Multi Robot Navigation and Mapping for Combat Environment

Senior Project Proposal

By:

Scott Tipton

&

Nick Halabi

Project Advisor:

Dr. Aleksander Malinowski

Date:

May 12, 2010

Project Summary

The Multi Robot Navigation and Mapping for Combat Environment project will safely enable a robot to navigate through an indoor or outdoor urban combat environment. The first robot, which would be inexpensive or expandable, will be in charge of mapping the environment and any obstacles or dangers. The second robot, which represents supply caravans or troops, will then use the map generated by the first robot and use a path finding algorithm to determine the best path through the environment that avoids all obstacles and threats. The overall goal of this project is to guide autonomous supply caravans or troops safely through a combat zone.

Detailed Description

The robots that will be used for this project are the Pioneer 3D-X series robots. The 3D-X model has 8 sonar sensors in the front and sides that can get readings from + 90 degrees to -90 degrees. It has two wheels up front that allow the robot to move in any direction and an additional wheel in the back for stability. Each robot is connected to a laptop via a USB port. The laptop runs the actual C++ program (created in Visual Studio) that will control the robot. A program called ARIA then interfaces the C++ program with the robot. Our program is divided into two modes, one for each robot. Mode 1 is responsible for mapping the unknown/combat environment and relaying that map to the server via a laptop with a wireless network connection. Mode 2 is responsible for retrieving the map through a wireless laptop connection, and safely navigating through the environment. Another PC is connected to the server in case manual override would ever be required. Additional sensors (IR and metal detection) will be added later and connected to a Silicon labs 8051 microcontroller, which will then send the information to the ARIA program.

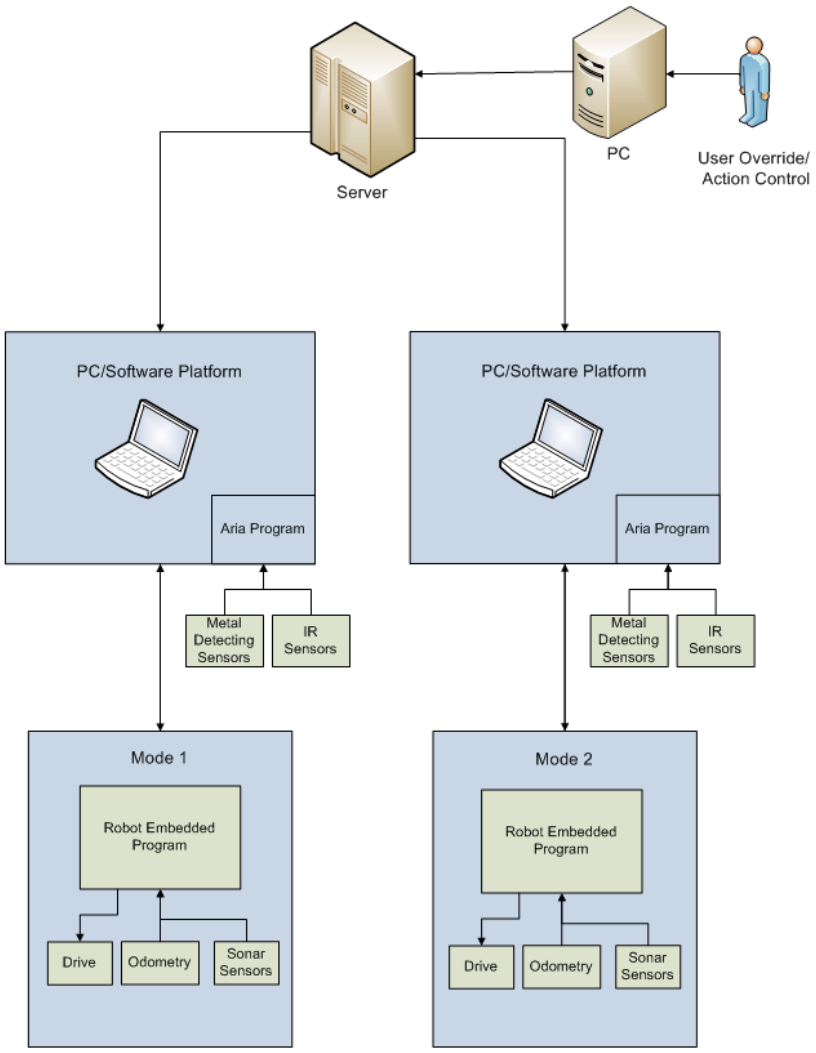
Current Project Goals

- Robot Navigating
    - Find and travel to closest wall/object
    - Position robot in a specific position to wall(s)/object(s)
    - Left/right wall following
    - Determine if sensors more accurate than sonar sensors will be necessary. If so, integrate the sensors into ARIA if possible.
    - Identify appropriate sensors for combat-like environment.
    - Acquire and integrate sensors for simulated combat-alike environment (metal detector?)
    - Develop communication framework, allowing server/central command to override local control algorithms and remote control robot
- Environment Mapping
    - Research and develop algorithms to map an unknown environment

- Research available ARIA or Pioneer robot compatible software for mapping
- Develop framework to contribute maps to server/central command and update maps from the server
- Research and develop algorithms to locate robot by matching its current map with the global map available on the server/central command

