Autonomous Firefighting Robot

University of Connecticut
Senior Design

Team Members:

Katherine Drogalis, Electrical Engineering
Zachariah Sutton, Electrical Engineering
Chutian Zhang, Engineering Physics

Faculty Advisor:

Professor John Ayers

Abstract

This project is in conjunction with the Trinity International Robot Contest, which took place April 2-3, 2016. It is a not-for-profit event that promotes innovation and creativity in the STEM field. The principles used in this design are such that enable our robot to be extended to a more robust system to be used to combat actual fires in residential or commercial settings. The main requirement of this project is to create a robot that is fully autonomous. This means that once the robot is started by the user, it navigates, searches for, and extinguishes the fire on its own, with no assistance or input from the user. In order to reach this goal, we made many critical decisions on motors, sensors, fire extinguishing mechanical parts and general design for our robot. Lastly, we wanted our robot to not only accomplish these requirements, but also be able to do so quickly and accurately.

Background & Rules

The aim of this project is to create a fully autonomous robot that can navigate a model home in search of fire, in the form of a burning candle, and then extinguish it. In the contest, the judge places the robot in the marked start location in the maze and presses the start button. The robot then listens for a 3.8 kHz frequency ±16%, signaling to begin. At the sound, the robot begins its autonomous search for the flame. These requirements are mandatory in order to compete in the competition. According to the contest rules, each robot is required to have a carrying handle, an LED indicating flame detection, a microphone, and a kill power plug. Finally, the robot must also accomplish the goal in the allotted time.

The arena that our robot must navigate is an 8*8 foot plywood square partitioned by walls to mimic rooms and hallways. The robot starts in the marked starting location in the maze and when prompted, navigates the maze to find and extinguish the fire. The flame that the robot must search for is in the form of a candle placed in a random room in the arena. We competed only in the first level of the contest, as this was the most resemblant of real-life scenarios. Since time was our project's main limitation, we didn't want to increase the scope of our project unless absolutely necessary.

In level one, there is a single arena with a set layout as shown in Figure 1. The walls of the rooms and hallways are smooth and painted white. We know the hallway and room locations

to a reasonable degree of accuracy. There is a 3-minute time frame allotted to complete this level.

The contest is separated into two categories, customized robots and unique robots. Customized robots are built from a kit while unique robots are designed completely by the user. Our robot was categorized as unique, as we selected individual components and designed an original system and structure.

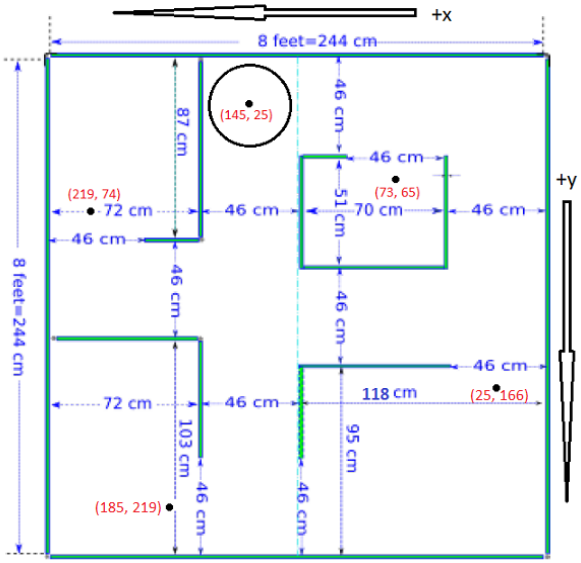


Figure 1. Level 1 Arena.

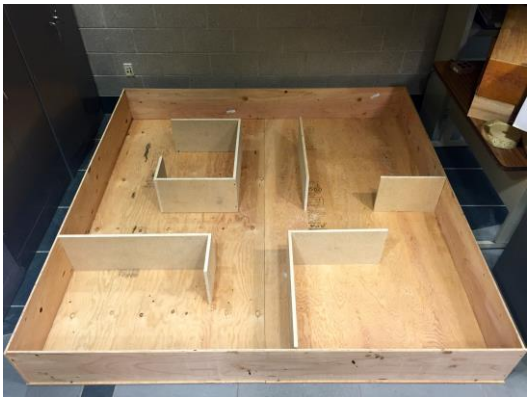


Figure 2. Test Arena.

Physical Design

The dimensional requirements for our robot are that it must not exceed 31*31*27 centimeters (Length*Width*Height). The material we chose for the platforms of the robot is polycarbonate "Lexan" sheet plastic due to its electrical insulating property, strength and resistance to cracking. It is also relatively cheap and easy to cut and fasten. The main shape of the platform is designed to be round in order to utilize the limited space. There are three main levels and a fourth sub-level of platform from the bottom to the top, which are all supported by threaded rod. The threaded rod gives us the ability to adjust the heights of our different levels if we decide to rearrange our components without having to spend the time or money to cut out a new design. The bottom level of our robot is where the two wheel and their driving motors are mounted. We also have two casters at the very bottom to help supporting the robot. Our power supply, which we chose to be rechargeable batteries, is located on the bottom level. The second

level holds the main range sensor (laser scanner) in the front. We discussed placing this sensor higher up on the robot, but ultimately decided to set it lower so that it doesn't overlook obstacles that are close to the ground. The required calculations for navigation are simplified having the center of the scanner is directly above the center of the line between the drive wheels. We also have our microcontrollers on either side of the laser scanner, a Raspberry Pi and an Arduino Mega. The third level holds the sensors and devices necessary for flame detection and extinguishing. These must also be positioned toward the front of the robot. The most important sensor arc to keep unobstructed is the 180 degrees in front of the robot. We are supporting the third level entirely from behind the scanner and have an arm that extends out to the front of the robot to ensure stability while also keeping the view of our scanner unobstructed. A smaller fourth level is positioned toward the back of the top of our robot that holds the start button, LED, microphone and kill-power plug (to be discussed in more detail later). We also have a handle built into the fourth level, as required by the contest, for easy lifting and maneuvering of our robot. Our core robot structure can be seen in Figure 3 and Figure 4 below, and the finalized part placement can be seen in Figure 5.

Figure 3. Robot structure front view.

Figure 4. Robot structure side view.

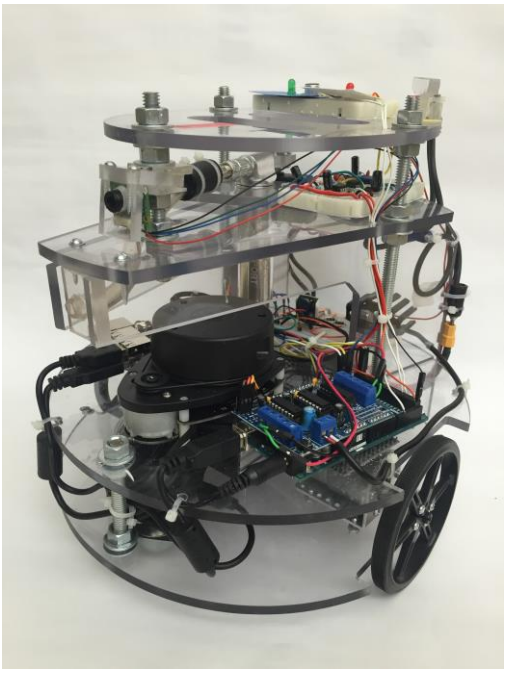


Figure 5. Finalized robot.

Technical solution

This is a complex system with many different parts that must function in unison. We can break our operations up into the following groups: navigation, which involves finding where the robot is relative to its surroundings and deciding where it should go next, movement, which involves the actuation of the robot, and search/execute, which involves looking for a flame and extinguishing once found. This last operation is ultimately the main function of the robot, but the other operations are essential to accomplishing it.

