Group B2: Robotic Indoor Plotting

Carnegie Mellon University Spring 2020 ECE Design Experience

Authors: Aditi Hebbar, Shanel Huang, and Alexander Bai

Abstract
This document outlines the design of an autonomously navigating robot that is capable of mapping the interior of a given building. This system will be implemented in the Spring 2020 iteration of the ECE 18-500 course. The output of the robotic system is a 2D floor plan of the given building, suitable for human navigation. This document outlines the system architecture, requirements, and implementation process for the device. We also explore existing similar projects and risk management we faced along the way.

Keywords: SLAM, Indoor Positioning, Rotating Lidar, Navigation, Obstacle Avoidance, Multi-thread, Real Time

I. Introduction

Current mapping and navigation systems lack accurate and comprehensive indoor positioning information. Accurate indoor positioning information has applications benefiting everyday consumers, facility administrators, and autonomous systems. In order to show specific details on indoor locations, current solutions such as Google Maps require businesses and establishments to upload their own floor plans. However, most buildings do not have this information available online, making it difficult for visitors to navigate indoors. The Robotic Indoor Plotting project will autonomously generate indoor floor plans. When placed indoors, the robot will explore the space, creating a 2D map using obstacle detection sensors, SLAM, and a rotating LIDAR. After traversing the indoor space, the robot will synthesize information into a readable format for the user to view on a web application.

II. Design Requirements

Our methodology to determine system requirements included taking into account ideal use case scenarios, technical limitations of our components, and project feasibility. To provide a good user experience, we require the device to complete a full scan on one charge and generate a floorplan accurate enough to allow for human navigation. We then identified core requirement areas of the system to be the mapping speed, floorplan accuracy, obstacle avoidance, and power consumption. To create quantifiable target metrics, we assume a mapping area of 2000 square feet, similar to that of our final test cases of Tepper Quad 2nd floor and Wean 5th floor.

For battery life, the iRobot Create 2 advertises a battery life of 2 hours with vacuum motors fully running. Since our project does not require power from the vacuum motors which use 50% normal operation wattage, the robotic base should realistically last us up to 4 hours per charge. The external power bank for the Pi and sensors will be supplying at 5V/3A, and we plan to acquire a 10Ah supply, which will last an estimated 3 hours. However, in order to provide the best user experience and reduce wait times, we created a tighter bound of an **operational time of up to 2 hours**. Combining the operational time requirement with our estimated map area of 2000 square feet then yields us a required **mapping speed of 16 square ft/min**. This total mapping time starts when the robot starts and ends when the map is fully generated. This speed requirement includes time taken for navigation inefficiencies and processing cycles. In cases where there are few obstacles, we expect this to be much faster.

The final floorplan output is required only to be accurate enough for a human to navigate with. We estimated that a human could still navigate to a mapped room effectively when there is up to **10% error in the mapped dimensions**.

Put all together, figure 1 summarizes the design requirements our system will need to meet in order for build a successful minimum viable product:

Figure 1.

Mapping Speed	Generate map faster than 16 sqft/min
Floorplan	Within 10% of true distance and depth value
Obstacle Avoidance	Zero collisions with active or passive obstacles
Power	Operational time of 120 minutes per charge

III. Architecture

The system architecture of the hardware components of the project consists of a Raspberry Pi, a spinning LIDAR, and an iRobot Create 2 (Figure 3). The software component consists of a server hosted on AWS running a React web application. The Raspberry Pi is powered by an external power bank, and in turn powers the rest of the system. The Roomba has a separate independent battery. See figure 2 for a complete system diagram of the project architecture.

For the base of the robot, we will be using an iRobot Create 2 for physical movement of the entire system. The iRobot Create system provides an Open Interface API to communicate with the robot case. This software interface is used to control and manipulate the robot's behavior, as well as read in from its sensors through a series of commands. It has multiple built-in ultrasonic sensors that can be used to detect cliffs and obstacles at the same height of the robot. The Roomba interfaces with the Raspberry Pi via a serial connection through a ribbon cable.

To process information from the rest of the hardware, we will use a Raspberry Pi 4 microcontroller. This version contains 4GB of RAM and includes all the necessary ports to connect our system gracefully. The Pi runs all of the control code, taking in data from the LIDAR and Roomba and turning it into commands for movement. The majority of the software packages we plan to use are compatible with the Raspberry Pi 4, and the extra processing power allows for extension and scaling of this project in the future.