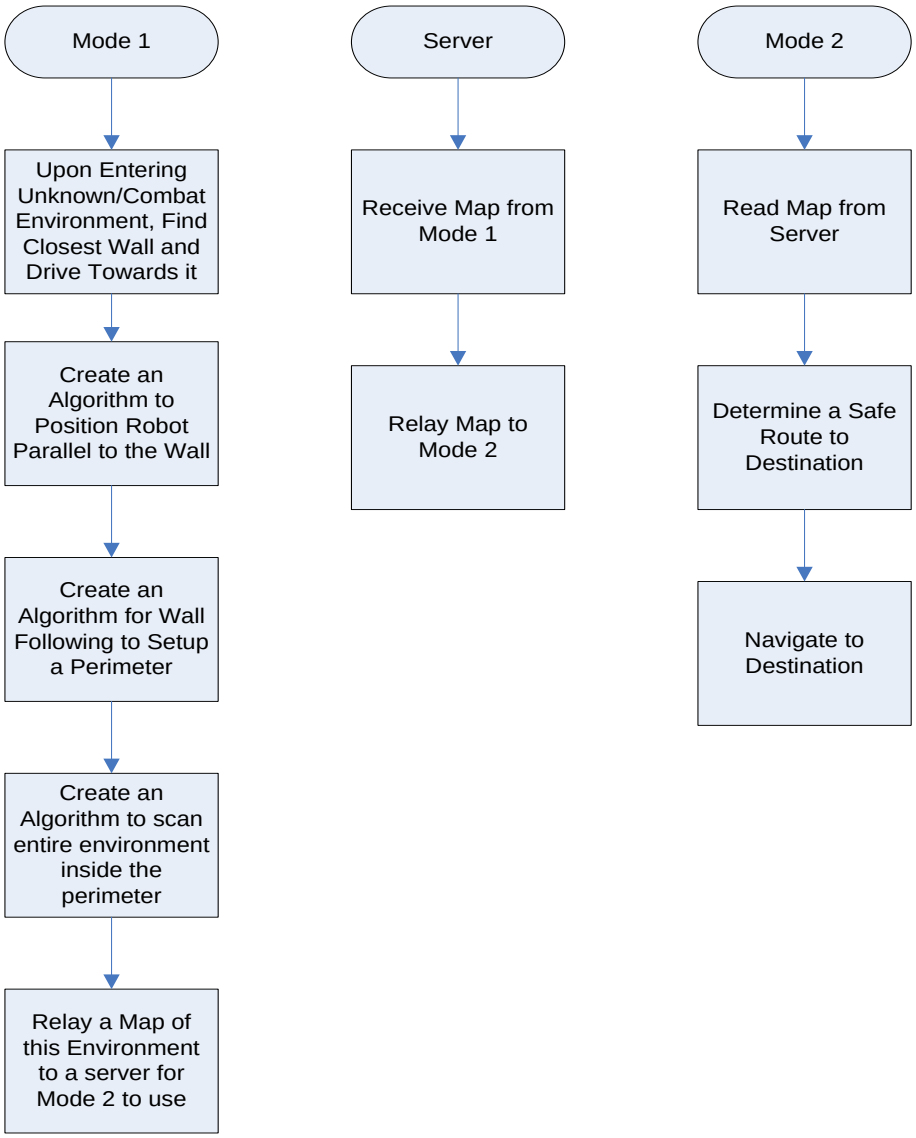
- Other Tasks
  - Create digital maps of the real-life-alike environments for computer simulation
  - Setup the infrastructure server for multiple robot cooperation/coordination
  - Weekly website update on project progress

Figure 1: High Level System Block Diagram



The high level system block diagram is divided into three main subsystems: mode 1, server, and mode 2. The first subsystem is the mode 1 subsystem. This subsystem's prime responsibility is to map an unknown/combat environment and send the map to a central server for storage. The second subsystem is the server subsystem which receives and stores the map from mode 1. When the map is completed, the server then sends that information to mode 2. The last subsystem is the mode 2 subsystem, which reads the map sent from the server. Based on the map received, mode 2 determines a safe route to the destination and then navigates through the environment. These three subsystems are all explained in better detail in Figure 2.

Figure 2: Subsystem Block Diagrams



Software

Mode 1: Wall Following

Mode 1, or wall following, was the first subsystem to be created for this project. To start, we had the robot use all eight sonar sensors to find the closest wall within 180 degrees ahead. The robot would then turn 180 degrees and scan again to see if there wasn't a closer wall behind the robot. Once the closest distance was recorded, the robot would turn in a counter-clockwise direction until the front two sonar sensors found the exact distance within plus or minus 2%. When the robot was finished turning to the closest distance, it would then drive in that direction until the robot was within 600 millimeters of the wall. At this point the robot would turn parallel to the wall and begin its wall following algorithm.

Dr. Malinowski required that our robot stay at least 600mm away from the wall to ensure accurate sonar sensor readings which would be needed to create an accurate map of the environment. Our first attempt at wall following just had the robot rotate itself a certain number of degrees based on how far away it was from the wall. For example, if the robot was at 800mm at one point in time and 1100mm at another point in time, the robot would turn more at the 1100mm distance to correct itself. This code was all completed with a large amount of if and else if statements that used distance as a determining factor for how much to turn. This code was later revised, and the 16 conditional statements were replaced with an error formula which can be seen below.

```
error = (((xr90 + ideal_distance)/2)-xr90);
if(xr50 <= (xr90+.3*xr90) && d_diff<50) angle_v=3;
else if(xr50 >= (xr90+.3*xr90) && d_diff>-50) angle_v=-3;
else if (error > 100){
angle_v = .025 * error;
slow_speed = .75 *speed;
}
else {
angle_v = .025 *error;
slow_speed = speed;
}
```

The first line in the formula determines the error which is related to how far away the robot is from its desired location of 600mm from the wall. The second and third lines check the sonar sensors at 90 degrees and 50 degrees and compares their values to one another. These lines were necessary to prevent the robot from over-correcting itself on faster turns which would cause a great deal of oscillation along the wall. Angle_v was the angle velocity of the robot which determined how fast the robot would turn. The error would be multiplied by a constant (.025) to set the speed that the robot should turn. This effectively implemented proportional control on the robot. Also, if the error was

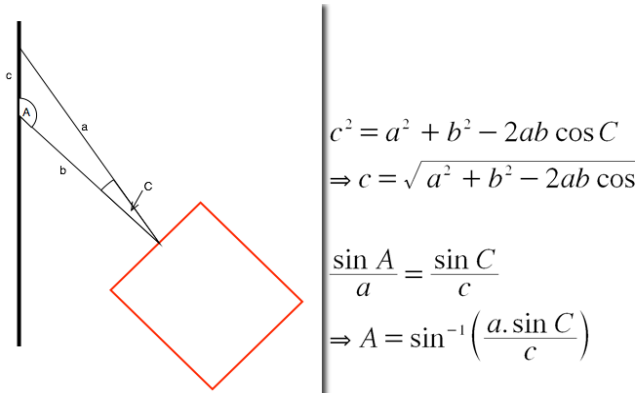
greater than 100, the robot slowed down to 75% of its normal speed to help prevent over-correction. In the case where the 90 degree sensor didn't detect a wall or was too far away from the wall it was following, the robot would turn 90 degrees over a period of 3 seconds and then head straight for 2 seconds unless a wall was up ahead in which case it would turn and face parallel to the wall. This allowed the robot to turn into other rooms and to correct itself with very little oscillation if for some reason it got too far away from a wall (ex. wall suddenly drops off). A picture of an environment mapped with this algorithm can be found in figure 3. Note that on the map, blue represents the robot trail, green represents an area the robot hasn't been before but because of the sonar sensor readings it is known to be clear, orange represents obstacles, and black is an unknown area.

