WORCESTER POLYTECHNIC INSTITUTE
ROBOTICS ENGINEERING PROGRAM

Team 10 Final Report

SUBMITTED BY
Taylor Bergeron
Samuel Milender
Daniel Oates

Date Submitted : 12/16/17
Date Completed : 12/15/17
Course Instructor : Prof. Putnam
Lab Section : B01

Table of Contents:

1.0 Introduction:

The challenge for B term RBE2002 is to design and program a robot to traverse a maze to locate a flame, report the location of the flame, put it out, and return to where it started. There is a cliff, marked by a two-inch black line, located throughout the maze that the robot must avoid or else fall off the edge of the table. The flame is created by a taper candle, and will be located between 10- and 30 cm off the table. When returning the location of this candle, the robot should report the xy coordinates, so where in the maze the candle is, as well as (for extra credit) the z coordinate, or how far off the table the flame is. The robot may use any extinguishing device if it does not damage the electronics given in the lab kit, and may optionally travel back to where it began for extra credit.

2.0 Design Methodology:

The design chosen was determined by the desire to have a slender robot, as well as one that has the best organization of electronics. There are three levels to the robot.

2.1 CAD Design

The first contains two Polulu motors for the two drive wheels, along with a mount for a freely moving omni wheel. As slippage is one of the main concerns due to smooth rubber wheels driving on a slippery and waxy white board table, there are two drive wheels with the VTC at the midpoint between the two motors. This means if the robot slips the sonar sensor senses the robot is farther away, and a correction is made. The original design for the robot called for two drive wheels and two free omni-wheels, as this is what the team defaulted to for robots in the past courses. However, after printing the first design with four wheels, it quickly became apparent that if there was any dips or bumps in the table's surface, one of the drive wheels could lose traction, with the weight of the robot resting on the two omni wheels and the remaining drive wheel. This would cause the robot to turn in place. Therefore, if there are only three points of contact, they will all always be in contact with the ground. Two of the wheels are drive wheels, with the third being an omni so that there is stabilization, but it does not determine the robot's movement. There are 3D printed motor mounts, connected to the first and second layer, so that the motors do not move at all, and the VTC is exactly where it is calculated to be. The battery is placed on the bottom layer as well, right next to the drive wheels, so that there is less of a chance of them losing traction. There is also a lip on the front of the robot, two inches from the axis of the drive wheels. This is for the line sensor to stick out in front of the robot and detect the two inch black line that appears directly before the cliff. This placement also makes it so that the robot can turn in place after the sensor sees the line, and not fall off the edge of the table.

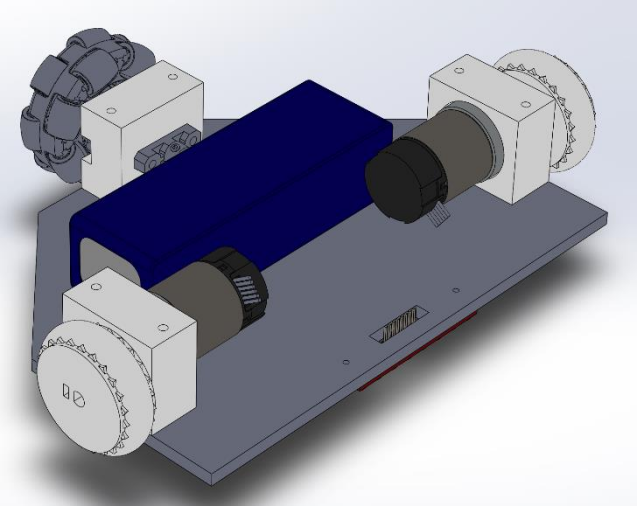

Figure 1. Bottom Layer of Robot

Layer two includes the four sonar sensors, as well as the electronics. The Sonar sensors are attached to the robot using custom designed 3D printed mounts, as for calibration purposes they cannot move between runs. There are four- one on each side of the robot, to detect the four walls surrounding the robot. The robot always starts following the left wall, so each sonar sensor is perpendicular to the wall it is measuring. The mounts also act as spacers between the second and third layer, giving the electronics enough room.

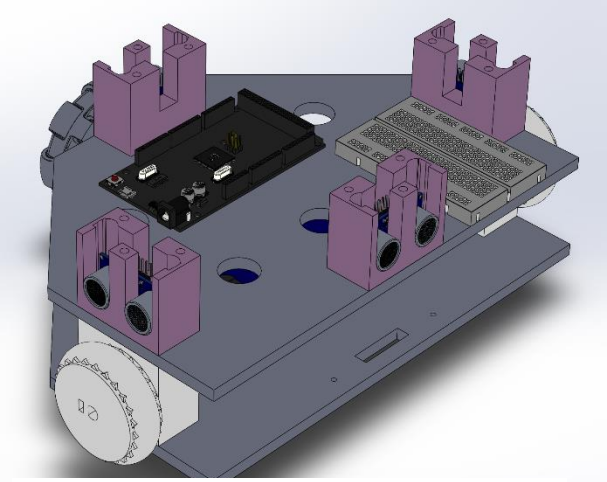

Figure 2. Second Layer of Robot

The third layer of the robot contains all the mechanisms for the pan-tilt system. The servo for the pan is embedded in the third layer so it does not move. This moves a 3D printed part for the pan. The pan arm was chosen so the robot can pan 90 degrees from the center of the robot to the right side of the robot. This is because there is no need to pan left as the wall is directly to the left. The tilt servo is embedded in the pan tower, and moves the tilt arm up and down. The tilt arm is designed to attach to the fan shroud, and the fan shroud to the fan. This is because with the way 3D printing works, if the shroud and tilt arm were connected, the tilt arm would be printed with short resin strands (width) rather than long resin strands (length). This would mean the arm could snap with the weight of the fan, which is why the original one-piece design was redone into a two-piece design. The shroud was created with solid sides, not grated sides, so that the air flow could be directed into a stream onto the flame once it is detected. The flame sensor is taped to the top of the shroud, as this was the simplest way to attach it. This way the arm/shroud tilts up and down and pans left to right at once, as the two servos can move at once, looking for the flame.

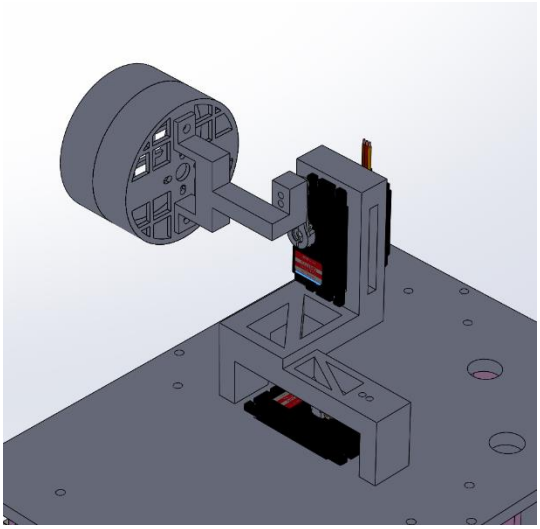

Figure 3. Pan-Tilt Fan System of Robot

The software for the robot spans over two platforms: the state-machine control systems code on the Arduino Mega 2560 (in C++) and the Matlab code which handles the UI and field mapping on the laptop control station. The two platforms communicate via standard serial over Bluetooth protocol, with an Hc-06 Bluetooth module on the robot to facilitate this communication.

In order to simplify, manage, and organize the large amount of C++ code and simultaneous processes needed to successfully control the robot, object-oriented design, namespaces, and state machine programming were heavily employed.

Each namespace encapsulates a system or process in the robot, including odometry, sonar, wall-following, and Bluetooth communication. Most namespaces contain setup() and loop()

functions analogous to the Arduino setup() and loop(), which handle namespace initialization and continuous updates, respectively.

