {0, 0};  //tracks rotation of wheel and edges from encoders
  float dist = 0;         //Distance tracking
  bool breakvar = NOTHING;  //Breaks while loop when high

  IRCam_TurnDirection(UP);  //Turn IRCamera up
  ResetDist();          //Reset distance count on ripple counter

  //Set speed of motors to turn
  analogWrite(lfmotor, rotspd);
  analogWrite(lrmotor, 0);
  analogWrite(rrmotor, rotspd);
  analogWrite(rfmotor, 0);

  //Turn until degree is reached or flame is detected
  while ((dist < (((float)deg) / 360.0 * (rdist))) && (breakvar == NOTHING)) {
      IRCam_GetData();
      dist = GetDistance(SSrot);
      if (IRAlert[0] == DETECTED) {
      Stop();
      SeekFlame(175);
      breakvar = DETECTED;
      }
  }
  Stop();

  //Updates current direction for grid naviagation
  if (curdir + deg / 90 < 0) {
      curdir = (curdir + deg / 90) % 4 + 4;
  }
  else {
      curdir = (curdir + deg / 90) % 4;
  }

  ResetDist();          //Reset ripple counter
  Line_ReadSensors(); //Read line sensors
  ignoretime = 1;     //Ignores immobile timeout
}

//Calibrtates turn data to use. Press calibration button, wait until robot has turned 360 degrees then press again
void CalibrateTurn(int rotspd) {
  bool cal = digitalRead(calibrationPin);
  if (cal == DETECTED) {
      //Acknowledge Command
      delay(1000);

      //Wait until button is released to begin calibrating
      while (cal == DETECTED) {
      delay(10);
      cal = digitalRead(calibrationPin);
      }

      //Right Turn algorithm
      int RTrot[2] = {0, 0};
      float dist = 0;

      analogWrite(lfmotor, rotspd);
      analogWrite(lrmotor, 0);
      analogWrite(rrmotor, rotspd);
      analogWrite(rfmotor, 0);
      while (cal == NOTHING) {
      cal = digitalRead(calibrationPin);
      dist = GetDistance(RTrot);
      }

      //Record distance travelled for 360 degrees
      Stop();
      rdist = dist;
      ResetDist();
      delay(1000);

      //Left Turn
      cal = digitalRead(calibrationPin);

      //wait until Button is released to turn again
      while (cal == DETECTED) {
      delay(10);
      cal = digitalRead(calibrationPin);
      }
      int LTrot[2] = {0, 0};
      dist = 0;

      analogWrite(lrmotor, rotspd);
      analogWrite(lfmotor, 0);
      analogWrite(rfmotor, rotspd);
      analogWrite(rrmotor, 0);
      while (cal == NOTHING) {
      cal = digitalRead(calibrationPin);
      dist = GetDistance(LTrot);
      }
      Stop();
      ldist = dist;
      ResetDist();
  }

  //If button is not pressed, use expected distances
  else {
      rdist = (360.0 / 360.0 * (3.14159 * roboDiam));
      ldist = (360.0 / 360.0 * (3.14159 * roboDiam));
  }
}

//Make robot go forward
//Input dutcyc - Speed of travel (max 255)
//Input goaldist - Distance to travel forward (inches) (input -1 to go forward unspecified amount)
//Input loff - offset for left motor
//Input roff - offset for right motor
void Forward(int dutcyc, float goaldist, int loff, int roff) {
  int FTrot[2] = {0, 0};
  float dist = 0;
  bool motoroff = 0;
  int lspd = dutcyc - loff;
  int rspd = dutcyc - roff - minwind;
  int cnt = 0;
  int lcnt = 0;

  //Imposes limits on motor offsets
  if (lspd > 255) {
      lspd = 255;
  }
  if (lspd < 0) {
      lspd = 0;
  }
  if (rspd > 255) {
      rspd = 255;
  }
  if (rspd < 0) {
      rspd = 0;
  }

  analogWrite(lfmotor, lspd);
  analogWrite(rfmotor, rspd);
  analogWrite(lrmotor, 0);
  analogWrite(rrmotor, 0);

