Roaming Spirit:

A Framework for Localization and Navigation on a Raspberry Pi

A Major Qualifying Project
Submitted to the Faculty of
Worcester Polytechnic Institute
in partial fulfillment of the requirements for the
Degree in Bachelor of Science
in
Robotics Engineering
By

Troy Hughes

Guillermo Vincentelli

MQP AW1-AUT1
Submitted on: April 27, 2016

Adviser

Alexander Wyglinski

Abstract

To facilitate the development and research of autonomous, indoor mobile robotics, the Roaming Spirit platform was created to implement localization and navigation. The Roaming Spirit platform is a light weight system (319 grams) intended for implementation on both terrestrial and aerial vehicles. The platform successfully navigated a Turtlebot through over 150 feet of hallway space, handling hallways intersections, static obstacles, and occasional interference from people walking past its line of sight. It also successfully created a map of the hallway. For a future iteration of this work, locating a safe location for further aerial testing would allow for debugging issues with the software to avoid damage to both the Roaming Spirit system and to the aerial vehicle.

Acknowledgments

We would like to thank Professor Alexander Wyglinski for his incredibly enthusiastic and priceless help on this massive project. We also would like to thank Nils Bernhardt and Lukas Brauckmann for flying across the Atlantic and giving us all of their time, effort, and expertise to complete this project. We couldn't have done this without you. Danke shcön.

Executive Summary

The aims of this project were to create a cost effective, cross-platform, indoor-localization-and-navigation framework for any type of indoor mobile robot. The goal of the framework was to enable any robotic system to implement the project (assuming it met the minimal requirements of the framework) in an easily adaptable and expandable manner. This would allow the framework to be independent of the sensors that are used for providing it data, as long as the data acquired could be sent to the framework in the correct format. It would also remove the dependency of the robot upon any particular map-making algorithm and any particular localization and navigation algorithms - so long as the algorithms chosen can be made to fit the interface of the framework.

The proposed solution to this problem included four key components: a processing unit with which to execute the localization and navigation commands/information; a sensor with which the system can procure data on the surroundings for the localization and navigation to occur; an algorithm that will produce a map and update it over time while new data is fed into it, such as Simultaneous Localization and Mapping (SLAM); and interface-driven software design to allow independent extension of the project by other users.

As a processing unit, the Raspberry Pi was chosen over other systems due to its large user community as well as the project group's familiarity with the Raspberry Pi. Several of the limitations of the Raspberry Pi (such as its limited processing power) were already known and could be taken into consideration during the planning stages, and libraries for different sensing systems and/or SLAM algorithms were available with guides on how to install and utilize them. As a result, the project group could save time that would otherwise be spent on attempting to alter the design when unforeseen limitations and issues arose, as well as save time in reading data sheets and documentations which - unlike the tutorials found - could be unclear or incomplete.

The sensor for data collection, an Asus Xtion Red-Green-Blue and Depth (RGB-D) camera, was chosen for its relatively low cost (around $270.00), low weight, and its functionality. The purpose of the sensor was to acquire depth information about the surrounding environment. LiDAR, or Light Distance And Ranging sensors accomplish this goal, and are used on projects ranging from Google's Self Driving Cars (SDC) to Boston Dynamic's Atlas. However, since part of the goals of for this platform required it to be cost effective, the group could not choose a LiDAR for the sensor as it would immediately negate that goal - LiDAR devices are as of 2016 still prohibitively expensive for the group (upwards of $1,000.00). Instead, however, it is possible to mimic the data returned by a LiDAR through use of an RGB-D camera, such as the Microsoft Kinect and the Asus Xtion. In order to ensure that the system is as cross-platform as possible (e.g. available for aerial indoor robotics as well), the weight of the sensor was taken into consideration, and so the lighter Asus Xtion was selected. The Xtion was then stripped of the hard external plastic to remove more weight and was then prepared for deployment.

The localization and navigation algorithm implemented was a lightweight, simple-to-implement system called BreezySLAM. The algorithm was designed to easily interface with Python - the project group's programming language of choice - and was written with C bindings to dramatically speed up the processing. Using a previously created algorithm such as BreezySLAM allowed the project group to program in a high level language such as Python. This allows for the opportunity to use several existing online libraries that implement C bindings, thus giving run-time execution comparable to lower level languages like C. BreezySLAM was also significantly less processor intensive than other SLAM algorithms that were found, as the algorithm was designed to fit on a Raspberry Pi for Do-It-Yourself (DIY) projects.

The software development contained a very simple yet important approach: design to the interface. By encapsulating all varying sections of the code behind interfaces, the body of the software is dissociated from the specific implementations of the these portions of the code. This means instead of the main algorithm needing different, custom code to use different sensors, one can program the sensor to the interface, and the algorithm can use each sensor without needing major changes to its own code.

The outcome of this implementation plan would be a low cost platform that would have the capacity to localize and navigate indoors. One concern of this solution was the processing power of the Raspberry Pi: would the Raspberry Pi be able to run the algorithm fast enough to provide sufficiently low-latency information for the robot to keep it safe? This concern was factored in while developing and designing navigation algorithms.

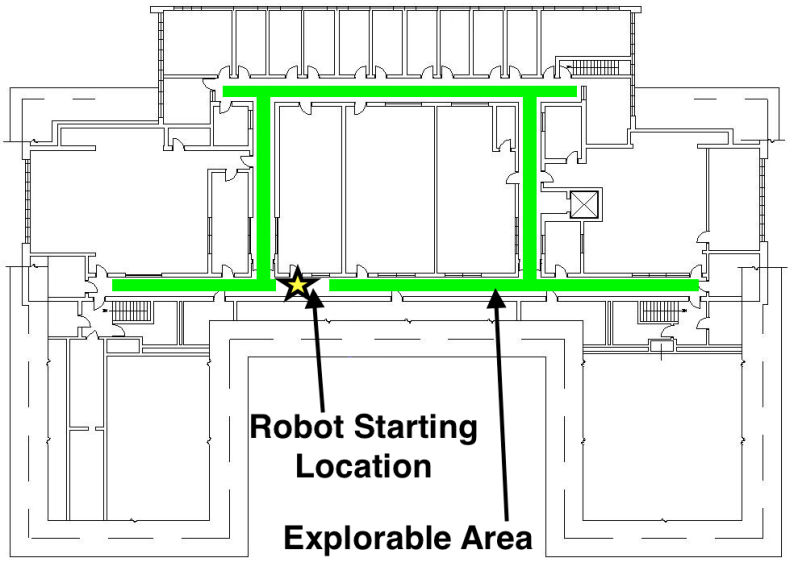


Figure 1: This is an old floor plan of the third floor of Atwater Kent, an academic building on Worcester Polytechnic Institutes campus. The shaded region on the map shows the explore-able region while the star represents where the robot started for each trial.

All of these considerations and decisions lead to the creation of the Roaming Spirit Platform. The system itself was successful in the tests provided. The robot was able to safely localize and navigate within a building. Provided with the area shown in *Figure 1* the system was able to produce a map that is similar in nature *Figure 2*.

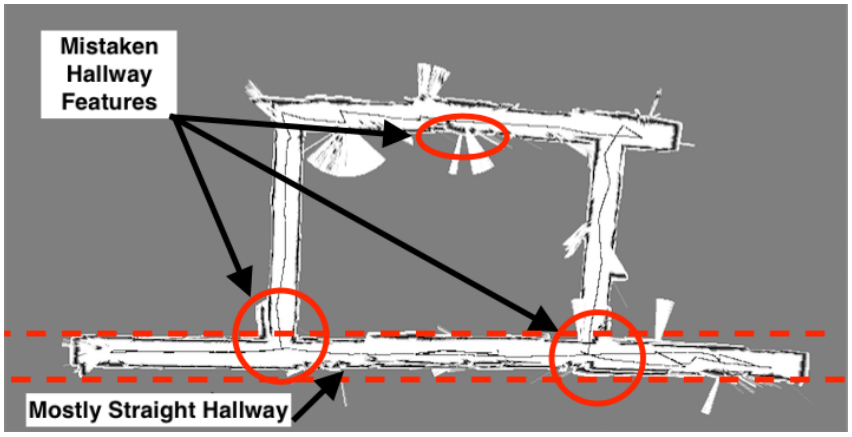


Figure 2: This map is an analyzed version of the output of the BreezySLAM library after a completed test. This shows the floor in *Figure 1* through the BreezySLAM algorithm.

It is important to note, that during the course of the project it was found that BreezySLAM does not perform loop closure and has a very low latency for adding objects to and removing objects from the map. This likely causes the Random Mutation Hill Climbing SLAM to provide poorer results as compared to the Deterministic SLAM. Additionally, from our tests, it seems probable that Deterministic SLAM produces better maps due to its higher reliance of the odometry data provided from the robot. This was one limitation of the success of the project, however, and would require more research into the SLAM aspect of the project.

Overall, this system was proved to be successful and as such can be continued in the future. With newer versions of the Raspberry Pi (Raspberry Pi 3) it may be possible to reconfigure the project to utilize a more demanding SLAM algorithm to get better results. The proof of concept also means that the system could be created in a lower level, faster, language like C/C++ to allow processing gains that would make room for better SLAM algorithms.

Contents

1 Introduction and Motivation **1**
- 1.1 Current State-of-the-Art . . . 1
- 1.2 Current Technological Challenges . . . 2
- 1.3 Proposed Solution and Requirements . . . 2
- 1.4 Report Organization . . . 3

2 Implementation Considerations and Choices **4**
- 2.1 Operating Environments . . . 4
- 2.2 Embedded Computing Technologies . . . 5
  - 2.2.1 Raspberry Pi 2 and Odroid C1 . . . 5
  - 2.2.2 Intel Compute Stick . . . 6
  - 2.2.3 Intel Next Unit Computing . . . 6
- 2.3 Sensing Technologies . . . 7
  - 2.3.1 Sonar . . . 7
  - 2.3.2 Depth Lasers . . . 8
  - 2.3.3 Stereo Camera . . . 9
  - 2.3.4 Mono Camera . . . 10
  - 2.3.5 RGB-D Sensor . . . 11
- 2.4 Localization . . . 12
  - 2.4.1 Waypoint Localization . . . 12
  - 2.4.2 WiFi Localization . . . 13
  - 2.4.3 SLAM . . . 14
- 2.5 Navigation . . . 16
  - 2.5.1 A Star . . . 16
  - 2.5.2 Tentacle Navigation . . . 17
  - 2.5.3 QR Code Navigation . . . 18
- 2.6 Software Organization . . . 19
  - 2.6.1 System Testability . . . 19
  - 2.6.2 Software Design . . . 20
- 2.7 Robotic Platforms . . . 21
- 2.8 Chapter Summary . . . 22

3 Proposed Approach and Expected Outcomes **23**
- 3.1 Picking a Design . . . 23
- 3.2 Selected Design . . . 23
- 3.3 Implementation Plan . . . 23
- 3.4 Expected Outcome . . . 24
- 3.5 Chapter Summary . . . 24

4 Implementation **25**
- 4.1 Asus Xtion and BreezySLAM . . . 25
  - 4.1.1 Asus Xtion Drivers and Communication . . . 25
  - 4.1.2 BreezySLAM Map Development . . . 26
- 4.2 Modular System Interface . . . 27
  - 4.2.1 Data Acquisition and Sensor Interfacing . . . 27
  - 4.2.2 Robot Controls and Interfacing . . . 28
- 4.3 System Overview . . . 29
  - 4.3.1 The Main Loop . . . 29
- 4.4 Frontier Exploration . . . 30
  - 4.4.1 System Initialization . . . 31
  - 4.4.2 Exploration Algorithm . . . 32
  - 4.4.3 Filling Out The Map . . . 34
- 4.5 Drone Testing . . . 35
- 4.6 Turtlebot Testing . . . 36
- 4.7 Chapter Summary . . . 37

5 Experimental Results **38**
5.1 Safe Movement 38
5.2 Distance Traveled 38
5.3 Localization and Navigation 39
5.4 Chapter Summary 41

6 Analysis and Future Projects **43**
6.1 Safe Movement and Travel Distance 43
6.2 Localization and Navigation 44
6.3 Chapter Summary 44

7 Conclusion **46**

8 Appendix **50**

List of Figures

1 This is an old floor plan of the third floor of Atwater Kent, an academic building on Worcester Polytechnic Institutes campus. The shaded region on the map shows the explore-able region while the star represents where the robot started for each trial. . . . . .
2 This map is an analyzed version of the output of the BreezySLAM library after a completed test. This shows the floor in *Figure 1* through the BreezySLAM algorithm. . . . . .
3 Projected growth of robot industry between 2010 and 2025 [1]. 1
4 The Intel NUC is a series of smaller versions of a regular desktop computer (i.e, a mini-computer). 6
5 This figure depicts a generalized example of an Ultrasonic Sensor emitting an original wave and said wave bouncing off an obstacle and reflecting back to the sensor. The sensor is then able to use different characteristics about the reflected wave to calculate the distance to the obstacle. 7
6 This is a diagram of the Vex Ultrasonic Sensor sensing the distance to a wall. As shown in the diagram, a sound wave is emitted by the sensor and the reflection of the wave is later sensed. With the fixed distance between the emitter and receiver, the system is able to calculate the straight line distance (the plane of the sensor to plane of the contact point) that the sound traveled. Photo courtesy of vexrobotics.com [23]. 8
7 This diagram shows two cameras with one tree within both their field of view. By using the field of view and basic trigonometry it is possible to get a distance calculation to the tree in view [27]. 9
8 Experimental Stereo Camera Mount machined for testing and proof of concept. This mount would allow two cameras to be mounted at a fixed distance to ensure that back-end calculations can rely on a constant distance. 10
9 a: Generating a vector by comparing the location of the red ball over multiple frames (overlaid into one picture). b: The result of using OpenCV's optical-flow methods to track features in a video of a highway, thus generating the colored lines. Images courtesy of OpenCV Docs. 11
10 a: The Xbox Kinect b: The Asus Xtion. Both of these sensors are RGB-D sensors, utilizing an infrared dot matrix in its calculation of depth. Note: Images are not to scale. Images courtesy of Microsoft and vr-zone.com, respectively. 11
11 Experimental output depth map of the XBox Kinect Sensor. The XBox Kinect calculates the depth by emitting an infrared array of dots, . The left image is the represented depth images outputted by the XBox Kinect and the right image is the RGB image 12
12 Wall-E would be able to determine in which room - *e.g.* localize - he is in if he could detect color, read the labels, or in some way know where or how he moved. 12
13 Image of Amazon Robotics system in action. The QR codes are the small white squares on the ground, the robots scan these when they are over top of them and move on from there. [36] 13
14 This figure depicts a graph of colored squares representing the strength of the system at that point, with Access points at locations (60,70) and (110,10). WiFi fingerprinting uses a graph similar to this to attempt to localize the system. Meaning, if the robot is in a square with a value of 40db, then there is only one location on the grid it can be - at location (20,10). 14
15 a: A robot with GPS (which gives a location accurate to the blue ellipsoid) and odometry moves with some predictive model (its velocity) to a new location. The Kalman filter combines the sources of information to create a new estimate, in purple. b: The graph depicts the combination of the two sets of data to create a new position that has less error (depicted by the ellipsoid thickness). 15
16 Depiction of how A* works. A*, by design, will create the optimal path between two points. The start and end points can be seen on the figure, along with the optimal path. The expanded nodes are representative of the locations that A* considered while building its path. This photo courtesy of Stanford University [39]. 16
17 Depiction of how Tentacle Navigation works. The yellow lines are the possible routes that the vehicle can take. The white regions describe obstacles in the environment. The tentacles "feel" out where the areas with obstacles are, and thus guide the car towards open areas. [40] 17

18 Depiction of how Tentacle Navigation works given a goal. The system extends tentacles out in given directions to sweep its surroundings and discover the only viable paths to take. In the figure, the system extends tentacles about the car to discover the edge of the lanes and the car in front of it. It then reasons about the plausible paths that can be taken and follows that tentacle [40]. 18
19 These figures depict R2 utilizing QR codes to navigate. As R2 identifies different codes it is able to calculate the location of the astrodroid within its environment. 19
20 This is a Unified Modeling Language (UML) diagram for the Strategy Pattern. It depicts the parent class, the robot, that contains a realization of the Sensor interface. This realization could be either some type of RGBD Sensor or LiDAR. This diagram was created for example purposes, in a real UML diagram, the interface would document the function definitions that the realizing classes would need to implement. 20
21 This figure shows the Parrot AR Drone 2.0 unmodified (a) and with the modifications made while the robot was running payload tests (b). 21
22 This figure shows the Parrot AR Drone 2.0 unmodified and with the modifications made while the robot was running payload tests. 22
23 This figure represents a gantt chart of the tasks and goal for implementation in the project. This covers the seven weeks representing the B Term of the 2015-2016 school year at Worcester Polytechnic Institute 24
24 These maps are different variations of the parameters (input and map size) given to BreezySLAM. As the parameters change, the ability for BreezySLAM to make the map changes (this change also depends on which version of BreezySLAM is being used, deterministic of RMHC). The roaming Spirit platform utilizes deterministic SLAM with parameter values of 1000 iterations and 20 mm and 1 degree of rotation. The size of the map should be appropriately chosen for the area to be mapped. 27
25 This diagram is a flow chart of the Roaming Spirit Platform's implementation of initialize(). It was implemented on top of Pythons Socket library. 29
26 This is a flow chart depicting the main loop of the Roaming Spirit System. As shown, the system updates the navigation information, and then converts that command to robot move data. Then, it couples that move data and a sensor scan to update the SLAM algorithm. Finally, it logs the position and repeats. The system uses defensive programming strategies to ensure that every step of the way valid data is being passed along and the system can continue without undocumented error (as can be seen with each of the 'is Good?' conditionals). 30
27 This flow chart represents the actions taken by the Frontier Exploration thread that is utilized to maintain the path directions generated by the Roaming Spirit platform. 31
28 This flow chart represents the actions taken by the Frontier Exploration thread when a new route is generated. The system generates the tentacles and builds frontiers from their tips. It then locates the midpoint of the largest frontier and converts it into robot space and returns said value so the Roaming Spirit Platform may direct the robot correctly. . . 31
29 These side by side figures show the viewing angle and range of different sensor types that could be used for the system. 32
30 An illustration of how the system utilizes tentacles to search the direct area. It also visually depicts the systems knowledge post search. 33
31 This flow chart represents the actions taken by the Frontier Exploration thread when it begins to execute the tentacle generation portion of the route generation. 33
32 This is a visual representation of the assumption made by the algorithm. In this image, the system has made its assumptions about the different areas between the tentacles. . . . 34
33 Generated map that has been marked to identify the location of the robot and the location of the unexplored frontiers. 35
34 This is a flow chart depicting the main loop of the Roaming Spirit System. 36
35 This diagram shows the implementation of the Asus Xtion on the Turtlebot. The cyan obstruction prevents the Xbox Kinect - attached to the Turtlebot - from displaying its own infrared dot array, potentially interfering with the Asus Xtion. The Asus Xtion is then placed in front of the obstruction to view the surrounding environment. Image modified from Clear Path Robotics [43] 36

36 In the actual implementation, the obstacles used are the pink Post-it notes covering the infrared display and the camera. A spring-action electronics board holder supports the Asus Xtion, and the Asus Xtion is placed close to the center of the Turtlebot. 37
37 This is an old floor plan of the third floor of Atwater Kent, an academic building on Worcester Polytechnic Institutes campus. The shaded region on the map shows the explore-able region while the star represents where the robot started for each trial. 39
38 This map is a parameterized BreezySLAM maps created from a log file of odometry data to view the maps that BreezySLAM created. The odometry data is taken from the Parrot AR 2.0 Drone while it was walked around the third floor of an academic building on Worcester Polytechnic Institutes Campus (Atwater Kent). Both trials use SLAM parameters that are set to trust the BreezySLAM library significantly more than the output of odometry data. 40
39 These figures are two separate trials of the roaming spirit platform driving the Turtlebot around the third floor of an academic building on Worcester Polytechnic Institutes Campus (Atwater Kent). Both trials use SLAM parameters that are set to trust the odometry of the robot significantly more than the output of BreezySLAM. 41
40 This map was created by using a Turtlebot hardware and the default ROS packages software (default move base stack and gmapping algorithms). The map is of the third floor of the academic building Atwater Kent on Worcester Polytechnic Institutes Campus. 41
41 This image shows the a sketch of a Turtlebot in the test conditions used during the course of this project. This image also depicts the vision range of the Asus Xtion. All objects found in the green region will be accurately recorded. Anything detected within the 'sensor deadband' area will cause invalid measurements. 43
42 This is a closeup of the bottom right hallway intersection of 39.b. It shows the error in the Roaming Spirit system caused by the BreezySLAM failure. 44
43 This is a diagram depicting the overview for the software from beginning to end. This shows how the Main thread initializes each of the different classes that are utilized in the Roaming Spirit platform as well as the different threads (main and navigational) that spawn off at run time. Finally, this diagram shows the 'server' thread that is used to broadcast the map in real-time to allow users to watch as the map is created. 50