2.2 Sensors

Each sensor on the robot has a very specific purpose integral in the robot's performance. TheBno055 IMU was chosen as the velocity and direction of the robot is important when navigating a maze and returning back to the start position. Four HcSr04 Ultrasonic Sensors were chosen as these are reliable and accurate sonar sensors for measuring the distance from the sensor itself to the corresponding wall. Other than pure were used for field mapping, and omnidirectional ranging. When it came to detecting the cliff, a QTR-8A Line Sensor was chosen to detect the black lines at the cliffs to avoid them, and this detection threshold was calibrated to the practice field. As for the motors, Polulu has a built in encoder, with a resolution of 3200 counts/rev. This is a high resolution, giving very precise values for the amount the axis has turned, which is why it was chosen for both computing the robot velocity and computing the position of the robot VTC in conjunction with the IMU. Finally, the KEYES 760-1100nm Light Detector was utilized to detect the flame presence, and therefore it is used in the process of determining the angle of the flame to the robot.

2.3 Namespaces

The following namespaces are used:

RobotDims : This namespace contains all physical dimensions of the robot necessary for internal computations. Organizing all of this information into a separate namespace allows for quick access and modification if the mechanical robot platform is redesigned.

IndicatorLed : This namespace utilizes the built-in LED on pin 13 of the Arduino Mega to indicate robot state and errors by turning on, off, and blinking. It is primarily used to indicate serial and I2C communication errors, helping reduce debug time if any robot wiring comes unplugged.

MotorL and MotorR : These namespaces contain the Arduino pin connections for motor control via the Pololu dual MC33926 motor driver. Each contains an instance of the <DcMotor> class, which facilitates voltage control and distance tracking via encoders. Their setup() functions initialize the motor objects and attach the necessary encoder interrupts.

Odometer : This namespace interfaces with the motor encoders and the Bno055 IMU to perform real-time odometry, heading, and velocity calculations. The positioning algorithm uses vector integration for accurate position tracking regardless of the shape of the path taken. Details on the theory and implementation of this algorithm are discussed in the analysis section.

Sonar : This namespace handles wall distance ranging via the four HcSr04 sensors mounted at the front, back, left, and right of the robot. Given the behavior of the sonar sensors, it is possible for all four sensors to take a combined time of up to 0.15 seconds to update all four sonar distances. If the program were paused for this amount of time during sonar collection, this would limit the loop frequency of the program to at most 6.6Hz, which would severely limit odometer accuracy and smooth PID control. For this reason, an <HcSr04Array> class was developed which uses interrupts and Timer1 to continuously sweep through all four sensors and get accurate distance readings without pausing the program to wait for each individual ping to return. Additionally, this class returns a zero distance when a ping is not returned, allowing for higher-level systems to ignore bad sonar data.

MatlabComms : This namespace interfaces with the Hc-06 Bluetooth module and facilitates serial communication with the laptop control station. It utilizes a <BinarySerial> library which supports exchanging bytes, ints, floats, and strings with other Arduinos and Matlab, which has a similar <ArduinoSerial> library which uses the same communication protocols. All communications begin with Matlab sending a message type byte to the Arduino (either begin, stop, or get-data), which the Arduino reads and responds to appropriately. If either side doesn't receive a correct response or message within pre-defined timeout periods, communication errors are initiated on both ends and the robot shuts down.

DriveSystem : This namespace initializes and controls the drive system via the MotorL and MotorR namespaces. It contains PID controllers for both absolute heading and drive velocity, and uses these to control the voltages applied to each motor to achieve its desired motion. A <PidController> class is used to hide the repetitive PID control computations from the user.

WallFollower : This namespace contains a state machine and PID controllers which use sonar and the cliff sensor to perform left-sided wall-following. When enabled, it directly controls the DriveSystem namespace. In certain states, it can be paused so that the main robot state machine can take over the DriveSystem to put out the flame. Once the flame is put out, the wall-follower re-enabled in the state at which it was paused so it can return home. Details on the state machine and PID control systems are elaborated on in later sections.

PanTilt : This namespace controls the pan-tilt servos which direct the fan and flame sensor. An <OpenLoopServo> class was developed which allows the user to control the angle and velocity of each servo while precisely knowing the angle of the servo at all times. This accurate feedback is important for localizing the position of the flame on the field. This namespace also contains two state machines, one for performing a left-to-right pan sweep, and the other for a down-to-up tilt sweep. When ran simultaneously, the flame sensor is moved throughout a wide range of angles while the robot drives in order to aid in detection of the flame.

FireBot : This namespace contains the main robot state machine, the flame sensor, and the fan for putting out the flame. The fan is controlled by a <BrushlessMotor> class. It is separated from the main ino file uploaded to the Arduino so that debug code can be quickly inserted and uploaded in place of its main setup() and loop() functions. Its setup() function initializes all namespaces, the flame sensor, and the fan, and waits for the begin message from the laptop control station. Once the message is received, it enters the loop(). The loop() function continuously updates the odometry, pan-tilt system, and checks for messages from the control station. It also runs the main robot state machine.

2.4 State Machines

Along with the namespaces, this robot employs four simultaneous state machines. The main robot state machine initially allows the wall-following state machine to control the drive system, and the pan-tilt state machines to sweep in search of the flame. Once the flame has been detected, the main state machine pauses the wall-following state machine, disables the pan and sweep state machines, and assumes full control of the pan-tilt and drive systems to locate and extinguish the flame. Once the flame is extinguished and the robot has returned to the wall, the wall-following state machine is unpaused and the robot wall-follows until it has returned to its home position.

Each state machine is described in greater using the following shorthand:

<Name of State Machine> (<name of file containing it>)

Brief: <brief description of state machine function>

States:

<STATE_NAME_1>: <Brief description of state task>

<Transition condition 1> → <STATE_TO_GO_TO>

<Transition condition 2> → <STATE_TO_GO_TO>

…

<STATE_NAME_2>: ...

The following is a description of each state machine, with all states and transitions listed.

Pan State Machine (PanTilt.cpp)

Brief: Sweeps the pan servo left and right between angle limits defined in software.

States:

- PAN_RIGHT: Turn pan servo right
    - Servo at maximum pan → PAN_LEFT
- PAN_LEFT: Turn pan servo left
    - Servo at minimum pan → PAN_RIGHT

Tilt State Machine (PanTilt.cpp)

Brief: Sweeps the tilt servo up and down between angle limits defined in software.

States:

- TILT_UP: Turn tilt servo up
    - Servo at maximum tilt → TILT_DOWN
- TILT_DOWN: Turn tilt servo down
    - Servo at minimum tilt → TILT_UP

Wall-Follower State Machine (WallFollower.cpp)

Brief: Performs left-sided wall-following using sonar data and IMU.

States:

- STOPPED: Do nothing (not controlling drive system)
- FORWARD: Perform left-wall-following
    - Left wall disappears → CHECK_LEFT
    - Cliff detected → BACK_FROM_CLIFF
    - Front wall gets close → TURN_RIGHT
- CHECK_LEFT: Drive a small distance forward (timer-based)
    - Near cliff → BACK_FROM_CLIFF
    - Left wall still gone → PRE_TURN_LEFT
    - Otherwise → FORWARD
- PRE_TURN_LEFT: Drive forward for fixed time to make it around corner
    - Timer elapsed → TURN_LEFT
- TURN_LEFT: Make a 90-degree left turn
    - Turn complete → POST_TURN
- POST_TURN: Drive forward at fixed heading
    - Left wall detected → FORWARD
    - Cliff detected → BACK_FROM_CLIFF
    - Front wall gets close → TURN_RIGHT
- BACK_FROM_CLIFF: Drive backwards from cliff (timer-based)
    - Timer-elapsed → TURN_RIGHT
- TURN_RIGHT: Make a 90-degree right turn
    - Turn complete → POST_TURN

Main Robot State Machine

File: FireBot.cpp

Brief: Controls high-level robot tasks of finding and extinguishing the flame and returning home.