The spinning LIDAR will be used to detect dimensions of objects and walls surrounding the robot. The RPLidar A1 model we're using allows for detection of objects 360 degrees around the robot, and up to 6m. Since it outputs a 2D point cloud on a horizontal plane, it does not give us any vertical data. For the initial prototype of this project, the Lidar sensor will be placed at a height directly above the robotic vacuum to reduce noise scan obstacles close to the floor.

Figure 2. System level block diagram

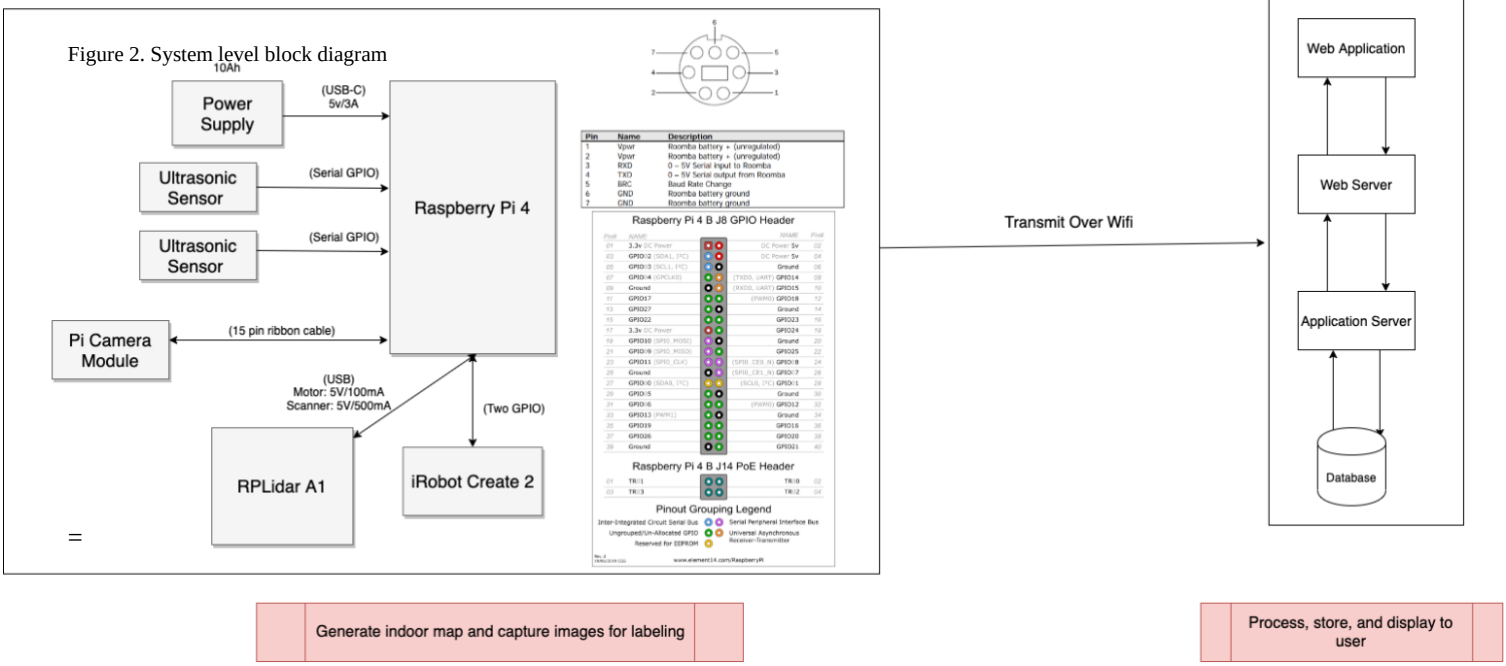


Figure 3: Robot system structure

flexibility and customization is building and wiring our own chassis. However, with the limited time and tools available, we decided it would not be wise to spend most of our efforts on a large mechanical undertaking. We found a few existing hobbyist bases such as the DirtDog or Smartibot, but they lacked a great deal of connectivity, stability, and battery life. With this in mind, we chose to use the iRobot Create 2. It is well-documented with an Open Interface software tool, developer-friendly, and designed specifically for robotics projects. There are several built-in sensors that are easily interfaceable, and the base is large enough to support our

IV. Design Trade Studies

A. *Robot Base*

The robotic base is responsible for physical movement of the entire system. Based on our target metrics, we need a hardware system large and strong enough to support the rest of the equipment in the system. One option that would allow for

frame and other components. for robotics projects. There are several built-in sensors that are easily interfaceable, and the base is large enough to support our frame and other components. We later found that depending on the firmware version of the iRobot Create 2 model, the odometry data may be corrupted. This was the case in our project, and we had to periodically check and reset our robot base if the sensor readings were wrong.