List of Tables

1 The table compares the different considerations among the investigated embedded computing systems that we found during our research [17], [19], [21]. Since we were more concerned with ensuring that the project had the best chance of succeeding within our time constraints, rather than seeking to choose the most powerful computing system or the lightest, we sought to use the Raspberry Pi as it was the computing system with which we had the most experience, and the most mature product of those that we investigated. We decided that some of the computational trade-offs of using the Pi in favor of the other systems were outweighed by the security of using a system that has had much interest by the general population for the longest time. 6

2 This table depicts the results of trials for safe control of a robot via the Roaming Spirit platform. In this table, it shows 10 trials and the number of times that an event occurs. The events listed in the table are visible collisions, invisible collisions, wall blind scenarios, wall blind associated collisions. A visible collision is a scenario where the robot can see what it into. An invisible collision is a scenario where the robot cannot see what it's running into. A wall blind scenario is what happens when the robot mistakenly gets too close to the wall such that it should be able to perceive it but cannot. A wall blind collision is when the robot is in a wall blind scenario and collides with the wall/obstacle. 38

3 This table displays the safe distance that was traveled by the robot in a given trial. The distances displayed are 60, 120, and greater than 150 feet. The 60 and 120 feet duration were chosen as part of the requirements for the project. 150 feet was decided as it was the a good cut off to delineate trials that were much longer than the 120 feet, while still displaying some of the trials that made it past 120 feet but did not go significantly longer. 39

Acronyms

- **ARM:** Advanced RISK Machine
- **BSSID:** Basic Service Set Identification
- **DARPA:** Defense Advanced Research Agency
- **EKF:** Extended Kalman Filter
- **GPS:** Global Positioning System
- **IMU:** Inertial Measurement Unit
- **Intel NUC:** Next Unit Computing
- **KNN:** K-Nearest Neighbors
- **LiDAR:** Light Detection and Ranging
- **MAC:** Media Access Control-Address
- **Open SSL:** Open Secure Sockets Layer
- **PR2 Robot:** Personal Robot 2
- **PTAM:** Parallel Tracking and Mapping
- **PVC:** Polyvinyl Chloride
- **QR Code:** Quick Response Code
- **RGB-D:** Red, Green, Blue, Depth
- **ROS:** Robot Operating System
- **SDC:** Self Driving Car
- **SIFT:** Scale Invariant Feature Transform
- **SLAM:** Simultanious Localization and Mapping)
- **SONAR:** Sound Navigation and Ranging
- **SURF:** Speeded Up Robust Features
- **SVM:** Support Vector Machine
- **TDD:** Test Driven Development
- **USB:** Universal Serial Bus
- **USD:** United States Dollar
- **V-SLAM:** Visual SLAM
- **WALRUS:** Water And Land Remote Un-manned Search Rover
- **WiFi:** Wireless Fidelity
- **WPI:** Worcester Polytechnic Institute

1 Introduction and Motivation

Globally, the market for robotics is expected to grow from a 26.9 billion USD industry in 2015 to an estimated 67 billion USD industry by 2025 [1]. As of 2015, the distribution is approximately 7.5 billion USD on military robots, 11 billion USD on industrial robots (such as automated welders), 5.9 billion USD on commercial robots (such as crop monitoring and medical robots), and 2.5 billion USD on personal robots (such as the Roomba vacuum cleaner) [1].

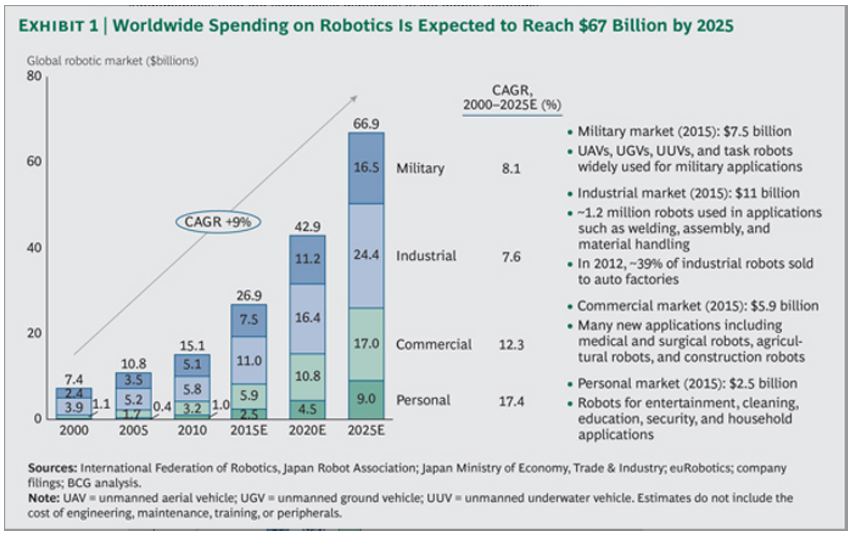


Figure 3: Projected growth of robot industry between 2010 and 2025 [1].

This large scale economic growth in personal, commercial, industrial and militaristic industries, have caused heavy demand for the innovation of more intelligent robotic systems. Universities, such as Worcester Polytechnic Institute (WPI), Lawrence Technological University, and the University of Pennsylvania (UPenn), have created undergraduate and graduate programs in robotics [2],[3],[4]. This has lead to development of a wide range of robots, such as WPI's search and rescue WALRUS robot [5], or UPenn's swarm quadrotor project [6].

As the demand for robotics rises, innovation for a robot's ability to sense its environment and innovation for a robot's ability to interact with its environment likewise increase. Similarly to humans, a robot's ability to interact with its world is limited by its ability to sense its world. Autonomous vehicles, such as Google's Self Driving Car (SDC), could not safely navigate its lanes without knowledge of their surroundings; Google spends upwards of $75,000 USD for each LiDAR sensor alone so that their SDC can attain local environmental information [7]. Likewise, in order for cars to plan routes to a final destination, they need to know their current location. This can be done through the use of a GPS unit or through probabilistic approximation of limited scope mapping. Irrespective of the method, perception is what allows robots to function. For this reason the project group decided to focus on the utility of robotic perception for the scope of the project.

The goal behind this project is to create a cross platform perception system that allows researchers to expand their robot's functionality, so that the researchers themselves do not need to learn how to implement localization and navigation. It is the belief of the project group that a platform to perceive localization and navigation information and return location and navigation commands would simplify robotic systems and allow developers to focus on the higher functionality that mobility could provide. While outdoor localization and navigation is a difficult project, we believe that preparing something capable of competing with the methods developed by SDC developers is not possible within the time allotted for this project. Therefore, the focus of this project is to create a cross platform localization and navigation system that communicates localization and navigation information to the robotic system on which it is attached.

1.1 Current State-of-the-Art

Indoor localization and navigation is a complicated problem, though it has a few existing solutions; a common solution is to utilize Simultaneous Localization and Mapping (SLAM) to discover the surround-

ings of the robot as it moves [8]. Although these algorithms have been proven to work, the sensors used for their implementation varies widely for each robot.

For example, iRobot's Ava 500 is a robot that navigates indoor areas through use of a LiDAR, three 3D imagers, three Sonar sensors, and bumper switches. Although this solution is highly effective, it comes at a steep price: the Ava as a whole costs $69,500 USD [9]. Not all development groups can afford to implemented three 3D imagers alongside a LiDAR.

For a more cost effective solution, developers can choose to purchase the Turtlebot platform for $2,000 [10]. The Turtlebot come equipped with a RGB-D (Red Green Blue Depth) camera that interfaces with encoders and a gyroscope, which it uses when implementing SLAM for navigation. The Turtlebot utilizes the Robot Operating System (ROS), which allows for easy development. The major downside to this platform is that it is limited to the domain of ground vehicles. This platform could not be implemented on an aerial drone without major changes to the sensor algorithm.

Similarly, Carnegie Mellon and the Sensible Machines's Office of Naval Research were able to develop a custom quadrotor that could support their 100 grams of computing and up to 387 grams of sensors to drive their algorithm and navigate indoors [11]. This specific project showed the capabilities of SLAM navigation and its possibilities in indoor aerial robotics.

1.2 Current Technological Challenges

Cost and implementation difficulties are two of the largest challenges facing current fully robotic solutions and partially complete (*i.e.* component) solutions. Complete solutions, such as Boston Dynamic's Atlas, can cost upwards of 2 million dollars to purchase. Although the system comes fully equipped with 4 force sensors for the hands and feet, 2 IMU's, stereo cameras, LiDAR, pressure sensors, encoders, and potentiometers, 2 million dollars can be difficult to justify for smaller development groups - many universities that have robots received the units as grants from companies or government agencies, such as DARPA. On the other hand, systems such as the Turtlebot start at $2,000 but are far less versatile. Although the Turtlebot comes with fully implemented localization and navigation through ROS, the potential for additional augmentations to the base is limited to lightweight alterations. This is due to the use of the Kobuki base in the Turtlebot: according to the Kobuki User Guide (version 1.0.6), the maximum payload for the Kobuki base is 5 kg [12].

Alternatively, roboticists in need of assistance with software can utilize ROS. ROS was designed to abstract the communications between robotic systems, such as sensors, much like how a normal operating system abstracts the hardware interactions from the user. ROS is used on a number of robotic systems including the PR2, Turtlebot, and can even be implemented on Atlas. The downside to utilizing ROS stems from its large computational overhead, somewhat steep learning curve, and the complexities of implementation when the target robot that already has some type of software framework implemented. There are scenarios in which utilizing ROS is not necessary - the introduction of ROS would created added complexity and software overhead causing the overall execution time to slow. In this scenario, it is important that the roboticists implement their own system organization; which is complicated.

1.3 Proposed Solution and Requirements

We propose to create a system that will abstract the task of devising a system for indoor navigation for any robotic platform. Given the wide range of environments with which one could use this system, our proposed solution will focus on specific indoor environments. The objectives of the project are as follows:

- The system will allow for fixed height navigation within a static indoors environment:

  (a.) At minimum, our proposed framework will allow the system to navigate a hallway at a fixed height without collision. The system would be considered successful if it's able to traverse 60 feet of hallway, 80% of the time or if the system able to determine that progressing forward is either too dangerous to itself or the surrounding environment. At 60 feet, the framework would only be required to traverse a straight hallway containing no hallway intersections, thus 60 feet will represent the simplest form of the challenge.

  (b.) At an improved level, the system must be able to function in non-straight hallways, as well as hallway intersections and doorways. The system must be able to coexist in an environment with no more than one, non-harming, dynamic obstacle at a time. The system would be considered successful if it's able to traverse 120 feet of hallway (including at least one hallway intersection or 1 strip of non-straight hallway) 80% of the time. During this evaluation, the

system must be able to detect at least one of the doorways in the testing environment. At 120 feet, the framework would be required to traverse a straight hallway containing at least one hallway intersections, thus 120 feet will represent an advanced form of the challenge.

(c.) The reach goal of this system is the implementation of Simultaneous Localization and Mapping. The system must be able to traverse while making an accurate map of its surroundings.

- An accurate map will be defined as a map with which the system can reliably use as a way to navigate through while avoiding obstacles.

- The system will establish best estimate localization:

(a.) At minimum, the system will work towards a probabilistic model of where the robot could be, narrowing the locations down over time. At the end of a trial, the proposed framework should be able to estimate the location of the robot on a map.

- This implies that localization will be dependent on navigation and it's ability to both navigate while learning about it's environment to update the probabilistic model.

(b.) At an improved level, the system must be able to probabilistically estimate the correct room/area the robot is located in. This involves the system detecting and protecting against 'teleporting' - or seemingly phasing through walls - while making a best effort to place a node on a map within a 6 foot radius of its location at end of a trial.

- The six foot radius is where the robot has an 80% chance of being within the radius.

(c.) The reach goal of this system is the ability for the system, given a map, to be able to localize to within a 3 foot radius. Given a map, but no other previous knowledge of its location on the map, the system should be able to determine its position on the map before the end of the run.

- This reach goal is independent of the navigational goal of implementing SLAM, which includes localization.

1.4 Report Organization

Following this introductory chapter, the report will begin by outlining the implementation considerations, choices, and fundamental concepts that were considered for this project. It will then take a chapter to discuss the proposed approach for the solution, followed by a chapter on implementation of the specifics of said approach. This will be followed by a chapter reporting the results of the project and a chapter discussing the limitations and future considerations of the project. Finally, there will be a chapter for concluding the project.

Each chapter will start with a high level introduction detailing the information contained in the chapter. Each chapter will be broken down into sub-chapters, delving into the different components of the chapter in greater detail. Finally, the chapters will end with a chapter summary to make an effort to highlight the take-aways within the chapter.

2 Implementation Considerations and Choices

In the following section, we illuminate the logic and processes that led to our final implementation of the Roaming Spirit system. We explore and analyze the different kinds of technologies and algorithms available, and our reasons for their resultant implementation, if used at all.

2.1 Operating Environments

The domain in which mobile robots operate is massive; there are aerial, terrestrial, aquatic, and deep space environments that robots - in some shape or form - have traversed. These robots can range from interplanetary robots such as NASA's Curiosity rover [13] that must face the harsh, rugged, Martian environment to iRobot's Roomba, which might face hardwood or carpeted floors. A sensor suite that can that can be generalized to such a large domain is simply not feasible. Given the environmental considerations needed when designing aquatic or deep space robots, such as water proofing and thermal shock resistance, we decided to focus instead on the more environmentally similar domains of terrestrial and aerial robots.

This domain still poses a large challenge. However, given that aerial robots and terrestrial robots are increasingly popular among the general population, we decided to decrease our sensor suite's domain between indoor and outdoor robots rather than between aerial and terrestrial robots.

For both aerial and terrestrial robots, the outdoor domains have a series of similar considerations. Assuming one is not simply dealing with a consumer's backyard, these robots must be able to move fast enough to be useful, and therefore the sensor system must be quick enough to deal with the rapid change of position. Any sensors used must take into consideration the effects of the sun, any nearby traffic, highly random obstacles, sudden sounds, electromagnetic interference from cellphones and radios, and survive collision while moving at speed.

Commonly, roboticists will deal with some of these issues by operating away from environments in which they do not want to consider possible interference (such as operating in open fields to avoid issues with collision). Additionally, roboticists may implement some type of manual override to handle unforeseen circumstances and guide the robot through particularly complex environments. With such precautions in place, robots need not worry so much about high performance sensors. The trade-off, of course, is that such robots must be constantly monitored or their environment must be carefully chosen.

When dealing with aerial robots outside, one cannot operate within a 5 mile radius of any airport without first communicating with the airport and its control tower [14]. This poses a special problem for us, since Worcester Polytechnic Institute is 4.8 miles away from Worcester airport (IATA Location Identifier ORH) [15]. Additionally, it is against university policy to fly robots with recording devices of any kind near dorms due to privacy issues. Worcester also tends to be very windy, which can pose problems when dealing with stabilization of the aerial robots. We did have the option to drive to a few open areas away from airports, since both of us drive.

In the indoor domain, aerial and terrestrial robots need not move particularly quickly, since the distances required to travel are smaller. Sensors used in this environment also need to take into account the effects of indoor lighting, sunlight through windows, electromagnetic interference from generators, cellphones, WiFi, bluetooth, and survive collisions moving at speed. In many cases, the robots cannot access GPS information since the signal has trouble penetrating through buildings. WiFi localization is one form of localization available for the indoor robots, but triangulation requires three separate access points.

However, indoor environments tend to be more deterministic because there are more similar attributes between indoor environments than similar attributes between outdoor environments. For example, one can more easily make assumptions (such as assuming the floor will be flat) that one cannot reliably make with outdoor environments. Since the number of uncertainties between indoor environments, with the exception of warehouse-like buildings, do not generally have a wide range of open space, making navigation somewhat difficult for aerial vehicles that cannot hover. That being said, there is also no wind to destabilize them (although drafts and fans can be a concern), so drifting is minimal.

Since we can make reasonable assumptions about the expected environment in which an indoor robot may operate (such as a low probability of rain in a building), we can decrease the domain that we must deal with while still having a sensor suite that can generalized for use on many indoor robots. An outdoor sensor suite would require more work for the same type of generalizable solution, due to the more deterministic nature of indoor environments. Additionally, needing to drive away each time we wished to test the sensor suite on aerial robots is too much of an inconvenience, considering that we can

test indoor aerial robots without needing to leave the campus. Therefore, we have chosen to create the sensor suite for indoor, mobile robots.

2.2 Embedded Computing Technologies

For the scope of this project, there were two processing related choices. The first choice was to do all of the processing off-board of the system. This means that the sensor suite would relay information to a base station where the data would be processed. The base station would then either make decisions and send motion commands, or send the processed data back the system to make its own decisions. The second choices was to design the system to do all of its processing on-board. Such a system would require the implementation of a system robust enough to efficiently sense its surroundings, create decisions from the data, and return the decisions in a timely manner.

We knew of the Raspberry Pi as a common and popular embedded computer used in several automation projects. Using the Raspberry Pi as a standard for comparison, we researched several other embedded computers, such as the Odroid C1. We then explored alternative computing methods, such as the Intel compute stick, and finally into larger compact systems have better processing methods, such as the Intel NUC. The subsections below will outline the most viable solutions discovered and the accompanying strengths and weaknesses.

2.2.1 Raspberry Pi 2 and Odroid C1

The Raspberry Pi is a popular embedded computer, selling more than four million [16] boards to a community that has been growing since 2009. From our research, its top competitor is the Odroid C1, this section will compare the two and provide a reasoning on why one was picked over the other.

As seen in *Table 1*, when it comes to processing power, the Odroid is more powerful than the Pi. Even though the Raspberry Pi utilizes a Cortex A7 over the Odroid's Cortex A5, the Odroid operates at 1.5 GHz compared to the 900 MHz for the Raspberry Pi. This allows for much faster processing of code, which in turn helps with response time and reliability of a robot. The faster speed was also revealed when utilizing the Open SSL command line tools speed option to measure the time required for testing its cryptographic algorithms; displaying the Odroid as faster than Raspberry Pi by about 40 to 50% [17] [18].

However, the Raspberry Pi is a more mature product. The Raspberry Pi was one of the first embedded computers to hit the market in 2011 [19]; its development community is large and continuously growing. The Raspberry Pi has more community based resources (videos, books, blogs, and forums) compared to the Odroid. As a result, less research would be needed to fully utilize the Raspberry Pi as compared to the Odroid. Since our greatest limiting factor regarding this project is time, we decided that it is more effective to implement the Raspberry Pi than to use the Odroid.

Table 1: The table compares the different considerations among the investigated embedded computing systems that we found during our research [17], [19], [21]. Since we were more concerned with ensuring that the project had the best chance of succeeding within our time constraints, rather than seeking to choose the most powerful computing system or the lightest, we sought to use the Raspberry Pi as it was the computing system with which we had the most experience, and the most mature product of those that we investigated. We decided that some of the computational trade-offs of using the Pi in favor of the other systems were outweighed by the security of using a system that has had much interest by the general population for the longest time.

Traits	Raspberry Pi 2	Odroid	Intel CS	Intel NUC
Price	$35	$35	$150	$260
UART	Yes	Yes	Yes	Yes
I2C	Yes	Yes	No	No
SPI	Yes	Yes	No	No
Processor Speed (GHz)	0.9	1.5	1.3	2.4
Memory (GHz)	1	1	2	2
Onboard Network (Mb)	10/100/1000	10/100/1000	10/100/1000	10/100/1000
GPIO pins	40	40	0	0
Weight (grams)	45	40	50	907
Time In Market	2009	2012	2015	2013

2.2.2 Intel Compute Stick

During our research, we discovered the Intel Compute Stick. As seen in Table 1, this thumb-drive sized embedded computer weighs at just over 50 grams, and is advertised to be a fully functional, general purpose computer [20]. This thumb-drive computer was seen as a possible replacement to the Raspberry Pi; with similar weight and similar processing specifications [21] the Intel compute stick would provide ease of integration into alternate systems due to its thumb-drive shape and size. We also considered using the Intel stick as a complementary processing unit for the Raspberry Pi. By offloading resource intensive actions, such as visual processing, onto the Intel Stick, the Pi would be able to spend more of its processing resources in analyzing any other sensors.

However, the Intel Stick was released at the start of this project. This means that, while there was much anticipation for the product, there was no community of users or support for it. This means that any complications with Ubuntu distributions, driver utilities, or bugs in general might not have been discovered or resolved. While the product is doing well, the Intel stick did not have the maturity for us to confidently use it; therefore it was not implemented in this project.

2.2.3 Intel Next Unit Computing

The Intel Next Unit Computing (NUC) is another Raspberry Pi replacement we researched, as shown in Table 1. It is a two pound computer that, depending on the model, contains an Intel Core i3 processor up to an i7. The NUC series are effectively desktop computers, but smaller in size.

Figure 4: The Intel NUC is a series of smaller versions of a regular desktop computer (i.e, a mini-computer).

This system would all but eliminate the concern of processing resources, thus providing rapid processing speeds for lower latency in our project. However, due to its weight, the Intel NUC greatly limits the type of robot one can use, particularly when considering drones. Therefore, we decided that an embedded PC such as the Raspberry Pi or the Odroid would serve a better purpose than the NUC.