States:

- SEARCH_FOR_FLAME: Wall-follow and perform pan-tilt sweep
    - Flame detected → ZERO_PAN_SERVO
- ZERO_PAN_SERVO: Stop robot and set pan servo to angle 0 in prep for pan sweep
    - Pan servo at 0 → GET_FLAME_HEADING
- GET_FLAME_HEADING: Sweep pan servo right to determine heading of flame
    - Pan servo at maximum pan → TURN_TO_FLAME_HEADING
- TURN_TO_FLAME_HEADING: Turn towards flame and zero the pan servo
    - Heading stabilized and servo zeroed → DRIVE_TO_CANDLE
- DRIVE_TO_CANDLE: Drive forwards to flame and measure time it took
    - Within predefined distance → LOWER_TILT_SERVO
- LOWER_TILT_SERVO: Set tilt servo to minimum tilt in prep for tilt sweep
    - Tilt servo at minimum tilt → GET_FLAME_TILT
- GET_FLAME_TILT: Sweep tilt servo up to determine tilt angle of flame
    - Tilt servo at maximum tilt → AIM_AT_FLAME
- AIM_AT_FLAME: Aim tilt servo towards flame tilt
    - Pan-tilt aimed at flame → EXTINGUISH FLAME
- EXTINGUISH_FLAME: Run fan at maximum speed to extinguish flame
    - Flame extinguish detected → CHECK_FLAME
- CHECK_FLAME: Keep fan running for pre-defined time
    - If flame comes back → EXTINGUISH FLAME
    - Otherwise → BACK_FROM_CANDLE
- BACK_FROM_CANDLE: Turn off fan and drive backwards to wall for the same amount of time it took to drive up to the candle.
    - Timer elapsed → TURN_TO_WALL

- TURN_TO_WALL: Turn to target heading according to wall-follower
    - Heading stabilized → GO_HOME
- GO_HOME: Wall-follow around the rest of the field
    - Within fixed distance of starting position → AT_HOME
- AT_HOME: Stop driving. Task complete!

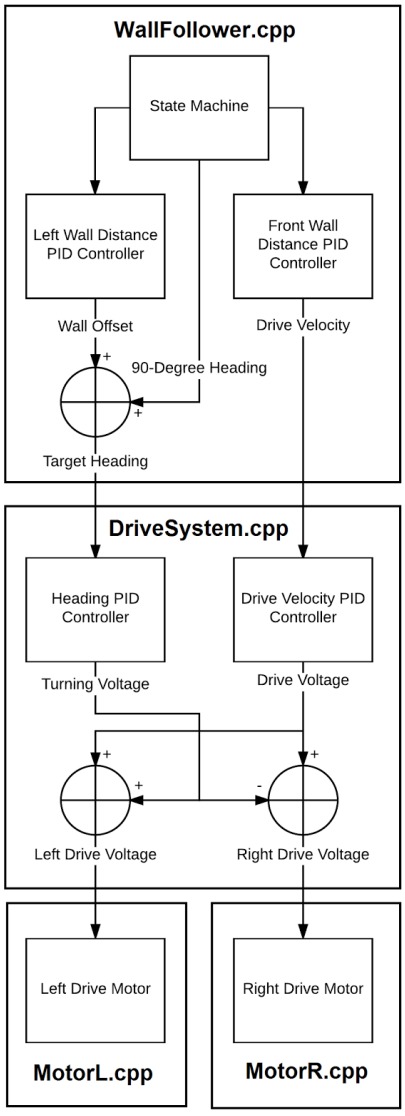

Figure 4. Drive PID Control Flow

Using multiple simultaneous state machines is a powerful tool in dividing and conquering complex automated tasks. However, the process of handing control of subsystems between different state machines is complicated and easy to do incorrectly. It is essential that only one state machine at a time ever attempts to control a subsystem (ex. Pan-tilt servos, drive motors, sonar, etc.). Additionally, some state machines can only be stopped in certain states without causing problems. For example, stopping the wall-follower during a timed driving state could cause it to drive an incorrect distance and risk colliding with a wall.

2.5 Drive Control

The drive control systems design implements high-level control flow. The motion of the robot base is controlled entirely by the wall-following state machine and a chain of parallel and series PID controllers. The high-level control flow is shown in Figure 4.

2.6 Drive System PID Controllers

The drive system contains PID controllers for drive velocity and heading. The drive velocity controller uses the velocity calculated in the Odometer namespace for feedback, and applies a drive voltage to both the left and right motors to reach the setpoint. The drive heading controller uses the heading obtained from the IMU in the Odometer namespace for feedback, and applies a differential drive voltage (subtracted from one wheel and added to the other) to turn to the desired setpoint. Both controllers have maximum and minimum outputs that restrict the total sum voltage on either wheel to within plus or minus 12 Volts.

Because the IMU outputs heading in the range of 0 to 2pi, there exists a toggle point in the heading output between 0 and 2pi that interferes with conventional PID control. Consider if the error were calculated as the traditional (target heading) - (heading). Two issues arise:

1. If the target heading is 0 and the robot overshoots at all, the error will change from 0 to 2pi and the robot will attempt to make another full 360-degree turn to reach the setpoint.
2. If the robot has a small positive heading and the setpoint is close to 2pi, the robot will make almost a full clockwise turn instead of a small counterclockwise turn.

To alleviate this problem, an angular transform is applied to move the toggle point of the error to 180 degrees out of phase with the setpoint, as shown below.

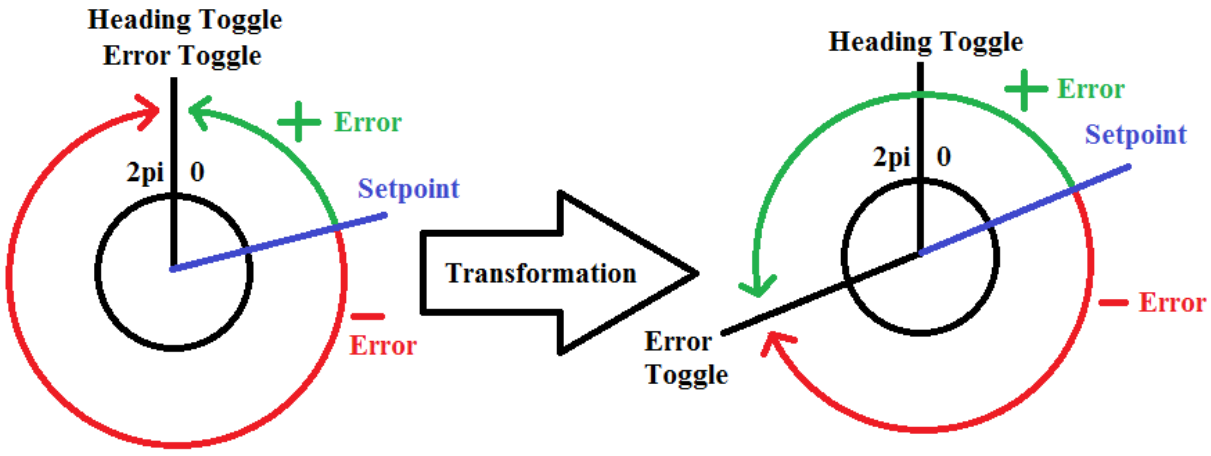

Figure 5. Heading Error Transformation

This transformation of the error solves both problems:

1. Because the toggle-point is opposite the setpoint, the robot never passes through the toggle point when turning towards the setpoint
2. The robot will always choose the smaller of the two possible turns to make (clockwise or counterclockwise). If the setpoint is exactly opposite to the current position, either choice is equally valid.

This transformation is implemented in the C-code below. Note that the transformation requires the angle setpoint to be in the 0-2pi range, so 2pi modulo math is applied first to put the target heading into this range.

```
// Convert target heading into 0-2pi range
if(ht > TWO_PI) {
    ht = fmod(ht, TWO_PI);
} else if(ht < 0) {
    ht = fmod(ht, TWO_PI) + TWO_PI;
}

// Compute heading PID error
float hError;
float hc = Odometer::heading;
if(ht <= PI) {
    if(hc <= ht + PI) hError = ht - hc;
    else hError = ht + TWO_PI - hc;
} else {
    if(hc <= ht - PI) hError = ht - TWO_PI - hc;
    else hError = ht - hc;
}
```

Figure 6. C-code for Heading Error Computation

Finally, the drive velocity PID controller achieves its setpoint almost entirely via integral control. Because drive voltage is linked directly to steady-state motor velocity with no derivative relationship, applying proportional control causes almost instantaneous changes in velocity and thus rapid unstable oscillation. Derivative control is also minimized because the encoder distances measured between loops (and thus the velocity) are noisy values. Taking the derivative of this noisy signal amplifies the noise further, leading to unstable motor drive voltages and minimal improvement in control. Additionally, more stable damping is provided by friction within the drive system itself. This pure integral control is analogous to ramping up the voltage until the wheels are driving at the correct speed.