B. Depth Detection Sensor

The depth detection sensor will be used to data for map generation. Initially, we considered a few custom-built sensors, such as an IR Dot Matrix, a multi camera CV system, or a stereo LIDAR, but each of these came with their own challenges with accuracy, tolerances, and manufacturing quality. To circumvent these issues, we decided to purchase an existing manufactured sensor.

Our first choice was the Intel RealSense d415, a stereo LIDAR system. It is well-documented, accurate, and integrates well with the rest of our system. However, this sensor requires excessive processing power to create the 3D point-cloud output. The specs of this sensor overshoot our requirements by a lot, while hindering our computational speed. We also considered the Microsoft Kinect 2 which processes more efficiently but is not very compatible with the rest of our components. The Kinect 2 requires additional cable connections to interface with the rest of the system, and has deprecated software support. In the end, we decided to use the RPLIDAR A1M8, a rotating lidar that generates a 2D point cloud output that is leaner. The output scans are customizable and accurate enough to support our use case.

C. Microcontroller

When considering which microcontroller to use, we settled on the Raspberry Pi 4 because of our team's experience with the device, as well as the compatibility with the other components of our system. This is the newest version of the Raspberry Pi, and gives us increased processing power and more ports to interface with. In order to reduce ramp-up time required to learn a new system and unique configuration for integration, we found that the Raspberry Pi's universal compatibility, detailed documentation, and developer-friendly environment is the best choice. We did consider using two Raspberry Pi's (one for receiving information, one for transmitting information), but the benefits of separating the system did not outweigh the integration efforts it would require.

D. Power

Originally, we had planned to use the Roomba's vacuum motor drivers to supply power to our Pi and sensors. The drivers had ample power, but with two drawbacks. The first is that the power came in at 12-16V/500-1500mA instead of the 5V/3A required by the Pi, meaning we would need to build a custom transformer circuit. The second drawback is that in order to draw power from these motor drives, we must send a signal to the Roomba, but we cannot do that without first powering the Pi, which means we would still need an external power jumpstart. In the end we decided on simply buying an external power bank for the Pi. Among the many options, we chose one that fulfilled our 5V/3A USB-C requirement, while also having 10Ah capacity to last our required 2 hours.

Upon completion of our project, we found that all necessary power requirements to run the system at a reasonable speed were met. We experienced no problems with real-time processing, and think that it more advanced software such as computer vision and extra peripherals could be easily added in the future.

E. SLAM

Since SLAM is a popular field of study, there are many existing SLAM packages available for free. Most notably, Google Cartographer is an open-source codebase along with some documentation for installation and usage. We were initially set on using Cartographer for its sophisticated algorithm and reputable developers, but decided against it for several reasons. Primarily, installation and setup of Cartographer was extremely problematic with our hardware. The most recent ROS compatible version had many dependency and environment bugs in just the installation process.. Additionally, we realized that we would need to hook into SLAM and modify some of the internal behaviors, which would be difficult for a complex system like Cartographer. Similarly, we found many compatibility issues when trying to install ROS' Gmapping module. Finally, we decided to forgo ROS packages and decided to use BreezySLAM. BreezySLAM is a well maintained Python SLAM package that does not require ROS. While simpler and less granular, BreezySLAM allows for an optimal balance of customization and ease of use. With this in mind, we tried several simpler SLAM packages and settled on BreezySLAM for its acceptable accuracy, ease of use, and ease of modification.

F. Navigation

Similar to SLAM, there are many open-source options available for navigation packages. Existing navigation packages are able to take in odometry and movement data to generate path planning and positioning. However, since the accuracy of our raw data is subject to error and tuning, we opted to write our own navigation algorithms instead alongside our control code for greater flexibility and testing options.