Figure 3: Initial Wall Following

One of the problems with the map shown in figure 4 is how the wall following algorithm handles ramps or slight depressions from the wall. If a ramp or temporary depression occurs, the robot oscillates for a while until it can eventually reach stability provided it is traveling along a straight wall. Up to this point, most of our environments were rectangular with straight lines and very few slopes (as in figure 3). Our current wall following algorithm could handle straight lines and turns well, but it would oscillate more on slopes which would make our map of the environment less accurate.

In order for our robot to safely map more complex environments, our wall following algorithm had to be improved to better handle slopes and wall depressions. The improved wall following algorithm that we implemented was developed by Mike Mensinger for Dr. Malinowski's robotic navigation course. The new algorithm was basically derived from figure 4 which uses the law of sines and cosines to determine the angle A, the robot needs to turn to position itself parallel with the wall.

Figure 4: Robot Trigonometry



In figure 4, variables a and b were replaced with the robot's 90 degree and 50 degree sensors. The necessary angle to turn was then calculated in figure 5 below. Turnang (turn angle) was then set to 50 - turnang to place the robot parallel with the wall. The angle velocity of the robot (how much to turn) was then set to negative turnang plus 1. The negative part was added because the robot software has the x-plane in the reverse direction. Plus 1 was added through experimentation since it appeared to make the robot more stable.

Figure 5: Final Wall Following Algorithm

$$c = \sqrt{xr90^2 + xr50^2 - (2 * \text{xr90} * \text{xr50} * \cos(\frac{40\pi}{180})}$$

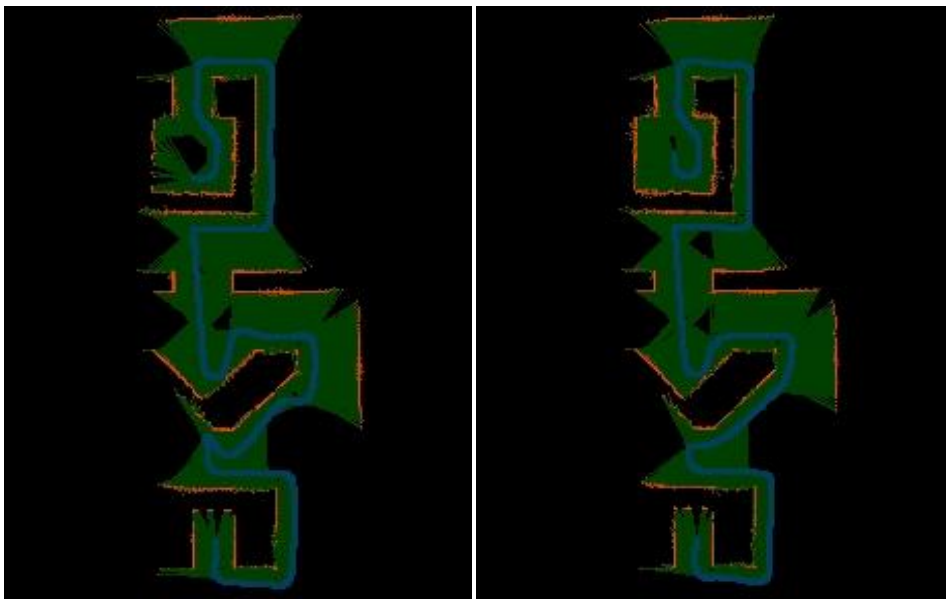
$$Turnang = \frac{\sin^{-1}(\frac{xr90}{c} * \sin(\frac{40\pi}{180}))}{\pi}$$

$$Turnang = 50 - Turnang$$

$$Angle\ velocity = -Turnang + 1$$

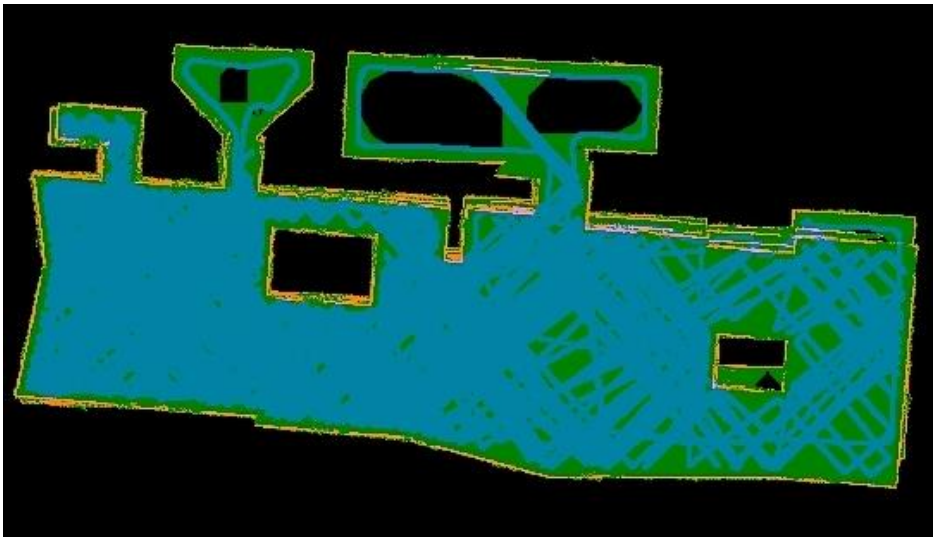
The maps in figure 6 were used as a test course for the two wall following algorithms to prove that the newly designed algorithm was better at navigating through non-standard environments. In figure 6, the map on the left hand side demonstrates the initial wall following algorithm, while the map on the right shows the newly developed algorithm. In areas where the angle of the slope is high, the initial wall following algorithm will over correct itself and take longer to stabilize as opposed to the new method.