2.3 Sensing Technologies

Accurate data from sensors is paramount to ensuring our system is capable of correctly detecting its environment. There are many aspects to consider when choosing sensors: one must consider the effects of potential signal interference, the behavior of the sensor while in a moving robot, and the range with which the sensor would be expected to operate. Additionally, one must ensure that the sensors on our system do not interfere with the other sensors on the robot (and vice versa). The following subsections will outline the different advantages/disadvantages to sensors observed while researching. It will also explain why a specific sensor was or wasn't chosen for use.

2.3.1 Sonar

One of the first sensors researched was a sonar sensor. This sensor works by emitting sound and measuring the response time of its echo to calculate distance. This type of sensor has been successful in safe navigation technologies for submarines and more recently for people who are visually impaired. G-Technology Group created a set of sonar glasses to provide the visually impaired with more information about their surroundings [22].

The speed of sound at room temperature is $340.28\frac{m}{s}$. Using the relationship $D = S * t$, where D, S, t are distance, speed, and time, respectively, one can determine the distance between the sensor and the first object to return a signal by measuring the time it takes to sense the signal.

Sonar is quite effective at detecting the presence of objects. As seen in *Figure* 5, the sensor is able to detect that something exists some distance away. We briefly researched whether an array of these sensors could provide the information necessary for localization and navigation.

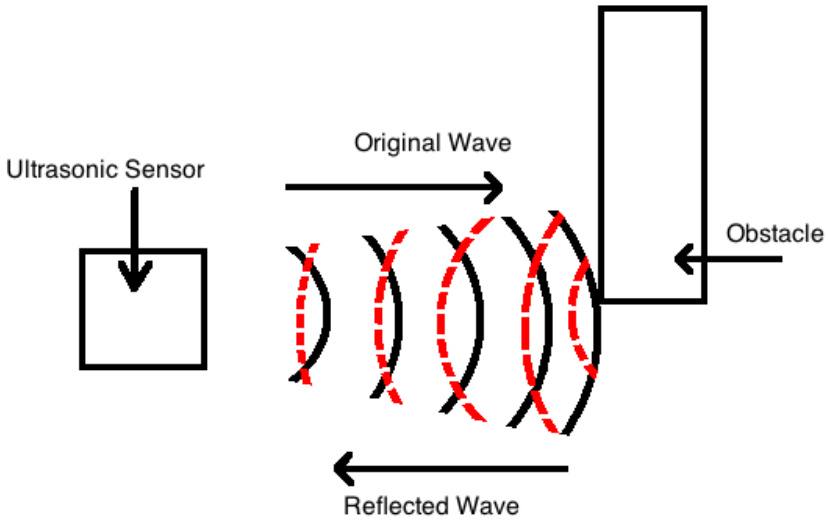


Figure 5: This figure depicts a generalized example of an Ultrasonic Sensor emitting an original wave and said wave bouncing off an obstacle and reflecting back to the sensor. The sensor is then able to use different characteristics about the reflected wave to calculate the distance to the obstacle.

For example, the Vex Ultrasonic Rangefinder (as seen in the diagram of *Figure* 6) utilizes a sound emitting speaker to create a high frequency wave that is then detected upon reflection by a sound detecting sensor. The Vex Ultrasonic Rangefinder then utilizes the time of flight to calculate the time that the wave was traveling, and thus the distance to the object.

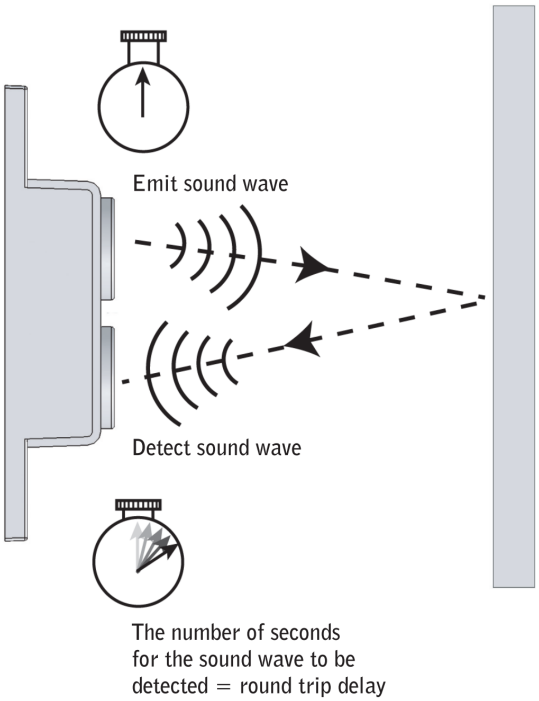


Figure 6: This is a diagram of the Vex Ultrasonic Sensor sensing the distance to a wall. As shown in the diagram, a sound wave is emitted by the sensor and the reflection of the wave is later sensed. With the fixed distance between the emitter and receiver, the system is able to calculate the straight line distance (the plane of the sensor to plane of the contact point) that the sound traveled. Photo courtesy of vexrobotics.com [23].

We found, however, that a large disadvantage of these sensors is the amount of information they provide. The sonar sensor provides information about a small, conical region at some distance away. The closest object within this cone is hit by the sound first, and the sensor receives its echo. Therefore, one only knows of the distance of the closest point in the cone, and any of the echos reflected from later objects are ignored entirely.

Such little data can make localization more difficult. Without perfect knowledge of how a robot moves, one needs to find unique features to help identify where a robot is with respect to its environment to help cope with error in the robot's odometry. Since only the closest point in a cone is returned, many of the features that are otherwise present in this cone are ignored.

Additionally, these sensors interfere with other sonar sensors. Since a sonar sensor sends out sound, and waits for the echo, any sound that a separate sonar sensor produces has the potential to be mistakenly interpreted by the first sensor as its echo.

While the alternative could be to provide a diverse number of supplemental sensors, we decided to look into other sensor types due to concerns about noise and signal clarity. For these reasons, the sonar sensors were not used in this project.

2.3.2 Depth Lasers

Another type of depth sensor researched were light based depth sensors. These utilize light emission to measure distance through either the angle of return or time of flight measurement. Both types of light sensors would provide a higher precision result due to the greater resolution of a thin beam of light. However, these sensors too only give a single point of distance data per sensor. One would either have to use multiple light to help identify unique features, or in some way rapidly move the single sensor to attain several different data points.

To solve this problem, we looked into rotational LiDAR systems. The rotational LiDARs have been implemented robots such as Google's SDCs [24] or Boston Dynamic's Atlas robot [25]. The LIDAR is a Light Direction and Ranging sensor that emits light to measure distances. The sensor alone is just a depth laser as referenced above, however most LiDARs implement a 360 degree rotational scanning as part of the sensor. This technique allows the depth laser to acquire a large amount of different data

points corresponding to the surrounding environment.

At first, these sensors were considered to be great candidates project. Over time however, these sensors turned out to be very difficult to find and purchase. Among the cheaper LIDAR we found initially (and after much research) was from VelodyneLidar.com, priced at $7,999 [26]. While this sensor could be perfect for the job, it is not within the budget of the project.

2.3.3 Stereo Camera

One method of depth sensing is to utilize a stereo camera system with a known distance between the two cameras. This allows the processing system to find similarities in the two images, measure the disparity between the images, and then calculate the depth. By utilizing a known distance between the two cameras, the system is able to calculate the depth of an object by comparing the location of an object between the camera. By representing the field of view of both cameras as triangles, along with the location of a point with respect to both cameras as a triangle it is possible to use trigonometry to calculate the distance to the point. The only requirement is that the field of view is known for both cameras, the photos are taken at the same time (e.g. the frame rate of the cameras are in sync), and that the cameras are aligned on the same horizontal plane and have a known vertical and angular error [27] [28] [29].

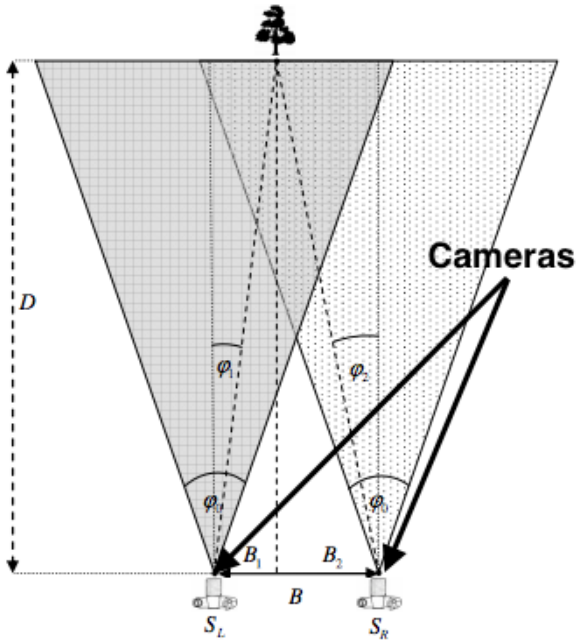


Figure 7: This diagram shows two cameras with one tree within both their field of view. By using the field of view and basic trigonometry it is possible to get a distance calculation to the tree in view [27].

In a paper written by Mrovlje and Vrančić, it was shown that the distance to the point, in this case the tree in *Figure 7*, can be calculated as a function of the distance between the cameras and the sum of the tangent of the related angles of viewing. This can be seen in equation 1.

$$D = \frac{B}{tan(\phi_1) + tan(\phi_2)} \tag{1}$$

Implementing this sensor would prove to be one of the lightest solutions available; a single Raspberry Pi cameras weighs only 3 grams. With two cameras and a light weight PVC mount, the experimental system created was just upwards of 50 grams. The experimental fixture we machined can be seen in *Figure 8*.

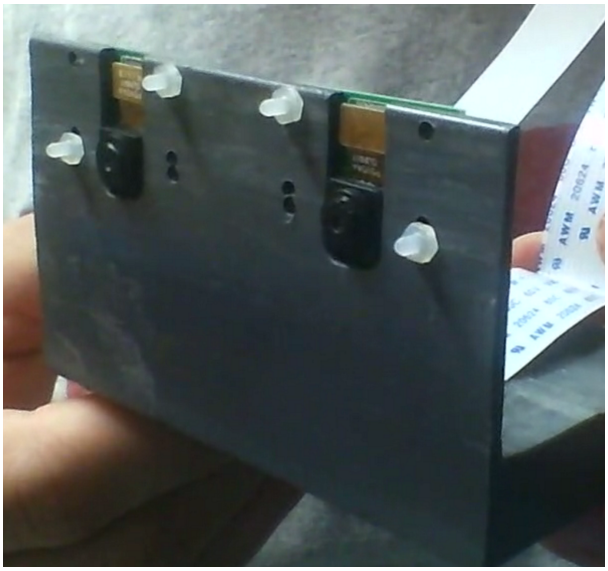


Figure 8: Experimental Stereo Camera Mount machined for testing and proof of concept. This mount would allow two cameras to be mounted at a fixed distance to ensure that back-end calculations can rely on a constant distance.

Unfortunately, because we had elected to use the Raspberry Pi for our sensor suite, stereo camera vision is not a viable solution. The Raspberry Pi has two 15 pin ribbon cable slots, with only one that is input capable. Because the Raspberry Pi cameras are 15 pin ribbon cable outputs, the Raspberry Pi cannot handle two of those type of cameras. As an alternative, it is possible to utilize two identical USB cameras. However, the USB cameras tend to be heavy and negate the weight benefits of the Raspberry Pi cameras. Though this solution was kept in mind as a last resort solution, it was not chosen.

2.3.4 Mono Camera

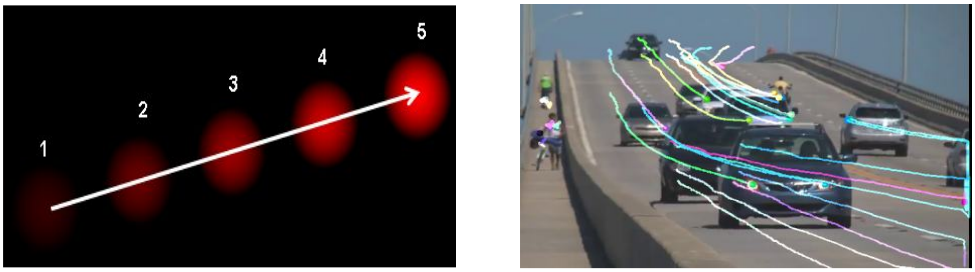
After discovering that the Raspberry Pi can only support one Raspberry Pi camera, we researched monocular-vision based localization and navigation. At first, it was very unclear how depth analysis occurred; instead we found a wealth of information regarding machine learning. It became clear that obstacle avoidance was possible using Support Vector Machines (SVM), K-Nearest Neighbors (KNN), and Neural Network (NN) algorithms [30]. All three of these algorithms worked well in scenarios with large amounts of data.

Basically, we would need to acquire video from the perspective of the camera and manually maneuver the camera through an environment that a robot might expect to traverse. We would then need to label all of the pixels that correspond to open space and those that correspond to obstacles. The machine learning algorithms then uses part of this data (known as the training set) to train what is known as a machine learning "agent". The agent is then tested on the remaining data (the testing set) to determine whether the agent needs to be retrained. Once trained, the agent should be able to take in new, unlabeled video and determine what is open space and what is an obstacle.

However, our research determined that the amount of data we would generate might not be sufficient to train these algorithms to deal with a sufficiently large amount of scenarios to be a useful solution. Additionally, the lack of depth information would still pose a problem for the algorithms; without the depth information the algorithms would only know if an object were within the view of the camera. The algorithms would not know how far away were these obstacles.

Further research showed a way to extrapolate distance information in a paper focused on virtual reality. The paper spoke about Parallel Transpose And Mapping, an algorithm by which one image is taken through by the camera and optical flow methods are applied as the camera translates to understand the distance the image moved [31]. After some distance, another image is taken and system calculates depth information similar to how stereo camera vision calculates depth information [31].

Optical flow is one of the key components of monocular vision. As images are taken by the camera, the images are compared to past images captured. With each image, a vector map is created based off the apparent motion of the pixels from one image to another. Across a large image, this process is able to create enough vector data to calculate a velocity [32].



(a) (b)

Figure 9: a: Generating a vector by comparing the location of the red ball over multiple frames (overlaid into one picture). b: The result of using OpenCV's optical-flow methods to track features in a video of a highway, thus generating the colored lines. Images courtesy of OpenCV Docs.

Although the PTAM and Optical Flow are robust algorithms for monocular vision systems, the project group had more experience with RGB-D Sensors and accompanying localization algorithms. In an effort to play to our strength, we elected to not use a monocular vision system.

2.3.5 RGB-D Sensor

The final sensor for localization and navigation was an RGB-D sensor (red green blue depth sensor), such as the Xbox Kinect. This type of sensor emits an array of infrared lights and measures the distance between the points in the array to measure distance from the sensor [33]. While the Kinect is large and heavy, smaller versions of the RGB-D sensor exist, such as the Asus Xtion.

(a) (b)

Figure 10: a: The Xbox Kinect b: The Asus Xtion. Both of these sensors are RGB-D sensors, utilizing an infrared dot matrix in its calculation of depth. Note: Images are not to scale. Images courtesy of Microsoft and vr-zone.com, respectively.

From there, further weight reductions can be made by removing the protective plastic casing and shortening the USB cable, reducing the weight from more than 200 grams down to about 100 grams [34].

These sensors are also easily integrated with the Raspberry Pi. During testing they we were able to get data with the Xbox Kinect, as shown in Figure 11, and all of the libraries are cross compatible with the Asus Xtion. While developing localization and navigation code, it would also be easily comparable with code in the Robot Operating System (ROS) libraries. The system would be able to run both sets of code and evaluate performance in an unbiased way.

Figure 11: Experimental output depth map of the XBox Kinect Sensor. The XBox Kinect calculates the depth by emitting an infrared array of dots, . The left image is the represented depth images outputted by the XBox Kinect and the right image is the RGB image

Due to prior experience with the Xbox Kinect sensor and the cross compatible libraries, it was our belief that the safest way to approach the problem would be with an RGB-D sensor. By doing so, we eliminate some risk from the project by utilizing sensors with which we have some familiarity.

2.4 Localization

Localization is the process of determining where one is with respect to a frame of reference. For mobile robotics, the frame of reference is generally the surrounding environment, usually described on a map of the surrounding environment.

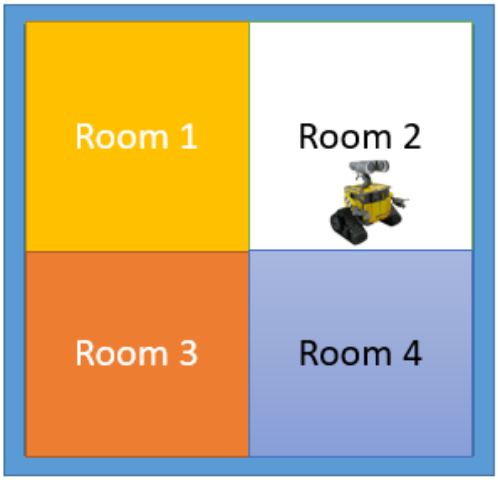


Figure 12: Wall-E would be able to determine in which room - *e.g.* localize - he is in if he could detect color, read the labels, or in some way know where or how he moved.

The robot's current position on the map is the goal of localization. The following sections will outline three different localization methods considered for the project: Way-point localization, WiFi localization, and SLAM or V-SLAM.

2.4.1 Waypoint Localization

Waypoint localization is the process of utilizing landmarks as reference points in space for where the system would be located. For the scope of this project, two specific implementations of waypoint localization were researched: QR code placement and sign reading.

QR code localization is the process by which QR codes are generated for different places in a building. These codes can hold different information. For example, a simple implementation would be to give the system a map of the environment and the map contains the location of all the QR codes on it. In this case, as the robot maneuvers around its environment, it is able to look up QR codes it sees and place itself on the map. This would mean that the limit of the localization of the robot would be the spacing

of the QR codes. As the codes get further apart, the uncertainty of location would get higher. In this case there would need to be a large number of QR codes placed in the environment and recorded on the map. Amazon Robotics, for example, does exactly this. Over time, their robots relay location and direction information to a cloud computational system which proceeds to give it information on how to continue [35]. The major downside to this system would be the setup requirements; outfitting a building with the necessary amount of QR codes could be impractical for our purposes.

Figure 13: Image of Amazon Robotics system in action. The QR codes are the small white squares on the ground, the robots scan these when they are over top of them and move on from there. [36]

Alternatively, we researched into methods of abstracting information from the robots surroundings that are already existing - to avoid the need to place QR markers around the building. One method that was a strong contender was the use of building signs and room labels. In most modern office buildings rooms are numbered and there exist signs that direct humans inside the building. Robots could potentially take advantage of these. Utilizing monocular camera vision from the Xtion or Kinect, the input could be paced through SVM and KNN machine learning algorithms to discern the room numbers and pre-determined features in the signs. This process has already been implemented with high accuracy for road signs [37], and we believe it could work for feature abstraction in localization. However, this system does not eliminate the need for the user to generate a map with accurate localization information.

Although waypoint localization is believed to be a viable system. An ideal system will require minimal setup from the user. In this case, the setup is too large to consider it a first choice. This system would, however, be a robust backup plan if alternative localization methods did not work.

2.4.2 WiFi Localization

WiFi localization is an alternative localization method that was researched for the project. The assumption was that with new router technology on campus, the new wirelss AC routers would bring easier localization information and precise results. This then required research into two specific WiFi localization techniques, WiFi fingerprinting and WiFi trilateration.

WiFi fingerprinting is the process of gathering wireless information within a building at regular intervals. With that information, a signal strength map is created based off the collection intervals, as seen in *Figure 14*. After the data is mapped, the our system would be able to examine the signals at a given point on the map, and pick the point that best estimates its own position.

This method can be highly accurate when the localization occurs soon after the fingerprint was taken. As the time between fingerprinting and localization grows, the variability of wifi signal begins to negatively impact the results. Thus, in order to ensure accurate localization, one would need to continually re-run the fingerprinting process.

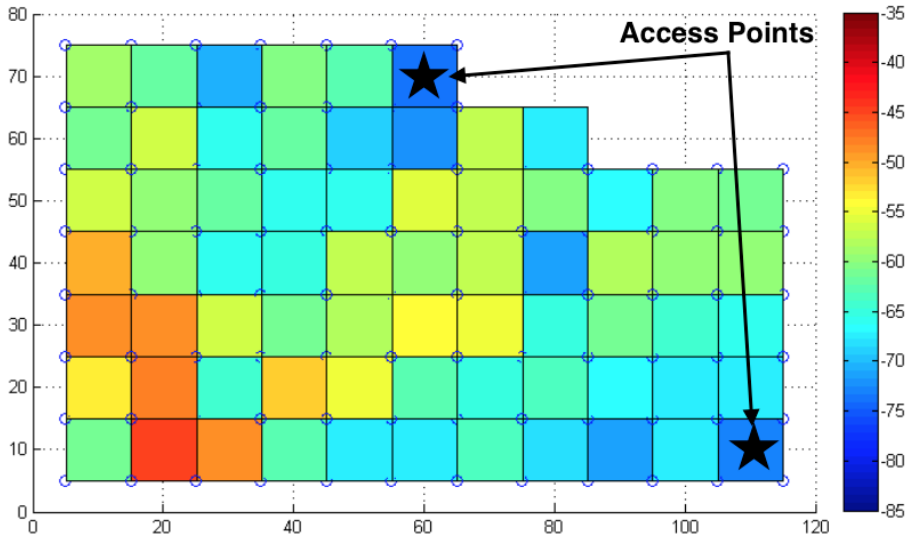


Figure 14: This figure depicts a graph of colored squares representing the strength of the system at that point, with Access points at locations (60,70) and (110,10). WiFi fingerprinting uses a graph similar to this to attempt to localize the system. Meaning, if the robot is in a square with a value of 40db, then there is only one location on the grid it can be - at location (20,10).