V. System Description

To use the Robotic Indoor Plotting robot, a user will set it down on the ground. Whichever way the robot is facing will be represented as 'North' on the orientation of the final user map. The robot will navigate until it has determined that all the explorable area has been explored and logged in a map. When finished, it'll automatically stop and send the final user

map as an image to display on a web application. Finally, the user can navigate to the web application and use the generated image with any other application for their specific use case. This is a breakdown of each system component:

*A. **Base***

The base itself is a nearly self-contained system. We will not be modifying the base except to remove the vacuum modules and install a mount on top for our other components. Power supply and sensor wiring are all handled inside the Roomba. If needed, the robot can dock wirelessly into a charging station to recharge the battery. The only connection to the base is in the form of a serial cable that directly connects into the Pi to send and receive data. Four of the DIN pins are for Roomba power, and one is used to change the baud rate, which we do not need since both Roomba and Pi default to 115200. The only data we will be transmitting through the input pin is movement commands. The base will handle no processing or navigation on its own.

*B. **Sensing***

Our sensing system consists of two parts: Roomba's internal sensors and the rotating LIDAR.

The RPLIDAR A1M8 is adhered directly on top of the Roomba at the center. Both power and data are transmitted through a provided USB 3.0 adapter from the Pi. We will be sending the LIDAR commands to activate the spinning motor and laser, with variable rotation speed and sampling rate for both. The sensor will output a 2D point cloud representing all the distances sampled by the laser, to be processed by the Pi for navigation and mapping.

Through the iRobot Interface, we can retrieve data packets containing the measurements from the robot's internal sensors. Of the 58 sensor packets available, we were particularly interested in the frontal ultrasounds, frontal push bumpers, and frontal cliff ultrasounds. Theoretically, these sensors alone should be able to provide enough data for obstacle detection. However, due to inherent firmware bugs, these sensors would permanently get stuck at certain values, rendering them faulty until restart. As a result, we had to incorporate intermediate LIDAR data for obstacle avoidance.

Though the robot does provide data on distance traveled and angle turned, another inherent firmware bug rendered these data packets completely unreliable. In place of this data, we implemented our own dead-reckoning odometry system.

*C. **Control***

All processing in our system will be handled by the Raspberry Pi 4, and all components will physically interface directly with the Pi. In total, we expect to use one USB port for the Lidar, and a serial connection with the Roomba robotic base, which leaves us plenty of extra connections if necessary. The Pi also will handle all the software computation within the system. This includes running SLAM, navigation and path planning, odometry sensor readings from the Lidar and Roomba, obstacle detection, and movement. Upon completing the operation, the Pi will transmit the captured data over to our server through Wi-Fi using basic FTP. Although a lot of different processes will be running on the Pi at the same time, we have planned accordingly in the case the Pi cannot handle the load. If needed, we are able to modify software processing requirements and power drawn from hardware components to an optimal solution.

*D. **Multithreaded Program Flow***

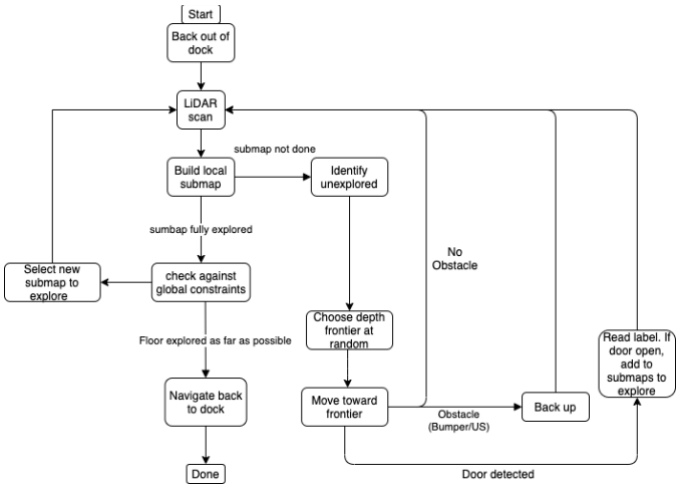
We separated our code into multiple concurrent threads for modularity and efficiency. The parent Main thread spawns a Movement, Obstacle, and SLAM thread, each of which persists until program shutdown. Most of the control flow is handled within the Main thread, along with navigation (explained below). Through a shared object, navigation instructions are passed to the Movement thread, which independently handles all robot movement and odometry. The SLAM thread is constantly monitoring the LIDAR data, feeding it into the BreezySLAM package, and maintaining a shared map object which the Main thread uses to navigate. The Obstacle thread runs as an event handler. When an obstacle is detected either through the Roomba data packets or the LIDAR hook, this thread will assume control of the entire system to prevent collision and update the map with new knowledge of the obstacle. All threads are running together in real-time, continuously updating the set of program and map information.