We are using an Arduino Mega for reading the flame sensor, controlling the drive motors, and reading encoder data. We have an H-bridge motor driver shield attached to the Arduino that handles generating a PWM signal for driving the motors. The Arduino is also used for running the status LEDs and start microphone discussed later.

We are using ROS (Robot Operating System) running on a Raspberry Pi 2 B to perform the navigation aspect of the project. The underlying principles of the navigation process are discussed below. The particular codes and ROS implementation of the process are discussed in the computation section later in this report.

Navigation

Navigation will involve two distinct, but connected operations: localization which is how the robot decides where it is, and path planning which is how the robot decides where to go.

Our main navigation sensor is a laser scanner that takes 2000 range readings every second. We have control over the scan rate so we can set the angular distance between consecutive readings. There are 64 bits of information for each range reading so the data rate from the scanner is 128 kbits/sec.

For navigating the arena environment, we considered implementing a Simultaneous Localization and Mapping (SLAM) algorithm. There are many online resources that describe possible implementations [1]. The main idea is to use features that can be sensed in the operating environment as reference points to check the estimate of the robot's motion and to create a map of the area. Since we are using a laser scanner it is relatively simple to detect corners and terminal points in walls. We can use these points as our landmarks.

SLAM is an attractive option since our robot would be able to perform in any random environment, however, we realized a few practical limitations in implementing it. First, the algorithm relies heavily on the ability to pick out landmarks in the environment and distinguish them from each other. We did not have a problem picking out the corners in walls to use as landmarks. However, distinguishing between them became an issue. In our particular operating environment there are a lot of walls. This means that we lose sight of many of our landmarks at any given time. This requires us to remember the locations of previously sensed landmarks so that we don't treat them as new ones the next time we see them. This proved to be theoretically challenging and practically even more so. Another disadvantage of SLAM is that it is inherently a complex operation. This works against us in two ways. First, we don't have a lot of time to finish programming the robot. Second, the contest is based on timed trials. If we want to have quick trial times, we don't want our on board computation to be too slow as this will limit the maximum speed we can run our robot at.

Because of these drawbacks, we decided to make use of all the information available to us regarding the layout of the contest arena. We know the exact layout of the level 1 arena which is never changing. We also decided to utilize the known start position for this level. We decided to make a single global map with this known information. This allows us to hardcode the map

into the robot's memory. We can then do away with the mapping aspect of SLAM and we have a somewhat simpler localization problem.

Localization

There are multiple methods for performing localization in a known map. One of the most flexible and adaptable seems to be a method called Monte Carlo Localization (MCL). The main advantage of this method is that it works fine in nonlinear operations. Our operation is nonlinear since we are rounding corners and our motion equations will contain sine and cosine functions.

MCL is based on an algorithm called a particle filter. Particle filters have more applications than just localization. For the specific purpose of localization we need a way to estimate the pose of the robot. The pose is the x- and y-position in the map along with the heading of the robot, so it can be thought of as a three dimensional space. We need to build a probabilistic distribution of the robot's pose. Ideally we should build the distribution in a way that allows for arbitrary features like multiple modes instead of making the assumption that it will always be Gaussian. A particle filter accomplishes this by building a distribution by first "guessing", or more accurately, sampling a proposal distribution. It then uses an observation to weight each sample. The weighted samples are then grouped into a new distribution which is then resampled. The algorithm runs recursively, updating and resampling each time new observation data is available. [1]

For our specific particle filter application of localization, our known global map is represented as a grid of 1*1 centimeter square cells. Since we have full knowledge of the layout of our global map, we have control over how it is programmed into the robot. We will indicate all cells that are walls, all cells that are inside rooms, and all cells that are part of hallways. Furthermore, we can define all the cells which are valid locations for the robot. Our robot is 28 cm in diameter. This means that any cells within 14 cm of any wall are invalid locations (relative to the center of our robot). We can even define cells in the middle of hallways as the most preferable positions. We can also set the cells that are in the center of doorways as points of interest since we need to search the rooms that the doorways lead to.

The robot's position will be modeled as being in the center of a cell. The (x,y) coordinates of the global grid map will be considered a global reference frame. The scanner measurements will be in body reference frame with (0,0) at the center of the robot, the x-axis

along the heading of the robot and the y-axis pointing to its left. The angle between the global x-axis and the body x-axis will be considered the robot's heading.

The samples that the particle filter takes will also be centered in cells. The initial distribution of the samples depends on the scenario. It is possible to uniformly distribute the samples over the entire map if the starting pose is unknown. The robot then needs to make initial random movements to allow the samples to converge to its location. These initial movements would waste time in a timed scenario. If the starting pose is known, a quicker option is to distribute the samples with a Gaussian distribution with the mean at the starting pose. The covariance of the distribution will determine how spread out from the mean the samples will be. The covariance can be adjusted depending on how certain we are of the exact starting pose. Once the robot begins moving the algorithm will work as follows:

1. Apply the motion given by the encoders to all of the current samples
2. Calculate what each sample should see based on where it is in the global map
3. Weight each sample based on how similar its observation is to the scanner observation
4. Build a new distribution from the weighted samples and resample from this new distribution

None of these operations are trivial and will all require significant computing power.

There are two main ways to represent the sample observations and scanner observation discussed in steps 2 and 3 above. The first is to express our map as an area with the wall corner and terminal positions indicated [2] [3]. To generate the sample observations we take the sample pose in the global map and determine the positions in the body reference frame of the corners that it would see. We will have to indicate in the logic which corners are connected by walls so that the sample pose won't be expected to "see through" the walls to corners on the other side. We can then process the 'r' component of the laser scanner data to find the positions of the corners in the scanner observation. Unlike the SLAM application, MCL doesn't require us to remember the locations of these corners, only to detect the local ones and compare their positions to those from the samples, making it a feasible option. This is the computationally simpler of the two methods and should be quicker so we plan on trying this first.

Another method is to have the samples and scanner build local grid maps. This method is also very effective with a laser scanner. The sample and scanner observations would be

represented as local grids in body reference frame with the value in the cells indicating the probability of that cell being occupied. For the scanner observation, the map is built by assigning a cell that corresponds to a laser range reading a high probability of being occupied. All of the cells between the center of the scanner and the reading are explicitly given a low probability of being occupied. All of the cells past the reading are given a probability of 0.5 [1] [3]. For the sample observations, we could pull "chunks" out of the global grid map which are translated and rotated according to each sample pose. These local grid maps could be thought of as images with probabilities instead of colors. So the scanner and sample observations could be compared using some image processing to find the most likely poses. This method is much more robust in terms of avoiding obstacles in the environment that are not included in the global map, but comes at the cost of being more computationally costly. We will use this method if the feature-based method doesn't perform well, but we will have to iterate the particle filter less often so overall performance should be roughly the same.

The fourth step in the algorithm, resampling, ends up being the most computationally costly. This is because building probability distributions is nontrivial. Fortunately, there are open source C++ scripts that do it for us. But it will be the time limiting factor in the computation speed.

Path Planning

Once the robot has a good estimate of its pose, it needs to decide where to go. Since we have full knowledge of the layout of the global map, we can define goal poses for the robot. In our particular case, these goal poses would be inside each room we need to search. A path can be defined on the same grid that is used for the localization. Given the robot's pose, a path can be found to a goal using a cost minimizing algorithm. This will result in the quickest path to the goal. Once this main path is generated, the robot needs to iteratively check the localization process for its pose relative to the main path and take measures to remain a close to the main path as possible.

Obstacle avoidance is also included in this task. If the laser picks up an unexpected obstacle blocking the main path, it needs to first take measures to avoid the obstacle and then replan the main path around the obstacle. If the obstacle is completely blocking a hallway, the new path must take a completely different way around the arena.

The output of this process is a velocity command for the center of the robot composed of a linear term and an angular term. This command is then translated to individual velocity commands for each drive wheel using the equations in the "movement" section below. These commands are sent to the Arduino which uses PID controller logic to ensure that the commands are executed accurately.