WiFi trilateration on the other hand is very similar to what cellphone towers do[38]. In places where multiple access points exist, it is possible to observe the strength of the networks for each access point. If the locations of the access points are known with reference to each other, then it is possible to gain an estimate of the location of the sensor based off of the strength of the network connections. This comes from the idea that WiFi signal decays with distance from the access point; therefore the number of decibels describing the signal increases with distance. Using geometry, it is possible to trilateralate the location with respect to the access points.

It was thought that this data would be reliable due to the new 802.11ac routers installed at WPI. However, during the testing phases they proved to be extremely difficult to use. In our university, the network is highly dynamic. The network is continuously reallocating Basic Service Set ID's (BSSID) within a network as well as cycling through the 16 to 64 possible Mac addresses for a given radio. It's also dynamically changing power levels based on external sensing data (such as when the airport nearby emits different signals or when the load on the network changes). The system posed a number of problems; the largest of which was access point connectivity. There were many times that the system was able to detect and measure more than 2 networks, however these networks were not always on the same floor. In certain parts of buildings, the system was able to detect WiFi from three floors up, but could not find the networks on its own floor. After weeks of testing and trials, this approach was abandoned, since the time requirements necessary to successfully interface with the WiFi system was not within the scope of the project on the campus WiFi.

2.4.3 SLAM

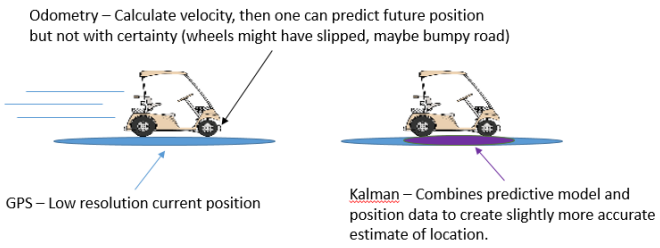
Simultaneous Localization And Mapping, or SLAM, is a process with which a robot makes a map of its surroundings as it determines where it is on the map. For the purpose of this example, assume a robot has odometry information and a LiDAR sensor. At the most basic level, when new odometry data is recorded, corresponding to the robot moving, the information from the LiDAR is processed to extract landmarks from the environment. The distance recorded by the LiDAR sensor with respect to these extracted landmarks are saved. This saved information is used to create a map that contains the position of the landmarks. Then, the robot combines its odometry information with the information of the LiDAR to determine where it has moved. Since odometry data tends to be unreliable over a long period of time due to the accumulation of error, the algorithm must compare previous LiDAR scans with the current LiDAR scans, and determine its current position based on the change of LiDAR scans and the odometry. Since it is possible that landmarks move (such as if a nearby human is picked), it is important that both odometry and LiDAR information be used to avoid over-reliance on potentially

bad data.

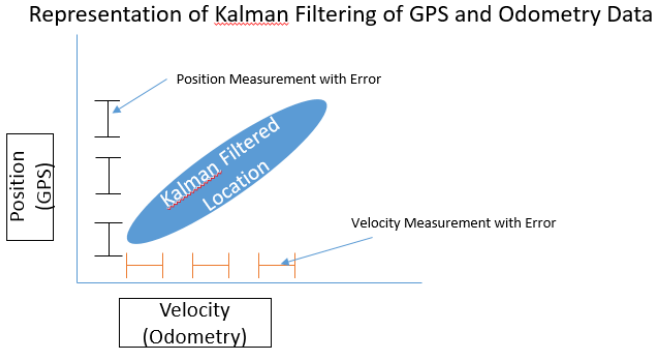
Since we have decided to use the RGB-D sensor, and the type of odometry available to use is expected to be different between terrestrial and aerial robots, we have decided to attempt SLAM for localization first before trying other algorithms. Additionally, this localization process is suitable for non-deterministic environments, since it creates its own map. No foreknowledge of the environment is needed, as opposed to QR codes, door reading, or WiFi localization. Additionally, the type of odometry used is not of particular concern so long as there is a model describing the movement of the robot and some way of measuring that movement with a sensor.

With the RGB-D camera as our primary sensor, we will be implementing SLAM in the following manner. Features are extracted using the RGB camera, using methods such as Scale-Invariant Feature Transform (SIFT) or Speeded-Up Robust Features (SURF) for the extraction. The depth data will record the information relating to the features in order to create a map. Then, the odometry information of the robot (terrestrial or aerial) and the depth data will be analyzed to create a probable location for the robot.

The type of algorithm used to fuse the odometry data and the depth sensor data is known as a Kalman filter. Consider a robot that moves from a point, A, to another point, B. The Kalman takes in current position - say depth information from a depth sensor - and a predictive model of how the robot moves - such as its velocity. Neither of these sources of information are perfect, but combining the two should at least give a better estimate for the actual location of the robot. So, based on the information of the current position from the depth sensor, the location of the last position, and the predictive model (the velocity), the information is combined and normalized to give a probably more accurate location than either sensor alone could provide. *Figure 15* below depicts this process.



(a)



(b)

Figure 15: a: A robot with GPS (which gives a location accurate to the blue ellipsoid) and odometry moves with some predictive model (its velocity) to a new location. The Kalman filter combines the sources of information to create a new estimate, in purple. b: The graph depicts the combination of the two sets of data to create a new position that has less error (depicted by the ellipsoid thickness).

Basic Kalman filtering requires linear functions that describe the sensor data. Most systems, however, are not linear, and so instead a different type of Kalman filter is used, called an Extended Kalman filter

(EKF). Our team has had previous experience with EKFs through using ROS, where it was already implemented for us in the gmapping module (used to convert Kinect depth for SLAM). However, we have not yet implemented our own form of EKF, and so we intend on borrowing the implementation from ROS to learn how best to use EKF with our sensor suite.

2.5 Navigation

Navigation is the process of a robot moving through its environment without collision. Mobile robots require some type of navigating ability to be useful, since unintended collisions can cause irreparable damage to the robot and its surroundings. There are several methods used for navigation, but we investigated the use of 4 different solutions for navigation.

2.5.1 A Star

A-Star, also referred to as A*, is a path-planning algorithm, generally used in a grid-layout map when used for robot navigation. Given a map, a starting point, and an ending point, the algorithm begins at the starting point and inspects all neighboring points. It is assumed that the robot can move to any point that is not an obstacle.

For each neighboring point, A* estimates the distance between said point and the goal: this is known as the heuristic. The heuristic does not necessarily have to be accurate to the actual distance. Instead, the heuristic is generally calculated as a straight line distance between the aforementioned point and the end point, irrespective of any obstacles that may be in the way. The algorithm also checks how far each point is from the original starting point. This true distance is added to the heuristic, and the sum is assigned to the aforementioned point as a cost. The algorithm then sorts the neighboring points in a stacked queue, with the smallest cost first and the highest cost last. It then chooses the smallest neighbor in the stack, and begins the search again, saving the chosen neighbor in a path-list.

This process continues until the algorithm has either reached the end goal, or has explored every possible route and determined no route to the end goal is possible (which happens if the goal is behind inaccessible areas, such as behind a closed door). Upon reaching the goal, the path-list will have all of the neighbors that reach the goal, and this path-list can be given to the robot as a series of move commands. *Figure 16* below shows the process of A* on a grid, where the lighter colored squares are near the start and the darker colored squares are need

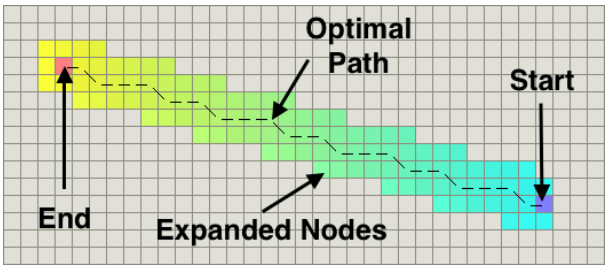


Figure 16: Depiction of how A* works. A*, by design, will create the optimal path between two points. The start and end points can be seen on the figure, along with the optimal path. The expanded nodes are representative of the locations that A* considered while building its path. This photo courtesy of Stanford University [39].

Since A* records the points it has searched, it can be inefficient when dealing with a large domain, requiring a large amount of memory. This issue can be reduced by increasing the size of the grids on the map. Then, there are less total points that need to be checked, since each would be separated by a larger distance. However, points that correspond to obstacles would be represented as being larger than they need to be, causing previously viable paths to become closed off. For this reason, the grid size is generally set to be about the same size as the robot. In this manner, any space that is not at least the size of the robot appears to be an obstacle, and any open space that is at least the size of the robot is clear for navigation.

2.5.2 Tentacle Navigation

A Star requires a map, since it requires a start goal and an end goal in order to create motion commands. Tentacle navigation, on the other hand, does not require a map for creating movement commands [40]. Instead, the RGB-D depth data is split into different segments. These segments describe potential routes with which the robot can navigate. If any segment detects an obstacle, that segment is not considered as a motion path. If all tentacles detect obstacles, the robot rotates to find other potential routes; otherwise it simply stops. *Figure 17* below illustrates this.

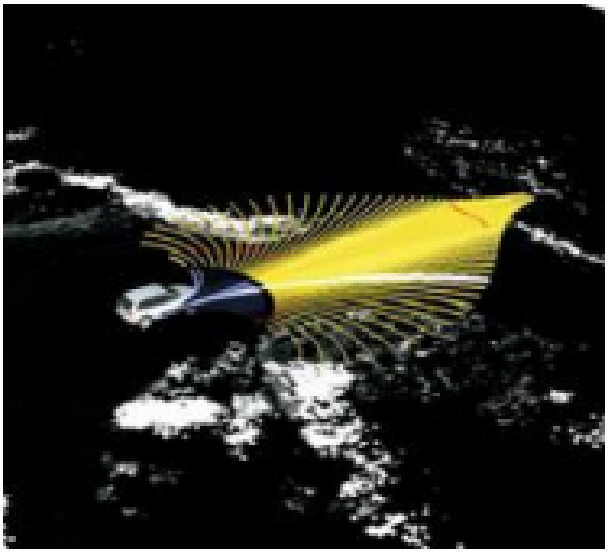


Figure 17: Depiction of how Tentacle Navigation works. The yellow lines are the possible routes that the vehicle can take. The white regions describe obstacles in the environment. The tentacles "feel" out where the areas with obstacles are, and thus guide the car towards open areas. [40]

The length of the tentacles become dependent upon the range of the sensor. Since the robot needs only be concerned with what the tentacles "feel", it does not need to store any other information about its surrounding, and it can delete tentacle information as soon as the tentacle stop "feeling" the information.

Assuming one needs to navigate a goal that is not immediately visible, however, one still needs a map. In this implementation, the end goal coordinates are given, and then another algorithm (such as A*) creates a path for the robot to move to the goal. The difference, however, lies in that the A* algorithm can be used with very large grids and choose to ignore obstacles, since tentacle navigation works to avoid any obstacles on its way to the goal. Therefore, the robot has access to a much greater domain than just with A* alone due to the diminished memory requirements. *Figure 18* below depicts a situation in which, given an end goal of driving down a street, the vehicle is able to continue driving towards its goal while avoiding obstacles.

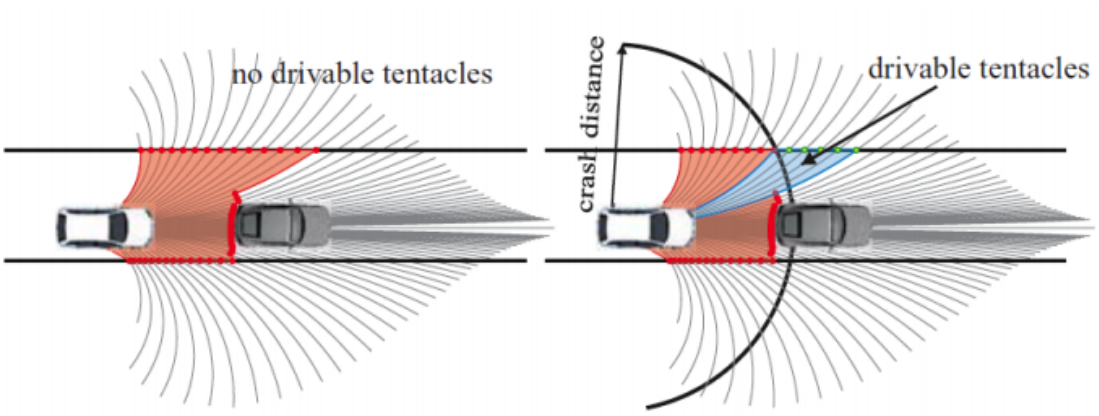
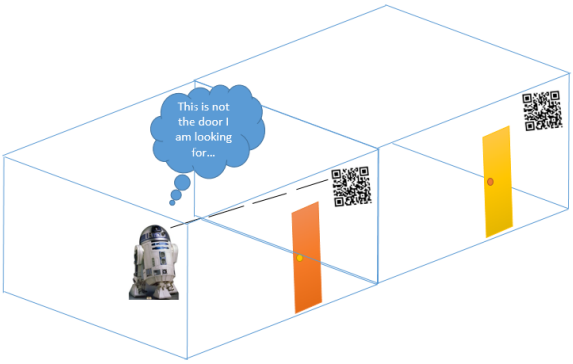


Figure 18: Depiction of how Tentacle Navigation works given a goal. The system extends tentacles out in given directions to sweep its surroundings and discover the only viable paths to take. In the figure, the system extends tentacles about the car to discover the edge of the lanes and the car in front of it. It then reasons about the plausible paths that can be taken and follows that tentacle [40].

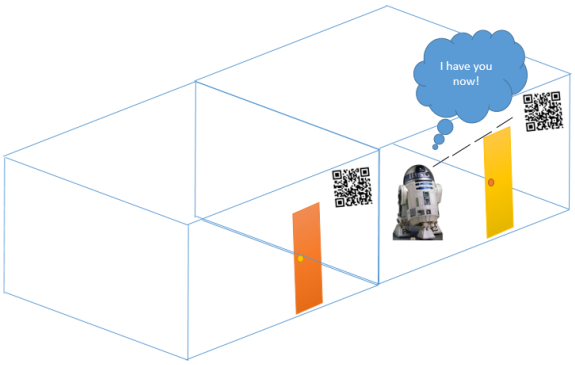
Since we are not quite as familiar with tentacle navigation as we are with A*, we will resort to using this method if the demand for memory is greater than that which the Raspberry Pi can supply.

2.5.3 QR Code Navigation

Similarly to how QR codes can be used for localization, they can also be used for navigation. Basically, as a robot approaches a QR code, it takes the location of the QR code in context to the mission that the robot is trying to accomplish, such as navigating from one end of a building to another. Then, knowing which QR code it saw, it can generate movements commands that would lead it to the next set of QR codes, eventually leading to the end goal.



(a) The R2 astrodroid identifies the QR code next to the door, and recognizes that it is not the door it is looking foor.



(b) The R2 astrodroid identifies the QR code next to the door in the adjacent room, identifying it as the correct door.

Figure 19: These figures depict R2 utilizing QR codes to navigate. As R2 identifies different codes it is able to calculate the location of the astrodroid within its environment.

2.6 Software Organization

One of the key concerns of the project is to ensure that the software is organized such it is easy to debug and improve upon. To do this, we researched into program design techniques and how object oriented programming principles can be applied to python programming (python being the preferred programming language).

2.6.1 System Testability

While coding, it is important to design with testability in mind. As a team, we looked into the use of Test Driven Development (TDD) as principle for all code written. TDD is the process of writing a test case and then modifying the code to fit that test case, and all past test cases (this is often called the red yellow re-factor method). This means that all code written passes all of the tests that have been written for it and therefore 'is tested.' However, the success of TDD depends on the validity of the test cases, and the ability to design tests that allow coding to happen in the proper increments. One concern with TDD in real life applications was how to recreate test environments and run conditions that are outside the control of the robot. To a point, the test can be recreated by logging all of the sensor data and running it through different algorithms but as the system becomes more complicated a virtual environment software would be needed to run complete tests.

Another key component to consider while designing the software for testability is software coupling and cohesion. Software coupling is a measure of how tight the code is bundled together and how easy it is to separate out individual classes for testing. Cohesion, on the other hand, is how well a 'module' (or group of classes that perform one complex action). Generally, modules that have a loose coupling also

have high cohesion. This means that each of the sub classes are able to be tested independently of other systems, and can dramatically improve results.

2.6.2 Software Design

Along with system testability is the importance of intelligent software design. Software design plays a key role in deciding early limitations and functionality of the system and how it will be impacted by changes that are applied in the long-term. One of the first steps towards intelligent design is to build an understanding of the problem and an understanding of the goal and how your software is supposed to solve the problem. In the case of this project, the Roaming Spirit platform is supposed to allow any type of sensor system (that provides the needed information about the robot's surroundings) to interface with the Roaming Spirit platform to provide any type of robot with localization and navigation instructions. This Implies the system must be able to easily add functionality without impacting the rest of the system. The safest way to do this is to design by contract, also known as program to the interface. For this reason interfaces will created for key abstractions within the project, where those key abstractions are is something that will be discovered and decided upon during the development process. However, there are specific design patterns that are brought to light when describing the goals of the software.

First is the strategy pattern. The strategy pattern allows the user to encapsulate specific variability in code into a specific type of object. For this project we can think of this as a sensor. What the sensor is and how the sensor works is a point of variability, but how the sensor interfaces with the rest of the system is not. This is because regardless of the type of sensor that is being used, the system still needs the same type of data, in the same format, in order for it to make sense of that data and use it. The strategy pattern can be seen in figure 20. The pattern shows that given a class, in this case we will call it robot, it will contains within it some number of objects of type interface, in this case sensor. The specific implementation of that sensor is is not necessary to describe to the system, as long as that sensor abides by the contract created between it and the interface then the parent class, the robot, does not need to know what the specific information is. In this way the system would be able to implement multiple different types of sensors, but as long as they work through one class that implements the interface, the system would have known problems.

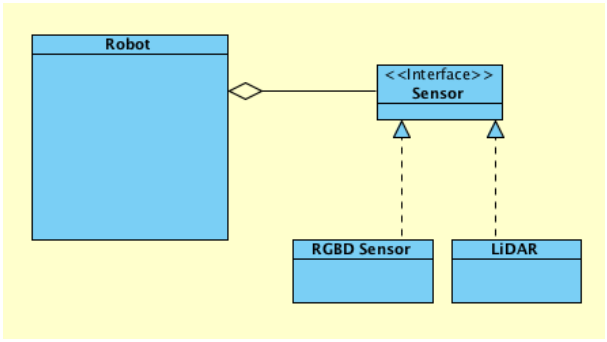


Figure 20: This is a Unified Modeling Language (UML) diagram for the Strategy Pattern. It depicts the parent class, the robot, that contains a realization of the Sensor interface. This realization could be either some type of RGBD Sensor or LiDAR. This diagram was created for example purposes, in a real UML diagram, the interface would document the function definitions that the realizing classes would need to implement.

Another pattern that could be used, is the state pattern. The state pattern is structured identically to the strategy pattern however it does contain some fundamental differences upon implementation. The state pattern utilizes Factory methods to replace the object that is contained by the parent class. In this case the state pattern could be used to maintain the search state of the robot while it is exploring and building a path, or it could also maintain the safety state of the robot. Unlike the strategy pattern where there exists one realization of the interface, the state pattern is continuously setting and resetting the realization. This pattern of setting and resetting the realization is what allows the software to implicitly track state without explicitly needing code to do it. this means that the robot, or parent class, would be ignorant to the state that it is in except that the interface realization would be the state and that would impact how other parts of the system functioned. In effect this would ensure clean and readable code,

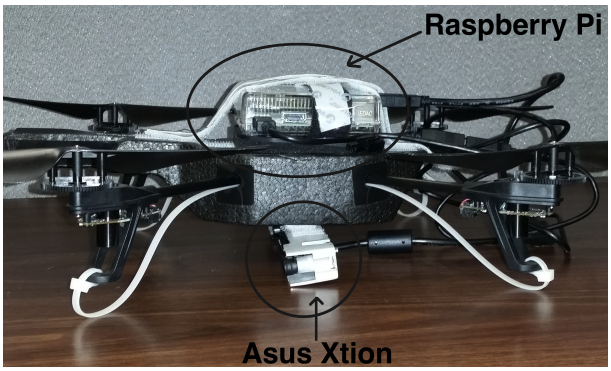
and the ability to implement new states and new functionality easily.