Figure 6: Initial and Final Wall Following



After the wall following algorithm was optimized, the interior of the environment needed to be mapped. While the robot was mapping the parameter of the environment, it would check to see if it's ever been at its current location before. If the robot detects its already been at a certain location before (meaning it reached its start point again), the robot would then randomly navigate the rest of the environment unless it runs into an unknown obstacle. If this occurs, the robot will navigate around the obstacle and then randomly navigate throughout the environment again once it is finished mapping the obstacle. To test this process out, we ran the robot during the night and generated the map seen in figure 7.

Figure 7: Environment Completely Mapped



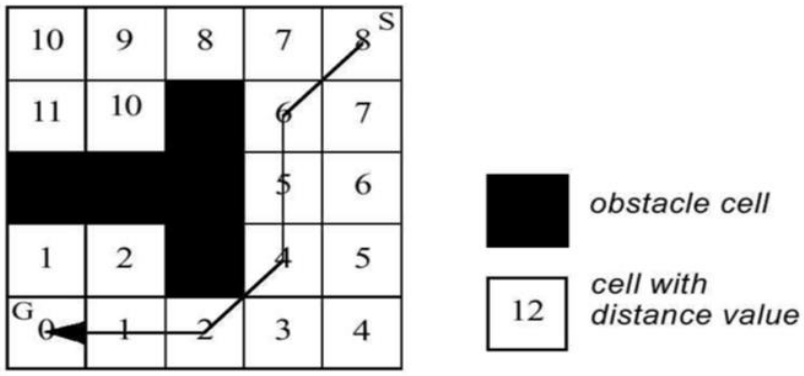
In figure 7, the a complete map of the environment can be seen including obstacles not on the parameter. Overtime the map becomes distorted or rotated due to wheel slippage since the robot only determines its current location based off its onboard odometer.

Mode 2: Path Planning

For mode 2, we developed two path planning methods for navigating to a destination while avoiding all obstacles. The first method that was developed was the grassfire technique which when given a complete map of the environment, can guarantee the shortest trajectory to the goal point.

With the grassfire technique, the goal position is the lowest number with wavefront expansion from the goal position outward. This wavefront expansion marks on each cell its distance to the goal. For a true grassfire technique, this process would continue until the starting position was reached. However, for this project, every cell is numbered even after the starting point has been reached. From the starting point, the robot will then continue to go to the next lowest numbered cell until the goal position is reached. Obstacles are set to a value of 32,000 to ensure they will never be the next lowest cell and therefore will always be avoided. See figure 8 below for an example of how the grassfire technique works.

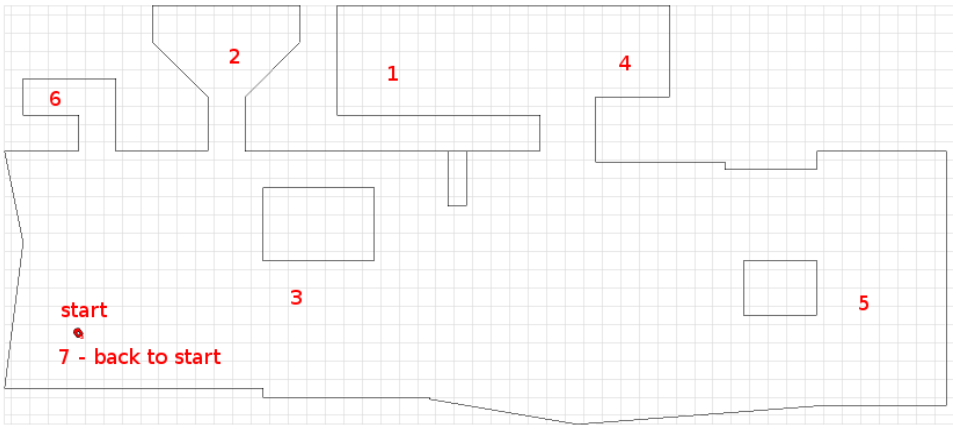
Figure 8: Sample Grassfire Approach



Note: for this project we modified the grassfire technique from the picture above and changed the goal coordinate to a 1 value, and unreachable areas to a 0 zero value.

To test the newly developed grassfire path planning technique, we created a test course on an environment where we had completely mapped the parameter and all obstacles in the interior. Figure 9 shows the test course we created.

Figure 9: Grassfire Test Course



The grassfire algorithm would have to navigate to each of the numbered destinations starting from the start point and then finishing back at point 7 (start point). With the way the grassfire technique is currently setup, the robot will go to its first destination (destination 1) and stop within four cells of the actual goal point. There it will wait for the user to enter in the next coordinates. Figure 10 shows the generated map from the simulation we ran on the test course.

Figure 10: Grassfire Path Planning

From figure 10 above, it is clearly evident that the robot traveled to each destination in the shortest distance possible without crashing into any obstacles. It also returned to its origin and awaited there for further instruction.

It's worth pointing out that the grassfire technique was used for this project because of the relatively low computational complexity of path planning compared to other methods. The main disadvantage of this method is the amount of memory that is used to create and store the grid. For large environments, the grid must be represented in its entirety. However, due to the falling prices of computer memory, this isn't really a concern for our project.

The second path planning technique that we developed for this project was the potential field algorithm. With potential field path planning, you have attractive forces such as a goal point, and repulsive forces such as objects. The attractive force equation used for this project was found in the course book, "Introduction to Autonomous Mobile Robots" as shown in figure 11. *Katr* is a positive scaling factor and *x-xgoal* and *y-ygoal* are the x and y distances from the robot to the goal point.