  //Go forware
  if (goaldist != -1) {
      while (dist < goaldist) {
      dist = GetDistance(FTrot);
      IRCam_GetData();
      if (IRAlert[0] == DETECTED) {
      SeekFlame(175);
      }
      }
      UpdateCoordinates(dist, curdir);
      ResetDist();
  }
  else {
      totdist = GetDistance(totrot);
  }
}

//Make robot go backwards
//Input dutcyc - Speed of travel (max 255)
//Input goaldist - Distance to travel forward (inches) (input -1 to go forward unspecified amount)
//Input loff - offset for left motor
//Input roff - offset for right motor
void Reverse(int dutcyc, int goaldist, int loff, int roff) {
  int FTrot[2] = {0, 0};
  float dist = 0;
  int lspd = dutcyc - loff;
  int rspd = dutcyc - roff - minwind;
  ResetDist();
  if (lspd > 255) {
      lspd = 255;
  }
  if (lspd < 0) {
      lspd = 0;
  }
  if (rspd > 255) {
      rspd = 255;
  }
  if (rspd < 0) {
      rspd = 0;
  }

  //  Serial.print("Lspd: "); Serial.println(lspd);
  //  Serial.print("Rspd: "); Serial.println(rspd);
  //  delay(50);

  analogWrite(lrmotor, lspd);
  analogWrite(rrmotor, rspd);
  analogWrite(lfmotor, 0);
  analogWrite(rfmotor, 0);
  if (goaldist != -1) {
      while (dist < goaldist) {
      dist = GetDistance(FTrot);
      IRCam_GetData();
      //if(IRAlert[0] == DETECTED){SeekFlame(230);}
      }
      UpdateCoordinates(dist, curdir);
      //Serial.print("Coordinates: "); Serial.print(coord[0]); Serial.print(","); Serial.println(coord[1]);
      ResetDist();
  }
  else {
      //totdist = GetDistance(totrot);
      //if(IRAlert[0] == DETECTED){SeekFlame(230);}
  }
  ResetDist();
}

//Makes robot turn right deg degrees at rotspd speed
//Input rotspd - Speed of turn
//Input deg - degree of turn
void RTurn(int rotspd, int deg) {
  int RTrot[2] = {0, 0};
  float dist = 0;
  ResetDist();
  analogWrite(lfmotor, rotspd);
  analogWrite(lrmotor, 0);
  analogWrite(rrmotor, rotspd);
  analogWrite(rfmotor, 0);

  //Keep turning until distance exceeds deg
  while (dist < (((float)deg) / 360.0 * (rdist))) {
      dist = GetDistance(RTrot);
  }
  Stop();

  //Checks for negative numbers
  if (curdir + deg / 90 < 0) {
      curdir = (curdir + deg / 90) % 4 + 4;
  }
  else {
      curdir = (curdir + deg / 90) % 4;
  }
  ResetDist();
}

//Makes robot turn left deg degrees at rotspd speed
//Input rotspd - Speed of turn
//Input deg - degree of turn
void LTurn(int rotspd, int deg) {
  int LTrot[2] = {0, 0};
  float dist = 0;
  ResetDist();
  analogWrite(lrmotor, rotspd);
  analogWrite(lfmotor, 0);
  analogWrite(rfmotor, rotspd);
  analogWrite(rrmotor, 0);
  //Serial.print("Left: "); Serial.println(deg);
  while (dist < (((float)deg) / 360.0 * (ldist))) {
        //CheckRotState(LTrot);
        dist = GetDistance(LTrot);
        //      Serial.print(LTrot[0]); Serial.println(LTrot[1]);
        //Serial.println(dist);
  }
  //Serial.print("Ldist: ");Serial.println(dist);
  //  Serial.print("Final Dist: "); Serial.println(dist);
  Stop();
  //Serial.print("Test: "); Serial.println( deg/90);
  if (curdir - deg / 90 < 0) {
        curdir = (curdir - deg / 90) % 4 + 4;
  }
  else {
        curdir = (curdir - deg / 90) % 4;
  }