2.7 Robotic Platforms

In order to test the Roaming Spirit platform it is necessary to have systems on which to test. Due to cost restraints of the project, all robotic testing systems will have to be done on robots that are easily accessible. The two types of robots with the highest access during the course of this project are the Parrot AR Drone 2.0 and a Turtlebot. The Parrot AR Drone 2.0 is a hobbyist quadcopter that was accessable during this project and the Turtlebot is a relatively inexpensive (relative to others) land based mobile robot that is often used for research or teaching applications.

The Parrot AR Drone 2.0 is, as stated, a hobbyist quadcopter. This means that the drones technical lifting specs are not widely publicized because the average user does not need to know this. Concerned that the drone would not be able to lift the amount of weight needed to be a viable test robot, the external protective foam was removed for additional payload. After a number of empirical tests utilizing the now protectionless drone, increasingly heavier weights, and fully charged batteries, a maximum acceptable payload range was found to be between 300 and 350 grams. *Figure 21* shows the Parrot AR Drone both as the unmodified drone that was originally received, and as the stripped down system used for payload testing.

(a) This is the Parrot AR Drone 2.0 unmodified, though without the removable foam guard.



(b) This is the Parrot AR Drone 2.0 before flight for payload testing.

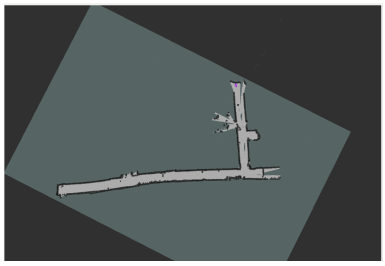
Figure 21: This figure shows the Parrot AR Drone 2.0 unmodified (a) and with the modifications made while the robot was running payload tests (b).

In order to use the Parrot AR Drone 2.0 as a testing platform, there will be specific administrative concerns that will have to be resolved. Payload testing occurred in a small square room that could be quarantined so that there was no concern with hurting humans. However, this is not something that can be easily accomplished for long hallways in buildings. This could raise the concern or injury and make it difficult to fly indoors.

The Turtlebot is, as stated, a land based mobile robot that is often used for research or teaching applications shown in 22-a). The system utilizes ROS (Robot Operating System), which is an application that dissociates programs from each other to allow highly mobile development (analogous to an interface). The difference is that ROS would allow multiple programs of different languages (or even different

computers etc.) to communicate. ROS uses a Topic Subscriber system, similar to a star topology to accomplish this. In order to interface with the Turtlebot, some type of ROS handling program/class would have to exist.

(a) This is a Turtlebot.



(b) This is a map generated by the ROS Gmapping Packages and displayed through the program RViz, a ROS Visualizer.

Figure 22: This figure shows the Parrot AR Drone 2.0 unmodified and with the modifications made while the robot was running payload tests.

As a land based mobile robot, the Turtlebot would act as an extremely safe decision for testing. There would be no concerns about losing control during flight and damaging the Roaming Spirit hardware. As a research platform, the Turtlebot also has significantly more reliability. The system is used by many experimenters/roboticists and if there were concerns there would be a development community to turn to for assistance. The Turtlebot would also provide an alternative map making system by which Roaming Spirit could be compared. *Figure 22*-b shows a map that can be generated by the default programs provided on the Turtlebot. This map can be used as a comparison to the Roaming Spirit System.

The Parrot AR Drone 2.0 is the preferred system of testing. If the Parrot AR Drone 2.0 can be used for testing than a safe argument could be made for the Turtlebot (due to the similarities in how the systems move, principally differing in the land based vs. air vehicle). If, for administrative reasons or weight reasons etc., the Roaming Spirit system can not be tested on the Parrot AR Drone 2.0, the Turtlebot will be a safe and reliable alternative.

2.8 Chapter Summary

This chapter outlines the requisite knowledge for project/paper. This chapter starts by communicating the environments that this system could operate within in Section 2.1. Then, in Section 2.2 the chapter begins an in-depth look into the different embedded computing technologies that could be utilized on this framework. This includes a look at the Raspberry Pi 2 and Odroid C1. In Section 2.3 this chapter observes the specific functionality of different sensing technologies (e.g. sonar, depth lasers, stereo/mono camera systems, and RGB-D sensors). After which, the chapter follows with an analysis into localization methods (e.g. waypoint localization, WIFI localization, or SLAM) and navigation strategies (e.g. A Star, Tentacle, and QR code) in Sections 2.4 and 2.5. After which, the paper outlines the importance of software design in Section 2.6, and the different robotic platforms by which the system could be tested upon in Section 2.7.

3 Proposed Approach and Expected Outcomes

The following subsections will outline the design process approached to create a cross-platform robotic system that can abstract the task of indoor localization and navigation for robotics platforms. This solution will be structured such that aerial and ground vehicles can utilize the system. This would include differential, and Holonomic drive robots, which allow aerial and ground vehicles to implement this solution. Finally, this chapter will discuss the expected outcomes of this project.

3.1 Picking a Design

While picking a design for this project the design criteria are as follows:

- System that weighs no more than 390 grams such that it is light enough to be on land and aerial vehicles.
- System is designed such that it is easily expandable for different sensors and SLAM algorithms.
- System allows for fluid motion of the parent robot - this means that the system is not stopping intermittently due to the need to recalculate a new path.

These criteria translate into the key components to be observed when selecting possible solution: processing methods, sensing methods, and run-time algorithms. All project research revolved around these components and special interest was taken to see how they would interact; said understanding would help identify future risks and complications. It was with this mindset, and the knowledge learned in while researching that physical solutions were selected.

3.2 Selected Design

1. The platform will utilize a Raspberry Pi ARM board in order to capitalize on the large supporting community and prebuilt interfacing capabilities between the Raspberry Pi and the Asus Xtion. By picking a embedded computer with a large community, it lessens the strain of solving every little bug. Instead, there is a large community with solutions and suggestions to a myriad of different, yet similar, problems.
2. The system will utilize an Asus Xtion as the primary sensor on the system; this will allow the system a light weight reliable depth sensor that we are comfortable using. This sensor will be the primary input for localization and navigation algorithms. However, it is important to note that software must be modular enough to accept different types of sensors given that the developer writes the appropriate sensor class.
3. The localization functionality will be SLAM based. Whether this is V-SLAM or normal SLAM is something that will be determined after time has been spent on implementation. This project group will not be writing a SLAM algorithm, to do so is outside the scope of this project.
4. Navigation functionality will be A Star focused, with Tentical Navigation as a backup. The memory requirements for A Star can slow the navigational algorithm, therefore the implementation of Tentical Navigation would improve the memory efficiency and improve the algorithmic decision outputs.
5. This system is focused on cross platform functionality. This includes aerial robotics; systems of which are highly weight sensitive. For that reason, a large consideration in decision making was weight. From the associated research, the Asus Xtion weighs 200 grams and the Raspberry Pi 50 grams. Theoretically, this places the weight of the system at 250 grams. As a safety factor, and to protect against the variance of results found while researching, the maximum weight of the system should be not more than 350 grams.

3.3 Implementation Plan

This section itemize and elaborate on the high order steps for development in this project. These steps are key in evaluating the success of the project and will therefore play a large role in the results section.The full time line of the development portion of this project can be seen in *Figure 23* . The generalized steps can be seen below.

1. Correctly interface with the Asus Xtion and Breezy SLAM algorithms (the SLAM Library used in this project) to create maps of hallways. This would include first getting output from the Asus Xtion, and working to find the Breezy SLAM parameters that work well with the Asus Xtion.

2. Design a software by which different types of robots can be controlled by the system. This will require a modular program design that provides an interface for other robots. This would ensure modularity.

3. Devise and implement an algorithm for world exploration. This algorithm will prioritize the safety of itself and its surroundings, as well as explore locations with the most unknown space.

4. Implement an in-map navigation program that allows the robot to move within the map that it previously created, while working to revamp the map as it travels.

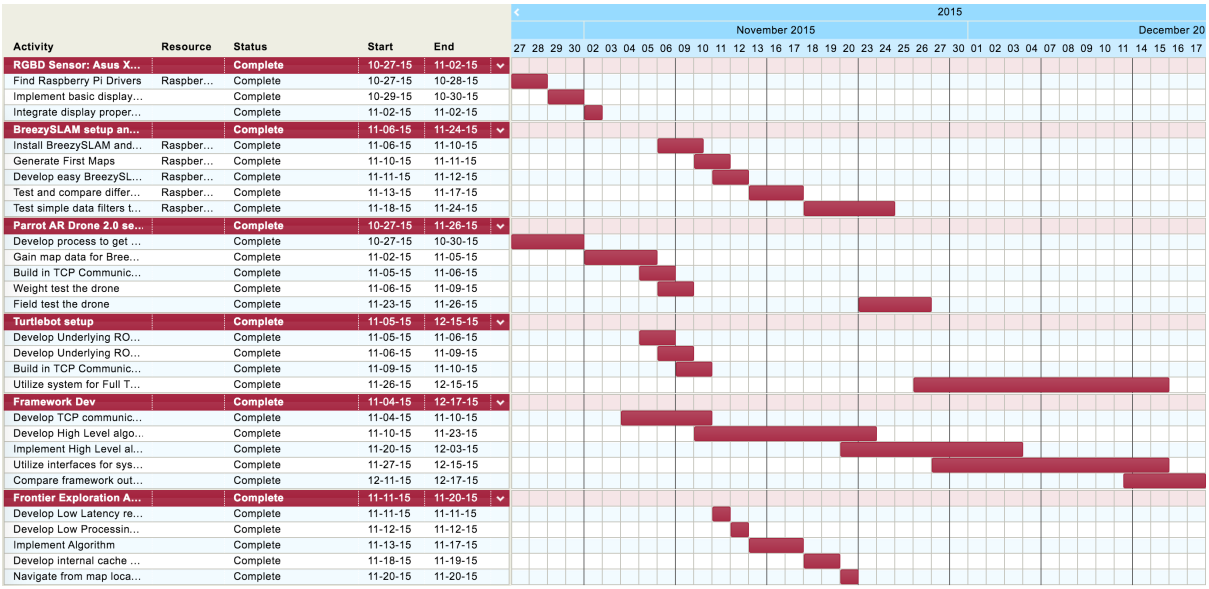


Figure 23: This figure represents a gantt chart of the tasks and goal for implementation in the project. This covers the seven weeks representing the B Term of the 2015-2016 school year at Worcester Polytechnic Institute

3.4 Expected Outcome

When this project is complete, it is expected that there will be a platform a user integrate with their robot to allow their robot to perform indoor localization and navigation. It is important to note that the parent robot of the system must be capable of relaying the proper odometry data to the platform. By using this system, the robot will be able to traverse indoor environments, while it is possible that the platform could correctly localize and navigate within rooms the platform will be focused to hallway navigation. This system is expected to provide safe travel to the robot it is attached to, meaning the system will not allow the robot to collide with any type of system that it can perceive.

3.5 Chapter Summary

This chapter focused on the course of action of the project group. It detailed the key components that the project grouped identified while selecting the design for the system. Then, this chapter explained each of the design decisions that were used and why. Finally, the chapter laid out a time-line for development in the gantt chart in *Figure 23*.

4 Implementation

The following section will explain the actions taken towards implementation of the Roaming Spirit Platform. This section is itemized into sub-sections documenting each step of the development process. This will start by explaining the implementation of the Asus Xtion and Breezy SLAM; it will focus on the BreezySLAM drivers, an understanding of the Breezy SLAM library, and how they can be interfaced together.

The following section will detail the modularity of the system and how the software is designed. This will cover data acquisition, sensor interfacing, robot controls and data management as observed through the lenses of software design. Algorithm's and technical explanations can be observed in the following frontier exploration and in-map navigation sections.

The frontier exploration section will cover the algorithms devised for motion planning and exploration. This will lead into the final section on in-map exploration. This section will explain in-map navigation and the algorithms utilized in the Roaming Spirit Platform.

4.1 Asus Xtion and BreezySLAM

The following section will explain the actions that must be taken to utilize the Asus Xtion and BreezySLAM. In order to utilize the Asus Xtion with the Raspberry Pi the proper drivers must be installed. The following sections will also cover how BreezySLAM was utilized and the settings applied for Map development.

4.1.1 Asus Xtion Drivers and Communication

The Asus Xtion was made to use the OpenNI drivers (used by the XBox Kinect as well). With the advent of Apple's acquisition of Primesense in November of 2013, the OpenNI website was shut down. The website (www.openni.org) now leads directly to the Apple store web page now. Fortunately, another company, Occipital inc., maintains the OpenNI 2 drivers (compatible with both the Asus Xtion and the Kinect). The drivers for Mac, Windows, and Linux are on [41].

A Python wrapper for OpenNI that allowed drivers interfacing was found at [42]. The wrapper allows Python to interface with the Xtion directly and came with sufficient content to allow us to access the RGB and Depth sensor data. The Operating System used on the Raspberry Pi was Raspbian Wheezy, a Linux derivative made for the Raspberry Pi. Therefore, the linux openNi drivers were downloaded.

It was found that the example wrapper given was insufficient to access the sensor data. The example code was:

```
from primesense import openni2

openni2.initialize()     # can also accept the path of the OpenNI redistribution

dev = openni2.Device.open_any()
print dev.get_sensor_info()


depth_stream = dev.create_depth_stream()
depth_stream.start()
frame = depth_stream.read_frame()
frame_data = frame.get_buffer_as_uint16()
depth_stream.stop()


openni2.unload()
```

For the Raspberry Pi, there were a few, resolvable issues with this code. Most importantly, the second line:

```
openni2.initialize()     # can also accept the path of the OpenNI redistribution
```

This initialize function did not work because the Raspberry Pi could not find the installation of OpenNI. By utilizing the path as an argument for *openni2.initialize()* this was solved:

```
path = '/home/pi/devel/OpenNI2/Packaging/OpenNI-Linux-Arm-2.2/Redist'

openni2.initialize(path) # can also accept the path of the OpenNI redistribution
```

Additionally, more settings were needed to access the actual data streams. For the Asus Xtion, we needed to add a second library and set up the depth stream:

```
from primesense import _openni2 as c_api #this import is also needed

depth_stream.set_video_mode(c_api.OniVideoMode(pixelFormat
                                = c_api.OniPixelFormat.ONI_PIXEL_FORMAT_DEPTH_1_MM,
                                        resolutionX = 320, resolutionY = 240,
                                        fps = 30))
```

According to the documentation, these settings set the size of the output data stream, determine the magnitude of the depth data, and cap the frames-per-second rate of capture of the camera.

The streamed data is saved in a numpy array. Numpy, a library for Python that supports matrix multiplication and multi-dimensional arrays, is widely used among scientific Python libraries. As a result, visualization of the depth data was very simple. OpenCV was used to visualize the data by opening the "$frame\_data$" using the $cv2.imshow$ command, as it uses numpy arrays:

```
frame = depth_stream.read_frame()
frame_data = frame.get_buffer_as_uint16()

cv2.imshow(frame_data, ''Showing the depth data") # ARGS: image name,
                                                  #optional image label
```

4.1.2 BreezySLAM Map Development

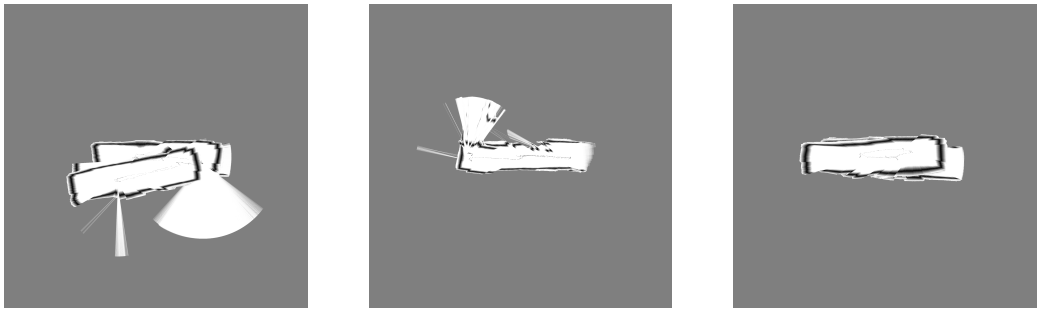
BreezySLAM is an open-source Python library for Simultaneous Localization and Mapping (SLAM). BreezySLAM is based on CoreSlam, and works with C extensions. The extensions maintain the speed of programming in C, but with the benefit of being able to program in Python. BreezySLAM was developed for using a LiDAR as the sensor, and one can choose to improve the results with other odometry data (such as encoders). Fortunately, BreezySLAM was easy to install on the Raspberry Pi. A manual for installation on different operating systems is provided online. Furthermore, BreezySLAM provides premade classes for sensors and robots. These can be subclassed to create individualized interfaces for custom robots and sensors.

For the map development, there were two algorithms tested. The first one was called "deterministic SLAM". This algorithm did not use the data provided by the LiDAR to estimate the robots position; it used only the odometry data. The deterministic SLAM was used for checking the odometry data, so no adjustments were made to the algorithm.

The second one was the "RMHC SLAM". RMHC, or Random Mutation Hill Climbing, is the algorithm for estimating the new position, and RMHC itself is a local search algorithm that reduces the tendency to become stuck in local maxima as compared to regular hill climbing. RMHC SLAM calculates an initial position based on the odometry data and then uses a particle filter to improve the accuracy of the result. To fit the given needs of the system, we adjusted some of the parameters for the RMHC algorithm. There are three parameters provided regarding RMHC:

The first parameter is the amount of maximum performed iterations. Some intermediate tests found that all values beyond 1000 iterations no longer noticeably improved the output but added to the computational load. For that reason, the maximum number of iterations was set at 1000. Some intermediate results can be seen in *Figure 24*. The second and the third parameters belong to the movement of the robot. One is the "σ" of rotation and the other the "σ" of translation. Those parameters are used by

RMHC to calculate the new position. A value of 20 mm for the translation and a value of 1 degree for the rotation provided the best results.



(a) (b) (c)

Figure 24: These maps are different variations of the parameters (input and map size) given to BreezySLAM. As the parameters change, the ability for BreezySLAM to make the map changes (this change also depends on which version of BreezySLAM is being used, deterministic of RMHC). The roaming Spirit platform utilizes deterministic SLAM with parameter values of 1000 iterations and 20 mm and 1 degree of rotation. The size of the map should be appropriately chosen for the area to be mapped.

The size of the map is another setting that can be set in the BreezySLAM setup. The size is given in pixels and meters. As is stated in *Figure 24*, the size of the map that BreezySLAM is using should be set appropriately to the chosen area to be mapped. BreezySLAM does not rescale the area as it maps, therefore it is important to avoid setting the area too large as to leave areas of pixels unused. One conclusion that can be drawn from this is that the number of pixels influences the quality of the resulting map. If there are insufficient pixels, the estimation of a new position suffers and provides poor results. The downside of a larger pixel count, however, is that the computational time rises with more pixels, which decreases the response time of the system.

For the map, the remaining parameters are the quality of the map and the so called "hole width". Both do not have influence to the positioning: rather, they affect different aspects of the display of the map. The quality of the map is a value between 0 and 255, where 255 is equal to the maximum quality. It determines how defined the edges of the walls are. The "hole width" refers to the size of a wall. So every object detected as a wall is added to the map and assumes this fixed thickness.

4.2 Modular System Interface

Modular code design is a key component of any large code base. The following sections will discuss how the software was built to allow different sensors to be utilized with the system. It will also explain what steps were taken to simplify the ability to expand the project to any type of robot. Finally, it will discuss the way that data is managed between software layers and how decisions are made between them.

4.2.1 Data Acquisition and Sensor Interfacing

For the purpose of this project the primary sensor used was an Asus Xtion. However, if, in the future, other experimenters or project groups wanted to use other sensors with this system, it has been designed to accept those and easily incorporate them into the system. To do this, a sensor interface was created by implementing the Laser interface in the Breezy SLAM library. The sensor interface has two main functions that all sensors must implement:

- **scan**: Returns an array of data that represents the scanned data from the sensor.
- **shutdown**: Safely shuts down the sensor so that the system can shut down.

By implementing the Laser interface from the Breezy SLAM library, it implies that all sensor data should mimic that of a LiDAR. For example, while the Asus Xtion provides a depth image of the area in front of the system, the only part of that image that is used is a single 'row' of values that approximate the sensor value as if it were a single line LiDAR scan. In order to use other sensors, such as a Microsoft Kinect, a class would need to be created such that it implements the Laser interface and returns the

data as required for said interface. This allows the system to be improved/modified at the need of the user.

4.2.2 Robot Controls and Interfacing

To guarantee a high modularity system, the decision was made to write an interface which can be implemented on every robot. The only requirements to the robot are four different command methods. While the implementation may differ, each method is named as follows and holds the given requirements:

- **move forward**: Implemented to drive the robot forward and return a touple of the movement data.
- **turn left**: Implemented to turn the robot left and return a touple of the movement data.
- **turn right**: Implemented to turn the robot right and return a touple of movement data.
- **wait**: Implemented to stop all robot movements and return a touple of stationary movement data.

while returning the following data to the system:

- **traveled distance**: Returns the distance that the robot has traveled in since the last query. Value should be returned in millimeters.
- **angle of rotation**: Returns the angle that the robot has rotated in since the last query. Value should be returned in degrees.
- **time since last call**: Returns the length of time since the last query. Value should be returned in seconds.