2.7 Wall-Following PID Controllers

The wall-follower namespace contains two PID controllers: one for left wall distance, and one for front wall distance. The wall-follower state machine operates under the assumptions that all walls are about perpendicular to each other, and that the robot began perpendicular to these

walls as well. However, small errors in angles between walls and in the initial placement of the robot are always present, and not having a way to correct for them is problematic. If the robot is driving even 3 degrees a-parallel to a wall, over the course of multiple feet, this can cause the robot to either drift towards or away from the wall by an amount sufficient to cause a collision or trigger an incorrect state change.

Thus, when the robot drive along a wall, it has an ideal desired angle (either directly towards +/-y or +/-x), and an offset added to it as computed by the left wall PID controller. This controller takes the sonar distance measurement to the left wall and applies a proportional correction. If the robot is closer to the wall than desired, this small correction slightly steers it away, and vice versa if the robot begins to drift too far away. The sum of the ideal drive angle and the correction is fed as the setpoint into the heading PID controller in the drive system.

It was also found when approaching a wall to the front, if the robot went directly from a forward velocity immediately into a right turn, the robot would slide slightly towards the wall during the turn and sometimes come close to colliding with it. To alleviate this problem, another proportional controller takes the front wall distance as an input and outputs a setpoint for the drive system velocity PID controller. This maximum output of this distance controller is the desired drive velocity when not close to a front wall at all, and the minimum output is zero. A large proportional term is used so that the robot only begins to slow down when it is about 10cm from the setpoint. In all other cases, the controller is saturated, and it is as if there is no proportional control on the velocity at all. This was precisely the desired behavior.

The act of chaining PID controllers in series by setting the output of one to the setpoint of the next is a powerful technique for precise control, but is has important limitations to consider. For one, rigorous mathematical analysis of the higher-level controller is incredibly complex due to the fact that the controller output does not immediately take effect in the physical model. For example, when the front wall distance controller outputs a drive velocity, that velocity is neither instantly nor exactly achieved in the robot due to the fact that it is controlled by a separate PID controller. Each chained controller adds time delay between the high-level desired response and the actual system response. Ultimately, the simultaneous control of four variables: heading; drive velocity; left wall distance; and front wall distance, would be better suited for a single more complex controller such as LQR. PID was chosen for its familiarity, ease of implementation, and fast subjective tuning capabilities.

2.8 Matlab Software

2.8.1 Introduction and Purpose

When dealing with the debugging of large, complex, multifaceted robotic systems, it is essential to be able to answer the question, “What is the robot thinking right now?” Where does it think it is? Where does it think its surroundings are? What are its current tasks and states? While a basic 16x2 onboard LCD display can maybe display the robot’s position and state, it is difficult to read while the robot is moving and particularly difficult to catch a bug if it occurs over a short period of time. The Matlab user-interface implemented solves both of these problems and adds additional useful functionality to the robot. A full visualization of the UI as seen on the laptop control station is shown below.

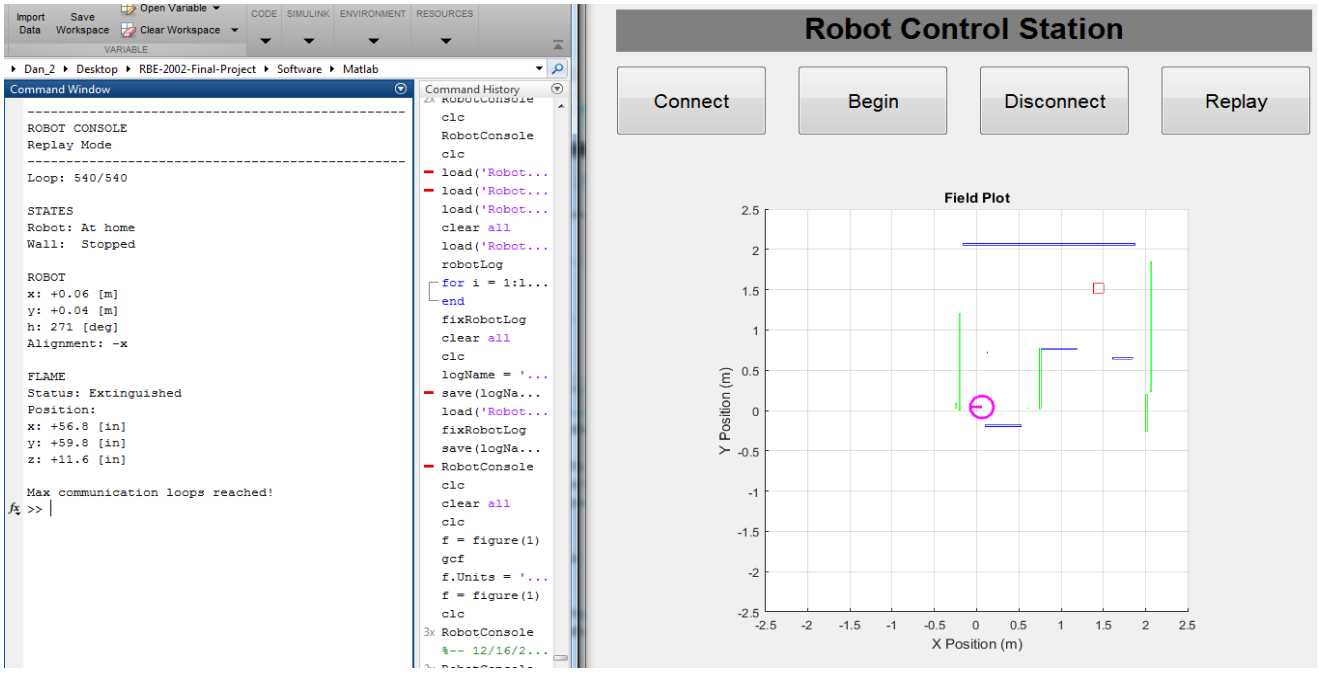

Figure 7. Full Matlab UI as seen on Laptop Control Station

The left side is dedicated to raw text output, including robot state, odometry, and flame position estimation. The right side contains the user-interface buttons and real-time field map. Details on the individual facets of this UI will be discussed in detail in the next sections.

2.8.2 Features and Benefits of Matlab UI

Remote Start and Stop

The UI contains 'Connect', 'Begin', and 'Disconnect' buttons for connecting to bluetooth, starting the robot, and stopping the robot. This means that once the robot is initially places and powered on, it can be operated completely hands-free.

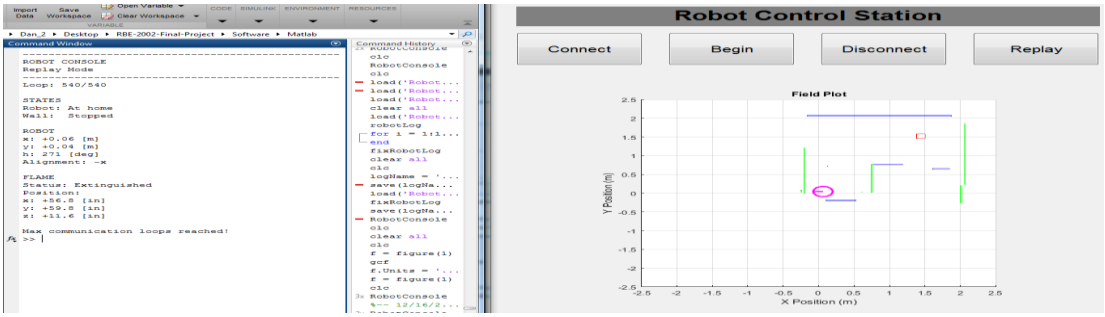

Figure 8. Matlab UI Buttons

Large Data Transfer

Instead of being limited to the 32 characters of an LCD display, the only limit on data sent to the UI for display is the Bluetooth serial baud rate. It was found that a baud rate of 115200 caused frequent communication errors due to dropped bytes. Lowering the baud rate to 57600 completely eliminated this problem. At this rate, sending over the robot states, position, heading, sonar ranges, and flame position (a total of 42 bytes) to Matlab takes less than 6 milliseconds - a negligible amount of time when executed only a few times each second.