The repulsive force equation was also found in our book, but wasn't derived with respect to x and y like the attractive force equations were. This derivation is necessary since our robot is operating in the x and y plane and needs attractive and repulsive forces from both x and y directions to correctly navigate. We went ahead and derived the repulsive equation in MatLab with respect to x and y and obtained the repulsive force equations shown in figure 11. In figure 11, *krep* is a scaling factor and *dist* is the distance from the robot to the obstacle defined as *dist=sqrt((x-xo)^2+(y-yo)^2)*. Here, *xo* and *yo* are the coordinates of the robot and x and y are the coordinates of the obstacle. Minus Rradius (robot radius) was later added to increase the distance of the repulsive force to decrease the chance of the robot crashing into an unknown obstacle or wall.

Figure 11: Potential Field Forces

Attractive:

$$double\ ddx = -Katr * (x - x_{goal})$$
$$double\ ddy = -Katr * (y - y_{goal})$$

Repulsive:

$$dx = Krep\ (x - xo) * \frac{Rrep_{inv} - \frac{1}{\sqrt{dist}}}{(dist)^{3/2}}$$

$$dy = Krep\ (y - yo) * \frac{Rrep_{inv} - \frac{1}{\sqrt{dist}}}{(dist)^{3/2}}$$

$$dist = \sqrt{(x - x_o)^2 + (y - y_o)^2} - Rradius$$

With the potential field equations in place, we ran a simulation where we set a goal point that had a single obstacle in the way (see figure 12). From the map shown in figure 12, you can clearly see that the robot is attracted to the goal coordinate and then is immediately repulsed once it detects an obstacle in the way with the sonar sensors. However, once the robot navigates around the obstacle, it then reaches its destination and stops.

Figure 12: Potential Field Working

Even though the results in figure 12 look promising, one of the largest drawbacks of potential field path planning is the occurrence of local minimas which appear when multiple repulsive forces neutralize the attractive forces, causing the robot to either come to a standstill or circle indefinitely in a certain region (see figure 13). When a robot reaches a local minima, it's attractive forces are reduced to zero, giving the appearance that it has reached the lowest part on the map (goal). For this project, to determine whether the robot was trapped in a local minima, we relied on the odometry of the robot as well as its current speed which we denoted as velocity coefficient. Once detected, we then temporarily deploy our grassfire technique to extract the robot from the local minima.

Figure 13: Local Minima

To detect whether the robot is in a local minima as opposed to the goal, we check the robot's current velocity coefficient and compare that with its current destination value. If the velocity coefficient is less than four, meaning it has slowed down considerably, and the odometer on the robot verified it hadn't reached its goal yet, then a local minima had

been detected. Also, in the case where the robot doesn't slow down, and continues to loop in a circle indefinitely, we would check the odometer on the robot to determine whether the robot was making progress towards the goal. We created a counter called old_value that would return to zero if the robot had traveled a certain distance to the goal. If for twenty five consecutive times the current distance value of the robot is greater than or equal to old_value minus ten or less than or equal to old_value plus ten, then a local minima has been detected. We also added a condition where the velocity coefficient has to be above five, so that when the robot is closer to the actual goal and slows down considerably, the robot won't think it is in a local minima since it hasn't moved far in the last twenty five cycles.

After a local minima is detected, we would switch all control of the robot over to the grassfire approach. With the grassfire approach deployed, the robot would be able to navigate out of the local minima regardless whether or not a complete global map of the environment was provided. While the grassfire approach is running, we also run the potential field technique in the background since the grassfire approach can't determine when it has exited a local minima. When the potential field has determined that for thirty consecutive cycles it has continued to get closer to the goal, the grassfire approach exits and all control is returned back to the potential field technique. From here on out the potential field path planning technique will continue to the goal unless another local minima is detected in which case the entire process will start all over again. In figure 14, the robot escapes from the first local minima depicted in figure 13, and then enters and leaves another local minima before reaching its destination.

Figure 14: Local Minima Correction

With the potential field and grassfire techniques both fully operational, it's worth pointing out why we decided to implement them both into this project as opposed to just sticking with one. The grassfire technique works best in a situation where you have a complete map of the environment with no or very little changes occurring in real time. With a complete map the grassfire approach will always guarantee the fastest time to the goal

point since it will take the shortest path possible. However, if a complete map of the environment is not available, anytime the robot encounters an unknown obstacle, it will have to recompute the entire map at every unknown point along that obstacle until it eventually passes the obstacle.

The potential field technique, on the other hand, doesn't read a map of the environment but instead calculates forces on the fly whenever the sonar sensors pick up obstacles along the way. The potential field in most cases won't take the shortest trajectory, but it will usually reach a given destination faster than the grassfire approach if a map of the environment isn't given or is incomplete. The potential field technique is also faster at maneuvering around obstacles that suddenly appear in the robot's trajectory.

Overall, both path planning techniques have their advantages and disadvantages. We decided to keep both techniques for this project because depending on the situation, one technique will have a clear advantage over the other. Since this robot is designed to be as safe and efficient as possible, multiple path planning options is a necessity for this project.

Server/Central Command:

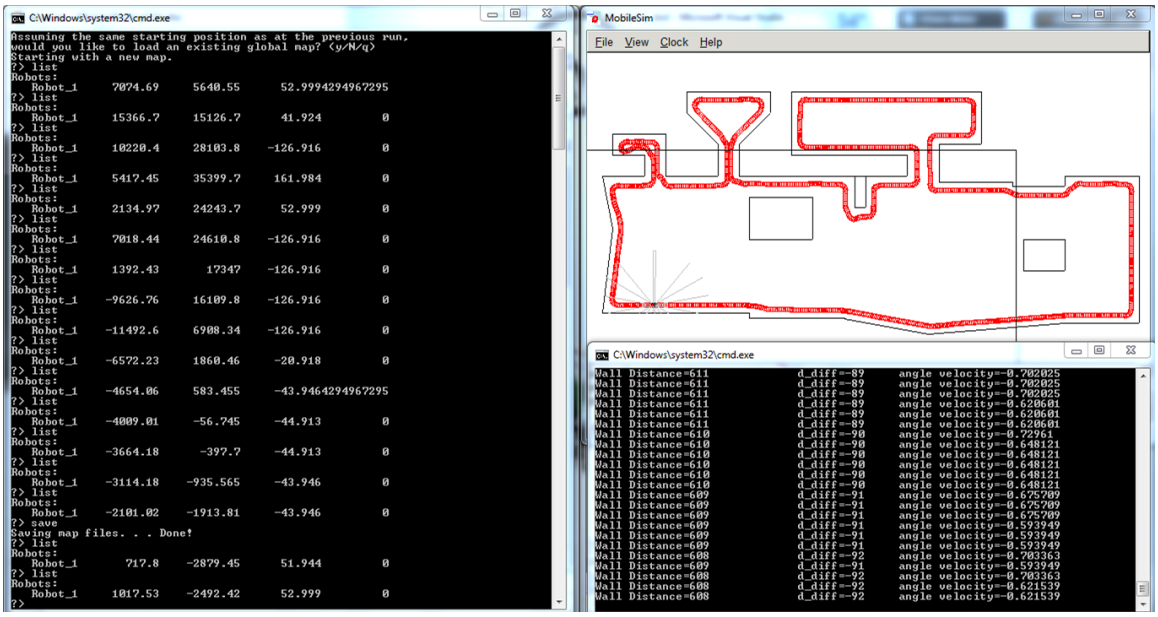
The last software subsystem that was created for this project was the server framework/central command. This framework allows to setup a centralized server where one robot could scan an environment and create a map, while another robot could then use that map to safely navigate to a destination within that same environment. In setting up the server we used Dr. Malinowski's initial multicast framework program as a base and added options and features on top of that.

The server is able to view all the robots connected to its network and their current status which consists of x, y, and theta coordinates and how many seconds it has been since the last good connection. The server can save the map at any time by simply typing the word "save" into the terminal. The server also has a control mode with five basic options: map environment (wall following), path planning (choose grassfire or potential field), manual override (joystick), random movement, and quit. With the implementation of the server framework came a change in the startup process of a robot. From now own, anytime a robot is turned on, it asks for the robot ID (any name without spaces) that the server can use to identify the robot. The server will then use that robot ID when sending various control commands.

Any control mode command can be turned on by typing "m robotID mode_number." The joystick program was created by Dr. Malinowski and runs in its own thread. To active the joystick program turn the joystick on (j robotID) and then run its control command (m robotID mode_number_for_joystick). The other control commands work in the same way except they don't need to be turned on since they don't run in a separate thread.

When using one the of the path planning techniques, the coordinates have to be entered first or else the robot will just travel to its starting point. To set a goal coordinate, type the following command: "g x_coordinate y_coordinate theta_coordinate." Once the coordinate is set, a user could even manually switch in between the two modes halfway through the travel, and it will still reach its destination. An example of the server in operation can be found in figure 15.

Figure 15: Server/Central Command



In figure 15, the window on the left is the server running which is currently listing the x, y, and theta coordinates of the robot was well as the last time the robot was active (0 means that robot has always been active). The windows on the right show the robot performing mode 1 wall following which the server executed with the following command: "m robotID mode1_number."

With the server infrastructure setup, the software subsystems portion of this project is now complete.

Hardware

Metal Detectors:

The first part of the hardware for this project was building the metal detectors. The metal detectors arrived as kits, and needed to be soldered. The initial metal detector schematic depicted in figure 16, shows that the detectors run off of a 9 volt source. Circuit analysis

was employed to change the circuit to running off of a 5 volt source as shown on figure 17. An optical isolator was also added to the circuit, giving the microcontroller protection from any possible issues emanating from the metal detectors.

Figure 16: Initial Metal Detector Schematic

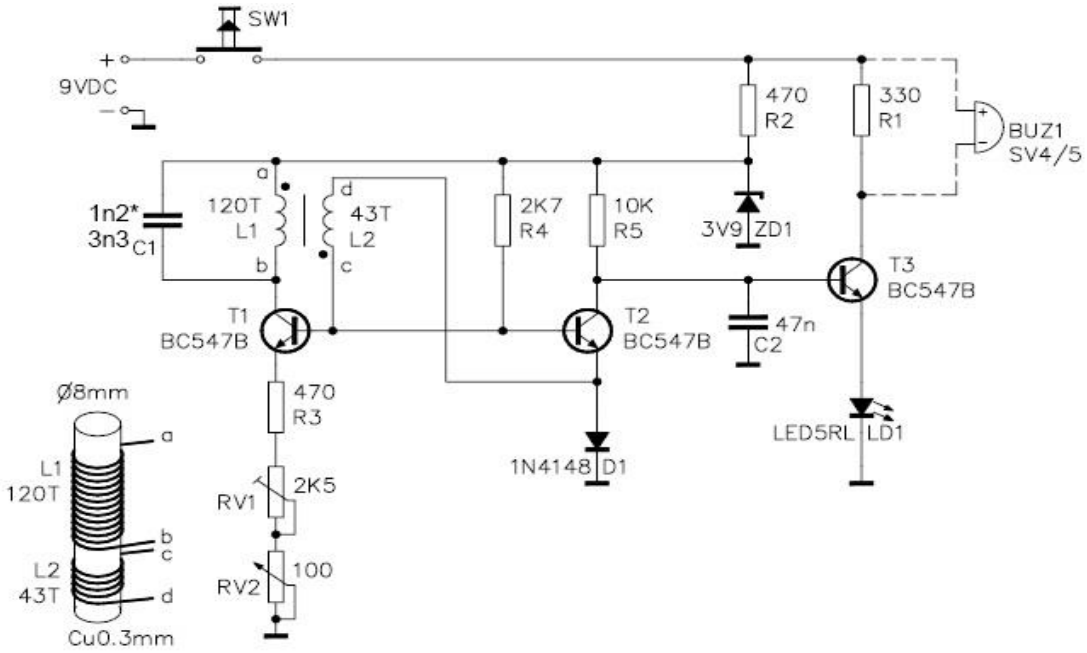
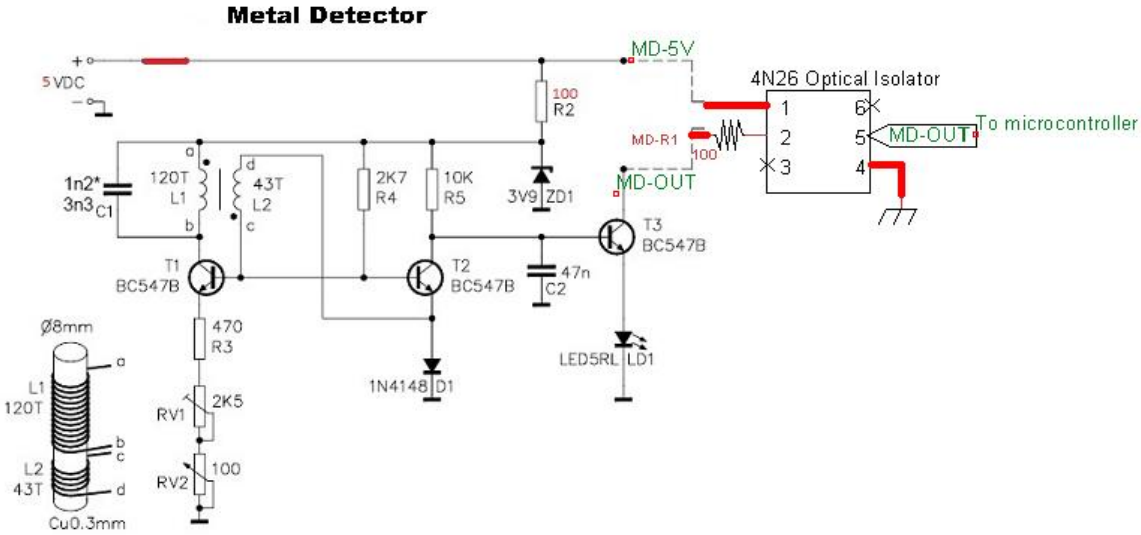


