Project Final Report

ECE 4532; Fall 2017; Group 02
Puvanai Avakul, Steven Frederiksen, Zachary Yee

Objectives

Our main objectives are to deploy a team of three rovers to perform basic navigation, confirm and detect position of new and known objects in an existing mapped space:

- All rovers able to navigate around a space by utilizing an existing map data
- A Seeker Rover to detect and confirm positions of known and new objects
- A Mapping Rover that provides a RADAR-like system
- Observable real-time positions of all rovers from a GUI

Design

Our system consists of four entities: Base Rover, Seeker Rover, Mapping Rover, and the server/control interface. All of our rovers are equipped with the chipKit Max32 microcontroller board, chipKit motor shield, Arduino wireless shield, and the RN-XV WiFly Module. The rovers share the same motor code module, WiFly code modules, internal struct message format, and external JSON message format.

Base Rover

The Base Rover is the simplest rover, with no other sensors other than the motor encoder. It is designed to do basic navigation in a space with an existing map data from the server as well as direction input from the user interface.

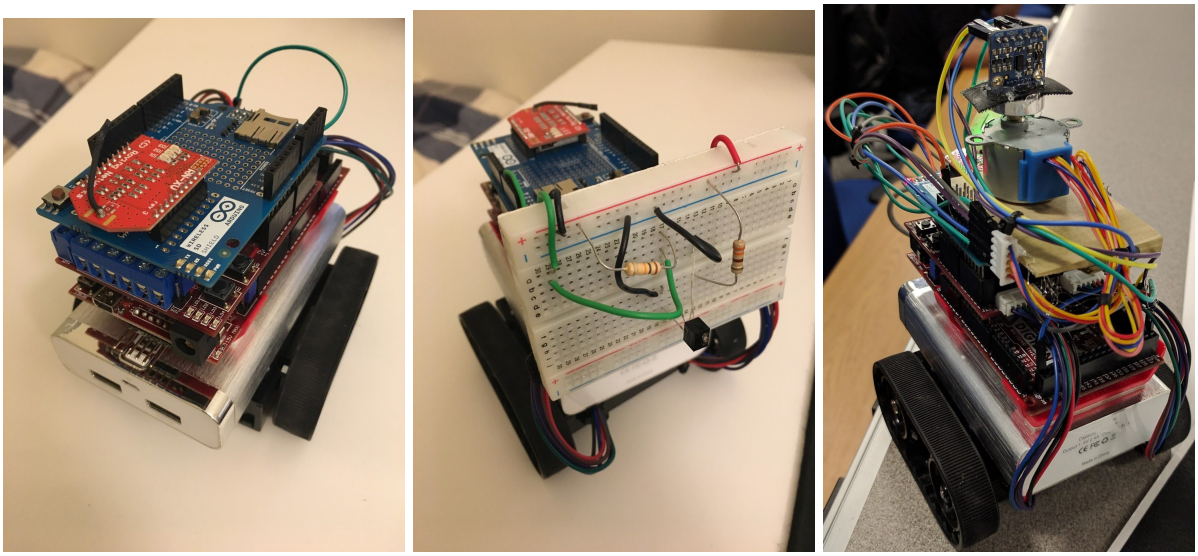
Seeker Rover

The Seeker Rover is designed to be used with the Mapper Rover to confirm the positions of known and new objects. The Seeker Rover features all the functionality of the Base Rover, but with an addition of the short-ranged IR sensor; the QRD1114. The sensor can be used to detect objects within the 0.5-1.0cm range. Equipped with this sensor, the Seeker Rover can detect any objects in the sensor's range and immediately stop the rover's movement to prevent a crash as well as notify the user of the detected object.

