Path Optimization Using STL Specifications

Konrad Rozanski
Electrical Engineering
University of Notre Dame
Notre Dame, IN
krozansk@nd.edu

***Abstract*—This report details the progress that was made over the course of the spring semester in translating a map of a region into an STL formulation that could be used to determine an optimal route. Beginning with a description of the process used to develop the ROS nodes and their structuring, this paper describes several scenarios, the problems encountered, possible solutions that have been tried, and plans for future work.**

***Index Terms*—Signal Temporal Logic (STL), Object Oriented Programming (OOP), Robot Operating System (ROS), Simultaneous Localization and Mapping Algorithm (SLAM), Light Detection and Ranging Sensor (LIDAR)**

I. Introduction

In planning the project for this semester, the work done last semester was considered. Being that the prior semester was my first semester working in the Discover Lab, it was a fairly simple project mainly concerned with helping me gain a proper understanding of ROS. That said, I thought the principles with which the project was developed were worth further pursuing. Seeking to mirror the fashion in which Amazon uses robots to retrieve boxes within their warehouses, Patrick McGrath and I came up with a project that consisted of using Optitrack to locate a box and robot within a region. Using these coordinates, our ROS nodes outputted a set of coordinates relative to the Turtlebot such that they could be properly fed into the topic that was used to drive the Turtlebot. The Github repository containing the code can be found here: https://github.com/KonRoz/Pusher.

In the case of that former project, the route to the box was planned using a heuristic approach. After publishing the processed goal coordinates to the `move_base_simple/goal` topic, the Turtlebot's path planning algorithm took over completely. After experimenting for a few weeks, it became apparent that this heuristic approach would oftentimes output results that were unsatisfactory with the path being unnecessarily long, passing through obstacles within the region of interest, or, worst of all, simply generating no path to the object. As such, I decided to tackle this problem of path planning over the course of this semester. Initially, my overall goal was to explore alternative methods of path planning. At Vince's suggestion, I decided to take the approach of translating the map into a STL formulation and utilizing the robustness function to determine the best path.

As was first discovered last semester, such projects often take much longer than expected to complete. In the case of this semester, a great deal of time was spent dealing with poor resulting paths. Fixing the aforementioned issue required sifting through several aspects of the project including the STL formulation of the region, the optimization algorithm's parameters, and the control system used to model the Turtlebot. The code for this project can be located on Github in the following repository: https://github.com/KonRoz/OccupancyGridToSTL. All instructions necessary to properly run the project are included in the README file.

There are six total files included in the project repository. However, three form the core of the project: `STLtranslator`, `ROSstuff`, and `Optimizer`. The other files in the repository represent variations of the three prior files. They were introduced while trying to fix the problems that were encountered. That said, `ROSstuff` is the master file. It uses an instance of the `STLtranslator` and `Optimizer` to process the map and determine an optimal route. The idea was to isolate the three main processes involved in this project – interacting with the Turlebot, creating an STL formulation, and determining an optimal route – in such a fashion that all three can be easily accessed and changed.

II. STL Overview

STL can be described as a type of time dependent Boolean logic. Unlike in Boolean logic, STL formulas are qualified using a time domain. That said, STL is defined over a signal $y(t)$ which is continuous over the entire time domain. In addition to the Boolean logical operators of NOT, AND, and OR, STL possesses time dependent ALWAYS and EVENTUALLY operators. These various operators can be used to define predicates. One can then combine predicates to create specifications that can be further combined to create one global specification describing the desired behavior of a system. This is particularly useful because one can define this global specification in terms of *Robust Semantics* [1]. In doing so, a given signal no longer discretely satisfies or does not satisfy the specification. Now, there is a quantitative indication of how well a signal satisfies the given specification. And so, an optimization problem can be defined where the goal is to find the signal that best satisfies the robustness function. In the context of this paper, this is particularly useful as one can create a cost function which can be minimized to find the optimal route through the region.