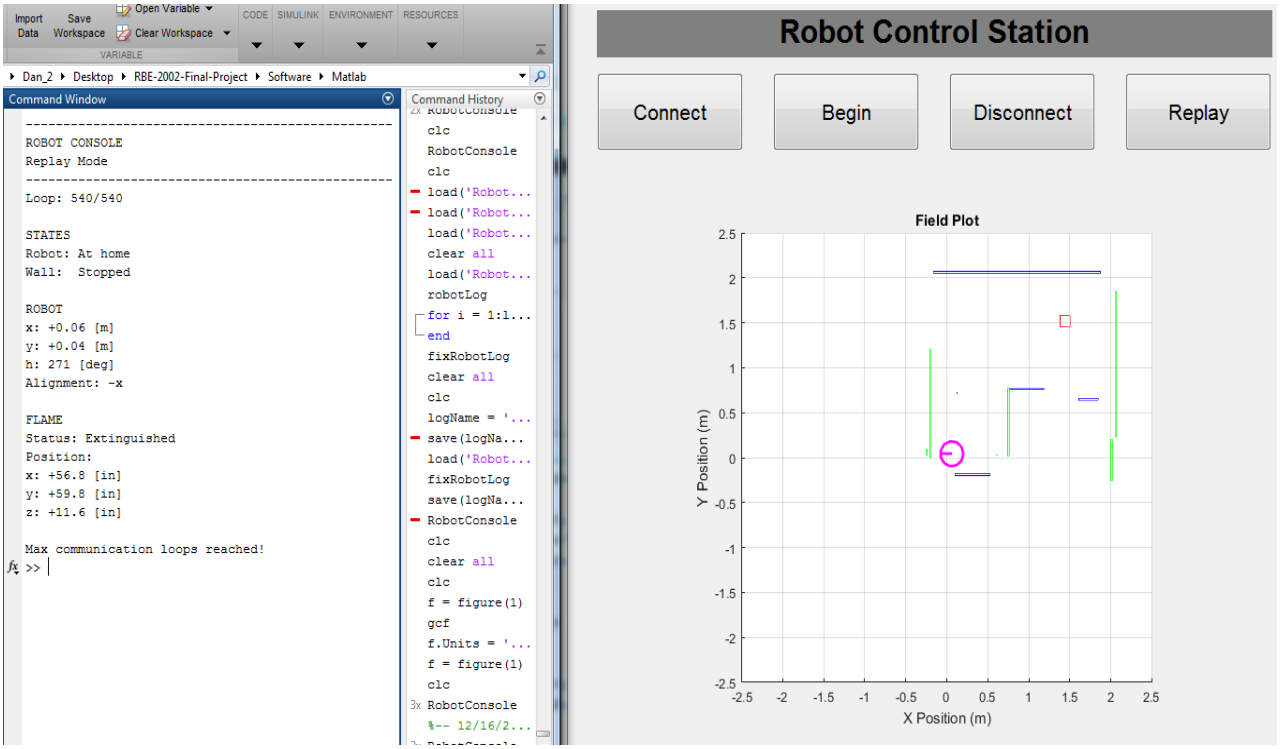

Figure 9. Text Output of Matlab UI

Robot Visualization and Field Mapping

The 2D field plot allows the operator to visualize the robot odometry and sonar. The robot's position and heading estimations as well as sonar ranges are plotted in real time with less than half-second lag. Additionally, the data is used in a basic field mapping algorithm which plots estimations of the locations of walls around the robot as it moves. This algorithm is discussed in more detail in a later section.

Replay Mode

Each time the robot attempts the task in real life, the data from the robot at all times during the run is saved sequentially in a Matlab data file. The 'Replay' button on the UI loads this data and displays it over time the same way it would have during the actual field test. This allows the operator to run the same robot demo multiple times in a row to aid in bug fixing without the need

to attempt to recreate the bug in real life. This replay technique was also used to gradually improve the field mapping algorithm on only a few robot run data sets.

2.8.3 Field Mapping Algorithm

The goal of the field mapping algorithm was originally to perform SLAM and correct wheel slippage using sonar. However, after the transfer from the 4-wheel to the 3-wheel drive base, wheel slippage became negligible, and instead the mapping was simplified and adapted for use as a simple visualization tool. Due to the simple layout of the field, the map approximation is simply a list of walls aligned to the x- and y-axes. Both fall under the parent Matlab class <SonarWall>. The <SonarWallX> and <SonarWallY> classes inherit from this base class and represent walls aligned parallel to the x- and y-axes, respectively. X-walls consist of a y-coordinate and minimum and maximum x-coordinates, and Y-walls consist of an x-coordinate with minimum and maximum y-coordinates. Depending on the alignment of the robot to the field, points from the front & back sonar are either added to X-walls or Y-walls (or neither), and vice versa for the left and right sonar points. If a point doesn't fit any existing walls, a new wall is created from the point. Walls are removed from the list over time if they are too small or contain too few points. The net result is often a very good approximation of the shape of the field.

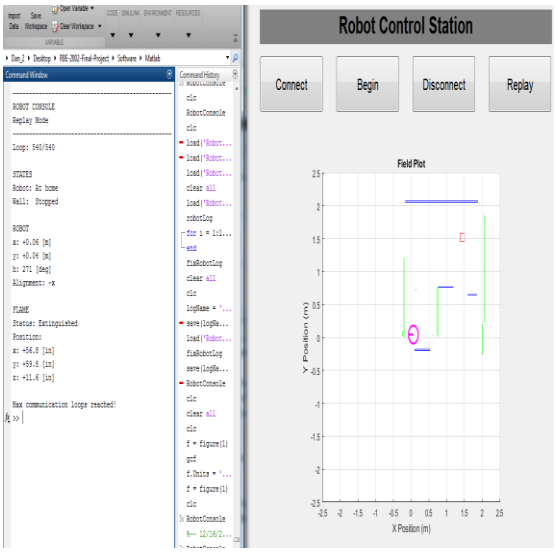

Figure 10. Final Field Plot from Successful Robot Performance

X-walls are plotted in blue and Y-walls are plotted in green. The candle base position is plotted as a red square, and the robot itself as a magenta circle with the line pointed in the forward-facing direction according to the IMU.

3.0 Analysis:

3.1 Drivetrain

The mass of the robot was around 2.1 kg, which translates to around 20.58 N. Assuming a friction coefficient of 1, this would make the traction the same, around 20.58 N. The stall torque of the drive motors we used was 1.2 Nm, and using 0.035m radius wheels, this would make the maximum drive force around 68.6 N. In other circumstances, having the maximum drive force be greater than the traction of the robot would cause wheel slippage, and while the robot has some wheel slip, it is driven so slowly that there is not enough to cause a problem that cannot be compensated in the code.