Mapping Rover

The Mapping Rover is designed to provide a RADAR-like system that detects new and known objects around itself within a radius of 40cm. Similar the other the rovers, it features the same basic functionality of the Base Rover. However, it is also equipped with the VL53L0X

distance sensor and a stepper motor. The distance sensor is able to measure distance from 50mm to 1200mm, and the stepper motor has 513 steps per revolution. With these two devices, the Mapping Rover is able to provide the RADAR-like system that we wanted. Because of the wires, we have the stepper motor oscillates back-and-forth instead of constantly rotating in one direction.



Figures 1,2,3. Fig. 1 is the Base Rover (leftmost). Fig. 2 is the Seeker Rover (middle). Fig. 3 is the Mapping Rover (rightmost).

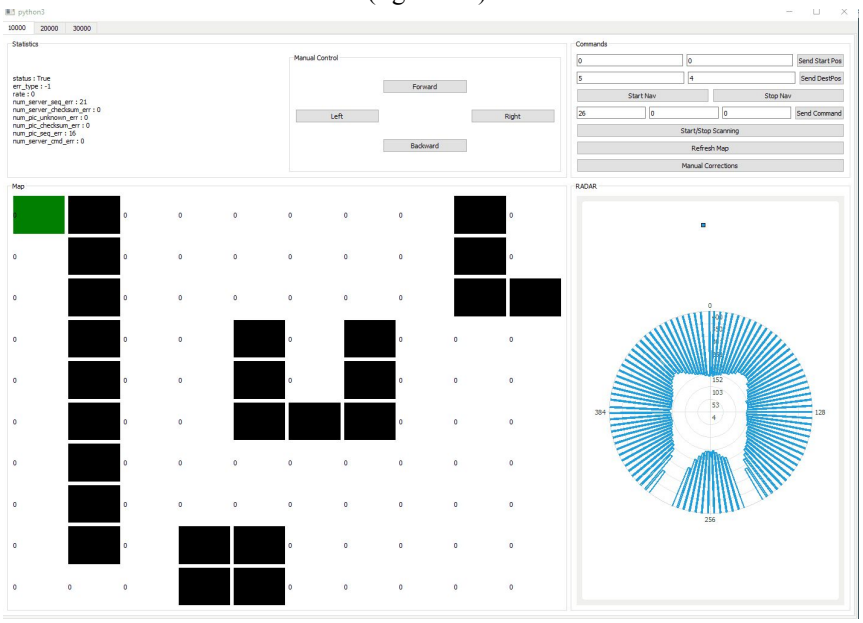


Figure 4. Control Interface for the Mapping Rover. On the top left is the statistics output such as message and error rates. On the bottom left is the map; black blocks are obstacles and the green block represents the rover's current position. On the bottom right is the RADAR output. On the top right are the buttons and input fields for the various predefined commands.

Server and Control Interface

The server's design was to be a simple bridge between all of the rovers and the user. This included passing through all of the various command and response messages as well as properly parsing and handling debugging and state change messages. The final functional requirement was that the server was to act as a database frontend for the rovers and the user alike. Data stored there would be the map used in navigation as well as automated testing inputs.

The control interface was designed to be primarily an observation tool while also giving the user the ability to directly control what was being seen. This is reflected in Fig. 4 where most of the window is taken up by the map, the statistics output, and the RADAR output. The user would be able to passively observe the state of each rover in their own respective tabs. Finally, the interface was to have predefined commands such as setting the destination and starting/stopping RADAR scanning as well as the ability to send an arbitrary command in our standard message format.

Message Format

For most of the messages in our system we used a standard message format based on a three 32-bit field C struct. The fields included the message type and two data fields. This design was chosen to simplify message conversion between different PIC threads as well as the JSON-to-struct conversion when communicating with the server. Additionally, the standard format allowed for the communications protocol between the internal PIC threads to be fast and efficient.

The other message formats used in our system were for communicating between the rovers and the server in a more efficient manner. The first is a JSON string that was the raw message sent over the WiFi connection. Depending on the message being sent this string could contain a well-formatted converted C struct with a checksum and message number or it could contain a debugging message without the other overhead. The last message format was for sending the map from the rover to the server more efficiently and it encoded 100 points on the map into a single string.