The robot and the Raspberry Pi communicates via TCP. TCP is an internet protocol which provides a reliable and error-checked connection between two members in an IP network. The system packages command messages or pertinent data into single byte messages and then utilizes the TCP connection to relay the messages between the robot and the system. By using this type of implementation, it not only makes the system modular, but it allows robot-to-system connections to exist independent of the physical connection (wireless, hardwired over Ethernet, or even a connection to localhost).

The main thread implements a class called NetworkVehicle which represents the connection to the robot. This class is responsible for all connection handling; this includes an efficient connection establishment, exception handling, and a clean shutdown. The class provides following methods to the main thread:

- **initialize()**: This method is called at the beginning of the routine. At first a TCP connection to the robot is established. The connection setup shown in the code snippet below runs the blocking command *self.connection, address = self.socket.accept()*. This command is blocking, therefore the system will wait for a connection. If the connection fails during the TCP authentication or tries to connect to an invalid user the system will shutdown and return False. After the connection is established, the method sends the initialize-command to the robot and waits for an answer. After the robot has initialized successful, this method returns and the main thread can work on. How this method works in the Roaming Spirit Platform can be seen in *Figure 25*.

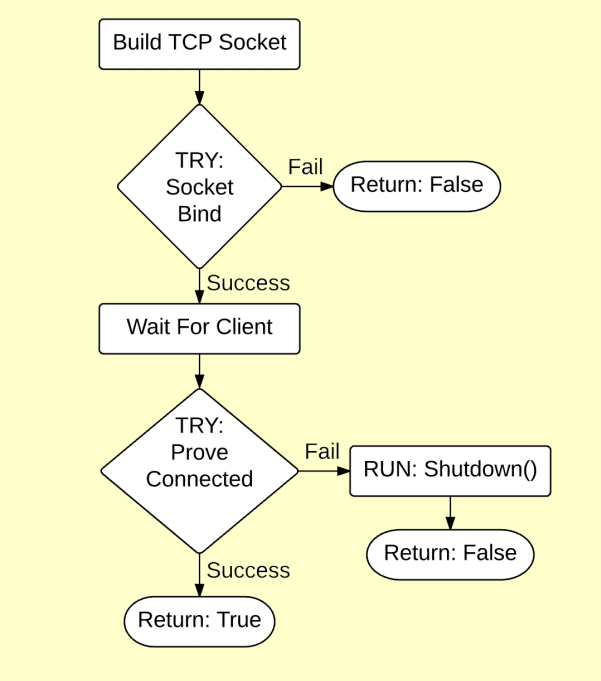


Figure 25: This diagram is a flow chart of the Roaming Spirit Platform's implementation of initialize(). It was implemented on top of Pythons Socket library.

- **move(cmd)**: This function implementation pass the move command to the robot at which point it will run one of the aforementioned mobility commands (move forward, turn right, turn left, wait).
- **getSize()**: This function is used to allow the system to know the size of the robot represented as a square. This should provide the largest side length of the robot as it helps decide what the robot can and cannot fit through on the map.
- **shutdown()**: This method is called at the end of the routine. It tells the robot to shutdown and then completely closes the TCP connection.

4.3 System Overview

As shown in *Figure 43* the Appendix, the system is divided into six main modules. The one which controls every other module is the main thread. Besides the main thread there are the four main modules (Sensor, Vehicle, Navigation and SLAM) and one auxiliary module (Server). As shown in the chart the server does not interact with the rest of the system and runs in its own thread. The server module is used to get the map from the SLAM algorithm so that a client can monitor the production of the map in real time.

The four main modules are all initialized directly after the start of the main thread. Each initialize function is a blocking function which only returns after the whole module is started and ready to work. Of the main modules, only the Navigation module has its own thread to run expensive route calculations in the background. If, at any point during the operation of the system, the main loop starts terminating each module will get called to safely shutdown.

4.3.1 The Main Loop

The main loop was approached with top down programming design in mind. The code was written such that others can understand the large ideas and others who wish to interface with this program can easily modify the functionality. *Figure 26* shows a diagram for the main loop.

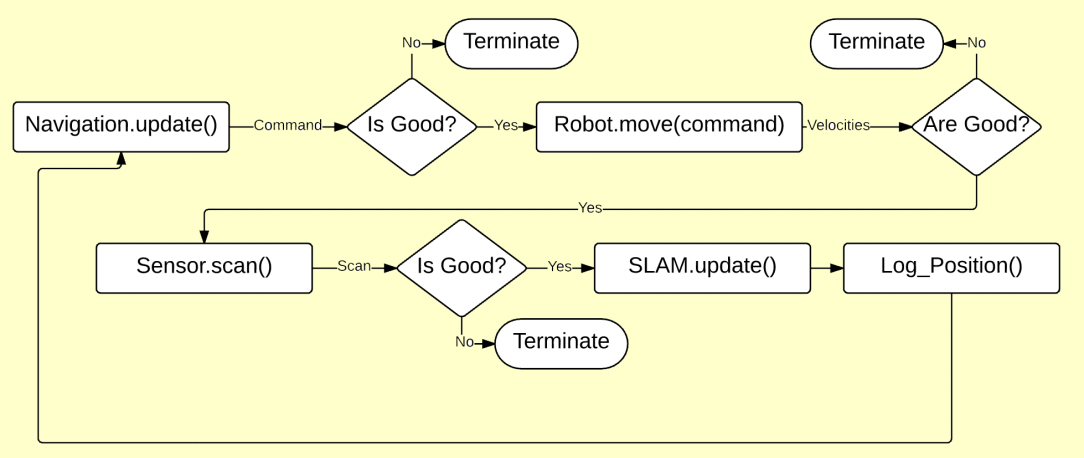


Figure 26: This is a flow chart depicting the main loop of the Roaming Spirit System. As shown, the system updates the navigation information, and then converts that command to robot move data. Then, it couples that move data and a sensor scan to update the SLAM algorithm. Finally, it logs the position and repeats. The system uses defensive programming strategies to ensure that every step of the way valid data is being passed along and the system can continue without undocumented error (as can be seen with each of the 'is Good?' conditionals).

The main loop starts by calculating the next move command for the robot. This starts by passing the previous iterations sensor scan to the navigation module (in the case that it is the start up scan, it passes the initial scan to the navigation module). The navigation module then calculates the next movement for the robot. If the entire room is explored or the module crashes the module will return *None* and the main loop will begin to shutdown the program.

Once the move command is returned, it is passed to the vehicle module. Because the vehicle module is designed based off an interface, it is only known that the module will take in the data and returns the odometry data since it was last queried. The command will then be sent to the vehicle implementation; depending on the implementation it may try and execute that command on the robot or it may not. The main loop then checks to see if the values returned by the vehicle module is *None*. If it is, it is possible that there was an error in communication with the robot or some other type of external problem that forces the robot to stop. The main loop then alerts the user via print statement and terminates the program.

The main loop then checks to ensure that the odometry data is not *None*. If the odometry is invalid, the system reports out to the user via print statement and begins to terminate the program. If it is valid, the main loop will query the sensor module to retrieve scan data. If the scan is of length zero, then the sensor cannot make a valid scan and the system begins to shut down.

If all modules work correctly, the odometry data and the previous scans are passed to the SLAM module which calculates the new position of the robot and updates the map with the new scan data. Before restarting the process, the main loop gets the location from the SLAM library and records it as the trajectory for logging.

4.4 Frontier Exploration

The frontier exploration classes of this project are implemented such that they act as a navigation stack. While reading this section, it is important to remember that the navigation stack is executed as its own thread in the system. This means that upon initialization, a 'navigation' thread is spawned begins running in the background. The navigation thread is used to manage the route that the robot will be instructed to follow. This means that during 'recalculation' times (moments where the navigation thread needs more time than was allotted) the robot is instructed to execute the *wait()* command in order to preserve safety.

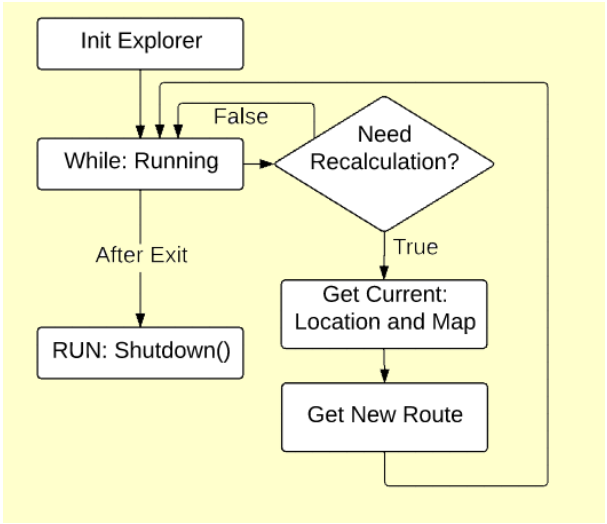


Figure 27: This flow chart represents the actions taken by the Frontier Exploration thread that is utilized to maintain the path directions generated by the Roaming Spirit platform.

When this daemon-like thread is started it begins the intermittent function of polling internal data structures to check if the navigational route needs to be recalculated. If it does, the system obtains the most recently generated map and the robots location in the map and calculates the new route. This process can be seen in *Figure 27*.

This section will outline the actions carried out to expand frontiers and build a map. This will start within the 'Init Explorer' section of *Figure 27* with a comprehensive explanation of the 'start-up conditions' and steps to resolve the problems that they create. Then, it will delve into the actions taken during the 'Get New Route' portion of *Figure 27* and explain the general case exploration algorithm that is used to discover a majority of the world around it (an overview of this process can be seen in *Figure 28)*. Lastly, this section will explain the end cases that can occur at the end of frontier exploration and how they were resolved.

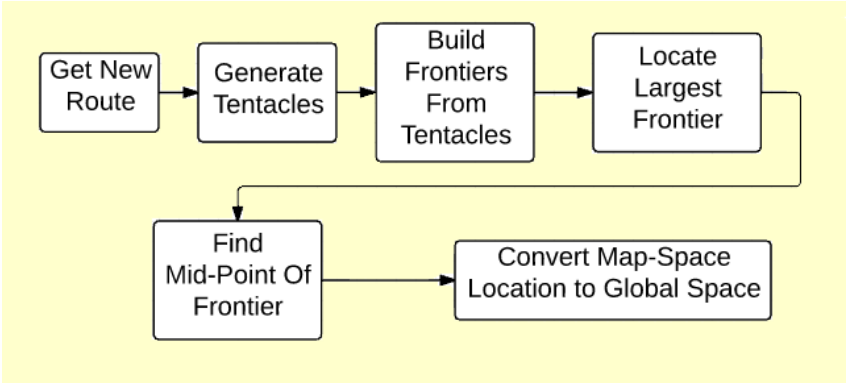
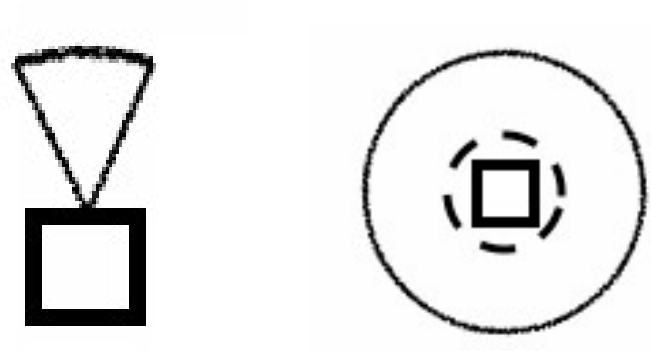


Figure 28: This flow chart represents the actions taken by the Frontier Exploration thread when a new route is generated. The system generates the tentacles and builds frontiers from their tips. It then locates the midpoint of the largest frontier and converts it into robot space and returns said value so the Roaming Spirit Platform may direct the robot correctly.

4.4.1 System Initialization

Upon start up, the system begins scanning the area in front of the sensor to get an initial understanding of its surroundings. Due to the modularity of the system, the specifics of what the system sees would differ with each implementation of a sensor. In the case of the Asus Xtion, the sensor would detect the cone of space in front of the robot. Conversely, if the Asus Xtion were swapped out for some type of LiDAR the system would detect an annulus of data with the system at the center point of the annulus, as in both cases there is a minimum distance and a maximum distance either kind of sensor can measure.



(a) Top down cross-sectional view of what the system would sense if using an Asus Xtion-like sensor

(b) Top down view of what the system would sense if using a LiDAR-like sensor.

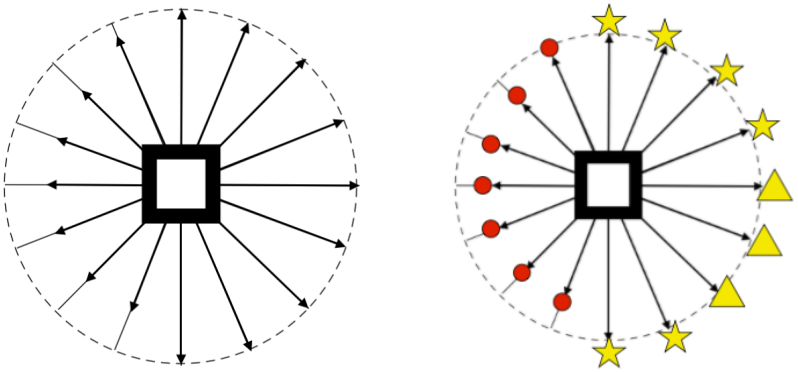
Figure 29: These side by side figures show the viewing angle and range of different sensor types that could be used for the system.

Due to prior experience in frontier exploration, it was well known that placing the robot at the centroid of an explored region significantly decreases the run time of large map exploration. This means, that if the system were to use any type of sensor that was not a 360° LiDAR (or produce an output that resembled that) the system would take significantly longer for exploration. In order to stop this, a start-up functionality was added to the algorithm to ensure that regardless of the sensor type, the system was able to create a similar starting ground for exploration.

At the start of each program the system runs through an initialization period. During this time it ensures that all the subsystems are started and prepared to run. At this point, the system begins the exploration phase. This starts with the robot rotating about its axis for 360°s. This command does assume that the robot is a differential drive robot, however alternate steering mechanisms will provide similar responses. This action allows the robot to circumscribe itself within a region of explored area. Once the robot has completed its circle (placing it at the center point of a circle or a point along an annulus, for differential and ackermann steering respectively) the robot is able to start path planning.

4.4.2 Exploration Algorithm

From past experience using a Raspberry Pi for traversing large collections of data, one of the primary concerns while designing the exploration algorithm was the processing and memory requirements. The BreezySLAM map displayed the known and unknown area through a 1,000 by 1,000 array - which means that if the algorithm explored *most* of the array every time the algorithm was executed, it would utilize a large amount of time for processing. In order to ensure the safety of the system and its accompanying robot, the algorithm must have a high responsiveness to external stimuli (e.g. dynamic obstacles or static obstacles to detect while moving).



(a) Searching stage of exploration algorithm, before termination.

(b) Representation of post search state, marking the circles walls, triangles as empty space, and stars as unexplored area.

Figure 30: An illustration of how the system utilizes tentacles to search the direct area. It also visually depicts the systems knowledge post search.

The algorithm begins after the start-up process has ended. At this point, the system uses its location on the map and searches outward from that point. The algorithm does not search *all* of the points, it extends 'tentacles' radially outwards until they reach either a pre-defined length or locate unknown space or an obstacle. *Figure 30* displays both the initial searching state of the tentacles and gives a visual representation of the post search state for the tentacles (a and b respectively). A flowchart of this process can be seen in *Figure 31*.

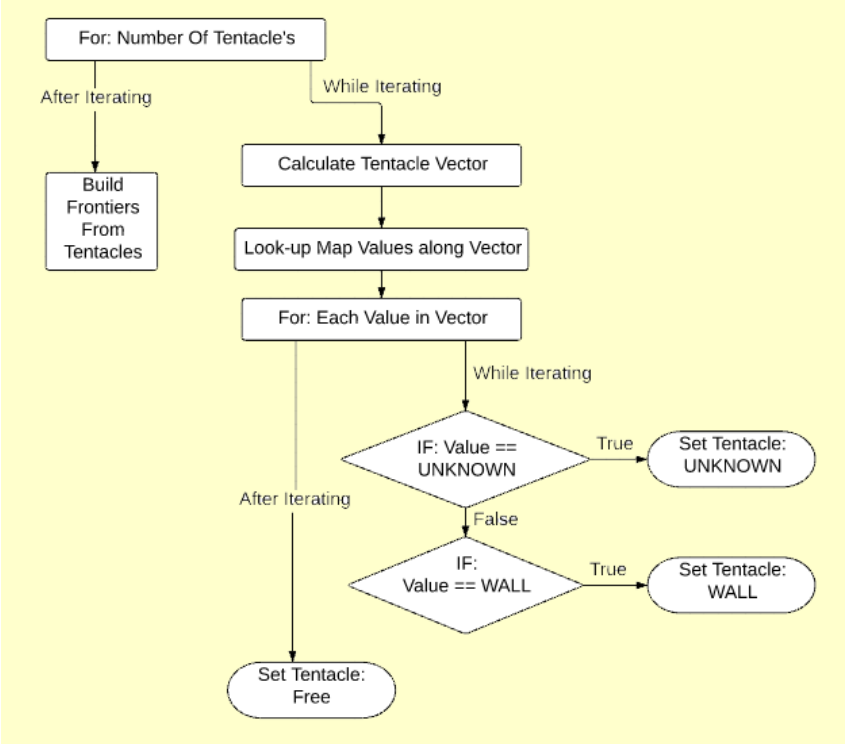


Figure 31: This flow chart represents the actions taken by the Frontier Exploration thread when it begins to execute the tentacle generation portion of the route generation.

After search, the algorithm finds the terminal points of the tentacles it makes an assumption about the safety of the systems surroundings. The system assumes that whatever exists at that point exists for a distance equal to half the width of the robot in the direction of the next point. At first, this meant that the number of tentacles was a function of the distance from the robot the system would explore

along with the width of the robot, however over time it was found that the number of tentacles could be increased to try and improve the explored regions around the robot.

Figure 32 displays a visual representation of what the algorithm would believe about its surroundings. At this point the algorithm classifies each data point based off its surrounding data points. By doing this, the system is able to detect large regions of 'unknown' space and classify that region as either a frontier or discard that data as either an error or a location that the algorithm will later detect through other motions.

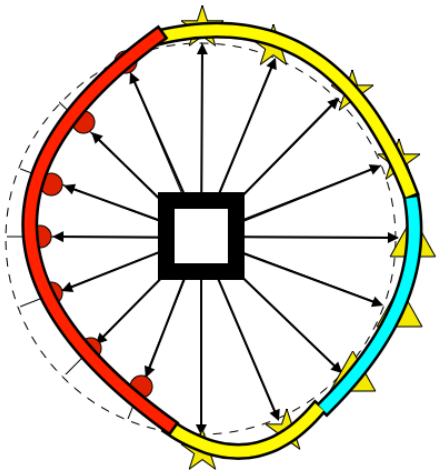


Figure 32: This is a visual representation of the assumption made by the algorithm. In this image, the system has made its assumptions about the different areas between the tentacles.

As the algorithm begins to finish, it utilizes the location and direction that the robot is on the map and makes a decision about which frontier (or point on the map) it will drive to next. This means that if the robot is located in the center of a hallway intersection it would need to make a decision about its next actions; should the system drive straight down hallway A or should it turn and drive down hallway B. In reality, it shouldn't matter which the robot does as long as they're both done. However, this system defaults to prioritize driving straight.

4.4.3 Filling Out The Map

One the robot has explored such that there are no frontiers directly detected by the tentacles used for navigation, the system needed a way to explore areas that were previously determined to be a low priority for exploration. For example, if the robot were located at the star in *Figure 33*, the system would need a way to navigate the robot to either of the two x's located at the top of the map or the right most side of the map.

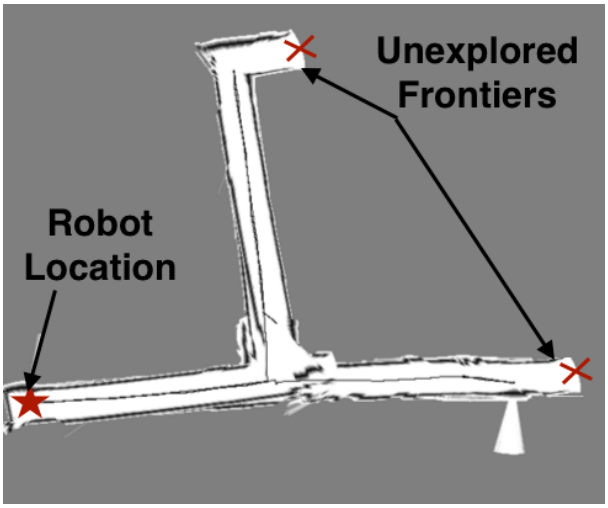


Figure 33: Generated map that has been marked to identify the location of the robot and the location of the unexplored frontiers.