Movement

This is a differentially steered robot. There are 2 DC gear motors independently controlling 2 drive wheels. The rotational velocity of a DC motor is proportional to the voltage applied to it. We are supplying voltage to the motors with a Pulse-Width Modulation (PWM) signal from the Arduino. We have a motor control shield attached to the Arduino which amplifies the 5 V max signal from the Arduino to a 12 V max PWM signal to drive the motors.

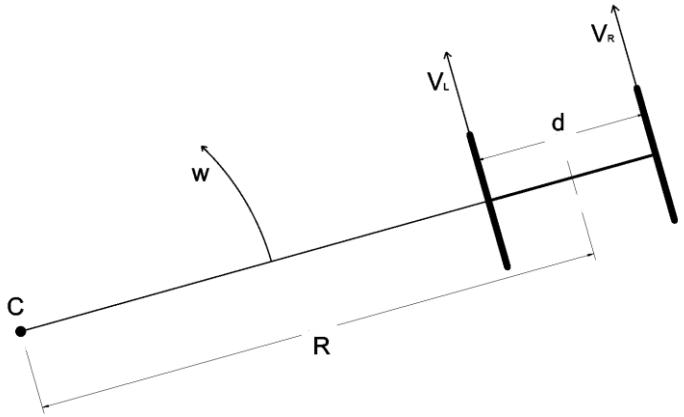


Figure 6. Model of the differential steering

The kinematics of differential steering are shown in Figure 6. The motion of the robot can be modeled as angular velocity around some instantaneous center of rotation. In Figure 4, $\mathbf{V_R}$ and $\mathbf{V_L}$ are the linear velocities of the right and left wheels, **R** is the radius around some instantaneous center **C**, and **ω** is the angular velocity around **C**. Figure 6 and the following equations illustrate this.

The velocity of the center of the robot **V** is given by the average velocity of the wheels or

$$V = \frac{V_R + V_L}{2} = \omega R$$

by the radius of rotation times the angular velocity. So:

The wheels must each have the same angular velocity around the center of rotation since they are

$$V_R = \omega\left(R + {}^{d}/_{2}\right), \quad V_L = \omega\left(R - {}^{d}/_{2}\right)$$

on a fixed axis so:

$$\omega = \frac{V_R - V_L}{d}, \quad R = \frac{d}{2}\left(\frac{V_R + V_L}{V_R - V_L}\right)$$

And:

This model allows for the extreme cases of straight line travel and pivoting around the center point.

1. If $\mathbf{V_R} = \mathbf{V_L}$, then we have forward linear motion in a straight line. R becomes infinite, and ω is zero.
2. If $\mathbf{V_R} = -\mathbf{V_L}$, then R = 0, and we have rotation about the center point of the robot.
3. If $\mathbf{V_L} = 0$, then we have rotation about the left wheel. In this case R = b. Similarly, if $\mathbf{V_R} = 0$, then R = -d and we have rotation about the right wheel.

In general, the output of the path planning operation is **V** and **ω**. The equations above are used to convert these values to $\mathbf{V_R}$ and $\mathbf{V_L}$ which the respective wheels are then driven to.

Computation

We have two computing devices for the project. First we have an Arduino Mega. Since they have unmatched I/O capabilities, Arduinos are popular for the uses of collecting sensor data and controlling motors and other actuation devices. We will program the Arduino to control the drive motors, read encoder data, read data from the thermopile (discussed later), and actuate the extinguishing device. It will also interface with the devices like start buttons and status LEDs (also discussed later).

We also have a Raspberry Pi model 2B. This is a powerful credit card sized computer that is capable of running an operating system. It has a 900 MHz processor (as opposed to 16

MHz on the Arduino) and 1GB of RAM. This will nicely facilitate the localization process. We will feed scanner data directly to the Pi since the data rate from the scanner is high.

We communicate between the Arduino and Pi via serial USB. From the Arduino, the Pi receives the data from the encoders that is used in the localization process. From the Pi, the Arduino receives velocity commands to apply to the motors.

One of our more computer savvy colleagues suggested that we consider using Robot Operating System (ROS). ROS is an open source pseudo operating system that runs on the Pi. It is very useful for scheduling operations. It has a structure that works in terms of nodes. A node is any operation that can be considered on it's own with some type of output. In our case there would be a node for receiving and processing scanner data, a node for running the particle filter, a node for path planning, and any other such operation. ROS automatically handles the interactions between these nodes and the scheduling of their operations. As soon as a node has a necessary result, ROS handles the communication of that result to the necessary nodes.

ROS needs to run on top of a Linux OS. We have Ubuntu 14.04 installed on the Pi. A base version of ROS with the minimum required software was installed via github. Individual packages can then be installed one at a time also from github.

ROS software is open source and there is a list of available packages from which we can choose applicable ones [4]. There is very good online documentation for ROS in general and for individual packages [5]. We are running packages in the 'navigation' stack [6]. Namely, the package 'amcl' performs the localization task. The 'map_server'package takes a .jpg image of the map and translates it into information useful to the localization and path planning operations. The 'navfn' package performs the main path planning, 'dwa_local_planner' does iterative path planning and obstacle avoidance, 'costmap_2d' maintains the grid maps used by the localization and planners, and 'move_base' combines these other operations to produce velocity commands. There are detailed descriptions of the workings of these software packages in their respective documentations [6].

Outside of the navigation stack, we are running three other packages. First, there is a package called 'rplidar' that is a driver for our particular model of laser scanner [7]. There is also a package 'astrid_nav_goals' that we wrote using the ROS action client functionality [8]. This node sends goal poses to the path planning nodes. These goal poses are particular to our application. There are 4 total goals, one for each room in the arena.

Finally, we found a package called ‘ros_arduino_bridge’ that implements serial communication between our boards [9]. This package comes with a sketch to be loaded onto the Arduino. It implements its own communication protocol in which the Pi sends various commands or information requests to the Arduino via single letter commands. The ROS side of this package converts tangential and angular velocity commands to linear velocity commands for each wheel. It also calculates odometry information given encoder ticks from the Arduino. On the Arduino side, there is a PID controller programmed to execute given velocity commands on each wheel. We expanded this Arduino code to add functionality for the flame searching and extinguishing discussed next.

In ROS, parameters are set for each package using .yaml files, and nodes are launched by running .launch files. The majority of the final month of the project was spent tuning parameters. The parameter and launch files, along with the sketch for the Arduino, are on a flash drive attached to the robot.

The open source characteristic of ROS was very helpful to us since we were somewhat lacking in programming expertise. There is a large community of contributors who write good documentation and were very helpful in a couple cases where we emailed to ask questions. The drawback to using ROS is that it is a true operating system with its own protocols and coding paradigms that can be quite difficult to understand for inexperienced coders like ourselves. Any code we wrote ourselves had to follow this style. Given a good understanding of the style, someone could definitely write their own packages to improve on the functionality of the available ones. But we tried and managed to keep our own coding to a minimum.

Search/Extinguish Flame

Hardware

For heat sensing we ordered a 16x4 thermopile array [10]. This is essentially a 64 pixel camera that assigns a heat signature to each pixel instead of a color so we are able to locate the candle once it is in the field of vision. The thermal sensor has a 60 degrees horizontal and 16.4 degrees vertical vision range. It will send the Arduino temperature data via Inter-Integrated Circuit (I^2C) communication. Experimental testing with this sensor has shown that it can detect a flame from a distance of approximately 1.5 meters away, which is far better than we were expecting. Because of the array setup of the thermal sensor, the closer we get to the candle, the

more precise we can locate the flame. This property is good for our flame extinguishing mechanism because the relative range of gas release is smaller than a fan.

There are several ways we discussed on how to extinguish the flame and we decided to use compressed gas. There is a bonus added to our score if we use compressed gas and it is effective because the compressed air will blow with a very high speed. Furthermore, in real life, compressed CO_2 is more effective and realistic to extinguish flames than fans or water spray because fans would blow more air (oxygen) to the fire and worsen the fire situation.