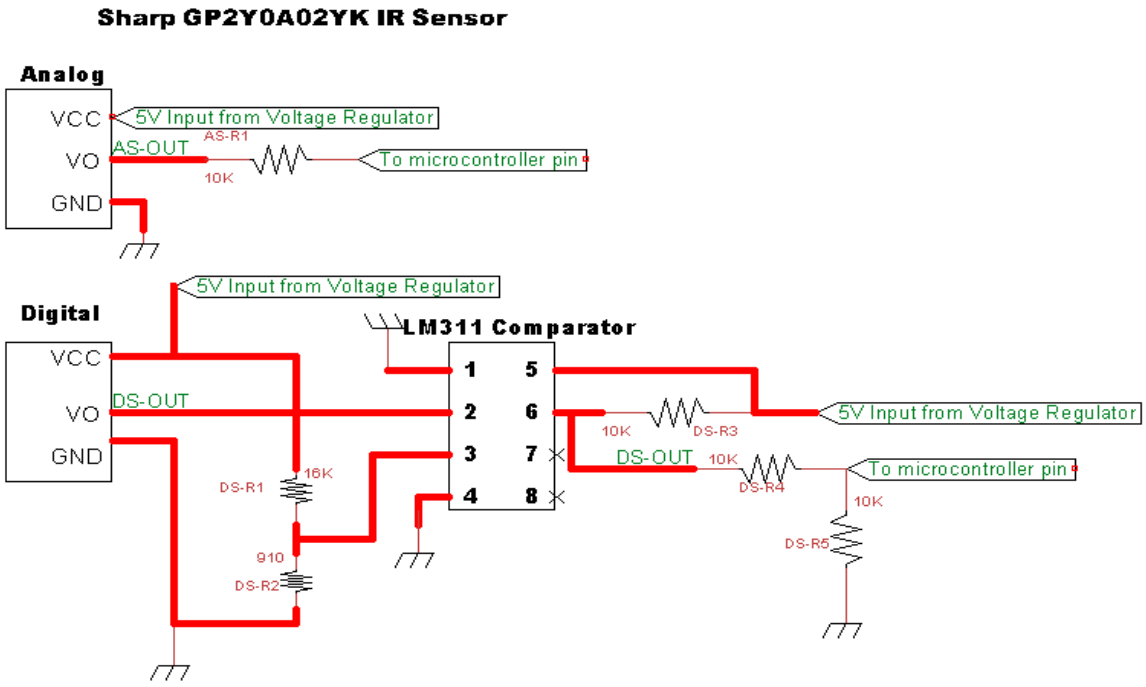
Figure 17: Modified Metal Detector Schematic



Infrared Sensors:

The robot has a total of 14 IR sensors mounted, 7 digital bump sensors, and 7 analog distance sensors. Figure 18 below shows the interfacing of both sensors to the microcontroller. For the analog sensors, a 10K resistor was placed between the sensor and the microcontroller A/D. This was added just for extra protection for the board. As for the digital sensors, their output was sent to the LM311 comparator which placed the voltage in the right range, and acted as a bump sensor. The distance chosen was 4 inches, but this distance can be edited by simply changing the resistor values.