$W_{robot} := 20.58\ N$

$\mu := 1$

$Traction := \mu \cdot W_{robot}$

$Traction = 20.58\ N$

$\tau := 1.2\ Nm$ Stall Torque of the Pololu Motors that were used

$r := 0.035\ m$ Radius of the Drive Wheels

$F_{drive} := \frac{2\tau}{r}$ Multiplied by two because of the two drive motors

$F_{drive} = 68.571\ N$ Maximum drive force

$Traction > F_{drive}$

Figure 11. Mathcad equations to find wheel traction

3.2 Flame Sensor

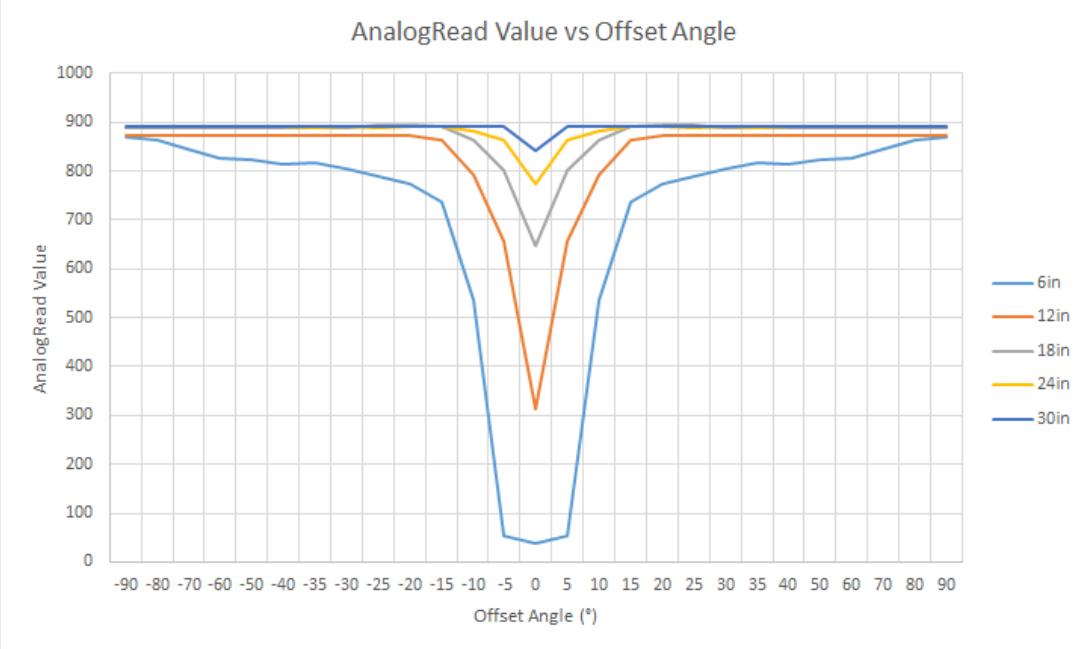

Figure 12. Flame Sensor Graph from Experimental Data Collected

Each of these lines consists of the data at a specific distance. There is no significant enough change above analogRead values of 800, and it was determined that eighteen inches was a sufficient distance from which to identify if there is a flame, so we chose a cutoff value of around 700 to determine if there is a flame. At twelve inches away, this means that a flame can be identified at an offset angle of 10°, and at six inches, a flame can be identified at around 15°, both of which are sufficient for their respective distances.

Extinguishing Mechanism

While the robot drives, the fan assembly with the flame sensor pans between 0° and 90° to the right, and tilts between around -30° below horizontal to around 30° above horizontal, to most efficiently check for the flame. The robot follows the left wall until the flame sensor reads below 700. At that point, the robot stops, and turns it's body towards where it saw the flame. The flame sensor is then panned to find the lowest measurement. The robot turns it's torso such that the front is pointed in the direction of said measurement. The robot drives in the direction of the perceived flame until the front sonar picks up that there is something around 20cm away, at which point it stops driving. The flame sensor pans again, to confirm that it is pointing on the correct z plane. It then tilts, and adjusts itself to the position that it detects the lowest measurement. This is where it has determined the flame is located. The fan spins up, and blows until the flame sensor no longer detects a value of less than around 840, at which point it determines that the flame has been extinguished. The robot reverses the same distance that it went forwards towards the candle, turns back to the wall, and continues following it around the field.

Analytic/Experimental Methods

The initial goal was to use the measurement of the flame sensor to determine direct distance between the robot and the candle, and so a test was conducted that reflected this goal.

Initially, a test was completed to check if the flame sensor was any more sensitive when turned in one direction or another. This test simply consisted of fixing the flame sensor at a certain distance away, and rotating it 90° left and right, at 5° intervals, and comparing the resulting readouts with those at the corresponding angles, but in the other direction. It was found that, at a fixed angle and fixed distance, no matter how the flame sensor was rolled, the output had little to no change. This meant that no abnormalities had to be accommodated in the code, and any data collected from the yaw could also be assumed to be equivalent for the pitch.

3.3 Testing the Flame Sensor

Figure 13. Flame Sensor During Testing

Figure 13 is a photograph of the experiment used to acquire the data that was used to calibrate the flame sensor. Contained within the green circle is the flame sensor, which, when the assembly was oriented at 0°, was pointed directly at the flame contained in the orange circle, represented by the white line. The boxes are taped to the floor next to the box with the flame sensor to make sure when the assembly was oriented at 0°, it was always at the exact same angle, removing the human error that could result in slight differences in the positioning. There is a protractor taped to the front of the assembly, and it was used in conjunction with the duct tape on the floor (accentuated by the red line) to measure the angle at which the flame sensor was pointed at the candle.

There are cross sections, also accentuated by a red line, at six-inch intervals from 6 inches from the candle to 30 inches from the candle. At each of these distances, measurements were taken from 0° to 90° at five-degree intervals.

3.4 Flame Position Calculation

Once the flame has been noticed by the flame finder, the robot turns the front of its body towards the flame, aims the flame sensor at it, and measures the sonar distance to the candle base. The candle base was modeled as a cylinder of average radius 0.07m as a simplifying assumption. The basic geometric model used is overlaid on Figure 14.

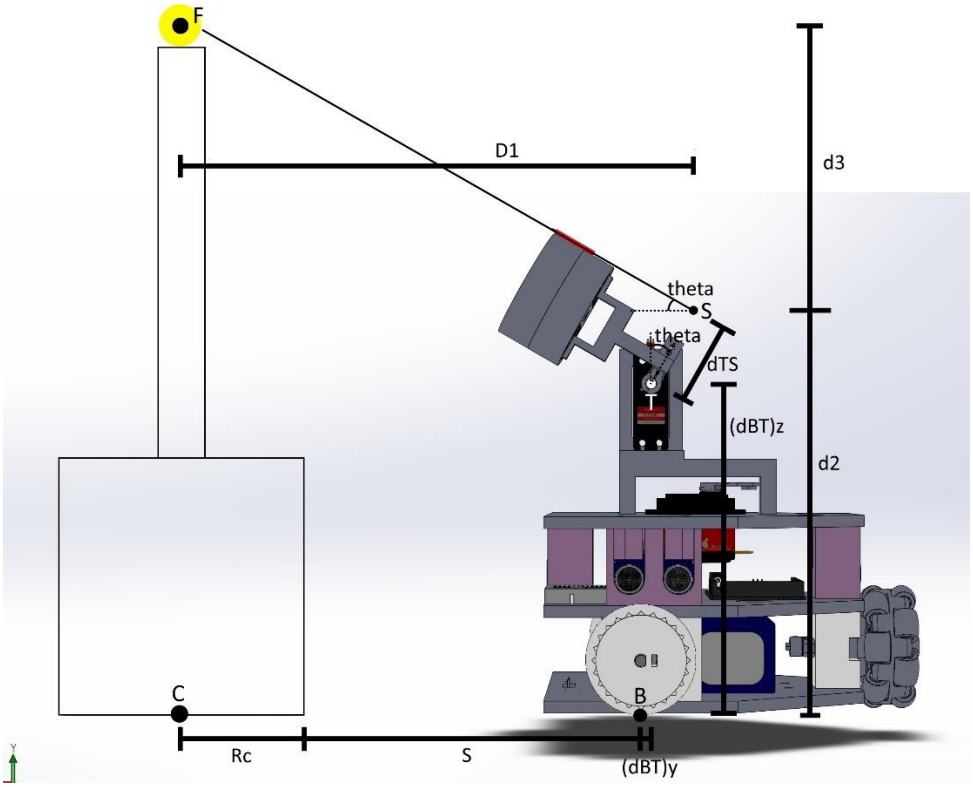

Figure 14. Points Used in Flame Localization

The points in the image are defined as follows:

- C: Center of candle base on ground level
- F: Center of flame directly above point C
- B: VTC of the robot at ground level
- T: Center of tilt servo in plane of points C, F, and B.
- S: Along the line of sight of the flame sensor and forming a 90-degree angle between points F and T

For the theoretical calculation, in order to obtain the (x, y, z) position of the flame, both the above-ground distance and straight-line ground distance to the flame must be calculated. A more detailed mathematical model of the system to make these calculations is shown in Figure 15.