We have found a tire inflator that holds a 16g CO_2 cartridge and releases the compressed CO_2 by pressing a button. We put the tire inflator at the back of our robot and connect to a nozzle which is mounted at the front of our robot approximately 18 cm high. The trickiest part of this operation is aiming the compressed gas. Since the nozzle is a relatively small diameter (~0.25 inch) it needs to be pointed directly at the candle flame to be effective. Because we have a precise thermal camera, there is a particular pixel in the array (the center column of pixels) that is aimed at the nozzle tip. We can then manipulate the robot's position until the flame is centered on this pixel. Once the robot is aimed at the candle, the compressed gas is released to extinguish the flame. The vertical field of view is depicted in Figure 7 below. The downward angle gives us information on the distance of the candle from the robot. An unexpected behavior was that the temperature read by the thermopile array also gets significantly hotter as the robot nears the candle. We can get distance information from this data as well.

The other challenging aspect was how to release the gas. We mounted a cam shaped piece of plastic on a servo motor to push the button to release the gas. We have the logic in our Arduino board that when we get to our position and are ready to extinguish the flame, the servo is rotated. This momentarily presses the button, releasing a burst of compressed CO_2. We found this design to be very effective.

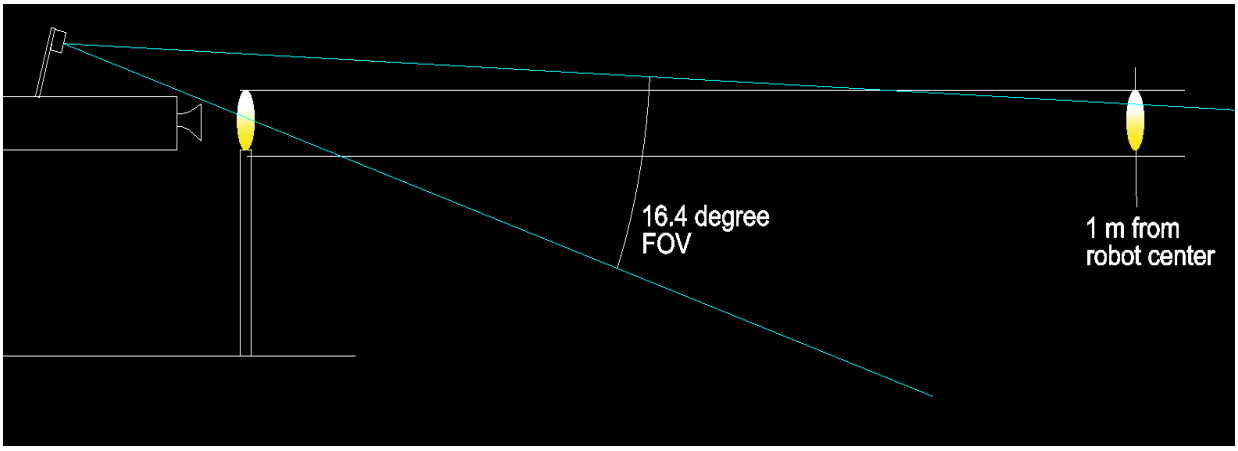


Figure 7. The vertical field of view setup.

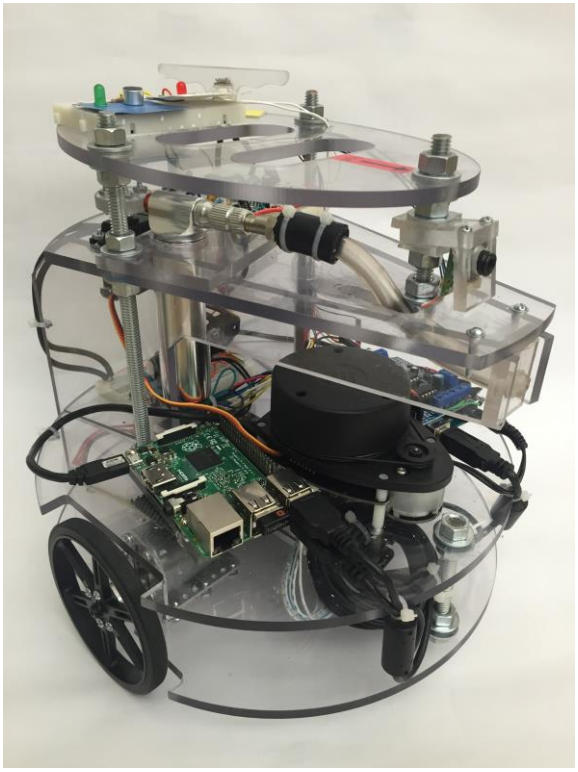


Figure 8. Left view of robot; flame extinguishing system.

Software

The programming for flame detection and extinguishing was done on the Arduino Mega microcontroller. To accomplish this task, we created three main functions, “candle scan,” “flame align,” and “extinguish.” The candle scan function is prompted by a command (called ‘g’ as shown in Figure 9). When called, it iterates over all 64 pixels of the thermal array sensor

checking for any pixel readings above 60 degrees Celsius. This temperature was chosen from experimental testing as a good threshold to determine the presence of a candle. The candle scan function then takes all pixel readings that are greater than 60 degrees and stores them in an array. It also takes the locations of each of those pixels, and stores them in a second array. If no “hot” pixels are detected, the function ends and the robot moves on. If one or more “hot” pixels are detected, the function sets a candle detect flag that prompts the next function, flame align.

The flame align function is called only when prompted by the candle detect flag and it takes the array of hot pixels and determines which of those is the hottest pixel. It then takes the array of locations and pulls out the location of that hottest pixel. The column location of the hottest pixel is then used to calculate the number of radians our robot must pivot on its center in order to align itself perfectly centered on that hottest pixel. This information is sent to our motor control to align the robot. During this process our robot is constantly moving closer to the flame and centering itself on it.

Finally, in the presence of a candle, when our robot is centered on the hottest pixel and if that pixel is greater than 280 degrees Celsius, the extinguish function is called. This function initializes a servomotor, which is used to press the button on the tire inflator, releasing the compressed CO_2. The 280 degree threshold was also selected from experimental testing as a good distance from the candle to extinguish it.

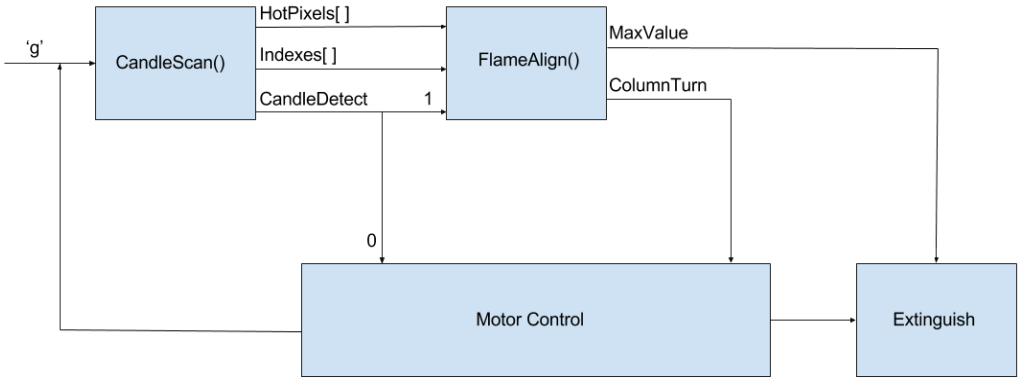


Figure 9. Flame extinguishing block diagram.