The end condition for the system program is when the robot completes full loop closure of an indoor space. The robot continuously takes 2D scans from the lidar and compiles historical information using SLAM. The robot continues until it recognizes that it is in a space fully enclosed by walls, and has mapped everywhere within the walls. This means that the robot recognizes it has explored all unknown frontiers, and stops the program. See figure 5 for navigation algorithm flow diagram.

All together, each thread in the program contributes to creating successive iterations of different representations of the mapped space. This starts with raw scans from the lidar sensor, represented as individual points of distance and direction in an array. Individually, this picture and 2D point cloud does not give us much information of what the room actually looks like. This collection of points is then given to BreezySLAM to generate a map of the current state of the surroundings. BreezySLAM will stitch together point clouds in an attempt to construct a global view. Our internal program then uses this map from BreezySLAM to create an internal data map with which we represent with data labeling each section of the map. Red is the robot's current position, green is the destination position, yellow is the proposed path the robot will take, blue are identified walls, and orange are caution buffers around known obstacles and walls. This map is updated by the program and the obstacle thread during each move sequence. Lastly, we strip the most updated data map of compression and labelled data to produce the final user map. The user map is a simple black and white representation of known walls and obstacles in the room, and includes a 1 meter

marker for scale. See this flow of information between the different maps in figure 6.

Figure 5. Navigation algorithm flow



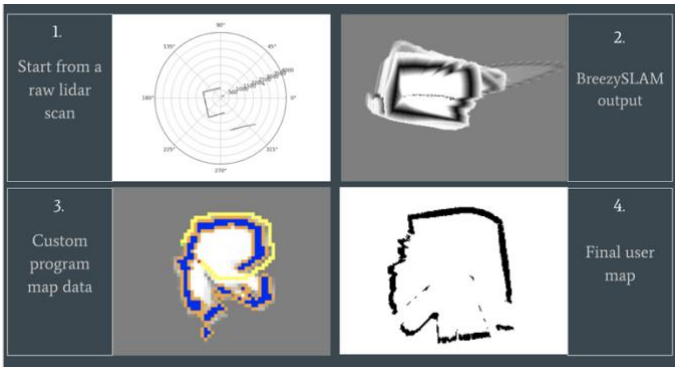


Figure 6. Program Data flow

E. *Navigation*

In order for the robot to generate a map of the unexplored environment, we built our own autonomous navigation system. Using the information collected from BreezySLAM, the lidar sensors, Roomba odometry data, and obstacle avoidance, we construct an internal data map that represents the current state of the program. This map keeps track of walls, known obstacles, current position, destination position, and paths (see Figure 7). With each cycle of movement and scanning during our program, this internal data map is updated. This system uses the map to do two things: choose a destination and find a path from the robot's current position to the selected destination.

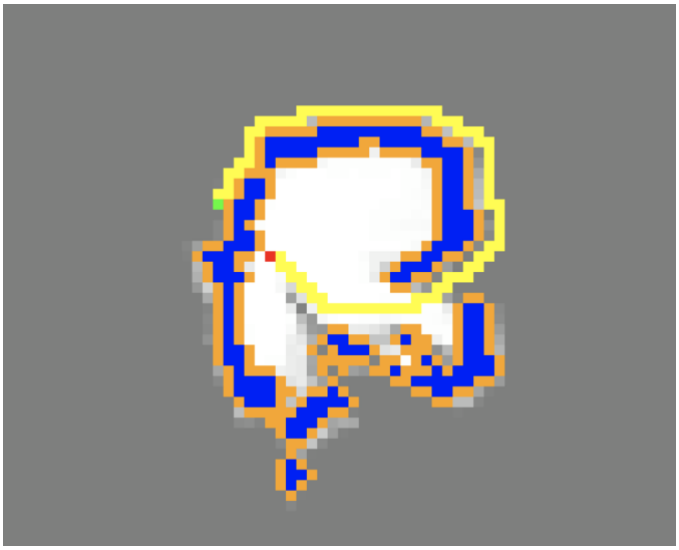
To choose a destination, the robot uses the current map representation to pick an unexplored area to navigate to. Because the internal data map keeps track of where obstacles, walls, and unexplored areas are, the program is able to spirally pick from a range of radii until there is a destination that is unexplored and navigable.

To determine whether a selected destination is able to be navigated to, the program plans a path from the current robot position to the given destination. The path planning algorithm takes in the grid representation of the current map, the start and end positions, and returns back its determined best path (if possible) using the A* search algorithm. We use the Manhattan distance heuristic for possible directions on the path, as keeping our robot's turns two the four cardinal directions will minimize error. Using the A* algorithm for path planning has the advantage of distance heuristics that find the "best" path, not just any path. This allows for heuristic tuning to avoid known obstacles, navigate through unexplored areas, and watch out for walls.