As I learned over the course of this semester, STL formulations have several advantages over other approaches. A particularly notable example includes the ability to specify complex combinations of time dependent actions that would be otherwise difficult to implement. Just as architects construct complex designs out of simple building blocks, STL allows for the construction of complex specifications from simple predicates. As is later illustrated with the description of the base specification class in `STLTranslator`, one can begin with a simple specification describing solely the obstacles within a region and build upon it with increasingly complex scenarios. Perhaps this means reaching one of two regions before continuing onto the goal. In short, the ease with which one can build upon scenarios using STL is very useful and gives it an edge over other approaches.

III. Generating Map of Region

The first step toward developing a path finding algorithm that could be used by the Turtlebot consisted of retrieving a map of the region and making sufficient sense of it to the point where it could be accurately translated into a STL formulation. When interested in creating a map of a region for use by the Turtlebot, one runs the SLAM algorithm. SLAM utilizes the Turtlebot's LIDAR to generate a grid of cells that carry values to indicate occupied and unoccupied regions. Being that the STL formulation for the region would be built out of formulas representing each occupied cell, the number of occupied cells within the region was of great importance. An increase in the number of occupied cells results in an increasingly complex computation when optimizing a signal over the robustness function.

As such, the goal was to start off with a sufficiently straightforward region with very few occupied cells so that the optimization would be completed quickly for easy debugging. In short, the goal was to achieve a high resolution for the generated map (resolution is measured in meters/cell). This is achieved by changing the delta parameter in the particular SLAM method that is being used. In the case of this project, this was done by adding a `set_delta` parameter to the launch file for `gmapping` so that the resolution of the resulting map could be changed as a parameter within the command used to run SLAM. Before leaving campus to work remotely, I was able to generate four maps with varying resolutions: 1 cm/cell, 10 cm/cell, 50 cm/cell, and 1 m/cell. An example is portrayed in Figure 1.

Having generated the desired maps, the next step was parsing the map such that the value of each cell within the map could be read. After running SLAM to generate a map, two files are created: a yaml file containing the map's metadata and a pgm file containing individual cell data. On their own, it is rather difficult to extract data in a useful form from these files. As such, a ROS node called `map_server` was used. `Map_server` takes in the yaml file from the map that you are interested in reading, and publishes the data in the easily read message type, `OccupancyGrid` (found in `nav_msgs`), to the `/map` topic. `OccupancyGrid` possesses the `MapMetaData` message which provides one with easily accessible information regarding the map's dimensions and resolution. It also contains an array of the cells that describes the layout of the region. These cells possess three values: 100, representing occupied, 0, representing open, and -1, representing unmapped. A node subscribing to the `/map` topic and retrieving the desired occupancy grid was written. It can be found in `ROSstuff`.

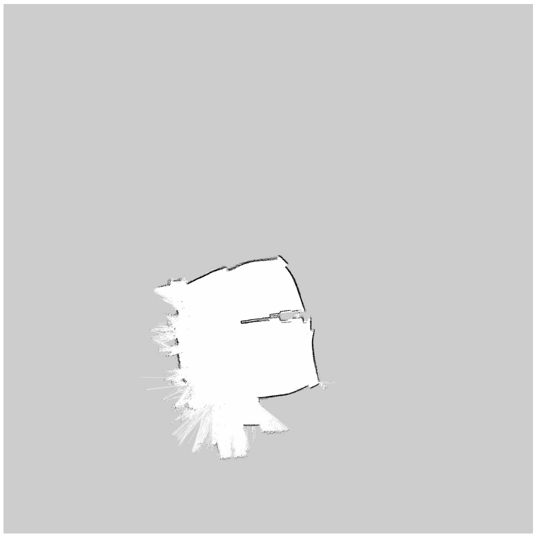
Fig. 1. A 1 cm/cell SLAM map where the grey cells represent unexplored regions with a value of -1, black cells represent occupied regions with a value of 100, and white cells represent free regions with a value of 0. Defining STL formulas for cells this small is generally unnecessary. As such, the SLAM maps used in Figures 10 and 11 were 1 m/cell and 50 cm/cell respectively.