However, because the system was implemented on a Raspberry Pi, there were concerns about what a frontier search would do in terms of processing power. A Star, for example, has a worst case memory complexity of the maximum number of states that could exist for search - which in our project would be a 1,000 by 1,000 grid (the average resolution of the map being built by BreezySLAM). As a result, the process could be extremely slow and costly. This is why the system utilizes tentacle navigation, but in this case it doesn't solve the problem of locating unexplored regions to explore.

For this reason, a change had to be made in the exploration algorithm that would allow the system to quickly find the next region to explore. The first component of this solution was to identify all of the waypoint set during exploration. Whenever the robot would reach a waypoint it would cache the waypoint along with the number of unexplored regions that exist around it. Then, when the robot came in contact with a dead end (in this case meaning an area with no frontiers immediately within it's tentacle search), it would be able to retrace it's path back to the location that previously had frontiers near it. From here, the system would recheck if there are still substantial frontiers, and it would try to navigate to them.

4.5 Drone Testing

The Parrot AR Drone 2.0 was to be the primary testing robot for this project. However, due to changes in previous space availability and the safety concerns about flying a drone indoors elsewhere (near people), the drone was later replaced with the Turtlebot for full systems testing. Prior to full system testing, the Drone was used to test individualized portions of the system during development. For example, by turning the drone on and having it transmit optical-flow based odometry back to the system which also had the Asus Xtion attached, it was possible to record all of the data and use it for a first line of testing. While this testing was effectively a crude simulation and therefore highly prone to human error, it provided a sanity check that incremental changes were, at least somewhat, moving in the correct direction without the need to do a full system test - which in this case would be autonomously flying the drone and observing the results.

This system was used mainly during the start of development, before each of the systems were developed far enough to utilize full system trials in a way that would provide useful results. In order to ensure that this system worked well, a class was created that implemented the robot interface, but was instead a 'virtual' robot in which all of the move commands returned but made no attempt to accomplish any actions. This system worked very well when the robot was not deciding where it was going, but was instead following a predefined path. During development it was realized that if a virtual system were to implemented around this system it would be possible to do similar tests as the development progressed. However, this was not something that was pursued in depth for this project due to time considerations.

4.6 Turtlebot Testing

All full scale testing was done on a Turtlebot. A specific implementation of the vehicle class was written for the Turtlebot. It extended a different class that handled all of the ROS interfacing, since ROS was used to control the motors of the robot. This setup is displayed in *Figure 34*.

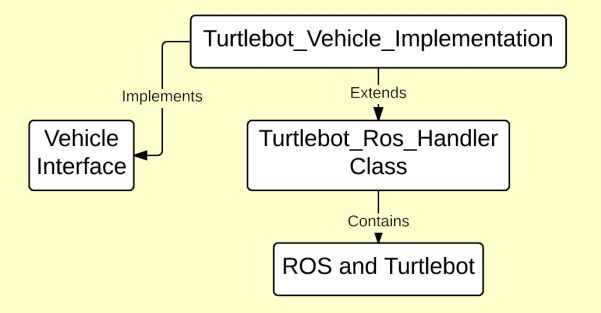


Figure 34: This is a flow chart depicting the main loop of the Roaming Spirit System.

Once the Turtlebot had the proper software to interface with the Roaming Spirit system, the hardware was temporarily attached to the Turtlebot. Due to the fact that the Turtlebot was loaned, no permanent physical modifications were made to the Turtlebot. Instead, the Asus Xtion was set in front of the Xbox Kinect (with special care to fully obstruct the IR sensors in the Kinect as to avoid excess noise in the system), as seen in *Figure 35*.

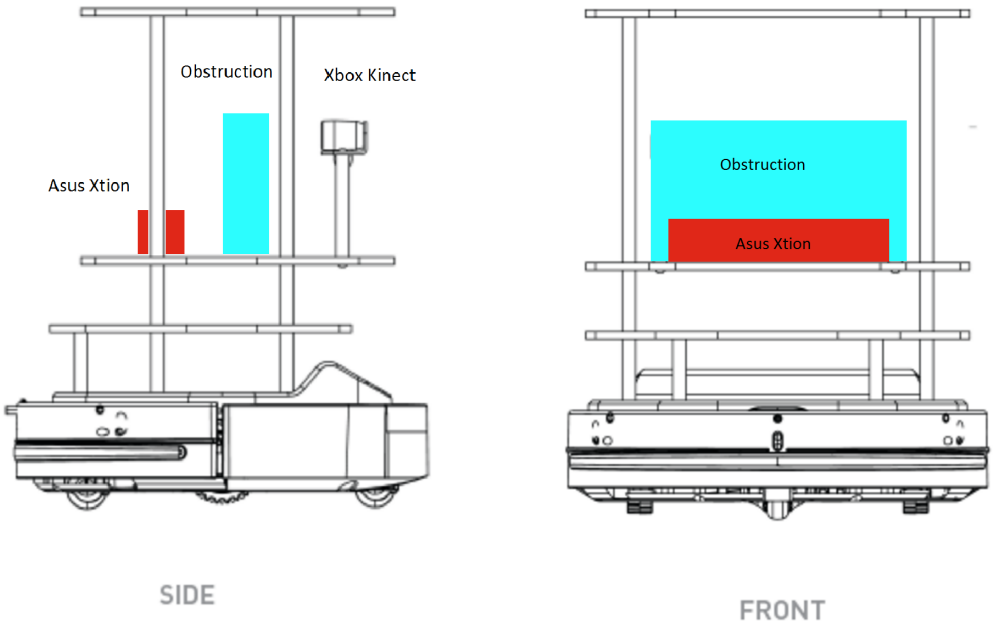


Figure 35: This diagram shows the implementation of the Asus Xtion on the Turtlebot. The cyan obstruction prevents the Xbox Kinect - attached to the Turtlebot - from displaying its own infrared dot array, potentially interfering with the Asus Xtion. The Asus Xtion is then placed in front of the obstruction to view the surrounding environment. Image modified from Clear Path Robotics [43]

Figure 36 shows the actual implementation of the Asus Xtion on the Turtlebot. Added to the original design were a spring-powered electronics board to support the Asus Xtion, and Post-it notes to cover the Xbox Kinect infrared dot matrix.

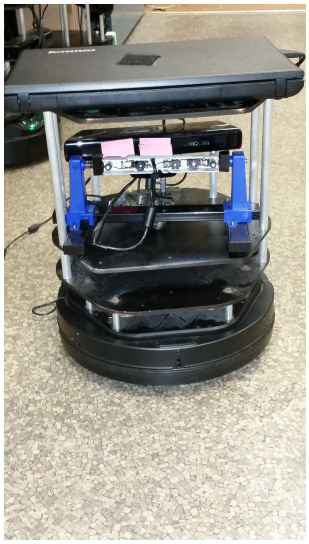


Figure 36: In the actual implementation, the obstacles used are the pink Post-it notes covering the infrared display and the camera. A spring-action electronics board holder supports the Asus Xtion, and the Asus Xtion is placed close to the center of the Turtlebot.

The Raspberry Pi and a battery pack were secured on the robot just behind the Xtion's support, and then the system was prepared to run. In order to execute the program, the *–help* argument was passed to the program and the instructions were followed accordingly.

4.7 Chapter Summary

This chapter outlined the various steps that were taken for the creation for the Roaming Spirit Platform. The chapter started with BreezySLAM and Asus Xtion installation and utilization requirements, highlighting the challenges encountered during implementation. This section then discussed the importance of modular software design and how programming to the interface would allow the system to be expandable to different robotic systems, thus allowing other developers to customize it to meet their needs. This chapter briefly spoke about the overview of the entire system, then expansively communicated the Roaming Spirit Solution to frontier exploration. Finally, it commented on the actions taken for testing the system with the Parrot Drone and Turtlebot.

Table 2: This table depicts the results of trials for safe control of a robot via the Roaming Spirit platform. In this table, it shows 10 trials and the number of times that an event occurs. The events listed in the table are visible collisions, invisible collisions, wall blind scenarios, wall blind associated collisions. A visible collision is a scenario where the robot can see what it into. An invisible collision is a scenario where the robot cannot see what it's running into. A wall blind scenario is what happens when the robot mistakenly gets too close to the wall such that it should be able to perceive it but cannot. A wall blind collision is when the robot is in a wall blind scenario and collides with the wall/obstacle.

Trial Number	Visible Collisions	Invisible Collisions	Wall Blind Scenarios	Wall Blind Associated Collisions
Trial 1	0	0	0	0
Trial 2	0	0	0	0
Trial 3	0	0	0	0
Trial 4	0	0	0	0
Trial 5	0	0	0	0
Trial 6	0	0	0	0
Trial 7	0	0	0	0
Trial 8	0	0	0	0
Trial 9	0	0	0	0
Trial 10	0	0	1	0

5 Experimental Results

The following subsections will communicate the results of the Roaming Spirit Platform. This will start by detailing the Roaming Spirit Platform's object avoidance performance as it guides a Turtlebot down a short hallway, as described by the Proposed Solution and Requirements (Section 1.3) After which, the performance of the Platform as it navigates though the full length of hallways is illustrated, per the Proposed Solution and Requirements. Finally, this section will detail the localization and navigation capabilities of the Roaming Spirit Platform. These results will be analyzed and compared to the goals in the Proposed Solution and Requirement. It is important to note that each subsection will report results that must hold to certain tolerances to be accurate. To accurately report the distance traveled, the robot must be as close to 100% safe - without imminent danger of collision or falls - as possible during every trial. Likewise, in order to evaluate map generation, the robot must be capable of traveling at least 60 feet to generate before using the generated data during map making comparisons.

5.1 Safe Movement

Table 2 outlines the results of 10 trials where the focus was to gauge the safety responsiveness of the system. The data recorded is as follows: visible collisions, invisible collisions, wall blind scenarios, and wall blind associated collisions. A visible collision is when the system acts such that it allows the robot to incorrectly make contact with something that was perceived by the Roaming Spirit System. An invisible collision is when the robot makes illegal contact with something that the Roaming Spirit Platform did not/can not/does not perceive. Wall blind scenarios are moments where the robot is positioned close enough to an obstacle such that the Roaming Spirit Platform cannot perceive the obstacle due to minimal viewing range limitations of the sensor. Finally, anytime that the robot is wall blind and the robot illegally contacts an obstacle it is recorded as a wall blind associated collision.

5.2 Distance Traveled

Table 3 is a metric for evaluating the distance that the robot was able to travel while maintaining its safety. If at any point the robot came in illegal contact with something in its surroundings the trial would have been considered over and the distance recorded. With interest in brevity the distances recorded were 60 and 120 feet as outlined in the proposed approach goals, and greater than 150 feet in effort to capture highly successful trials.

Table 3: This table displays the safe distance that was traveled by the robot in a given trial. The distances displayed are 60, 120, and greater than 150 feet. The 60 and 120 feet duration were chosen as part of the requirements for the project. 150 feet was decided as it was the a good cut off to delineate trials that were much longer than the 120 feet, while still displaying some of the trials that made it past 120 feet but did not go significantly longer.

Trial Number	Travel 60 feet	Travel 120 feet	Travel Greater Than 150 Feet
Trial 1	Yes	Yes	No
Trial 2	Yes	Yes	Yes
Trial 3	Yes	Yes	No
Trial 4	Yes	Yes	No
Trial 5	Yes	Yes	Yes
Trial 6	Yes	Yes	Yes
Trial 7	Yes	Yes	Yes
Trial 8	Yes	Yes	No
Trial 9	Yes	Yes	Yes
Trial 10	Yes	Yes	Yes

5.3 Localization and Navigation

For this project, the best measurement of localization and navigation capabilities as discussed in the proposed approach is to observe the types of maps that are created by the Roaming Spirit Platform. In order to test the platform an area is needed for localization and mapping to occur. The project group was capable of using the third floor of Atwater Kent, an academic building on Worcester Polytechnic Institutes campus. An old floor plan of the third floor shows the hallway's that the Roaming Spirit platform explored. It is important to note that while the floor plan in *Figure 37* is not the exact floor plan (the most recent which was not able to be published), the area able for exploration is accurately displayed.

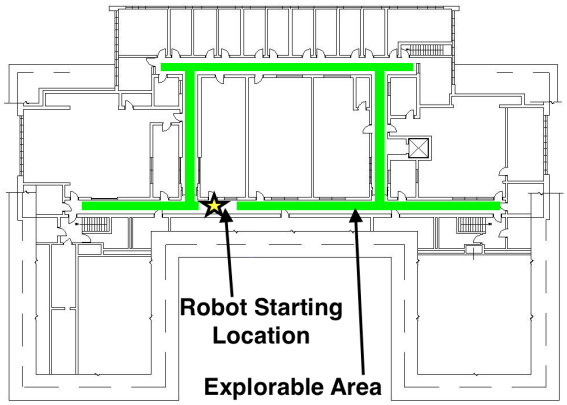


Figure 37: This is an old floor plan of the third floor of Atwater Kent, an academic building on Worcester Polytechnic Institutes campus. The shaded region on the map shows the explore-able region while the star represents where the robot started for each trial.

Map development varied by the weights that were used, this section will display the most relevant maps for the project, and their meaning will be discussed in following sections. For time considerations, most of the tests preformed (although not all that are displayed below) were kept to a 30-minute window.

In all following maps, there are 3 different colors with possibly 4 different meanings.

- **Grey** This is a majority of all maps and is unknown space. This represents regions that the system

has not seen/does not know what it is.

- **White** This is explored space that is empty. When looking down an empty hallway, all the area between the walls would appear as white.
- **Boldface Black Lines** These are 'obstacles.' This is a location that the system has placed some object by which the robot would hit if it drove there.
- **Non-bold Black Lines** These are an attempt to track the path of the robot. Typically, these will be one 'map pixel' wide and will exist within a hallway or near the wall.

In *Figure 38* were the results of map development tests. This map was generated by placing more trust in the SLAM algorithm as opposed to the odometry data. Odometry data was gathered and fed it into BreezySLAM over time to simulate the robot moving. Given the results shown, an on-robot test was not preformed.

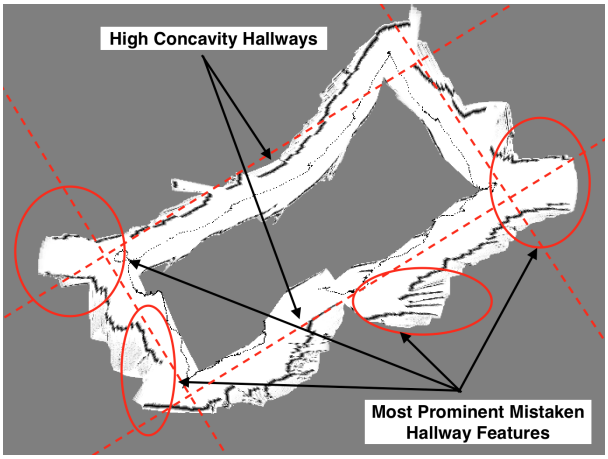
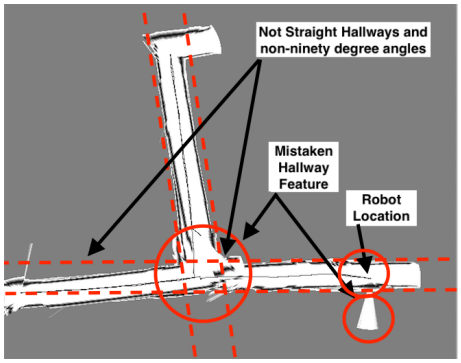
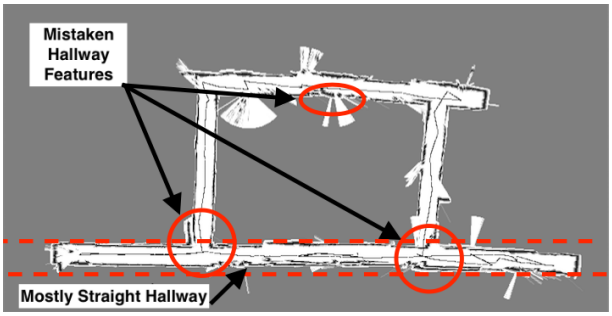


Figure 38: This map is a parameterized BreezySLAM maps created from a log file of odometry data to view the maps that BreezySLAM created. The odometry data is taken from the Parrot AR 2.0 Drone while it was walked around the third floor of an academic building on Worcester Polytechnic Institutes Campus (Atwater Kent). Both trials use SLAM parameters that are set to trust the BreezySLAM library significantly more than the output of odometry data.

Alternatively, the following maps were created by trusting the odometry more than the BreezySLAM output. In *Figure 39.a* the map was generated with the Turtlebot speed set to 0.1 of the total input speed (1). After 30 minutes the robot was stopped, the speed was upped to 0.5 and another trial was run to generate *Figure 39.b*.



(a) This is an incomplete map of the floor to be mapped. The Roaming Spirit Platform generated the map and was interrupted for time considerations.



(b) This is a complete map of the floor to be mapped. The Roaming Spirit Platform generated the map and properly declared that it was done exploring the floor. This map was created with no time constraints.

Figure 39: These figures are two separate trials of the roaming spirit platform driving the Turtlebot around the third floor of an academic building on Worcester Polytechnic Institutes Campus (Atwater Kent). Both trials use SLAM parameters that are set to trust the odometry of the robot significantly more than the output of BreezySLAM.

For comparison the Turtlebot was used with ROS to generate a map of the same floor. This was run utilizing the default parameters for the ROS Gmapping libraries and displayed through RViz. In order to get similar results to that of the BreezySLAM test, the Turtlebot was set to explore large frontiers and fill in the map. This trial was stopped after 45 minutes.

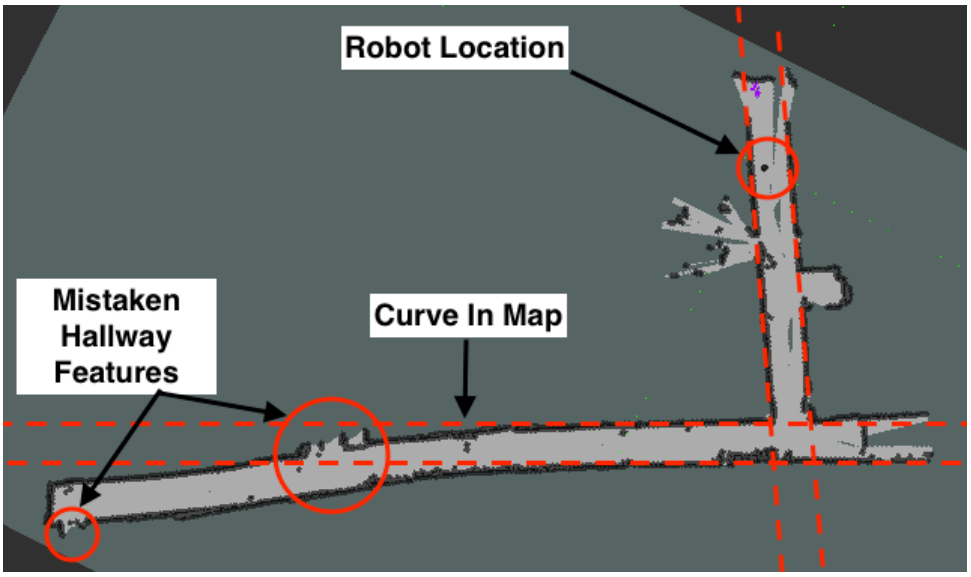


Figure 40: This map was created by using a Turtlebot hardware and the default ROS packages software (default move base stack and gmapping algorithms). The map is of the third floor of the academic building Atwater Kent on Worcester Polytechnic Institutes Campus.

5.4 Chapter Summary

During the section the experimental results for the roaming Spirit platforms ability to safely maneuver a robot for a given distance as well as localize and navigate within the map were reported. The system's ability to move safely was successful 100% of the time, with a given bug arising 1 out of 10 trials and not impacting the success of a given trial. The robot was also able to travel at least 120 feet on every single trial. This also included the robot's ability to travel that distance safely. Finally experimental results recorded the maps created for the systems localization and navigation. These Maps had a variety of success as well as shortcomings. As the results show, there were inconsistencies in hallway straightness as well as feature detection and recording. Also provided in this section was a map created by the Turtlebot

platform using one of the current state-of-the-art slam algorithms, gmapping [44].

6 Analysis and Future Projects

Thus far the paper has displayed the actions taken by the project group and objectively reported the results. The goal of this section is to analyze the project through the lens of the developers to view the shortcomings of the project and how they relate/can impact the functionality. This analysis will be utilized to propose suggestions for future projects as well as outline a direction this project could be taken in. This section will start with a discussion of the safe movement and travel distance results, then will analyze the localization and navigation results, and will terminate with general thoughts about the software or hardware.

6.1 Safe Movement and Travel Distance

Both of these sections were viewed as extremely successful in that there was a high rate of success for both of these. However, there is a bug that was experienced once although it never caused a problem. This bug was characterized as the 'Wall Blind Scenarios.' This means that the robot is allowed to get close enough to a wall such that the dead band of the sensor (in this case the Asus Xtion) is mistakenly being used as a reliable sensing method. This means if the robot is too close to a wall, it may not see the wall due to dead band sensing. In this case the Roaming Spirit Platform would be blind to the wall.