F. *Server and Web App*

Since the processing has already been completed by the time data is passed to the server, our web application is a very simple UI to display the resultant maps. It contains some basic information about the project and a page with the most recently generated maps in chronological order. The final set of maps to display to the user are saved onto the Raspberry Pi, then transmitted to the EC2 instance the web application is hosted on. This UI is written in React and uses NodeJS and Webpack packages. This web application is where additional features of user correction, labeling, and real-time generation visualization could be in the future.

Figure 7. Internal Data Map



VI. Project Management

A. *Project Schedule and Gantt Chart*

The Gantt chart (see Appendix B) outlines the proposed project timeline until completion. This timeline was created with the highest priority being to finish the minimum viable product before the end of the course. Although this Gantt chart

has been continuously updated throughout the course, there was one major shift in division of labor after converting to remote work. In order for this schedule to be as useful as possible, we've constructed it with practicality and feasibility in mind. With each component of the project, we've built in slack time in addition to integration time to cover unexpected issues that may arise. We've also identified and scheduled around known non-work days for school breaks and holidays. Additionally, we've scheduled in time for continuous documentation and presentation preparation.

B. Division of Labor

The implementation of our project is divided into three main areas: robotic programming, navigation, and the web application. Our origin division of labor was to each lead one area while assisting with another. Due to logistical issues, the division of labor was affected, as we started to work in remote locations in different time zones. We identified several modular parts of the project that can be worked on independently, and split them thusly:

1. Hardware Interfacing: Aditi.
2. Navigation/Movement: Alex, Shanel
3. Obstacle Avoidance: Aditi
4. Web Application: Shanel, Alex

Since time was of the essence during this project, we made the decision to keep all the hardware in one place, delegated to one person. This would eliminate shipping and transportation of the parts, and allow us to continue testing in the same. location. Aditi had all the parts with her, and led the majority of the testing and hardware interfacing changes.

Otherwise, Shanel and Alex worked on software components including navigation, path planning, the web application, and program representation and control flow.

After the transition to remote work, the majority of progress was done together in scheduled Zoom calls with all three members present. This turned out to be a beneficial because it ensured that nearly every part was completed with input from every member. Since we had to make decisions quickly and quite often throughout the process, it was helpful to have input from other team members who have context right away.

C. Budget and Bill of Materials

This project is scoped to fit within the limits of a $600 budget, not including borrowed materials from the Carnegie Mellon University Electrical and Computer Engineering Department. In order to fulfill the minimum viable product, the total cost of materials was estimated to be $356.59. However, due to unforeseen logistical limitations due to COVID-19, there were additional costs to buy monitors, keyboards, and other workstation setup materials. We also did not end up using the ultrasound sensors and Raspberry Pi Camera purchased for the original scope of the project. The final cost including those, is $515.68. For the full parts list and bill of materials, see Appendix C.

D. Testing Plan

For the robot module, we'll be testing the accuracy of the resulting dimensioned map with a series of rooms with varied sizes, shapes and clutter density. In addition to the accuracy of the final product, we'll be monitoring the total processing time and the success of obstacle detection. The planned idea test cases for the robotic module included:

1. Rectangular room - empty
2. Rectangular room - furnished
3. Irregular shaped room - empty
4. Irregular shaped room - furnished
5. Irregular shaped room - furnished with dynamic components

Specific identified test cases include Wean 5th floor (big, standardized building), and Tepper Quad (furnished with high traffic).

Quarantine measures greatly limited our testing capabilities. We originally planned not only to test in many of the large, structured buildings around campus, but also to build our own custom test courses out of cardboard or styrofoam. In the end, our only viable test cases were the bathroom, bedroom, and hallway of the apartment building where our team member with the hardware lived.

The bathroom is intended to function as a base test. This space is about. 129 square feet, and contains a couple of key obstacles along the walls including a sink, trashcan, and toilet. Since the room is so small, the program completes nearly instantly, with no movement. Little complex processing is done, and no navigation is performed. Our system was able to get 96% accuracy of the total mapped area.However, there are some signs of feature loss as result of SLAM processing and compression algorithms.