IV. Parsing and Translating the Occupancy Grid

From the subscribing node in `ROSstuff`, the occupancy grid is passed to `STLtranslator`. Written in an OOP format, the `STLtranslator` completes the translation of the occupancy grid into a complete STL formulation. At the moment, the file contains a large parent class, `BaseSpecification`, and one inheriting class, `ReachAvoid`. `BaseSpecification` generates a specification solely based upon the obstacles presented in the occupancy grid. I thought doing so was logical due to the fact that all derived specifications will contain formulas for obstacles within the region. A basic augmentation of the base specification parent class is observed in `ReachAvoid` where the final specification includes a goal region. In the future, far more complex specifications can be developed off of the base specification. In `BaseSpecification`, a final specification including every obstacle within the passed occupancy grid is developed. This was accomplished by passing every obstacle contained within a list to a method that returns a rectangular STL specification. Every STL specification is then combined using the STL equivalent of the AND logical operator, CONJUNCTION. Defining the STL predicates for each side of the rectangle was implemented with the help of Vince Kurtz's pySTL library. The Github page is https://github.com/vincekurtz/pySTL. It also provided support for the STL logical operators needed to construct the final specification for the entire region.

A visual representation of the fashion in which the specification for one cell was developed is displayed below in Figure 2. That said, it is important to note that the specification displayed in Equation 1 outputs a positive robustness if the signal satisfies all four predicates. That is, the predicates are defined so that when the signal is less than xmax, greater than xmin, less than ymax, and greater than ymin the resulting robustness is positive. Given that it is being defined for an obstacle, we want a positive robustness value to be outputted when the signal is outside the four boundaries defined by the predicates. Thus, the STL equivalent of NOT, NEGATION, is needed after all four predicates are combined using CONJUNCTION. The ALWAYS operator is also applied to the specification with the appropriate time bound as to ensure that the obstacles are avoided at all times. In ReachAvoid, `in_rectangle_formula` is used without NEGATION and with EVENTUALLY to specify a goal region.

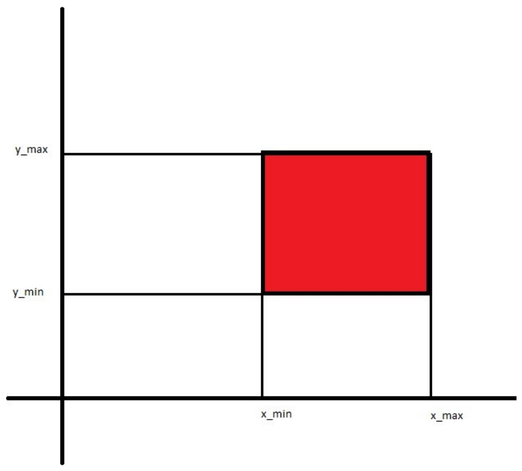

Fig. 2. This Figure illustrates the four parameters that are needed to define an STL formula for a cell that is occupied. See Equations 1, 2, and 3 for a derivation of the STL formula used to define obstacles in this project.

$$S(t) = \begin{bmatrix} x(t) & y(t) \end{bmatrix}^T$$
$$p_1 = ([1,0]S \geq x_{min})$$
$$p_2 = ([1,0]S \leq x_{max})$$
$$p_3 = ([0,1]S \geq y_{min})$$
$$p_4 = ([0,1]S \leq y_{max})$$
$$F_{inside} = p_1 \wedge p_2 \wedge p_3 \wedge p_4 \tag{1}$$
$$F_{outside} = \neg F_{inside} \tag{2}$$
$$F_{avoid} = \mathbf{G}_{[0,time]} F_{outside} \tag{3}$$

V. Generating Optimal Path

Possessing an STL specification for the region and a corresponding robustness function, the next challenge was to generate an optimal path through the region in question. It was decided that the best way to go about doing so was by defining a cost function that could be optimized using SciPy. I accomplished this by defining three functions: `cost_function`, `rho`, and `STL_signal`. `Cost_function` is the function which is directly passed into the SciPy minimize function. Optimizing the cost function resulted in a control signal, u, that could be plotted using the `STL_signal` method.