  ResetDist();
  //Serial.print("Current Dir: "); Serial.println(curdir);
}

//Stops robot dead in its tracks
void Stop() {
  analogWrite(lrmotor, 0);
  analogWrite(lfmotor, 0);
  analogWrite(rfmotor, 0);
  analogWrite(rrmotor, 0);

  //Update distance data
  totdist = GetDistance(totrot);
  UpdateCoordinates(totdist, curdir);
  totdist = 0;
  totrot = {0, 0};
  ResetDist();
}

//Return to the original location recorded with RecordLocation function
void ReturnOrigLoc() {
  int xoff = coord[0] - origLoc[0];  //Distances to travel in the x direction
  int yoff = coord[1] - origLoc[1];  //Distances to travel in the y direction

  //Correct in the x direction
  if (xoff > 0) {
        TurnDirection(LEFT);
        Forward(240, abs(xoff), 0, 0);
  }
  else if (xoff < 0) {
        TurnDirection(RIGHT);
        Forward(240, abs(xoff), 0, 0);
  }
  Stop();

  //Correct in the y direction
  if (yoff > 0) {
       TurnDirection(DOWN);
       Forward(240, abs(yoff), 0, 0);
  }
  else if (yoff < 0) {
       TurnDirection(UP);
       Forward(240, abs(yoff), 0, 0);
  }
  Stop();
  TurnDirection(origdir);
  servoPos = (180 - origIRCamdir * 90);
  FlameServo.write(servoPos);
  IRCam_Dir = origIRCamdir;
  origLoc = { -1, -1};
}

//Turn robot to face goaldir
void TurnDirection(int goaldir) {
  int tempdirl = curdir;
  int rcnt = 0;
  int lcnt = 0;
  int turndif = abs(curdir - goaldir);

  //Check distance to turn left
  while (tempdirl != goaldir) {
       if (tempdirl - 1 < 0) {
       tempdirl = (tempdirl - 1) % 4 + 4;
       }
       else {
       tempdirl = (tempdirl - 1) % 4;
       }
       lcnt++;
  }
  tempdirl = curdir;

  //Check distance to turn right
  while (tempdirl != goaldir) {
       if (tempdirl + 1 < 0) {
       tempdirl = (tempdirl + 1) % 4 + 4;
       }
       else {
       tempdirl = (tempdirl + 1) % 4;
       }
       rcnt++;
  }

  //Turn whichever direction takes less turns
  if (rcnt < lcnt) {
       RTurn(240, rcnt * 90);
  }
  else {
       LTurn(240, lcnt * 90);
  }
}
```

```
//Reset the ripple counter
void ResetDist() {
  digitalWrite(dist_reset, HIGH);
  delayMicroseconds(50);
  digitalWrite(dist_reset, LOW);
}
```

11.3 IR Camera Code

```
void Write_2bytes(byte d1, byte d2)
{
  Wire.beginTransmission(slaveAddress);
  Wire.write(d1); Wire.write(d2);
  Wire.endTransmission();
}

void IRCam_Init() {
  Wire.begin();
  Write_2bytes(0x30, 0x01); delay(10);
  Write_2bytes(0x30, 0x08); delay(10);
  Write_2bytes(0x06, 0x90); delay(10);
  Write_2bytes(0x08, 0xC0); delay(10);
  Write_2bytes(0x1A, 0x40); delay(10);
  Write_2bytes(0x33, 0x33); delay(10);
}

void IRCam_GetData() {
  byte data_buf[16];
  int i;

  int Ix0, Iy0, Ix1, Iy1;
  int Ix2, Iy2, Ix3, Iy3;
  int s;
  //IR sensor read
  Wire.beginTransmission(slaveAddress);
  Wire.write(0x36);
  Wire.endTransmission();

  Wire.requestFrom(slaveAddress, 16);          // Request the 2 byte heading (MSB comes first)
  for (i = 0; i < 16; i++) {
        data_buf[i] = 0;
  }
  i = 0;
  while (Wire.available() && i < 16) {
        data_buf[i] = Wire.read();
        i++;
  }

  Ix0 = data_buf[1];
  Iy0 = data_buf[2];
  s   = data_buf[3];
  Ix0 += (s & 0x30) << 4;
  Iy0 += (s & 0xC0) << 2;