Our initial plan for the flame extinguishing software was to have it be prompted by the raspberry pi when our robot is at a goal location in one of the rooms. The task of writing a ROS node that accomplishes this was found to be much more lengthy and involved than we had time for so we had to go a different route. We ended up writing a mock candle scan function that we

called “quick scan” that was placed in the main loop of the Arduino and only checked for the presence of a hot pixel. This function is called once every ~100 iterations of the main loop, essentially causing our robot to be constantly looking for the flame. When the quick scan function detects a hot pixel, the candle scan function is prompted to do a thorough check, and continue on with the regular extinguishing software. The benefits of this quick scan function is that our robot is constantly searching for a flame and in a real life scenario, this is beneficial. It makes more sense to be constantly searching during navigation in case a flame is detected in an unlikely location. The cons of having this function is that in the case of our particular robot, if a flame is detected before our robot is fully within a room, there is a chance that it could hit an obstacle because of the way our programming is set up. When the flame extinguishing system is in control of the motors, the Raspberry Pi is ignored by the Arduino, causing our obstacle avoidance system to also be ignored. In certain cases this has caused our robot to get caught on the corner of a wall in the process of approaching the candle to extinguish it. Given more time and more programming knowledge, this problem is fixable by implementing the extinguishing system onto the same processor as the navigation in order to keep the obstacle avoidance capabilities while extinguishing the flame.

Start (LEDs, microphone, kill power plug, handle)

The contest requires each robot to have 4 key elements, a flame detect LED, microphone, kill power plug, and a handle. The LED must be red over a white background, and is used to indicate flame detection. The microphone needs to be on a blue background and when it detects a sound of a specific frequency and amplitude it activates the robot. The kill power plug is used to immediately remove power from the robot’s sensor and control system and its drive system (when plug is removed, all robot systems are turned off) in case of emergency and needs to be on a bright yellow background. All of these devices, as well as a handle used to safely lift and move the robot will be placed at the very top of the robot.

For the microphone start, there is a protoboard circuit near the top of the robot which is a frequency to voltage converter [11]. This circuit outputs a DC voltage proportional to the input frequency. This voltage can be read and made to trigger operation when the correct frequency is detected.

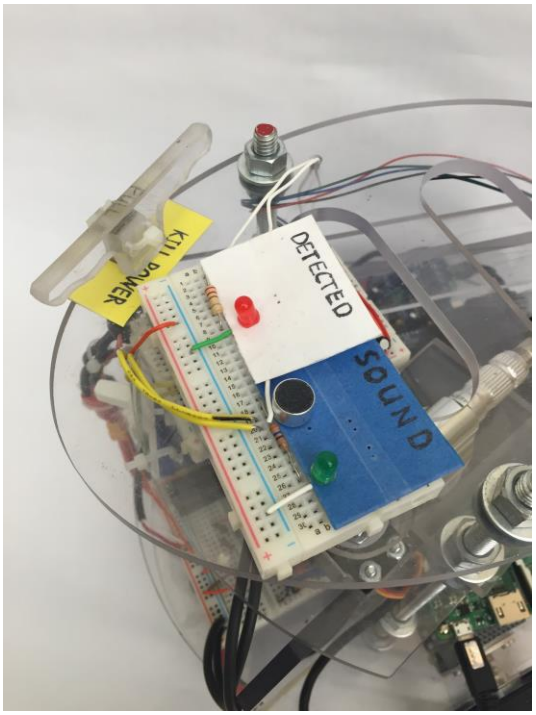


Figure 10. Start board and carrying handle.

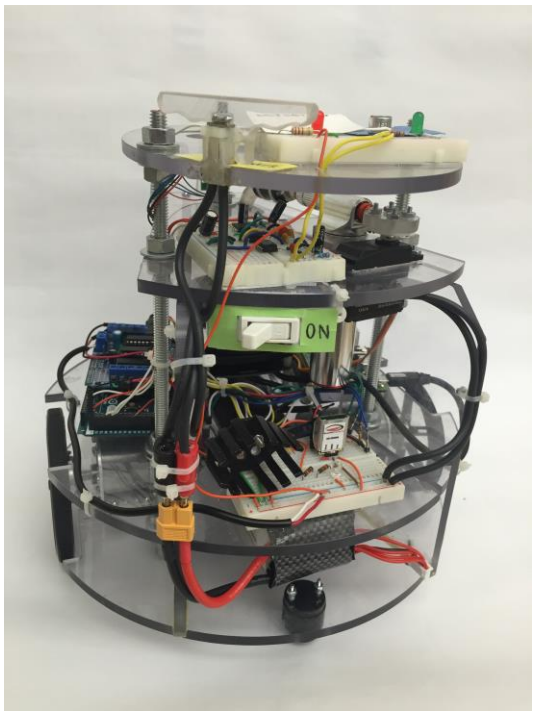


Figure 11. Rear view of robot.

Power Supply

To power our robot, we decided to use rechargeable batteries for convenience as well as to reduce expenses (replacing disposable batteries is an unreasonable option). Two packs of the batteries were ordered to prevent the situation (if we only have one pack) where we have to stop our test on the robot and recharge the battery. We did some rough analysis on how much total power will be needed and determined that the maximum current our robot will draw at any given time will be approximately 5 amps. From this, we decided that we wanted our battery to be about 5000 mAh and at least 12 Volts. We chose two packs of rechargeable Li-Polymer batteries with 14.8 Volts and 5500 mAh. The ones we chose are relatively compact since they are designed for airborne drones. They are also economical compared to other similar products we found.

A linear 12 V regulator is used to supply the Arduino, and a 5 V switching regulator is used to step down voltage for the Pi. We found that a 5 V linear regulator had to dissipate far too much power in stepping down from 14.8 V to 5 V.

Experimental Procedures

We performed preliminary testing on our two main sensors, the laser scanner and the thermal array sensor. To test our laser scanner we set up a room constructed out of old poster boards, positioned our scanner at two locations in the room and collected data using our Arduino connected to a computer. Our experimental setup can be seen in Figure 12 below. We found that the connection to the computer was not able to handle the amount of data being collected and only gave roughly 10% of the total data, which explains the gaps between data points in Figure 13 shown below. This experiment successfully proved the ability of our scanner to map out its surroundings. It is clear even from the 10% of data shown, the shape and orientation of the room. Our robot will be able to utilize the full amount of data collected as it will not have to send that data over a usb connection to a computer, thus giving an extremely accurate model of the arena.

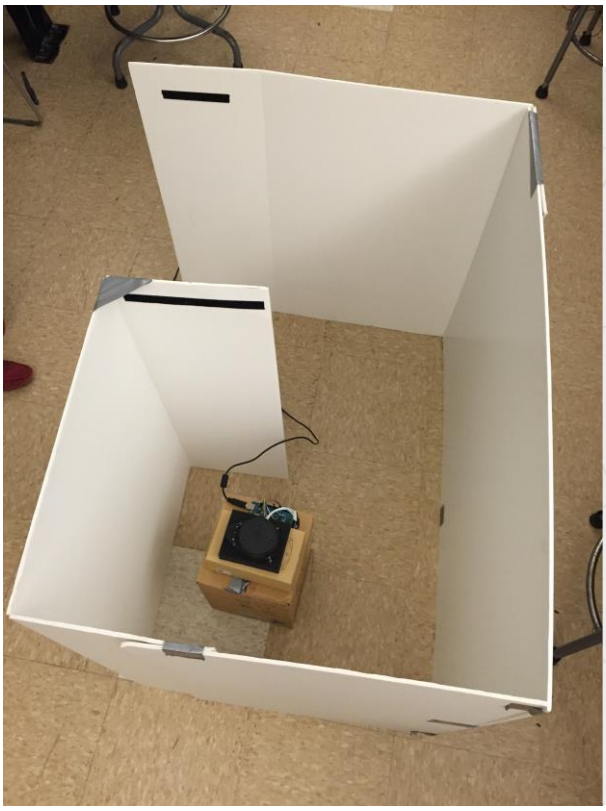


Figure 12. Experimental setup.