The `STL_signal` method that I ended up setting as the default is the double integrator. See Equation 4 for an illustration of the system definition. However, as is described later in the report, I tried using a single integrator in order to help get rid of the optimizer's tendency to get stuck in local minima. Before using the specification generated by `STLtranslator` for optimization, a rectangular set of predicates was added to bound the control signal. This was accomplished by using a method similar to `in_rectangle_formula` aside from the part of the signal (S is a 4xT NumPy array where row one contains the x part of the signal, row two contains the y part of the signal, and rows three and four contain the x and y components of the control) that is passed into the lambda function used to define the STL formulas for each of the control bounds. The particular optimization algorithm being used and related settings could be easily changed and greatly affected the results.

$$x_{t+1} = x_t A + u_t B, u_t : acceleration \tag{4}$$
$$A = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}, B = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{bmatrix}$$

VI. Optimization Algorithms

Over the course of the past semester, I have primarily experimented with Nelder-Mead and Powell. Both of these are black-box global optimization algorithms. As is later illustrated in Figure 7, Nelder-Mead takes a geometric approach. It belongs to the general class of direct search methods due to the fact that it does not construct a gradient as many of the other SciPy algorithms do. Likewise, it uses only some function values at some points in $\mathbf{R}^n$ – in the case of this project $\mathbf{R}^2$. The geometric means that was earlier mentioned is due to the fact that Nelder-Mead is simplex based [3]. In other words, it uses a simplex S (in this case a triangle), and evaluates the function values at all vertices. The method then uses transformations on S in order to minimize the function values at all the vertices. Computation is terminated when S becomes smaller than initially specified at execution. The paper from which SciPy implemented Nelder-Mead is noted in References [2].

Powell's method, the other algorithm that was utilized in this project, also does not use derivatives as it uses the function values instead. It is classified as a conjugate direction method. In short, the algorithm starts with an initial point and a set of search vectors. From this initial point, the direction vectors in the search set are minimized. The direction which results in the greatest decrease in the function's value is chosen and then used to move the initial point. That vector is then removed from the set, a displacement vector from the initial point to the next point is added to the search set, and the process of finding the direction of the largest decrease is repeated. Termination of the algorithm occurs after an arbitrary number of cycles over which no significant improvement is made [5]. The particular version of Powell's method implemented by SciPy is noted in References [4].

VII. Results and Challenges

After obtaining control signals that resulted in unsatisfactory results with the double integrator (e.g. signal passing through multiple obstacles or not passing through final goal), I set about trying to determine whether the issue was the STL formulation, the optimizer getting stuck in local minima, or something else. I combated the STL formulation problem by writing another Python file, `Test`, that allowed me to define a very simple occupancy grid that I could pass into instances of the `STLtranslator` and `Optimizer`. Doing so with the standard double integrator yielded very good results regardless of the optimization algorithm passed into SciPy.

The scenario portrayed in Figure 3 illustrates that the STL formulation is properly defined as the resulting path is good and the final robustness value is positive. Another illustration of planning a successful path in a simple region is portrayed in Figure 4 using a different obstacle layout and goal region. In both of the displayed scenarios the initial x and y velocities are zero and the route is computed over a time interval of 20 seconds. Figure 3 had an initial position $(-0.5, -0.5)$ while Figure 4 had an initial position $(0, 0)$.

Experimenting with the optimizer's parameters in these simple scenarios, it rapidly became apparent how much a small change can affect the final result. Simply changing the optimization algorithm can result in a completely different path. Successfully having verified that it was in fact not the STL specification that was causing the unsatisfactory paths to be outputted, I then surmised that the next most likely cause of the issue was related to the solver getting stuck in local minima. Such reasoning would explain why some optimization algorithms yielded significantly different results from one another in the same scenario optimizing over the exact same cost function. I went about addressing this problem in two ways. The first involved rewriting the `STL_signal` method in the Optimizer file such that the robot's dynamics were defined in terms of a single integrator as opposed to a double integrator. The second involved supplying the SciPy solver with a better initial guess.