Design Analysis

For the most part, our design turned out to be sound. However, the current design was developed over the course of the project and is a product of trial-and-error. The main point that was changed was the distance sensor for the Mapping Rover. Over many iterations we changed the sensor from an expensive, prefabricated 360-degree rotating LIDAR sensor to a cheap combination of a stepper motor and distance sensor.

Even now, the current sensor we are using can be improved. For example, the control interface, while robust, is too complex for our purposes. The calibration steps required to produce accurate values took around three seconds leaving a large window for errors to occur.

Additionally, the sensor was intermittently unreliable where it would produce garbage values after being used for a while. In the future, we would look for a simpler, more reliable device.

Our message format was another instance where our design could improve. While a standard, static message format made the communications between the PICs and the Server fast and simple, there was a lot of wasted space in most of the messages being sent. For messages that encapsulated commands which required no data (stop/stop navigation, ect.) two thirds of the message was being wasted. In the future, changing the message formats to be more dynamic would help reduce this data waste as well as allow for more complex messages that may improve efficiency in other ways. We did trend in that direction with the introduction of the non-standard message types described earlier but these were more of an "efficiency hack" than a directed design choice.

Similarly, the Server and User Interface were not the product of a single directed design choice. Instead both of them evolved over the course of the project and from various individual components. As such, in future iterations both of them should be designed independently and all at once. Especially with a focus on performance.

Implementation Analysis

Originally, we wanted to be able to generate a map of a space with the Mapper Rover. However, because of the time constraint, we scaled back from map generation to a RADAR-like system. For most other components of our system, they work as designed, except for the motor.

Motor

The motor implementation for rover movements is not as accurate as we wanted it to be. When the rover is moving at full speed, it will stop approximately 1-2cm beyond where it is supposed to. We suspect that this is because there is some braking distance involved. After adjusting the motor to half of the speed, the rover is able to stop with less braking distance. Another issue we have for the motor is that when the rover is moving in a straight line, the rover shifts slightly and when the rover is turning, the turns are not as accurate even after some calibrating. This is because we have no dynamic correction for the motors since we are only reading the motor pulses from one of the motor.

As such, in future iterations, more effort would need to be spent on creating a more robust motor control scheme. This includes automatic corrections as well as possibly introducing more sensors to detect error and inconsistencies.