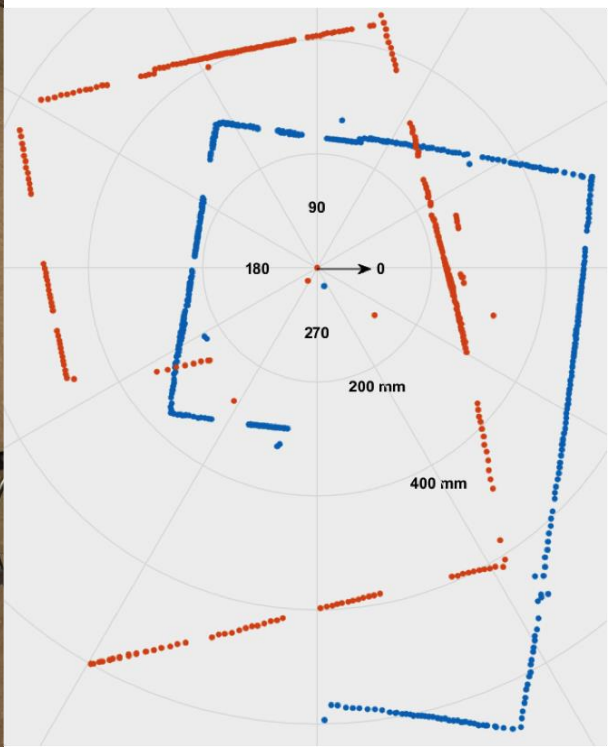


Figure 13. Experimental data.

To test our thermal array sensor, we set up an experiment in which it was aligned at the same height as our candle as seen in Figure 14, and took data at various distances to see how well it can detect a flame. From our data we found that the sensor can pretty clearly detect the flame from 1.5 meters away, as seen in Figure 15 below. Using MATLAB, we gathered the heat

map data which is color coded and each pixel is labeled with the temperature reading in degrees Celsius. At a distance of 1.5 meters it is clear that the flame is detected as 64.11 degrees Celsius and all surrounding pixels read approximately 20 degrees Celsius. As we get closer to the flame, our accuracy increases. From a distance of 20 centimeters we can clearly distinguish the flame by the deep red pixel that reads 394.34 degrees as shown in Figure 16. The total field of view of our thermal array sensor is also shown in Figure 17.

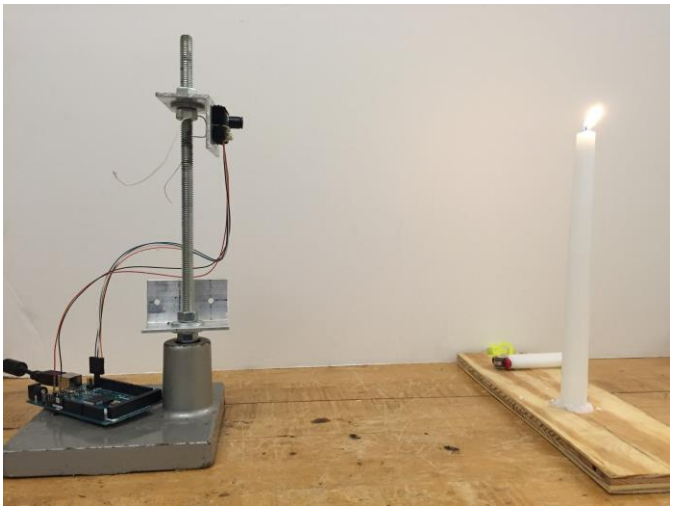


Figure 14. Thermal array sensor test setup.

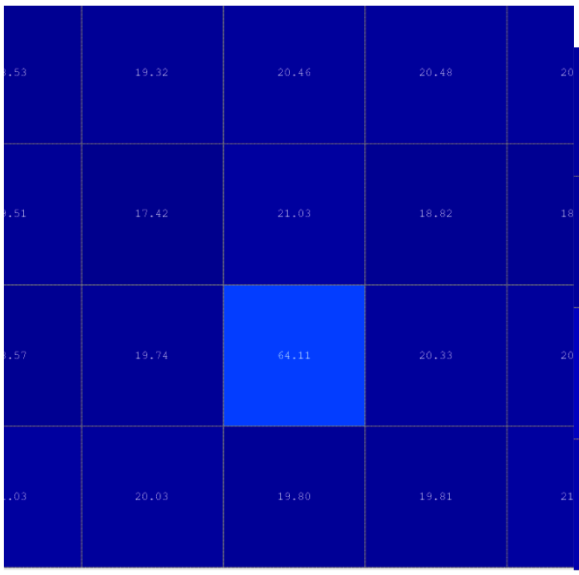


Figure 15. Heat map at 1.5m distance.

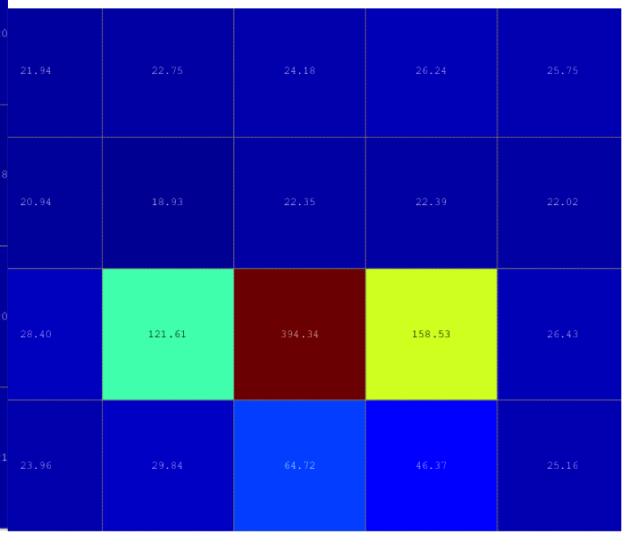


Figure 16. Heat map at 0.2m distance.

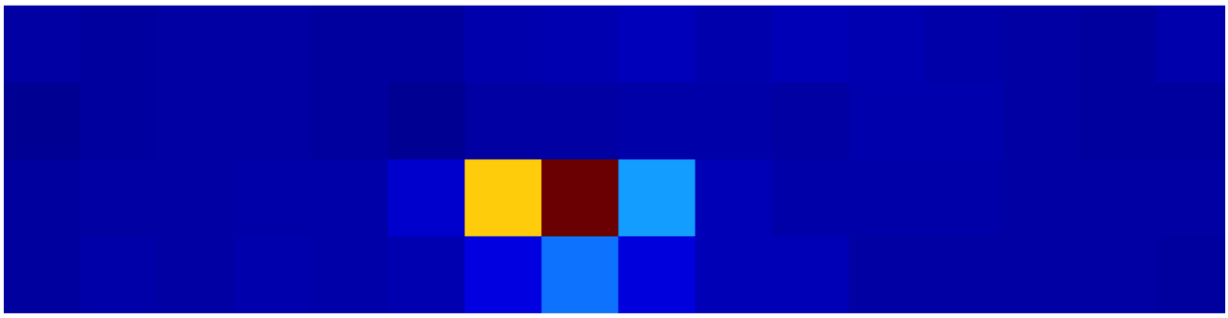


Figure 17. Total field of view.

Contest Results / Future Improvements

We competed in the Trinity College International Robot Contest on April 2, 2016, in which our robot won the IEEE Connecticut Outstanding Robot Award for the senior division. This award is given to "outstanding Connecticut-made robots based on the ability of the team to work together and solve clever engineering problems." We are very proud to have our work recognized and appreciated by the IEEE Connecticut section. Performance-wise, the contest didn't go as well as we had hoped, having no completed runs, but we are very proud of the level of intelligence of our robot. Our downfall in the contest was not a lack of ability of our robot, but in the nitty-gritty rules and setup of the contest that are not representative of real-life situations.