Starting with the former, I added a new file to the project, `OptimizerSingleIntegrator`. The biggest change in-

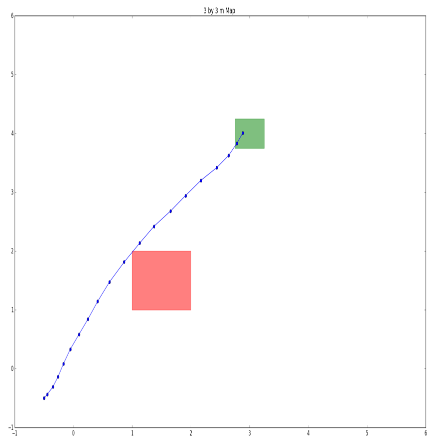

Fig. 3. Nelder-Mead was used to generate the above path through a very simple region defined by an STL formula with a single obstacle and one goal region. The resulting path is good as the final robustness value is positive, none of the signal points fall inside the obstacle, and the final region is reached successfully.

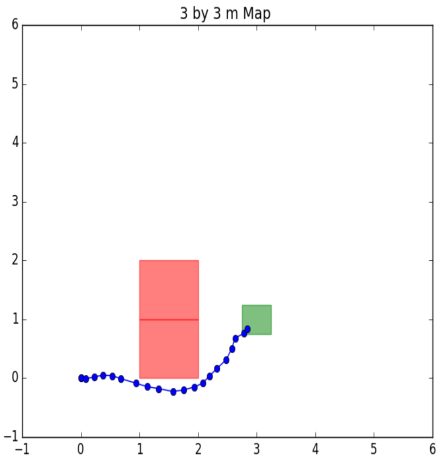

Fig. 4. A more complex scenario than Figure 3, this Figure adds an extra obstacle and moves the initial and final points so that the obstacles are completely in between them. Nelder-Mead successfully located the global minimum and avoided the local minimum represented by a path going right across the obstacles, straight to the goal. The final robustness was again positive and computation time was held under 10 seconds.

volved the A and B matrices as is shown in Equation 5. The control signal, u_t, now represented the velocity of the robot as opposed to its acceleration. Likewise, the initial position of the robot, x_0, was changed from a vector with length 4, $[x_0, x'_0, y_0, y'_0]$, to a vector of length 2, $[x_0, y_0]$. These changes yielded interesting results. In order to test my hypothesis that the change to the single integrator would yield fewer minima and better results, I found a scenario in which the double integrator produced an unsatisfactory result and re-ran it with the single integrator optimizer.

(5) $$x_{t+1} = x_t A + u_t B, u_t : velocity$$
$$A = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, B = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Unfortunately, the outputted path for both the single and double integrator were unsatisfactory. That said, the single integrator's path was more promising due to the fact that it largely avoided the obstacle region. This indicates that it did not get stuck in the same minima that affected the double integrator (See Figures 5 and 6). I have not yet tried a large amount of different scenarios in which the double integrator gets stuck in a particularly bad minimum (e.g. running right across the obstacles) and the single integrator comes up with a somewhat satisfactory path (e.g. avoiding the obstacles but failing to reach the goal region) to definitively say that the single integrator produces fewer local minima, and, thus, tends to produce better paths. However, I noticed in situations where the double integrator did not get stuck in local minima, the resulting path tended to be smoother than the one that was produced by the single integrator. In general, the single integrator tended to produce erratic paths without a significant reduction in the number of local minima. Thus, the the hypothesis that a single integrator would result in fewer minima to the point where finding a satisfactory path was more likely than with the double integrator was debunked.