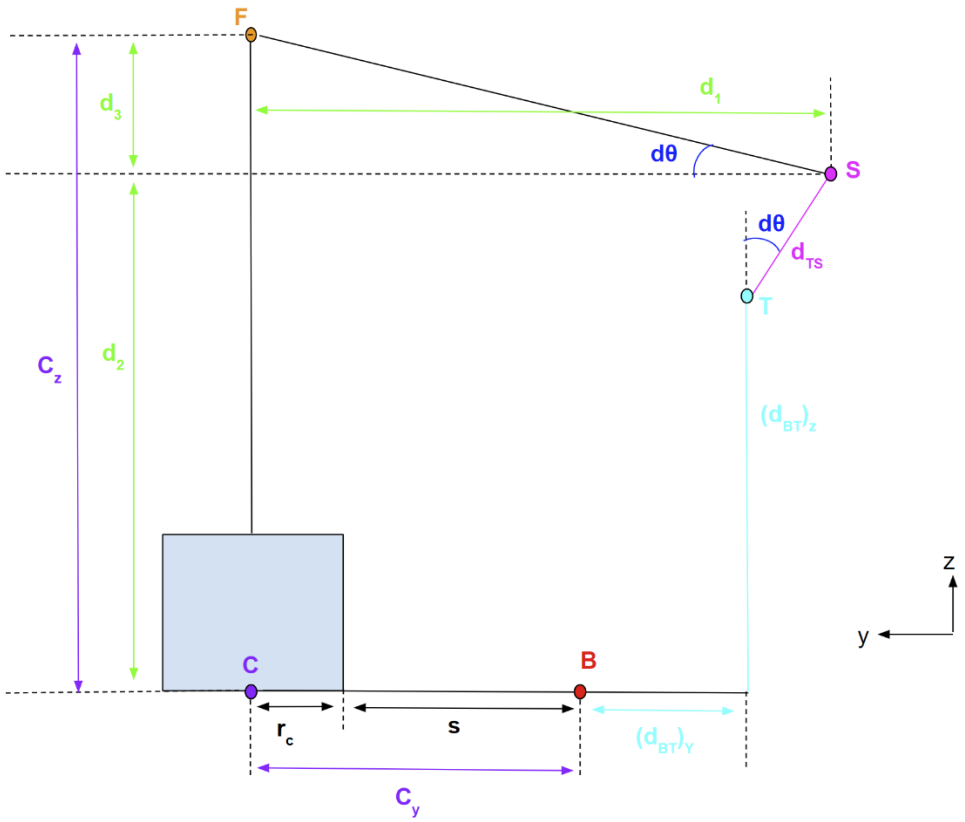

Figure 15. Geometric Model of Flame Localization

The variables are:

- s: Sonar distance measured from candle base relative to VTC at B
- r_c: Average radius of candle base
- $(d_{BT})_y$: Distance from B to T on robot y-axis
- $(d_{BT})_z$: Distance from B to T on robot z-axis
- d_{TS}: Straight-line distance from T to S
- theta: Tilt angle of servo aiming flame sensor directly at flame

Once each of these distances is either known (constant) or calculated, the computation of the horizontal and vertical flame distances cy and cz are calculated as follows:

$$c_y = s + r_c$$

$$d_1 = c_y + \left(d_{BT}\right)_y + d_{TS}\sin\left(\theta\right)$$

$$d_2 = \left(d_{BT}\right)_z + d_{TS}\cos\left(\theta\right)$$

$$d_3 = d_1\tan\left(\theta\right)$$

$$c_z = d_2 + d_3$$

In order to calculate the position of the flame relative to the starting position and orientation, a change of coordinate systems must be made. In the figure below, the coordinate system (x, y) at O is

fixed to the field at the starting position and orientation of the robot, and the system (x', y') is fixed to the robot and rotates with its relative heading h.

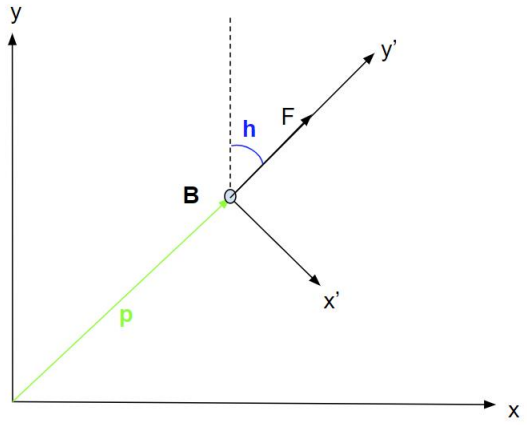

Figure 16. Flame Localization From the Starting Point

Given a known position displacement p and heading h, the flame position (fx, fy, fz) relative to the start can be calculated as follows from the previous calculated cy and cz:

$$f_x = p_x + c_y \sin(h)$$

$$f_y = p_y + c_y \cos(h)$$

$$f_z = c_z$$

3.5 Odometry Algorithm

The odometry algorithm used employs sensor fusion of the encoders and IMU.

The local rotating coordinate of the robot (x', y') centered at the VTC where y' is directed toward the front of the robot and x' is directed to its right. The robot makes small displacement arcs dL with its left wheel and dR with its right wheel about instant center C with instantaneous turning radius R, undergoing a differential angular displacement d-theta in the process. The VTC initially at point B travels in the arc dC to point B', undergoing an equivalent straight-line displacement dp.

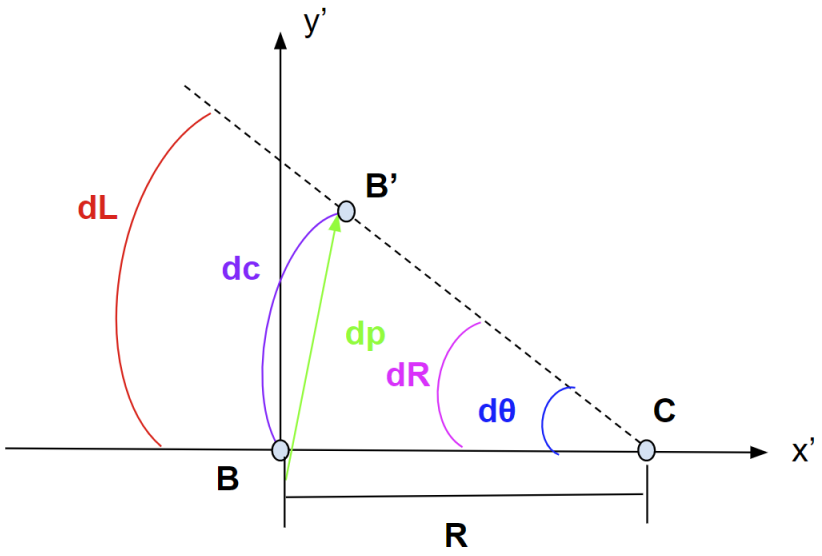

Figure 17. Infinitesimal Local Robot Displacement

Assuming dL, dR, and d-theta are all known, the following computation yields dp:

$$dC = \frac{1}{2}(dL + dR)$$

$$R = \frac{dC}{d\theta}$$

$$d\bar{p} = R\left(1 - \cos(d\theta)\right)i + R\sin(d\theta)\,j$$

Next, consider this displacement in terms of the global coordinate frame (x, y) centered at O. Point O and axis (x, y) are defined as Point B and (x', y') at the instant of robot initialization. Assuming at any given time the robot has already traveled a linear displacement p and angular displacement theta from its original orientation, the following figure shows the local displacement in (x', y') as seen from the global frame (x, y).

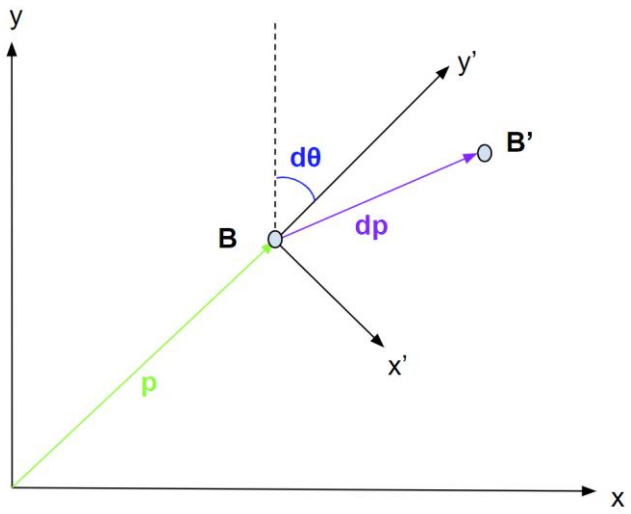

Figure 18. Robot Displacement in the Global Frame

In order to compute the new displacement, dp must be rotated from (x', y') into (x, y) before being added to p, via the following computation:

$$\vec{p}' = \vec{p} + R_\theta \cdot d\vec{p}$$

$$= \vec{p} + \begin{bmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{bmatrix} \cdot R \begin{bmatrix} 1-\cos(d\theta) \\ \sin(d\theta) \end{bmatrix}$$

Continuous iteration of this algorithm allows for the robot to maintain accurate position information regardless of the shape of the path taken.