One of the main problems we had was in the startup of our robot. The time constraint of the project caused us to skip out on setting up the Pi to automatically launch on startup. What this means is that in order to start up our robot, the Pi needed to be turned on, logged in with a username and password, and have the launch file executed manually. This was acceptable startup according to the contest judges, as the robot is still fully autonomous at sound activation, however it caused a load of issues in the actual competition. Unfortunately, we were unable to bring our monitor to the contest arena so when we manually executed the launch file, it didn't always actually launch. In one case, the Raspberry Pi had gone into sleep mode and when we clicked to launch it was actually prompting for the password to log back in. In another case, the launch file was accidentally minimized so clicking to execute ended up clicking nothing. With more time, these issues are avoidable by setting up the Pi such that the password is eliminated, sleep mode is deactivated, and the launch file executes on startup.

Another issue that affects the functionality of the robot is that our sound activated start is implemented on the Arduino microcontroller. Similar to writing the flame extinguishing

program, we found that writing a ROS node to control the sound activated start was difficult with our limited programing experience. We avoided this issue by implementing the functionality on the Arduino. When the robot is powered on, the Arduino ignores all commands from the Pi and listens for the frequency to activate. We have found that this method works, however the longer the robot waits, the greater likelihood that our robot behavior is negatively impacted when it finally initializes. In our own test arena, the sound activation, prompted by a team member, is usually 30 seconds to a minute after startup. In the contest, our robot could be waiting upwards of 5 minutes on the judges table before being placed in the arena and initialized. Because the navigation is constantly trying to localize during this time, sitting on the judges table for 5 minutes did not bode well for our localization. This caused our robot to spin around and perform recovery behaviors in order to attempt to figure out its location. 30 seconds of this behavior ended our trial in the contest as per the rules. For this function to work properly, it needs to be integrated into ROS which could be done by a reasonably experienced programmer.

Another issue we ran into was the processing power of the Pi. It runs Ubuntu and ROS without much trouble. But when we fully launch all of the operations involved, it uses ~90% of the processor. This resulted in latency issues. Particularly, the local path planner can only run at 2 Hz max and sometimes as slowly as 0.5 Hz. This is the node that generates velocity commands so this means we're getting new velocity commands only once a second or so. Because of this, we had to set very low velocity limits of 0.5 m/s to avoid the robot hitting walls and obstacles continuously. This does not help when doing timed trials. If the local path planner could run between 10 and 20 Hz instead, much higher speed limits could be used since the robot's velocity could be corrected more often. Fixing this would involve increasing the processing power of the robot. We would suggest replacing the Pi with the ODROID-C2 or ODROID-XU4 [12]. Ubuntu and ROS could be set up exactly the same way on these boards, but their processors run at 2 GHz as opposed to the 900 MHz on the Pi. This would give the robot processing power equivalent to a normal laptop computer and should solve many of the processing issues we had with the Pi. An added benefit of this switch would be in the graphics area. ROS has a very good GUI called 'rviz'. It can be used to visualize real-time navigation data in graphical form and can be used for tuning parameters while the robot is running. The graphics capabilities of the Pi were such that we were unable to use this GUI. It would've been hugely helpful, and switching to one of the ODROID boards may be worth it for this reason only. The Pi has already been removed from the

robot since it is a personally owned item. If it is needed for further work on the project, see the contact information at the end of this report.

We also decided to compete just in level 1 of the contest. This is because there is a single fixed map for this level. In level 2, there are 4 possible map configurations. We tried grouping these 4 layouts into a single global map and doing localization in that map. We ran into two issues doing that. First, we had to initialize localization with an unknown starting pose. The problems with this are discussed further in the following paragraph. Also, even once we did get it started this way, the localization struggled. This is because the 4 maps are actually *too similar* to each other. The localization needs the laser to pick up unique features in order to discriminate between possible poses. Since the 4 maps are really just the same map with slight variations in door locations, picking out a single location in one of them proved to be next to impossible. The best solution to this would be to scrap the localization and revert back to our original plan to do Simultaneous Localization and Mapping. There is a ROS package called 'gmapping' that does this that could simply be run in the place of 'amcl'. This would be an interesting thing to try since it would eliminate the need to have any knowledge about the map whatsoever.

Lastly, our team chose to do a fixed start location in the contest. This is most resemblant of a real-life scenario and also works best functionally for our system. The start location is given in the contest rules and we programmed it into our robot. Because we are given the exact start location, we decided to set a very precise start location in the system, as it lightens our computational load upon initializing. Unfortunately, in the actual contest, the judges paid little attention to the accuracy of robot placement in the arena and our robot was shifted upwards of 10 cm away from its given start location. Although this seems to be a minor shift, we did not account for such a large degree of error in this case. This likely contributed largely to our initial struggle to localize during the contest. This problem is avoidable by inflating the start position in the program, however it increases localization processing time initially. In our own arena, we are able to keep our very specific start location and place our robot precisely in the arena, speeding up this initial localization problem. In a real life scenario, the setup used in our own arena is much more accurate, as a robot will likely be placed in a permanent, designated docking station, waiting for initialization. There will be no question as to the initial location of the robot. It is possible to initialize the localization to work with an unknown start location and it would be an interesting thing to try. But in a timed contest situation it is likely to just waste time.

Results and Discussion / Conclusion(s)

The robot is highly intelligent and likely able to be extended to a wider-range of applications and scenarios. Smoke detectors that set off fire alarms could be used to initiate the sound-activated robot. Pre-programming the robot with a floor plan and room locations would make it easy to find and extinguish a fire before humans are even aware of the situation.

Many of the methods used in this project, specifically in the navigation aspect, are still under research and development. Anyone interested in coming up with possible improvements could write their own algorithms given some programming experience in C++ and/or Python.

It may be very challenging or even impossible to improve performance to the point that it would win in the Trinity Contest in particular since many of the robots there are smaller, lighter, microcontroller based, and are capable of very high speeds. But our robot is a versatile platform that could be used even just as a tool for navigation algorithm development.

Acknowledgments

Sponsor: University of Connecticut School of Engineering
Project Advisor: Professor John Ayers
Consultant on Computer Applications: David Paquette

Contact

For any particular questions about the project, please contact Zac Sutton at zcsttn@gmail.com or 860-906-2854.

References

[1] SLAM course by Cyrill Stachniss
www.youtube.com/playlist?list=PLgnQpQtFTOGQrZ4O5QzbIHgl3b1JHimN_
[2] Ioannis M. Rekleitis. *A Particle Filter Tutorial for Mobile Robot Localization*
Center for Intelligent Machines, McGill University, Montreal, Quebec, Canada
[3] Patric Jensfelt. *Localization Using Laser Scanning and Minimalistic Environmental Models*
Royal Institute of Technology, Stockholm, Sweden
[4] http://www.ros.org/browse/list.php
[5]http://wiki.ros.org/
[6]http://wiki.ros.org/navigation
[7]http://wiki.ros.org/rplidar
[8]http://wiki.ros.org/actionlib
[9]https://github.com/hbrobotics/ros_arduino_bridge
[10]http://www.robotshop.com/en/roboard-rm-g212-thermal-array-sensor.html
[11]http://www.seattlerobotics.org/encoder/200112/elik.htm
[12]http://www.hardkernel.com/main/main.php