It is important to note that this scenario only becomes a problem when the exploration algorithm decides that the Roaming Spirit Platform should explore in the direction of the wall that it is blind to. In the one time this occurred during testing the system corrected the mistaken wall when it later saw it correctly (as any SLAM algorithm should).

Furthermore, it's important to comment that part of this error is the fault of BreezySLAM. SLAM algorithms such as ROS's GMapping utilize efficient filtering over time that correct landmarks over time, instead of instantly as is the case with BreezySLAM. For example, if a temporary wall was put up during the start of mapping then removed with BreezySLAM, the first time the lack of temporary wall was seen it would remove it from the map. Whereas systems such as GMapping would hold the wall in the map for many sensor scans to ensure that it was creating the most accurate map possible.

It is equally possible that this is an unexpected side effect of the mounting of the Asus Xtion on the Turtlebot. As shown in *Figure 41* the Asus Xtion was mounted on the edge of the Turtlebot so that there was no external noise created from the plates it was resting on. However, by choosing to mount the sensor in that location, it did not properly allow for the dead band of the sensor, a problem that was thought to be solved in software.

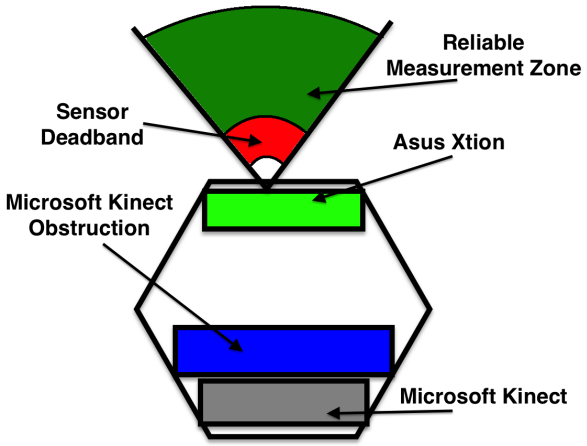


Figure 41: This image shows the a sketch of a Turtlebot in the test conditions used during the course of this project. This image also depicts the vision range of the Asus Xtion. All objects found in the green region will be accurately recorded. Anything detected within the 'sensor deadband' area will cause invalid measurements.

As displayed in the results section, the software was not enough to resolve the bug. Future projects using this system must take care to mount the Asus Xtion (or other vision systems to be adapted) such

that the deadband nearest to the sensor is accounted for. Any future project aiming to improve the Roaming Spirit platform can continue to research into/implement a lightweight SLAM algorithm that can handle the bug.

6.2 Localization and Navigation

Map development is a difficult property to objectively qualify and compare. For the purpose of this section, there will be a couple points that the two systems will be compared by:

- **Straightness of Measurements** Over long distances, does the system accurately represent the straightness or angular measurements of the inputted hallway?
- **Angle Assessment** Are corners represented by the proper angles? Does the map accurately represent 90° corners?
- **Noise in the Map** Are the given walls continuous? Are there random points in the hallway?

The first exciting observation is that the Roaming Spirit platforms output map tends to have straighter hallways than that of the ROS platform. This is believed to be one of the advantageous outcomes of trusting the odometry over the BreezySLAM library for map creation.

While assessing angles in both maps, there are no measurable differences in 40 and 39.b, however in 39.a there is a notable difference in angle. Due to the inconsistency of the angle representation the ROS system is deemed to be more reliable and thus better.

The tiebreaker comes in the observation of the noise in both maps. The ROS system created an output map with smooth walls, few holes in vision, and well connected corners. Conversely, the Roaming Spirit map is extremely noisy. There are many holes in the walls or walls in hallways from the system struggling to place itself on the map (as shown in *Figure 42*.

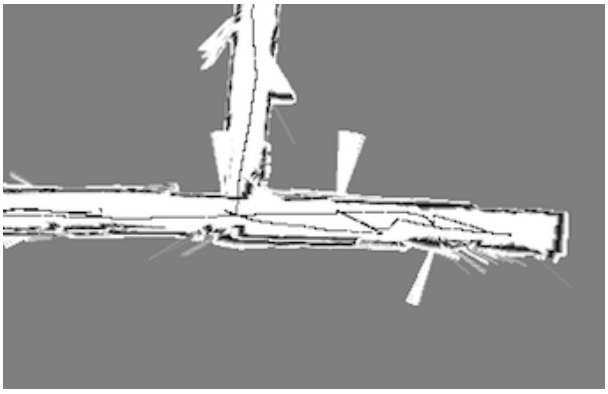


Figure 42: This is a closeup of the bottom right hallway intersection of 39.b. It shows the error in the Roaming Spirit system caused by the BreezySLAM failure.

Overall, this would state that, irrespective of system size or computational power, the Turtlebot and the ROS system are a superior system for map making. However, the Roaming Spirit system was developed for lightweight low processing requirement systems. Future projects could improve the Roaming Spirit platform by, as stated in the prior section, improving/developing a new/researching and discovering a new SLAM algorithm that is more reliable.

Another consideration to make is the language this project was completed in. This project was developed mainly in Python. Future projects could re implement the system in C/C++/Java in order to increase the speed of the system. By using one of these languages, it would be possible to utilize higher demand SLAM algorithms in order to improve map development.

6.3 Chapter Summary

This chapter focused on the outcomes of the project and how they can be improved in the future. The current state of the platform is such that it is possible to be implemented on a robot. However, this chapter raises the point that the Roaming Spirit Platform is currently limited by the SLAM library capabilities as well as the capabilities of the sensor utilized. The current recommendation of this project

group is to continue to improve and refine the platform before utilizing it on a robot. As better SLAM libraries are developed or discovered by other project groups the system will continue to improve and provide more reliable results.

7 Conclusion

The requirements for this project fit within two main categories, each containing quantifiable goals. These main categories and quantifiable goals, along with the result of each, are as follows:

- **Distance Requirement:** Safely traverse 60 to 120 feet of mostly static hallway environments.
- **Distance Result:** System safely traverse greater than 240 of a static hallway environment. This included at least 4 encounters with hallway intersections.
- **Map Navigation Requirement:** The system must be able to utilize its location within the map and location of other items within the map and travel to them.
- **Map Navigation Result:** The system was able to accurately move the parent robot from a given location to a set-point on the map.
- **In-map Localization Requirement:** The system must be able to identify where the robot is located in the map that it is generating. This can be an estimation ranging from 6 feet radius (basic goal) to 3 feet radius (reach goal) for end of trial localization. At the most challenging level of this requirement the system must utilize SLAM to locate the robot and update over time as the map is developed.
- **In-map Localization Result:** The platform was able to locate the robot within a 3 foot radius of the hallway and was accurately updated over time as the map was built and the robot moved.

As has been shown, the Roaming Spirit platform was able to accurately drive multiple types of robots for long distances while plotting a map describing its location and its surroundings. The platform was highly successful in ensuring the robots safety and its ability to travel long distance, and moderately successful at producing accurate and useful maps. The system was light enough to be mounted and flown on a stripped down Parrot AR Drone 2.0 as well as a Turtlebot. Overall, this makes the Roaming Spirit platform a success and shows that as SLAM algorithms evolve and processing units get faster and smaller Roaming Spirit can only improve.

Future projects involving the Roaming Spirit platform can take one of two paths: improve the systems that the Roaming Spirit platform are built upon (e.g. SLAM libraries, alternative processing systems, improved sensing libraries/systems) or expand the use of the Roaming Spirit platform (e.g. introduce mapping of rooms, handle stairways, detect doorways and windows). It is recommended that future projects first focus on improved sensing libraries/systems and the related SLAM libraries used for best results.

References

[1] A. Sander and M. Wolfgang, "The Rise of Robotics", www.bcgperspectives.com, 2014. [Online]. Available: https://www.bcgperspectives.com/content/articles/business_unit_strategy_innovation_rise_of_robotics/. [Accessed: 24- Sep- 2015].

[2] "Robotics Engineering - WPI", Wpi.edu, 2016. [Online]. Available: http://www.wpi.edu/academics/robotics.html. [Accessed: 24- Sep- 2015].

[3] "Bachelor of Science in Robotics Engineering", Lawrence Technological University, 2016. [Online]. Available: http://www.ltu.edu/engineering/mechanical/bachelor-science-robotics-engineering.asp. [Accessed: 23- Sep- 2015].

[4] "GRASP lab", GRASP lab, 2016. [Online]. Available: https://www.grasp.upenn.edu/. [Accessed: 24- Sep- 2015].

[5] "WALRUS Rover", WALRUS Rover, 2016. [Online]. Available: http://robot.neu.edu/walrus/. [Accessed: 25- Sep- 2015].

[6] "WALRUS Rover", WALRUS Rover, 2016. [Online]. Available: http://robot.neu.edu/walrus/. [Accessed: 25- Sep- 2015].

[7] C. Tannert, "Will You Ever Be Able To Afford A Self-Driving Car?", Fast Company, 2014. [Online]. Available: http://www.fastcompany.com/3025722/will-you-ever-be-able-to-afford-a-self-driving-car. [Accessed: 25- Sep- 2015].

[8] H. Durrant-Whyte and T. Bailey, "Simultaneous localization and mapping: part I", IEEE Robotics & Automation Magazine, vol. 13, no. 2, pp. 99-110, 2006.

[9] C. Davies, "iRobot Ava 500 'bot offers pricey telepresence", SlashGear, 2014. [Online]. Available: http://www.slashgear.com/irobot-ava-500-bot-offers-pricey-telepresence-17320938/. [Accessed: 27- Sep- 2015].

[10] "TurtleBot 2 —.0 Clearpath Robotics", Store.clearpathrobotics.com, 2016. [Online]. Available: http://store.clearpathrobotics.com/products/turtlebot-2. [Accessed: 27- Sep- 2015].

[11] B. Spice, "Autonomous Drone Flies in Dark, Tight Quarters To Assist Firefighting Inside Naval Vessels-CMU News - Carnegie Mellon University", Cmu.edu, 2016. [Online]. Available: http://www.cmu.edu/news/stories/archives/2015/february/firefighting-drone.html. [Accessed: 19- Oct- 2015].

[12] User guide 1.0.6 Kobuki Base, 1st ed. Kobuki, 2016.

[13] 2016. [Online]. Available: http://www.nasa.gov/mission_pages/msl/index.html?gclid=CNvwpYb29ssCFQZkhgodyjw [Accessed: 22- Jan- 2016].

[14] "Model Aircraft Operations", Faa.gov. [Online]. Available: https://www.faa.gov/uas/model_aircraft/. [Accessed: 23- Oct- 2015].

[15] Worcester Airport Website, "Massport - Worcester Airport", Massport.com. [Online]. Available: https://www.massport.com/worcester-airport/. [Accessed: 23- Oct- 2015].

[16] T. David Blaza, "It's not just Raspberry Pi", Embedded, 2015. [Online]. Available: http://www.embedded.com/electronics-blogs/say-what-/4439231/It-s-not-just-Raspberry-Pi. [Accessed: 24- Oct- 2015].

[17] "Raspberry Pi 2 / ODROID C1+ Development Boards Comparison", Cnx-software.com, 2015. [Online]. Available: http://www.cnx-software.com/2015/02/02/raspberry-pi-2-odroid-c1-development-boards-comparison/. [Accessed: 27- Oct- 2015].

[18] "Showdown: Raspberry Pi 2 vs ODROID C1 vs HummingBoard vs MIPS Creator CI20", YouTube, 2015. [Online]. Available: https://www.youtube.com/watch?v=vSgka1awkXs. [Accessed: 25- Oct- 2015].

[19] "Raspberry Pi Foundation - About Us", Raspberry Pi. [Online]. Available: https://www.raspberrypi.org/about/. [Accessed: 30- Oct- 2015].

[20] P. Carey, "Review: Intel's computer-on-a-stick: Awesome idea, but its day hasn't arrived", Phys.org, 2015. [Online]. Available: http://phys.org/news/2015-06-intel-computer-on-a-stick-awesome-idea-day.html. [Accessed: 29- Oct- 2015].

[21] "Intel® Compute Stick STCK1A32WFC, STCK1A8LFC Product Brief", Intel. [Online]. Available: http://www.intel.com/content/www/us/en/compute-stick/compute-stick-product-brief.html. [Accessed: 26- Oct- 2015].

[22] "Intel® Compute Stick STCK1A32WFC, STCK1A8LFC Product Brief", Intel. [Online]. Available: http://www.intel.com/content/www/us/en/compute-stick/compute-stick-product-brief.html. [Accessed: 26- Oct- 2015].

[23] Ultrasonic Sensors, 1st ed. VEX Robotics.

[24] J. Levinson, et. al. "Towards fully autonomous driving: Systems and algorithms", 2011 IEEE Intelligent Vehicles Symposium (IV), 2011.]

[25] D. Love, "This Is How Google's New $1 Million Robot Sees The World", Business Insider, 2014. [Online]. Available: http://www.businessinsider.com/atlas-robot-sensor-system-2014-4. [Accessed: 01- Oct- 2015].

[26] "VLP-16", Velodynelidar.com. [Online]. Available: http://velodynelidar.com/vlp-16.html. [Accessed: 12- Oct- 2015].

[27] J. Mrovlje and D. Vrančić, Distance measuring based on stereoscopic pictures, 1st ed. 2008.

[28] M. Mahammed, A. Melhum and F. Kochery, Object Distance Measurement by Stereo VISION, 1st ed. 2013.

[29] M. Kytö, M. Nuutinen and P. Oittinen, Method for measuring stereo camera depth accuracy based on stereoscopic vision, 1st ed. Aalto University School of Science and Technology, Department of Media Technology. [Online]. Available: http://www.helsinki.fi/∼msjnuuti/pdf/EI_2011_kyto_preprint.pdf

[30] "Machine Learning — OpenCV-Python Tutorials 1 documentation", Opencv-python-tutroals.readthedocs.org. [Online]. Available: http://opencv-python-tutroals.readthedocs.org/en/latest/py_tutorials/py_ml/py_table_of_contents_ml/py_table_of_contents_ml.html. [Accessed: 22- Aug- 2015].

[31] Georg Klien, "Parallel Tracking and Mapping for Small AR Workspaces - Source Code", [Online]. Available: http://www.robots.ox.ac.uk/ gk/PTAM/, [Accessed 17- Oct- 2015]

[32] OpenCV, "Optical FLow", [Online], Available: http://docs.opencv.org/master/d7/d8b/tutorial_py_lucas_kanade.htm [Accessed 7- Jul- 2015]

[33] Tim Carmody, "How Motion Detection Works in Xbox Kinect", September 3, 2010, [Online]. Available: http://www.wired.com/2010/11/tonights-release-xbox-kinect-how-does-it-work/ [Accessed 3- Aug- 2015]

[34] Mewgen, "ASUS Xtion Pro Live- Disassemble and weigh", March 8, 2012, [Online], Available: https://www.youtube.com/watch?v=mn80jJ4Io04, [Accessed 26- Oct- 2015]

[35] Pablo Valerio, "Amazon Robotics: loT in The warehouse", [Online]. Available: http://www.informationweek.com/strategic-cio/amazon-robotics-iot-in-the-warehouse/d/d-id/1322366, [Accessed 26- Oct- 2015]

[36] K. Lobosco, "Army of robots to invade Amazon warehouses", CNNMoney, 2014. [Online]. Available: http://money.cnn.com/2014/05/22/technology/amazon-robots/. [Accessed: 07- Feb- 2016].

[37] Min Shi, "Road and Traffic Signs Recognition using Support Vector Machines", [Online], Available: http://www.diva-portal.org/smash/get/diva2:518097/FULLTEXT01.pdf [Accessed 3- Oct- 2015]

[38] Marshall Brain and Tom Harris, “How GPS Receivers Work”, [Online]. Available: http://electronics.howstuffworks.com/gadgets/travel/gps2.htm [Accessed 12- Oct- 2015]

[39] Amit Patel, “Introduction to A*”, [Online]. Available: http://theory.stanford.edu/ amitp/GameProgramming/AStar [Accessed 14- Nov- 2015]

[40] F. von Hundelshausen et. al., ”Driving with tentacles: Integral structures for sensing and motion”, Journal of Field Robotics, vol. 25, no. 9, pp. 640-673, 2008.

[41] “OpenNI Drivers”, [Online]. Available: http://structure.io/openni. [Accessed 27- Oct- 2015]

[42] “Python Wrapper for OpenNI Drivers”, [Online]. Available: https://pypi.python.org/pypi/primesense, [Accessed 27- Oct- 2015]

[43] “Turtlebot 2”, [Online], http://www.clearpathrobotics.com/turtlebot-2-open-source-robot/ [Accessed 15- Mar- 2016]

[44] Willow’s Garage, “gmapping”, [Online]. Available: http://wiki.ros.org/gmapping. [Accessed 16- Mar- 2016]

8 Appendix

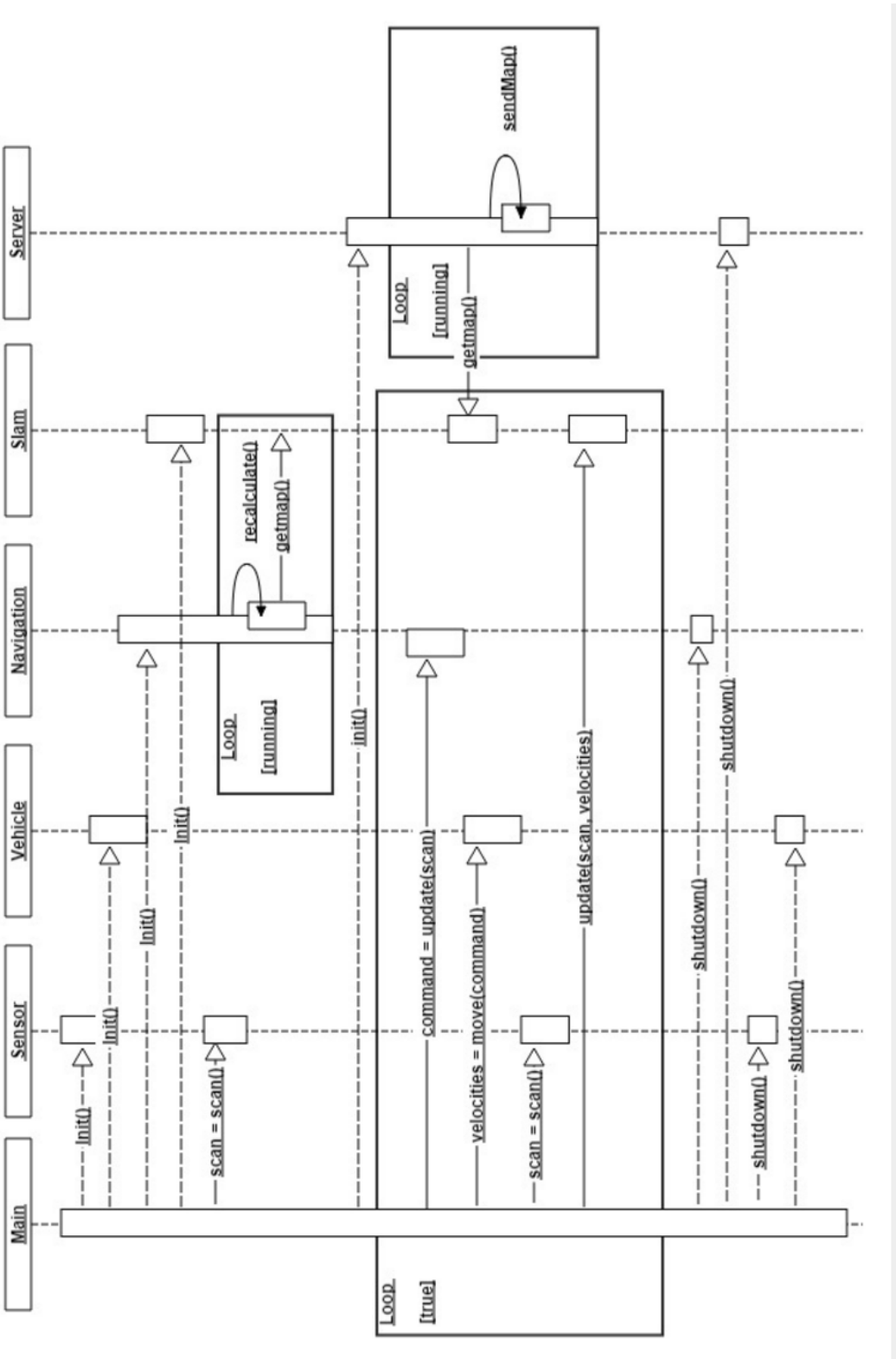


Figure 43: This is a diagram depicting the overview for the software from beginning to end. This shows how the Main thread initializes each of the different classes that are utilized in the Roaming Spirit platform as well as the different threads (main and navigational) that spawn off at run time. Finally, this diagram shows the 'server' thread that is used to broadcast the map in real-time to allow users to watch as the map is created.