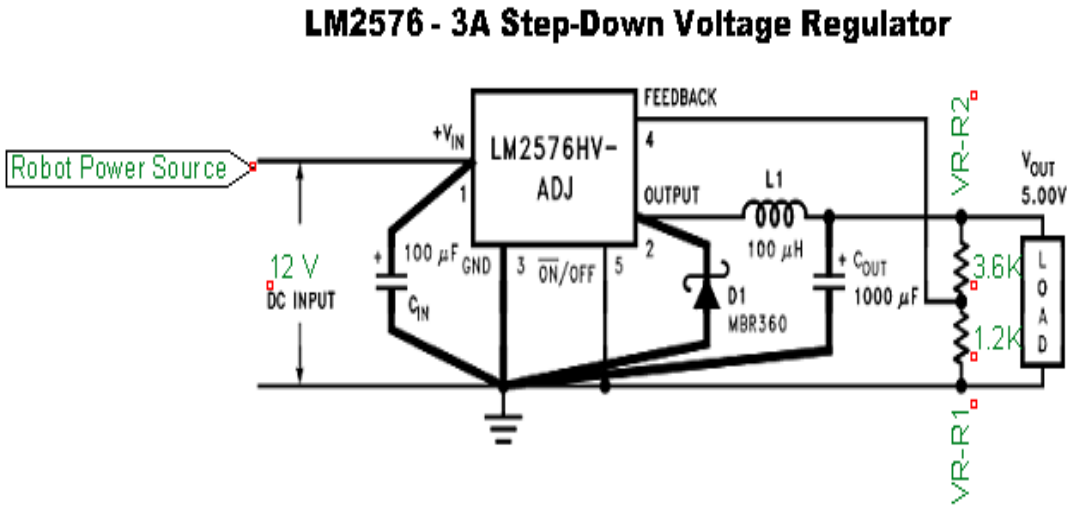
Figure 18: IR Sensors Integration



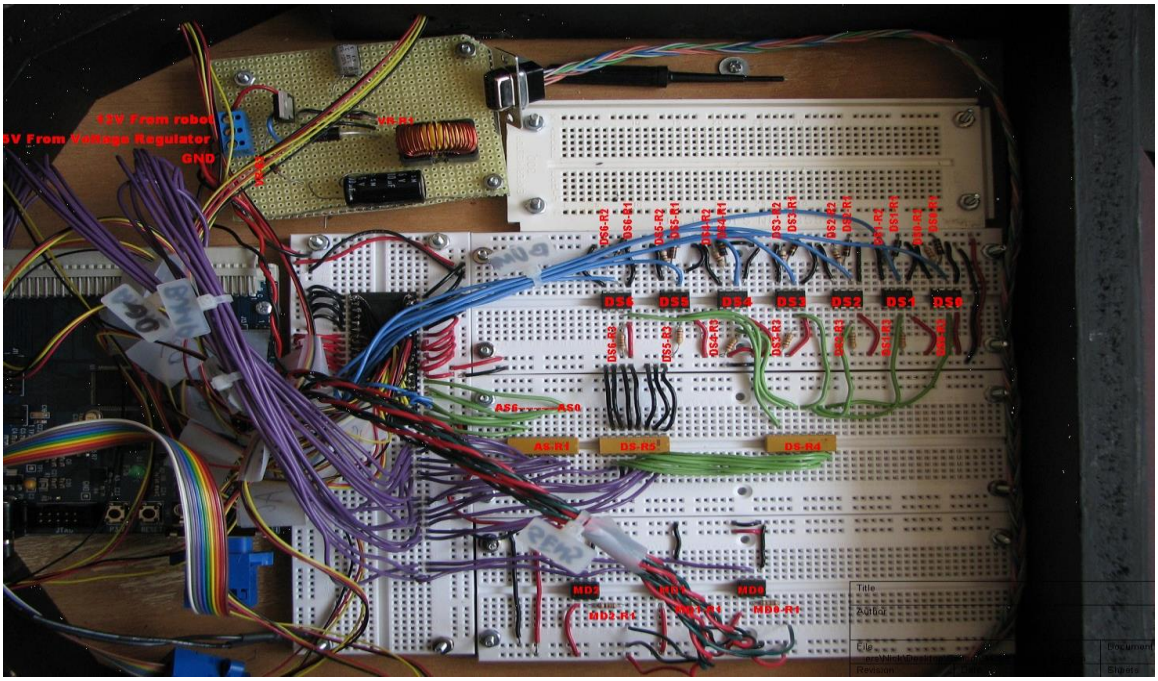
Voltage Regulator:

All of the hardware that was installed on the robot was designed to run on a 5V source. The internal battery on the robot is a 12V source, which needed to be regulated to 5V. The LM2576 Switching Regulator steps down the voltage from 12V to 5V and has a limit of a 3A current. Figure 19 depicts the regulator used for the robot. This regulator can also be adjusted to output values other than 5V, by modifying the resistor values VR-R1 and VR-R2.

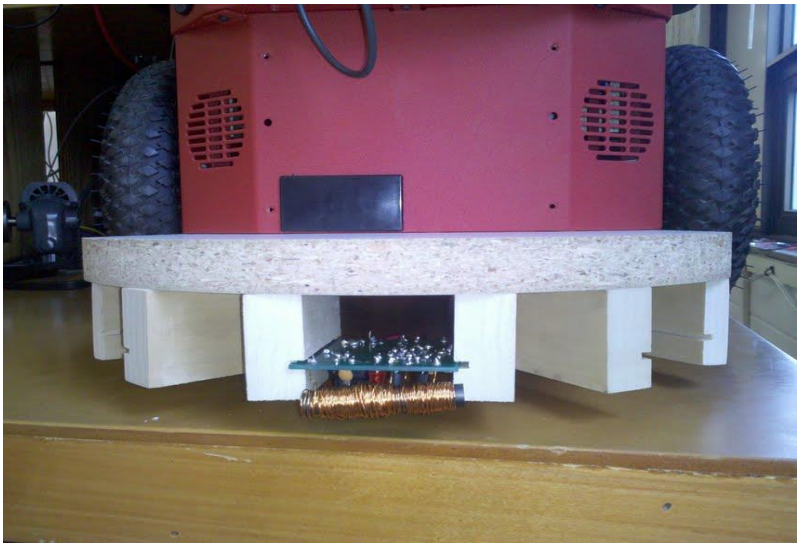
Figure 19: Voltage Regulator



Completed Circuitry:



Metal Detector Holder:



Results/Conclusions

All three subsystems (mode 1, mode 2, and server) for this project are complete. The robot can successfully map an environment using mode 1 wall following and navigate to a given destination within the environment using the path planning techniques provided in mode 2. The server infrastructure is also successfully setup and the server has complete control of all the robots operating on its network. All of the hardware and circuitry for the metal detectors, IR sensors and voltage regulator has been completed. The software for integration of the metal detectors and IR sensors to the robot is approximately 70% complete.

Bibliography

[1] Siegwart, Roland, and Illah R. Nourbakhsh. *Introduction to Autonomous Mobile Robots (Intelligent Robotics and Autonomous Agents)*. New York: The MIT, 2004. Print.

Equipment List

- Pioneer 3D-X
- Metal detector – Electronics123.com Product # Velleman K7102
- IR sensors - Sharp GP2Y0A02YK0F
- LM2576 Voltage Regulator
- Force Feedback Joystick
- Silicon Labs 80C51F120 + UART/USB adaptor
- Space in Jobst as testing environment