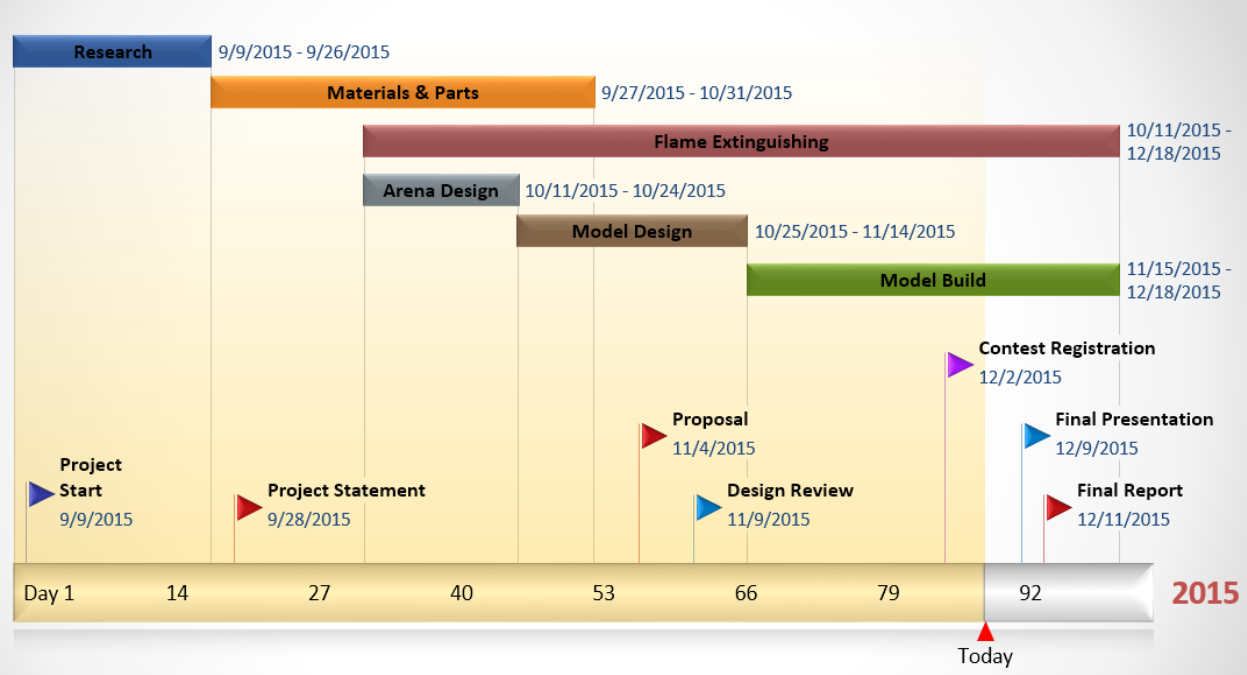
Appendix

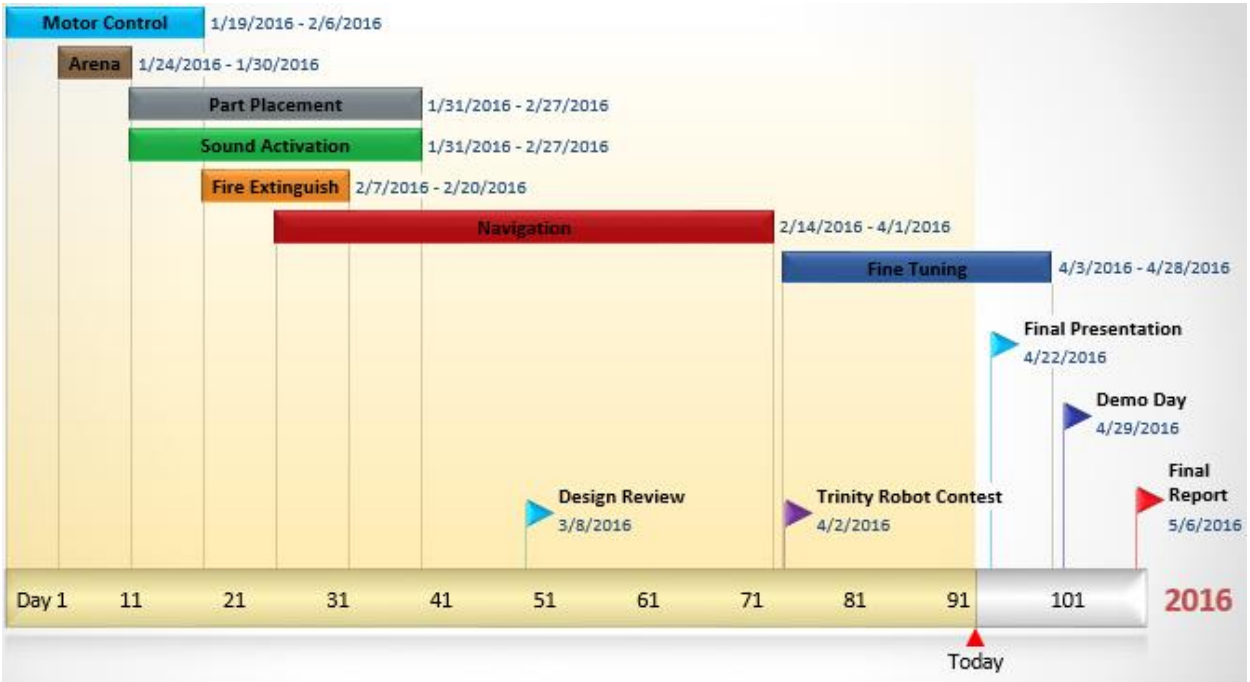
All code that can't be pulled from github is attached on a flash drive with the robot

Budget

Quantity	Item	Amount ($)
1	Arduino Mega 2560 Microcontroller Rev 3	37.39
2	Pololu 12V, 100:1 Gear Motor w/Encoder	79.9
1	Pololu 37D mm Metal Gearmotor Bracket (Pair)	7.95
1	Pololu Universal Aluminum 6mm Mounting Hub (4-40)	7.95
1	Pololu Wheel 90 x 10mm Black (Pair)	9.95
1	Pololu Ball Caster with 3/4" Metal Ball	2.99
1	RoBoard RM-G212 16X4 Thermal Array Sensor	120.17
1	RPLIDAR 360° Laser Scanner	398.9
1	Polycarbonate (PC) Sheet, Transparent Clear, Standard Tolerance, 1/4" Thickness, 24" Width, 48" Length	78.9
1	2Pcs 14.8V FLOUREON 5500mAh 35C 4S Lipo High Power Battery RC Battey Pack	83.99
1	TLP-2000 Tenergy Universal Smart Charger for Li-Ion/Polymer battery Pack (3.7V-14.8V 1-4 cells)	21.95
1	Mechanic Shop Poly Carbonate Cut	125
4	4*8' Plywood	80
	Total	1055.04
	Personal Expenses	
1	Raspberry Pi Microcontroller	35
1	Tire Inflator	18
3	Compressed CO2 Canisters (14 pk.)	42
1	Motor Driver Shield	10
1	Miscellaneous: (Nuts, Bolts, Wood Glue, Sandpaper, Screws, ICs, etc.)	30
	Total	135
	Grand Total	1190.04

Timelines


Research
9/9/2015 - 9/26/2015
Materials & Parts
9/27/2015 - 10/31/2015
Flame Extinguishing
10/11/2015 - 12/18/2015
Arena Design
10/11/2015 - 10/24/2015
Model Design
10/25/2015 - 11/14/2015
Model Build
11/15/2015 - 12/18/2015
Contest Registration
12/2/2015
Proposal
11/4/2015
Final Presentation
12/9/2015
Project Start
9/9/2015
Project Statement
9/28/2015
Design Review
11/9/2015
Final Report
12/11/2015
Day 1
14
27
40
53
66
79
92
2015
Today


Motor Control
1/19/2016 - 2/6/2016
Arena
1/24/2016 - 1/30/2016
Part Placement
1/31/2016 - 2/27/2016
Sound Activation
1/31/2016 - 2/27/2016
Fire Extinguish
2/7/2016 - 2/20/2016
Navigation
2/14/2016 - 4/1/2016
Fine Tuning
4/3/2016 - 4/28/2016
Final Presentation
4/22/2016
Demo Day
4/29/2016
Final Report
5/6/2016
Design Review
3/8/2016
Trinity Robot Contest
4/2/2016
Day 1
11
21
31
41
51
61
71
81
91
101
2016
Today