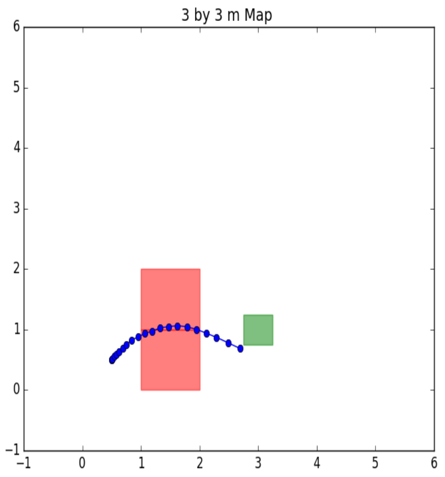

Fig. 5. This path was generated using the double integrator model. As is illustrated by the path passing through the obstacles, Nelder-Mead fell into a local minimum. The final robustness value was negative.

Seeing that the single integrator produced only marginally better results in certain scenarios, I surmised that the true problem was with the initial guess passed to the SciPy solver. It was hypothesized that a poor guess could result in getting stuck in local minima. I tested this hypothesis by writing another Python file, `Trajectories`, and modifying `Optimizer`. `Optimizer` now writes every control signal that it outputs to a cvs file called `controls.out` that is then read by

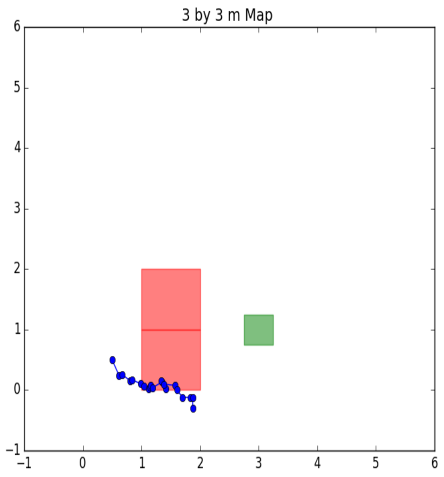

Fig. 6. This path was generated with the single integrator model. Although Nelder-Mead again fell into a local minimum, the resulting path possessed a robustness value that less negative than the robustness of the double integrator.

`Trajectories`. `Trajectories` then graphs several of these controls to give the user an idea of whether certain paths have been iterated over. `Trajectories`' output for the scenario detailed in Figure 5 is displayed in Figure 7. Figure 7 clearly shows that the selected optimization algorithm, Nelder-Mead, did not produce any controls that fell close to what one would consider the trivial route to the box. Thus, it can be assumed that the solver got stuck in local minima which happened to be represented by going straight across the obstacles to the goal region (See Figure 5).

Such a problem can be solved by either using a different solver which does not possess a tendency to get stuck in the same local minima or using a better initial guess. Using `Trajectories` to display the paths tested by the solver, it was revealed that Nelder-Mead has a tendency to test a sequence of likely paths that are closely related to each other. This made getting stuck in local minima quite likely. Thus, I tried using Powell's optimization method. I also provided the solver with an initial guess pointing straight up from the starting point at (0.5,1). And so, using both a different solver and better initial guess, I obtained the result shown in Figure 8. The improvement based on the better guess and new solver is best seen in Figure 9 where the plot outputted by `Trajectories` illustrates the fact that the trivial paths are part of the set tested by the solver. And so, using good initial guesses and solvers that do not get stuck in local minima appear to be the most promising avenues toward obtaining better optimization results.