3.4 Implementation of Odometry in C++

In reality, the distances dL and dR are measured via angular displacements of the wheel encoders multiplied by the radius of the wheels. The angular displacement d-theta is computed by subtracting samples taken from the Bno055 IMU. While it is possible to compute d-theta from the wheel encoders alone, doing so requires more computation on the Arduino Mega and is subject to drift and wheel slippage, while the IMU heading measurement is immune to wheel slippage and virtually immune to drift over the 5-minute time scale of the demo due to sensor fusion algorithms performed on-board the chip. The necessary vector and matrix operations are handled by a linear algebra library. Below is the C-code of the algorithm:

```
void Odometer::loop() {

    // Get heading and change in heading
    heading = imu.heading() - headingCalibration;
    float dH = heading - lastHeading;
    lastHeading = heading;

    // Get encoder distances since last update
    float dL = MotorL::motor.encoderAngle();
    float dR = MotorR::motor.encoderAngle();
    MotorL::motor.resetEncoder();
    MotorR::motor.resetEncoder();

    // Compute velocity
    float arc = (dL + dR) * RobotDims::halfWheelRadius;
    velocity = arc / velocityTimer.toc();
    velocityTimer.tic();

    // Compute delta position vector
    if(dH == 0) {
        deltaPos(1) = 0;
        deltaPos(2) = arc;
    } else {
        float R = arc / dH;
        deltaPos(1) = R * (1.0 - cos(dH));
        deltaPos(2) = R * sin(dH);
    }

    // Rotate and add delta position vector to position
    float ch = cos(heading);
    float sh = sin(heading);

    rotator(1,1) = ch;
    rotator(1,2) = sh;
    rotator(2,1) = -sh;
    rotator(2,2) = ch;

    position = position + rotator * deltaPos;
```

Figure 19. Odometry Algorithm C-Code

Some additional practical considerations were taken into account when implementing the algorithm in C-code. Note that the variables “arc”, “dH”, and “deltaPos” are equivalent to “dC”, “d-theta”, and “dp” as described in the algorithm derivation. First, on the small chance that the robot undergoes 0 angular displacement (dH) over the brief period, the turning radius calculation would have a divide-by-zero error, resulting in undefined behavior. Thus, if dH is zero, the differential displacement is calculated as if it were a straight line instead of a differential arc. Additionally, temporary variables holding the values of cos(heading) and sin(heading) are used to prevent the computations from being performed twice when assigning the values of the rotation matrix. The encoders are reset on each iteration in order to assure that displacements are measured relative to the previous iteration. Finally, a simple scalar velocity calculation is made by dividing the displacement arc by the time measured between iterations of the odometer. This calculation is used to track and control the robot driving speed.

3.7 Power Analysis

The following table shows the electrical breakdown of the various sensors and motors on the robot. This was done to make sure that the power required by the components never exceeds the supply voltage/current/power.

Component Description	Operating Voltage (V)	Quantity	Unit Nominal Current (A)	Unit Peak Current (A)	Ext Nominal Current (A)	Ext Peak Current (A)	Ext Nominal Power (W)	Ext Peak Power (W)
Arduino Mega	5	1	0.0005	0.0005	0.0005	0.0005	0.0025	0.0025
Motor Driver	5	1	0.0200	0.0200	0.0200	0.0200	0.1000	0.1000
Drive Motor	12	2	0.3000	5.0000	0.6000	10.0000	7.2000	120.0000
Brushless Fan	12	1	0.0000	12.0000	0.0000	12.0000	0.0000	144.0000
Bno055 IMU	5	1	0.0123	0.0123	0.0123	0.0123	0.0615	0.0615
Hc-Sr04 Sonar	5	4	0.0150	0.0150	0.0600	0.0600	0.3000	0.3000
QTR-8 Line Sensor	5	1	0.1000	0.1000	0.1000	0.1000	0.5000	0.5000
Hc06 Bluetooth	5	1	0.0080	0.0400	0.0080	0.0400	0.0400	0.2000

Totals	
Nominal Current (A)	0.80
Peak Current (A)	22.23
Battery Max Current (A)	44.00
Nominal Power (W)	8.20
Peak Power (W)	265.16
Battery Charge (Ah)	2.20
Nominal Life (min)	164.84
Peak Life (min)	5.94

Figure 20. Power Analysis

4.0 Results and Discussion

The robot performed as anticipated. Prior to the run, some problems were encountered with respect to finding the flame. In the old code, the robot stopped once it saw the flame, would then pan up and down to find the height of the flame, and then attempt to blow it out. The issue with this was the xy coordinate would be determined by how far away the flame is from the robot, which means it would be dependent on the flame sensor sensing small changes in the amount of light from the candle. Since the flame brightness changes very little which each centimeter away the robot is, this quickly became a very erroneous way of finding the xy coordinate. Also, since the robot was far away, even the z coordinate would be incorrect by a significant amount. A secondary concern was that the robot was too far away to blow out the flame quickly, and it would often take 10-20 seconds. Another problem with this method is that not only will flame brightness change only a little bit with centimeter changes in distance, but also with the wick size--a large wick will produce a large flame, and a nub of a wick will produce a small flame. Therefore, the distance from the candle cannot be reliably determined from the brightness of the flame, instead all data that determines the location of the flame must be collected during the run. With that in mind, new code was developed so that the robot still pans for the flame in the x, y, and z direction.

In the future, something that could be improved upon is the distance from the point of contact of the drive wheels with the ground and the bottom of the light sensor. Instead of being 4mm, which was enough to traverse the slight raise in height between the two demo tables, it would be more than twice that. Therefore, if the demo board is changed, it would optimally not affect the robot.

As for the cost breakdown, it is as shown in Figure 21.

Part	Category	Source	Quantity	Unit Cost	Ext Cost
Arduino Mega	Controller	Amazon	1.00	$14.99	$14.99
Dual Motor IC	Controller	Pololu	1.00	$29.95	$29.95
12A ESC	Controller	Amazon	0.25	$32.99	$8.25
50:1 DC Motor	Motor	Pololu	2.00	$39.95	$79.90
DS3218 Servo	Motor	Amazon	2.00	$18.93	$37.86
Bno055 IMU	Sensor	Amazon	1.00	$28.35	$28.35
HcSr04 Sonar	Sensor	Amazon	0.80	$9.99	$7.99
QTR-8A Line	Sensor	Pololu	1.00	$9.95	$9.95
Hc06 Bluetooth	Communication	Amazon	1.00	$9.99	$9.99
LiPo Monitor	Communication	Amazon	1.00	$5.99	$5.99
LiPo Battery	Power	Amazon	1.00	$18.09	$18.09
5V Step-Down	Power	Pololu	1.00	$14.95	$14.95
Acrylic Sheet	Mechanical	Amazon	1.00	$21.98	$21.98
TOTAL COST					**$288.24**

Figure 21. Cost Breakdown

5.0 Conclusion

Overall, the robot was a great success. It was able to reliably find the candle, and extinguish it. It successfully reported the (x, y) location of the candle within four inches, and was easily able to return home. It followed the wall very smoothly, due to the use of multiple interconnected PID controllers, and when the robot came upon a cliff, it was able to successfully navigate beyond it without falling off the table. The three-tiered design made assembly debugging simple, and allowed for an attractive final design. The odometry worked immensely well, and was able to reduce wheel slippage such that it made a negligible impact on the final measurements. The mapping algorithm worked quite well, and was able to consistently record a relatively accurate model of the maze. The bluetooth connection between the robot worked well, but would, at times, randomly disconnect for unknown reasons, requiring a restart of the robot to fix. In the end, we were unable to successfully detect the location of the flame on the z axis, but that was the only part of the original design that did not work. With a little more time, the problems with the z coordinate locator could have been ironed out, to result in a robot that works completely perfectly.