Software Testing Plan: Version 1.0

4/6/2023

Floor Explorer Algorithms Team

Sponsored By: Michael Leverington

Mentored by: Rudhira Talla

Members:

Jacob Doyle, Armando Martinez, Luke Domby, Aiden Halili, Vincent Machado

Table Of Contents

1. Introduction

This FEAT software gives users the ability to send programs to a viable robot, while simultaneously providing self-localizing and object avoidance utility. Our finished software has the potential to revolutionize robotics in academia while providing a strong backbone to a larger more applicable/realistic system. The application of these robotics ideologies offers more possibilities and potential for further education, which makes this a very exciting project for the team, Michael Leverington, and Northern Arizona University as a whole.

We intend on being able to put the software into a compatible robot, and apply correct configurations to make this software as robust as possible. This means that we must include extremely detailed documentation about how this software can configure into various actuators and sensors to perform accurately. Here we are going to take a look at these modules individually, and then work into their integration, and explain how a user can interact with this platform.

This document describes our testing agenda with both our individual modules as well as modules integrated into a functioning system. Last but not least, we will test the system and its interactions with a user and show how the robot can be easily configured or altered to meet the user's requirements and serve as a robotics platform. Our entire project has been built around the modularity of several working components that form a sort of hierarchy or behavior tree. The idea being; map-generation gives map information to navigation, navigation runs in parallel to our safety and object avoidance module, and our wifi-localization works independently from them all, except to hand information into our navigation module.

2. Unit Testing

Unit testing will be split up into our major module related sections individually as they form an essential basis for the overall functionality of the system. We test these modules in every external scenario to prove that the software works on its independent functions, then we will integrate several modules together in specific orders, to work our way up towards full system functionality. The split between unit testing and integration testing can become slightly ambiguous because even our individual modules depend on so many different interworking modules to function properly. Most of the under the hood software is however abstracted from the user, so we don’t necessarily have to dive too far into detail about these special cases.

2.1 Map Generation

Map generation was of the first completed modules of the project, as it serves an essential piece of our navigation technique. We test this by ensuring that a map can be generated in any environment, specifically an indoor setting. Testing several floor plan features that potentially cause the robot to behave inaccurately, or providing glass to ensure that the sensors work correctly when they are in proximity. After testing and proving that our map generation movements are suitable, we then prove that every possible datapoint is accounted for and provided in the map.

Mobility of the environment was the first task to test. We thoroughly dialed in our mobility algorithms, and created work-arounds for several interesting environment features that caused inaccuracy. Testing this section of map-generation took the most time to get correct, but it provided insight that we will use throughout our testing phase.

Our next test is the direct mapping ability. After visualizing the data of a constant environment, we could compare its generated map, both in real time and combined into a single map. As stated above, this is a piece of hardware and software that is abstracted from the user, as it does not serve them a purpose beyond learning about the environment. That being said, its integration into the larger system becomes crucial to product functionality, so we conducted thorough testing still. The real trick lies in the overall integration, which in this case we consider to be the conversion of the map into usable coordinates. We will discuss this further in section 3.1.

2.2 Safety and Object Avoidance

Safety and object avoidance is critical for real world applications. The robot will function in a busy environment with lots of changes happening consistently. Unit testing for this module is split into a few different tests, specifically to test the correctness of the robot response to the environment.

Beginning with cliff sensing, we have created a function that detects a cliff, and provides a series of motions to get away from it. If this function is always running in the background, it will continuously check for a cliff and move backwards if one is detected. We applied this to several different cliff encroaching angles, and proved that this function will indeed provide the appropriate reaction.

The same idea applies to our object detection and avoidance functions. We first test object detection in multiple locations in proximity to the robot, and prove that the object detection will always perform if the node is continuously spun in a cycle. We have several different vector angles on our sensor that checks if something exists in its path. It is very easy to see when a specific vector is triggered with simple print statements, so testing this was fairly straightforward.

We perform the same unit tests for the actual object avoidance algorithms, and show that the avoidance functions successfully move the robot away from encroaching objects, as opposed to just detecting that object. After tweaking several mobility algorithms, we finally had enough data to prove that our robot could perform correctly with most environmental settings.

Finally, we ensure that both cliff and object avoidance can work together, although cliff avoidance takes priority. We accomplish this using what is called a reentrant callback group. For this to work, we must put the robot in an environment where both a cliff and an object can be detected. A successful test would be that the robot backs up to avoid the cliff rather than attempting to avoid the object, which is in fact the result that we gathered. Several variations of this same idea did in fact provide sufficient test results to continue.