With a larger size and more obstacles, the bedroom functions as a more complete test. Scanning, processing, navigation, movement, and obstacle avoidance are all tested as the robot moves around the room. The bedroom that we set as our test case contains a cardboard box in the middle of the room, as well as a chair and desk. Our robot was able to navigate the room safely and generate a final user map representative of the room to an acceptable degree of accuracy (86.5%). Despite being a more comprehensive test, the relatively small size of the room limits operational time to about one minute.

We tried several times to test our system in the hallway, but kept running into some issues with the LIDAR that we were unable to debug without live monitoring of the data. Although the test ultimately was incomplete, we did confirm that in areas closer to our projected use case (large open areas with little furniture), our obstacle avoidance was much more robust

E. Risk Management

A. Power consumption and battery life

Theoretical calculations of power consumption result in a projected system battery life of ~3 hours. This meets our requirements, though we have no way to verify this metric, since the program never takes longer than a couple minutes to map out the use cases available to use at this time.

B. Logistics

Campus lockdown incurred the loss of access to important services, such as fabrication labs, soldering irons, workspaces, and hardware peripherals. These components were either scrapped or replaced with purchases through Amazon. Lack of testing capability and the necessity of testing through video call greatly hampered our test capabilities, resulting in slow development cycles.

C. Hardware bugs

The iRobot Create 2 turned out to be extremely unreliable for sensor data. Persistent hardware and firmware bugs resulted in fatal data corruption, garbage sensor readings, and regular system freezes which could only be resolved through a factory reset. A large amount of effort was spent poring through online documentation or support threads, of which there were thankfully many.

D. SLAM

Using BreezySLAM over Cartographer was a calculated choice, but it still came with many of its own risks. The relatively simple algorithm may have been easier to pick up and modify, but was also less sophisticated, resulting in less accuracy. Few customization options were available, again due to the simplicity of the package. Furthermore, when we compress the map for internal data processing, even more accuracy is lost.

VII. Related Work

As indoor positioning and navigation is a highly researched topic, similar iterations of our project have been carried out by Google and other university teams. However, the identified similar projects focus on optimizing SLAM and indoor map generation. We have yet to find another project attempting to add door detection and labeling on top of a produced map. We believe incorporating CV to add this information to the map is a unique and substantial addition to existing projects.

A. RPLidar A1 with RPi3

This is a simple project that uses a raspberry Pi, RPLidar, and an Adafruit PiTFT display to display data from an RPLidar onto a screen. It is a useful resource for learning how to read and process data from the same spinning lidar model we will be using.

B. Google Mapper

Cartographers is a project by Google that creates maps in real-time using a broad range of sensor configurations common in academia and industry. Although it is mainly used for 3D point cloud projects, we will still be using the same fundamental navigation algorithms for our 2D use case. The project also provides software that can be integrated with ROS, among other useful library functions for lidar navigation.

C. Capstone S19 Team BC (AutoMapper)

This is a Capstone project from 2019 that similarly uses a Roomba and RPLidar to create an autonomous mapping system and detects the presence of humans using a thermal camera. It's pretty similar to our own project in its basic movement and mapping functionality and uses a lot of similar parts.

VIII. AWS Credit Usage

We obtained a total of $100 in AWS credits at the start of the course. The sole use of the credits was to spin up an EC2 Micro instance to host our web application. Since we made sure to stop the instance when not in use, we ended up using only $9 worth of credits, leaving $91 remaining.

We would like to thank Amazon for providing these credits for educational purposes, and also CMU ECE for connecting us to these resources

IX. Appendix

A. References

Project Research

1. https://learn.adafruit.com/slamtec-rplidar-on-pi?view=all Specifications and data sheet for the RPLidar A1 part we used in our project.
2. https://opensource.googleblog.com/2016/10/introducing-cartographer.html Google's Open Source Cartographer ROS package used for research. We did not use this in the final project.
3. http://course.ece.cmu.edu/~ece500/projects/s19-teambc/ Previous Capstone AutoMapper project referenced in 'Related Works' section.
4. https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/45466.pdf Documentation on loop closure using 2D lidar slam. This information was useful when we were planning to write our own closure and end conditions for our program.
5. https://link.springer.com/article/10.1007/s10514-012-9298-8 Comparison of a couple known path planning strategies for exploration using SLAM. Some of these concepts related to conceptual search algorithms were used when constructing our own algorithm.

iRobot Create 2

6. https://www.irobotweb.com/~/media/MainSite/PDFs/About/STEM/Create/iRobot_Roomba_600_Open_Interface_Spec.pdf iRobot Create 2 Open Interface specification sheet, used for software interfacing with the roomba.
7. https://github.com/pomeroyb/Create2Control Encapsulating python API built on top of iRobot Open Interface. This library was used to control the robot using python commands.