After obtaining satisfactory results using the basic occupancy grid defined in `Test`, I decided to move on to the maps I created of the region in Discover Lab. The results are displayed in Figure 10 and 11. Figure 10 was obtained using the map with a resolution of 1 m/cell and Figure 11 was obtained using a map with a resolution of 50 cm/cell. Both

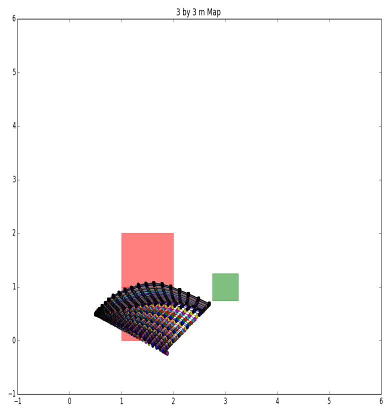

Fig. 7. These are the trajectories tested by Nelder-Mead in generating the route shown in Figure 5. `Trajectories` was used to plot every 50th trajectory outputted by the optimizer. It is apparent that Nelder-Mead was stuck in a local minimum due to the fact that none of the tested trajectories pass around the obstacle.

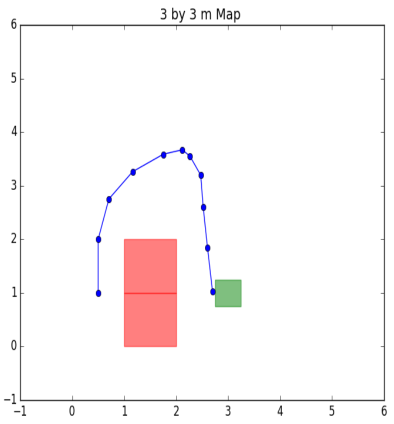

Fig. 8. Powell's method was used to generate the above path while using a double integrator model. Although it is not ideal due to the great distance between the obstacles and the peak of the trajectory, Powell's method was able to avoid the local minimum represented by crossing over the middle of the obstacles to the goal region in which Nelder-Mead was caught. The final robustness value for the path was positive.

employed the use of Nelder-Mead. It can be said that both paths are satisfactory. The results that were obtained using the sample occupancy grid in `Test` were largely replicated in the real world scenarios. That said, the computation time between Figures 5,6,8 and Figures 10,11 differed slightly with an increase of a few seconds for 10 and 11. This is most likely due to the fact that the STL formulation in Figures 10 and 11 is significantly more complex as the number of obstacles increased greatly.

VIII. Future Plans

As was detailed in this report, moderate success was achieved with translating a ROS `OccupancyGrid` into a

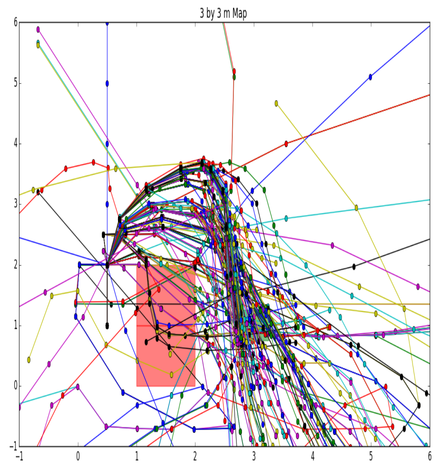

Fig. 9. These are the trajectories that were tested by Powell's method while generating the route shown in Figure 8. In contrast to the trajectories shown in Figure 7, many trajectories that do not pass through the region are tested. It appears that the algorithm was able to successfully determine that the global minimum was located around the top of the obstacle because the majority of the tested trajectories are concentrated in that particular area.

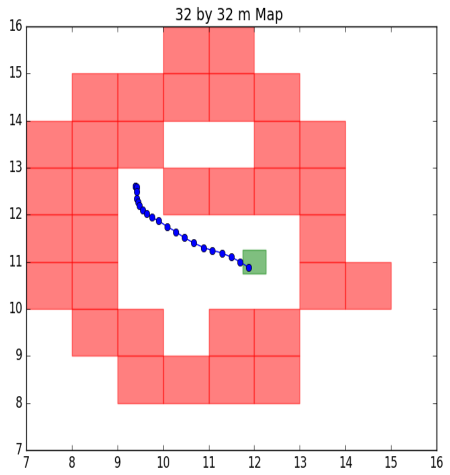

Fig. 10. Using the SLAM generated map with a resolution of 1 m/cell, this Figure illustrates the use of Nelder-Mead to generate a route through a real world scenario. The generated route successfully avoids local minima and locates the global minimum. The final robustness value was positive.