  Ix1 = data_buf[4];
  Iy1 = data_buf[5];
  s   = data_buf[6];
  Ix1 += (s & 0x30) << 4;
  Iy1 += (s & 0xC0) << 2;

  Ix2 = data_buf[7];
  Iy2 = data_buf[8];
  s   = data_buf[9];
  Ix2 += (s & 0x30) << 4;
  Iy2 += (s & 0xC0) << 2;

  Ix3 = data_buf[10];
  Iy3 = data_buf[11];
  s   = data_buf[12];
  Ix3 += (s & 0x30) << 4;
  Iy3 += (s & 0xC0) << 2;

  IRx_Filt0.add(Ix0 - 512);
  IRx_Filt1.add(Ix1 - 512);
  IRx_Filt2.add(Ix2 - 512);
  IRx_Filt3.add(Ix3 - 512);

  IRy_Filt0.add(Iy0 - 512);
  IRy_Filt1.add(Iy1 - 512);
  IRy_Filt2.add(Iy2 - 512);
  IRy_Filt3.add(Iy3 - 512);

  //Put data through filters
  flamex[0] = (int)IRx_Filt0.getMedian();
  flamex[1] = (int)IRx_Filt1.getMedian();
  flamex[2] = (int)IRx_Filt2.getMedian();
  flamex[3] = (int)IRx_Filt3.getMedian();

  flamey[0] = (int)IRy_Filt0.getMedian();
  flamey[1] = (int)IRy_Filt1.getMedian();
  flamey[2] = (int)IRy_Filt2.getMedian();
  flamey[3] = (int)IRy_Filt3.getMedian();

  for (int i = 0; i < 4; i++) {
      if (flamex[i] >= 510) {
      IRAlert[i] = NOTHING;
      }
      else {
      IRAlert[i] = DETECTED;
      }
  }
}

//Update coordinates based on dist distance traveled in dir direction
//Input dist - Distance traveled
//Input dir - Direction that distance was traveled
void UpdateCoordinates(float dist, int dir) {
  dist = round(dist);
  switch (curdir) {
      case (UP): coord[1] = coord[1] + dist; break;
      case (RIGHT): coord[0] = coord[0] + dist; break;
      case (DOWN): coord[1] = coord[1] - dist; break;
```

```
        case (LEFT): coord[0] = coord[0] - dist; break;
  }
}

//Save the location to return to later
//Input location - x,y coordinates that robot is currently at
void RecordLocation(int location[2]) {
  origLoc[0] = coord[0];
  origLoc[1] = coord[1];
  origdir = curdir;
  origIRCamdir = IRCam_Dir;
}

//Turn IRCam to face direction
//Input goaldir - direction to face camera
void IRCam_TurnDirection(int goaldir) {
  servoPos = (180 - goaldir * 90);
  FlameServo.write(servoPos);
  IRCam_Dir = goaldir;
}

//Turn on fan and correct direction until flame is extinguished
void PulseFan() {
  int twocand = 0;
  int totr = 0;
  int totl = 0;
  int swivWind = 50;

  //Checks for two candles
  if (IRAlert[1] == DETECTED) {
        twocand = 1;
  }

  //While flame is detected and the candle is within distance, try to extinguish flame
  while ((IRAlert[0] == DETECTED) && (USAlert[2] == DETECTED)) {

        //Turn on fan for a second
        digitalWrite(fanPin, HIGH);
        delay(1000);
        digitalWrite(fanPin, LOW);
        delay(1000);

        //Update sensor data
        for (int i = 0; i < IRCam_FiltSize; i++) {
        IRCam_GetData();
        US_ReadSensors();
        }

        //If two candles were detected, exit pulse
        if ((twocand == 1) && (IRAlert[1] == NOTHING)) {
        break;
        }

        //Correct direction if candle is slightly offset
        else if (IRAlert[0] == DETECTED) {
        if (flamex[0] < -swivWind) {
        LTurn(240, 10);
        totl++;
        }
        else if (flamex[0] > swivWind) {
        RTurn(240, 10);
        totr++;
        }
        }
  }
  //  RTurn(240,10*totr);
  //  LTurn(240,10*totl);
}
```

11.4 Line Sensor Code