2.3 Navigation

Navigation can be considered the heart of the system in our case. Dependent on the map generations' input, and in tandem with the safety and object avoidance module, whilst controlling our wifi-localization, this module has a lot happening inside of it. At this stage it is difficult to do a significant amount of unit tests on navigation, due to the fact that it is so heavily dependent on others, but let's take a look at our most basic unit tests.

First, we start with the most basic unit test, which is our point-to-point mobility. Given an end location in a Cartesian coordinate system, we have proven that the robot can navigate to the point. This is all dependent on an origin which is determined at robot boot up, which will come into play later during integration.

Not only must we ensure that the robot can follow a series of point-to-point commands, we must also ensure that odometry accuracy is maintained throughout this process. To begin this process, we direct the robot to its origin, (0,0). It doesn't actually matter which location is considered to be the origin at this stage. Here we measure a meter in every direction from the robot's perspective and lay some tape to mark this point. Since the coordinate system is mapped out using distance as the coordinate unit, we can easily move the robot forward one meter using the coordinate (0,1). We moved the robot to several different coordinates, beginning from this ambiguous origin, and proved the odometry accuracy to have a margin of error less than 15 centimeters from the expected location.

2.4 Wifi-localization

Due to the limitations of the sensors available to us, as well as our time constraints, our wifi localization is limited to confirming whether or not the robot has come to a stop in a physical position that matches where the robot believes it is virtually. This module is to be lightweight and only function when called by the main navigation modules. Wifi localization in the end has three parts: the ability to automatically collect and sort data, the ability to automatically store that data in an easily accessible file and the ability to compare new incoming data to the stored data to determine if it matches to an expected value.

The most basic test to start with is the data collection functions, we can run the functions through a central demo mode. The purpose is straightforward, to test if the functions can grab the mac address for the router nearby and its signal strength and sort them by said signal strength. Since we have run the functions through the demo function we can see the output of the eight strongest signals in the console. This can then be repeated as much as required to ensure the function works before moving on to the storing section. This set of functions will store the data recovered into a text file created in the same location as the wifi files themselves. This will allow us to visually confirm whether we have successfully recovered the wifi information and if it has been sorted correctly. Finally once we have run the demo for storing information at least once we can run the demo for comparisons and check if it matches the expected returns both in the location where it was originally recorded and in a physically different location from where the original tests were performed. The demo will then return a visual response in the console for the tester to see if it confirms a true or false on if it is near the expected location. The confirmation on whether the expected location matches or not should be within a foot for its margin of error.

3. Integration Testing

Similar to unit testing, our integration testing agenda is also broken down into several built upon applications of increasingly integrated modules.The most important being the integration of safety and object avoidance into our navigational functionality. We will stress this idea heavily in this section. Overall, our navigation module is the heart of the system, and is where all of the logic and data is handled and acted upon. We will see that the relationship between each module varies, but is still ultimately handled within navigation. For us to ensure proper integration, it is absolutely essential to have conducted thorough unit testing, as it abstracts from underlying components and ensures reliability when everything is working together.

33.1 Map Generation Integration

When integrating this module, we must show that the map (working individually with the laserscan library and a map server) can collect accurate data and hand it to our navigation module in a usable form. Let’s look a little closer at how it should perform. We have a set of mobility algorithms that move the robot along a wall, theoretically making its way around a room. In a building that is not confined to one room, we must apply the algorithms in several ways to ensure that we track the entire building. Once we ensure that we have the robot tracking out every possible location, we work with the LIDAR’s laser scan, to collect data along the robots path. The laser scan library takes care of the visualization of the collected data point, so we must prove that we have an accurate map to work with by the end of the module's lifetime. After multiple tests and slight mobility adjustments, we are in fact able to visually represent the building.

Now we move into the section of this module in which we transfer data into a usable coordinate system, which proved to be the most difficult task for the team to accomplish. To make this conversion into coordinates happen, we had to rely on another node/module, which handled this process for us. Given all the right information about a map, we can hand it off to the map server and it will spit out a map that our navigation module can use. Once again this is abstracted from the rest of the program from libraries given Nav2 provided by Ros2. Thorough testing on several differently sized environments, we proved that we can make this conversion happen consistently without error.

3.2 Safety and Object Avoidance Integration

We begin by ensuring safety and object avoidance would have total robot control in front of every other task in the stack. But only when certain conditions are met, should it actually produce movement. Proceeded then by the original task that was interrupted by our safety module. This parallelism functions similar to our reentrant callback group described in section 2.2, but is instead referred to as a multithreaded executor, working with entire nodes as opposed to callbacks within a node.