Navigation

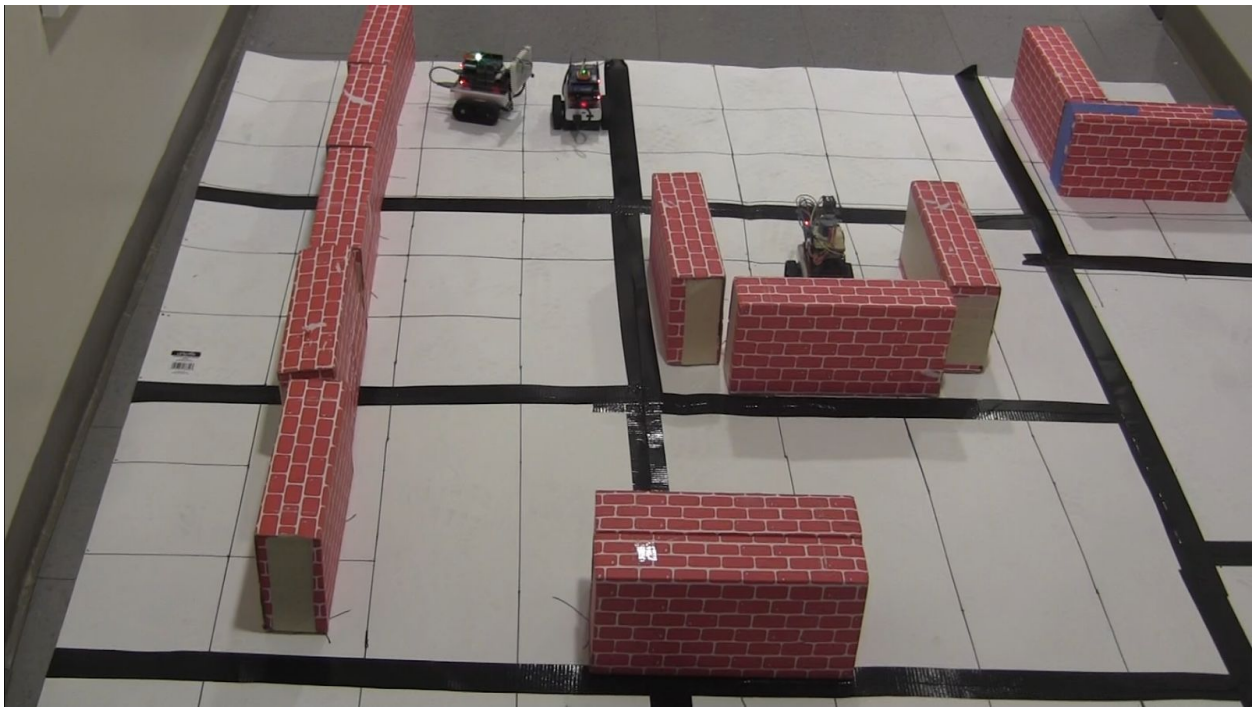
While functional, the navigation code had issues in robustness and efficiency. For example, when the rover is manually pushed, it is not detected by the navigation code. That along with other oversights such as overlooking failure states and not having automated testing code made the navigation code a poor long term solution.

Additionally, in terms of efficiency, the navigation code had issues when it came to map data transfers as well as having a complex state machine in order to translate the generated path to motor commands. In future iterations this code should be looked at for simplification and efficiency improvements.

RADAR System

The RADAR system consisting of the distance sensor and the stepper motor was straightforward to implement and gave no real issues other than being time consuming. As mentioned earlier, the control interface for the distance sensor is complex, and as such ended up requiring a port of and Arduino library for the device in order to function properly. If this sensor is kept, in future iterations, focus should be given to improving this port in terms of reducing the time spent configuring as well as improving reliability and consistency of the device.

Achievements



Figures 5. All rovers in a mapped space.

Most of the systems in our project were inspired from previous or standard works. Things such as the path generation code (pulled from a memory optimized A* algorithm), interrupt-driven design, and the server and user interface code (Qt Framework) all come from various standard sources. This eased the burden of designing these components of our system and gave us a good design pattern to remember for future works.

Other things such as good state machine design as well as implementing efficient and simple APIs came from knowledge gained in previous coursework. This project helped to refine

those skills. This was especially important when it came to the asynchronous nature of all of our component; where nothing was allowed to block when waiting for something to complete. Having a robust state machine made such functionality possible.

For the most part, however, the code implementing our systems was completely self-designed. This helped us learn how to program from the system perspective where there are many moving parts that need to by synchronized. Asynchronous and event-driven APIs became a core component of any component in our system to facilitate this.

Lessons Learned

Debugging Output Routines

The debugging output routines from Milestone One have taught us well. We were able to utilize them to solve multiple issues early in the stages of our project development. Based on these debugging routines, our team has improved upon the debugging routines from Milestone One that can only be seen from the use of logic analyzer by utilizing the WiFly and having the routines sending debugging messages over WiFi to the server. Because of this we were able to see exactly what is going on in each rover through the debugging messages.

Conclusion

Even though we were not able to deliver map generation and accurate rover movements, we were able to scale down our project and have a working prototype. The map generation was scaled down to a RADAR-like system and the rover inaccurate movements are compensated by manual corrections through the GUI. Overall, we found this project a challenging and valuable learning experience that will serve us well in the future.