```
//Initialize and calibrate line sensors
void Line_Initialize() {
  int line1 = 0;
  int line2 = 0;
  RunningMedian Line_1_FiltT = RunningMedian(20);
  RunningMedian Line_2_FiltT = RunningMedian(20);

  //Read 20 values for the median filter
  for (int i = 0; i < 20 * 2; i++) {
        line1 = analogRead(line1Pin);
        line2 = analogRead(line2Pin);

        Line_1_FiltT.add(line1);
        Line_2_FiltT.add(line2);
        delay(10);
  }

  //Select median value of collected data
  Line_1_Thresh = Line_1_FiltT.getMedian();
  Line_2_Thresh = Line_2_FiltT.getMedian();
}

//Reads in data from the ADC
//Enable commented lines to use median filter
void Line_ReadSensors() {
  Line_1 = analogRead(line1Pin);
  Line_2 = analogRead(line2Pin);

  //  int line1 = analogRead(line1Pin);
  //  int line2 = analogRead(line2Pin);
  //
  //  Line_1_Filt.add(line1);
  //  Line_2_Filt.add(line2);
  //
  //  Line_1 = Line_1_Filt.getMedian();
  //  Line_2 = Line_2_Filt.getMedian();

  //Sets alerts to high if exceeds threshold
  if (Line_1 > Line_1_Thresh + LineDif1) {
        LineAlert[0] = DETECTED;
  }
  else {
      LineAlert[0] = NOTHING;
  }
  if (Line_2 > Line_2_Thresh + LineDif2) {
      LineAlert[1] = DETECTED;
  }
  else {
      LineAlert[1] = NOTHING;
  }
}
```

11.5 Object Avoidance Routines

```
//Object Avoidance Routine when following a candle
//Input sThresh - activeate routine if object is below this threshold (cm)
void AvoidFlameObject(float sThresh) {
  int turnnum = random(0, 90);  //Degrees to turn
  US_ReadSensors();  //Update sensors

  //Right Sensor
  //Move slightly left then rescan for candle
  if ((USDist[0] < sThresh) && (USDist[0] != -1)) {
      Stop(); Reverse(240, 1, 0, 0);
      LTurn(240, 10); Stop();
      Forward(240, 2, 0, 0); Stop();
      RSpinSearch(240, 30); Stop();
      ignoretime = 1;
  }

  //Right-Middle Sensor
  //Move Left 6" then rescan for candle
  else if ((USDist[1] < sThresh) && (USDist[1] != -1)) {
      Stop(); Reverse(240, 1, 0, 0);
      LTurn(240, 90); Stop();
      Forward(240, 6, 0, 0); Stop();
      RSpinSearch(240, 180); Stop();
      LTurn(240, 90); Stop();
      ignoretime = 1;
  }

  //Left-Middle Sensor
  //Move Right 6" then rescan for candle
  else if ((USDist[3] < sThresh) && (USDist[3] != -1)) {
      Stop(); Reverse(240, 1, 0, 0);
      RTurn(240, 90); Stop();
      Forward(240, 6, 0, 0); Stop();
      LSpinSearch(240, 180); Stop();
      RTurn(240, 90); Stop();
      ignoretime = 1;
  }

  //Left Sensor
  //Move right slightly then rescan for candle
  else if ((USDist[4] < sThresh) && (USDist[4] != -1)) {
      Stop(); Reverse(240, 1, 0, 0);
      RTurn(240, 10); Stop();
      Forward(240, 2, 0, 0); Stop();
      LSpinSearch(240, 30); Stop();
      ignoretime = 1;
  }

  //Update Sensor Data
  for (int i = 0; i < US_FiltSize; i++) {
      US_ReadSensors();
  }
}


//Normal Object Avoidance routine
//Input sThresh - activeate routine if object is below this threshold (cm)
void AvoidObject(float sThresh) {
  int turnnum = random(0, 90);
  US_ReadSensors();