To test this parallelism, we created a node that simply moves the robot forward with no end goal or object detection mechanisms. We run this in tandem with the avoidance module, and observe the robot’s reaction. When no object is detected, we observe the robot moving forward continuously, until it does detect an object. From there, the safety module takes control and attempts to avoid the object, which is exactly the reaction we expected to see. Thus, proving the integration of safety alongside the navigation module. We do iterate this test over again in the next section, but with slightly differing conditions, to further expand on this idea.

3.3 Navigation Integration

In our most complicated section of integration testing. We see that the robot's task must always attempt to be executed, except for when our safety conditions trigger a stoppage. Explained previously in section 3.2. From there we block the navigation task and allow the robot to avoid obstacles. As long as any safety conditions are triggered, the robot will only move away from objects. We can then develop expected reactions from the robot when it is in certain hallway conditions, and prove that the parallelism of the navigation module does in fact hold true.

We begin by substituting a basic move forward function for our point to point mobility, for ease of testing. We then run this in parallel to our safety and object avoidance module, and show the robot with and without presented obstacles. Without any obstacles we observe the robot moving continuously forward, which is to be expected. A present obstacle, on the other hand, makes the robot halt and reverse until the object is out of the desired range. This is also to be expected. Once we have determined that the initial parallelism appears to be in order, we test out different combinations of obstacle proximity, and make sure that the robot can react appropriately for every environmental scenario. With slight algorithm tweaking, the robot was able to avoid most environmental conditions.

Controlling and handling of the wifi-localization module is also handled here in navigation, so we have a few things we must ensure to hold true in various scenarios. We must show that the wifi-localization in fact delivers us useful information about robot localization, and deliver a proper reaction to correct any faults.

3.4 Wifi-localization Integration

The wifi module, for the most part, functions independently from the rest of the modules with it only running when called upon by the navigation module in the specific locations recorded and noted beforehand. Testing this module’s integration with navigation involves ensuring that it successfully is recording its data when called by navigation.

The wifi module must be able to retrieve the appropriate data stored in its text data file according to the location given to it by the navigation module. We can confirm that it is doing this by having it print the location given to it by the navigation module to a debug file and as well as the information it retrieves. After this the wifi module will complete its comparison and print to the file the margin of error as well as its current location data. This will allow us to ensure that the wifi module is correctly comparing its data. Finally we will have the program print to the debug file its return, that is whether or not the program will return true or false, depending on if the functions decided its location match any in its database. By having these features built into the functions we can ensure that the module is working correctly whenever it runs.

4. Usability Testing

Common usability testing would look something like a client logging onto a server and ensuring that they can maneuver the product effectively. This isn't the case with a product such as this, as it is so much more back-end based than it is front-end based. Most of the front-end utensils are used by us developers, whereas users want to see the results that they desire by using the product. Lets look at the project's direct example, the tour program.

A user acquires the robot and wants to run our software. After all necessary downloads and configurations are made (which are specified in the product user manual) the user can boot up the robot in the environment that we want to map. Upon boot up, users have to ssh into the robot via our simple list of commands and execute the software. We can easily describe the process in which to visualize what the robot sees, which is our most valuable front-end technology. From there the user easily executes the map generation commands and stops the program when the environment has been fully mapped, we can see the map both before coordinates and after, so the user has a few tools to learn about the environment.

From there, the proper map conversions are made and the result is handed over to the navigation module in which we program tasks. We execute the appropriate file and gather the information in which to execute the tour program, and boom, the tour is running.

As you can see, the overall usability of the software is not meant for the general public at this stage. Many more alterations to interfaces and processes can be applied to better suit a users need, but this is clearly much more of an academic utensil than it is a public product. This could serve in a real life application as long as the user is fully aware of the robotics concepts that we work with.

5. Conclusion

As you can see, we have done thorough testing on our system throughout the entirety of the project. We designed the product in a way to abstract and provide modularity to users to make this process easier in totality, and the team agrees that this was the best way to approach a project of this magnitude. Starting from the ground up, testing every individual piece in order as we developed them, we were able to ensure that we would have no backtracking in this project. From individual modules to overall system integration, we understand that this software has a lot of moving parts. We must thoroughly test all of these varying conditions, and make sure that this software has life after initial development, so it is important to treat testing as equally important as any other phase of the development process.