SLAM

8. https://github.com/simondlevy/BreezySLAM Python SLAM packaged used to handle the SLAM component of the project. The rpslam module was highly integrated with the

SLAM portion of our project, and was tuned specifically for the RIP project use case.

Navigation and Path Planning

9. https://medium.com/@nicholas.w.swift/easy-a-star-pathfinding-7e6689c7f7b2 A* search algorithm conceptual explanation. Our path finding algorithm is based on this A* concept.
https://www.hackerearth.com/practice/algorithms/graphs/flood-fill-algorithm/tutorial/ Floodfill algorithm conceptual explanation. Our enclosed check to determine whether or not the robot is enclosed by walls uses this floodfill concept.

B. Parts List and Bill of Materials

Part Name	Buy/Borrow	Price
iRobot Create 2	Buy	$199.99
Raspberry Pi Camera Module V2.1	Borrow	FREE
Slamtec RPLidar A1	Borrow	FREE
Raspberry Pi 4	Buy	$87.97
SanDisk 16GB MicroSD Card	Buy	$5.92
Ultrasonic Distance Sensor HC-SR04 (x4)	Buy	$34.20
Power Bank	Buy	$28.61
Monitor	Buy	$70.00
Mouse & Keyboard	Buy	$21.00
Raspberry Pi 4	Buy	$35.00
Pi Power Supply	Buy	$7.00
Pi MicroHDM	Buy	$10.00
Velcro	Buy	$2.00
Epoxy	Buy	$5.00
USB-C to USB-C cable	Buy	$8.99
Total Cost:		$515.68

C. Gantt Chart

PROJECT TITLE	RIP - Robot Indoor Positioning	**TEAM MEMBERS**	Shanel Huang (shanelh), Alex Bai (albai), Aditi Hebbar (ahebbar)
GROUP	B2 (Tamal + Joe)	**TERM**	Spring 2020

					PHASE ONE					PHASE TWO					PHASE THREE				
WBS NUMBER	TASK TITLE	TASK OWNER	DUE DATE	PCT OF TASK COMPLETE	Week 1 (1/27-1/31)	Week 2 (2/3-2/7)	Week 3 (2/10-2/15)	Week 4 (2/17-2/21)	Week 5 (2/24-2/28)	Week 6 (3/2-3/6)	Week 7 (3/9-3/13)	Week 8 (3/16-3/20)	Week 9 (3/23-3/27)	Week 10 (3/30-4/3)	Week 11 (4/6-4/10)	Week 12 (4/13-4/17)	Week 13 (4/20-4/24)	Week 14 (4/27-5/1)	Week 15 (5/4-5/8)
1	Project Proposal and Design																		
1.1	Project Proposal Presentation	SH, AH, AB	2/3/20																
1.2	Design Review Presentation	AB	2/16/20																
1.3	Design Review Report	SH, AH, AB	3/2/20																
1.4	Compile Parts List and Order Parts	SH, AH, AB																	
1.5	Slack	SH, AH, AB																	
2	Robot Programming	AH & AB																	
2.1	Base control	AH & AB																	
2.2	Realsense control	AH & AB																	
2.3	Rpi Control	AH & AB																	
2.4	Integration	AH & AB																	
2.5	Slack	AH & AB																	
3	Navigation	SH & AB																	
3.1	SLAM	SH & AB																	
3.2	Generate Map	SH & AB																	
3.3	Integration with Robot Programming	SH & AB																	
3.4	Slack	SH & AB																	
4	Computer Vision	SH & AH																	
4.1	Build physical camera platform	SH & AH																	
4.2	Door and Obstacle Recognition	SH & AH																	
4.3	Text Recognition and Labeling	SH & AH																	
4.4	Integration with Robot	SH & AH																	
4.5	Slack	SH & AH																	
5	Web Application	AB & SH																	
5.1	Data Processing	AB & SH																	
5.2	Server/API	AB & SH																	
5.3	Database	AB & SH																	
5.4	Webapp (UI)	AB & SH																	
5.5	Slack	AB & SH																	
6	Presentation and Reports	SH, AH, AB																	
6.1	Interim Demo	AH	3/30																
6.2	Final Presentation	SH	4/26																
6.3	Final Report	SH, AH, AB	5/1																
6.4	Blog and Documentation	SH, AH, AB																	