  //Right Sensor
  //Move slightly left then continue in original direction
  if ((USDist[0] < sThresh) && (USDist[0] != -1)) {
      Stop();
      LTurn(240, 10); Stop();
      Forward(240, 4, 0, 0); Stop();
      RTurn(240, 10); Stop();
      ignoretime = 1;
  }

  //Right-Middle Sensor
  //Move Left 6" then continue in original direction
  else if ((USDist[1] < sThresh) && (USDist[1] != -1)) {
      Stop();
      LTurn(240, 90); Stop();
      Forward(240, 6, 0, 0); Stop();
      RTurn(240, 90); Stop();
      ignoretime = 1;
  }

  //Left-Middle Sensor
  //Move Right 6" then continue in original direction
  else if ((USDist[3] < sThresh) && (USDist[3] != -1)) {
      Stop();
      RTurn(240, 90); Stop();
      Forward(240, 6, 0, 0); Stop();
      LTurn(240, 90); Stop();
      ignoretime = 1;
  }

  //Left Sensor
  //Move right slightly then continue in original direction
  else if ((USDist[4] < sThresh) && (USDist[4] != -1)) {
      Stop();
      RTurn(240, 10); Stop();
      Forward(240, 2, 0, 0); Stop();
      LTurn(240, 10); Stop();
      ignoretime = 1;
  }

  //Update Sensors
  for (int i = 0; i < US_FiltSize; i++) {
      US_ReadSensors();
  }
}
```

11.6 Ultrasonic Sensor Code

```
void US_ReadSensors() {
  float USval[NumUS];
  float USmaximumRange = 140;
  float USminimumRange = 2;
  float IRminimumRange = 10;
  float IRmaximumRange = 140;

  //Reads all Ultrasonic Sensors
  USval[0] = Read_Range_US(US0_TrigPin, US0_EchoPin, USminimumRange, USmaximumRange);
  USval[1] = Read_Range_US(US1_TrigPin, US1_EchoPin, USminimumRange, USmaximumRange);
  USval[2] = Read_Range_US(US2_TrigPin, US2_EchoPin, USminimumRange, USmaximumRange);
  USval[3] = Read_Range_US(US3_TrigPin, US3_EchoPin, USminimumRange, USmaximumRange);
  USval[4] = Read_Range_US(US4_TrigPin, US4_EchoPin, USminimumRange, USmaximumRange);

  //Put data through filter
  US_Filt0.add(USval[0]);
  US_Filt1.add(USval[1]);
  US_Filt2.add(USval[2]);
  US_Filt3.add(USval[3]);
  US_Filt4.add(USval[4]);

  USDist[0] = US_Filt0.getMedian();
  USDist[1] = US_Filt1.getMedian();
  USDist[2] = US_Filt2.getMedian();
  USDist[3] = US_Filt3.getMedian();
  USDist[4] = US_Filt4.getMedian();

  //Set alerts to high if distance is less than threshold
  for (int i = 0; i < NumUS; i++) {
      if ((USDist[i] == -1) || (USDist[i] > ObjDistThresh)) {
      USAlert[i] = NOTHING;
      }
      else {
      USAlert[i] = DETECTED;
      }
  }
}

//Reads individual ultrasonic sensors
//Input trigPin - trigger pin for US sensor
//Input echoPin - echo pin for US sensor
//Input minimumRange - Minimum range able to be read from US sensor
//Input maximumRange - Maximum range able to be read from US sensor
float Read_Range_US(byte trigPin, byte echoPin, float minimumRange, float maximumRange) {
  float distance = -2;
  float duration = -2;

  digitalWrite(trigPin, LOW);
  delayMicroseconds(2);
  //Send Ultrasonic bursts
  digitalWrite(trigPin, HIGH);
  delayMicroseconds(25);

  digitalWrite(trigPin, LOW);

  //Read pulse from echo pin
  duration = pulseIn(echoPin, HIGH, 5000);

  //Calculate the distance (in cm) based on the speed of sound.
  distance = duration / 58.2;

  //If distance is less than the minimun range or greater than the maximum, return -1 as an error code
  if (distance >= maximumRange || distance <= minimumRange) {
       return -1;
  }

  else {
       return distance;
  }
}
```