STL specification that could be used to determine an optimal path through a given region. That said, there are many problems with the current results. A few notable examples include the slow computation time (i.e. up to 15 seconds) and unsatisfactory paths being outputted in certain situations. I believe slow computation time can be addressed through the rewriting of the ROS nodes in C++ due to the fact that Python is not optimal for speed in this situation. The problem of computation time is also related to the solver being used and the set parameters. Additionally, the outputting of

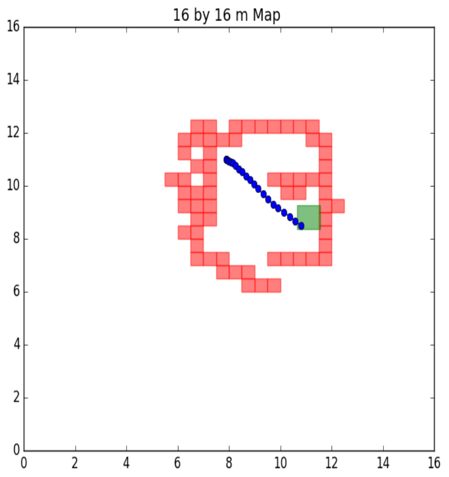

Fig. 11. This route was generated using Nelder-Mead. The initial position and goal region were kept the same as in Figure 10. The only change involved the resolution of the map becoming 50 cm/cell. The resulting route is satisfactory and the final robustness value is positive.

unsatisfactory paths can be tackled by trying other solvers. Thus, one of the first things I will try will be implementing new solvers. A few that I was unable to try include dogleg and BFGS. Assuming the reliability of this path optimization code can be improved, the next step will be to try writing a PID controller for the Turtlebot that can be fed the control signal. Other possible future plans include compiling different STL specifications from the `BaseSpecification` class. For example, go to one region before continuing onward to the goal region. In short, possibilities abound. That said, the most pressing problem is sorting out the reliability issue concerning getting stuck in local minima.

Acknowledgment

To conclude, I would like to thank Dr. Hai Lin for allowing me to work in his lab this semester. Likewise, I would like to thank Vince Kurtz for his assistance and expertise.

References

[1] Calin Belta and Sadra Sadraddini. "Formal Methods for Control Synthesis: An Optimization Perspective". In: *Annual Review of Control, Robotics, and Autonomous Systems* 2.1 (2019), pp. 115–140. DOI: 10.1146/annurev-control-053018-023717. eprint: https://doi.org/10.1146/annurev-control-053018-023717. URL: https://doi.org/10.1146/annurev-control-053018-023717.

[2] Fuchang Gao and Lixing Han. "Implementing the Nelder-Mead simplex algorithm with adaptive parameters". In: *Computational Optimization and Applications* 51.1 (Jan. 2012), pp. 259–277. ISSN: 1573-2894. DOI: 10.1007/s10589-010-9329-3. URL: https://doi.org/10.1007/s10589-010-9329-3.

[3] "Nelder-Mead algorithm". In: *Scholarpedia* 4 (2009), p. 2928. URL: http://www.scholarpedia.org/article/Nelder-Mead_algorithm%20AU%20-%20Singer,%20S.%20AU%20-%20Nelder,%20J..

[4] M. J. D. Powell. "An efficient method for finding the minimum of a function of several variables without calculating derivatives". In: *The Computer Journal* 7.2 (Jan. 1964), pp. 155–162. ISSN: 0010-4620. DOI: 10.1093/comjnl/7.2.155. URL: https://doi.org/10.1093/comjnl/7.2.155.

[5] Wikipedia contributors. *Powell's method — Wikipedia, The Free Encyclopedia*. [Online; accessed 29-April-2020]. 2020. URL: https://en.wikipedia.org/w/index.php?title=Powell%27s_method&oldid=950974472.