**ECE8833 Computational Intelligence**

Spring 2023

**Final Project**

**Multi-Waypoint Path Planning with Digital Twin Technology for Dynamic Priority-Based Waypoint Routing and Navigation**

Student: Kyler Farrar

Student NetID: kaf386

Student E-mail: kaf386@msstate.edu

Instructor: Dr. Chaomin Luo

Department of Electrical and Computer Engineering

Bagley College of Engineering

Mississippi State University

Date: April 10th, 2023

Table of Contents

[Introduction 4](#_Toc134584494)

[Project Description 5](#_Toc134584495)

[Global Path Planning: Self-Organizing Map 5](#_Toc134584496)

[Point-to-Point Path Planning: Graft-RRT 10](#_Toc134584497)

[Local Navigation: VFH+ 18](#_Toc134584498)

[Digital Twin Implementation 26](#_Toc134584499)

[Priority Waypoint Transition 27](#_Toc134584500)

[Hidden Obstacle Transition 29](#_Toc134584501)

[System Flow Overview 33](#_Toc134584502)

[Results 35](#_Toc134584503)

[Conclusions and Future Work 40](#_Toc134584504)

[References 41](#_Toc134584505)

[Appendices 43](#_Toc134584506)

[Appendix 1 – TSP Dataset 1 44](#_Toc134584507)

[Appendix 2 – TSP Dataset 2 46](#_Toc134584508)

[Appendix 3 – Map 1 52](#_Toc134584509)

[Appendix 4 – Map 2 53](#_Toc134584510)

[Appendix 5 – Map 3 54](#_Toc134584511)

[Appendix 6 – Full Output Map 1 55](#_Toc134584512)

[Appendix 7 – Full Output Map 2 57](#_Toc134584513)

[Appendix 8 – Full Output Map 3 59](#_Toc134584514)

[Appendix 9 – Block Diagram 61](#_Toc134584515)

[Appendix 10 – Main.m Code 62](#_Toc134584516)

[Appendix 11 – Map(s) Code 69](#_Toc134584517)

[Map1.m 69](#_Toc134584518)

[Map2.m 72](#_Toc134584519)

[Map3.m 75](#_Toc134584520)

[Helpers/CreateObstacle.m 78](#_Toc134584521)

[Helpers/CreateWaypoint.m 79](#_Toc134584522)

[Appendix 12 – Digital Twin Code 80](#_Toc134584523)

[ObstacleOverride.m 80](#_Toc134584524)

[PriorityOverride.m 82](#_Toc134584525)

[Appendix 13 – Self Organizing Map Code 85](#_Toc134584526)

[Helpers/GetClosestPoint.m 89](#_Toc134584527)

[Appendix 14 – Graft-RRT Code 90](#_Toc134584528)

[Helpers/g\_rstara.m 93](#_Toc134584529)

[Helpers/g\_rstarb.m 97](#_Toc134584530)

[Helpers/check\_globalvalue.m 101](#_Toc134584531)

[Helpers/check\_globalvalueII.m 102](#_Toc134584532)

[Helpers/check\_loacalvalue.m 103](#_Toc134584533)

[Helpers/CheckCollision.m 104](#_Toc134584534)

[Helpers/DistanceCost.m 106](#_Toc134584535)

[Helpers/graft\_extend.m 107](#_Toc134584536)

[Helpers/in\_distance.m 111](#_Toc134584537)

[Helpers/line\_check.m 112](#_Toc134584538)

[Helpers/ObstacleFree.m 114](#_Toc134584539)

[Helpers/rand\_center.m 115](#_Toc134584540)

[Helpers/RandomPoint.m 118](#_Toc134584541)

[Helpers/re\_path.m 119](#_Toc134584542)

[Appendix 15 - README 122](#_Toc134584543)

# Introduction

The effective navigation and routing of multi-waypoint scenarios in mobile robotic platforms is a complex and often-researched phenomenon within modern academia. Fundamentally, this problem is comprised of many core principles which find their roots in NP-hard computations within a real-time scope, thus, it is not uncommon for research work to find its basis within these fundamental problems in isolation rather than as a fully assembled system. A plethora of research exists examining and introducing new means for global path planning, point-to-point path planning, and local navigation. A novel technology that has recently garnered interest in both industry and academic settings, due to its usefulness in complex navigation problems, is the effective application of Digital Twin technologies. The primary concept behind a Digital Twin system is to create a virtual model of a real-world robotic platform in a specific scenario and simulate the platform’s behavior in order to understand its performance and optimality in various scenarios. The application of the Digital Twin concept has become feasible due to recent advancements in computer computational capabilities as they relate to simulation and modeling of real-world systems. This work focuses on the integrations of a global path planner, point-to-point path planner, and a local navigator into a unified system capable of navigating muti-waypoint scenarios accompanied by a Digital Twin simulation which responds to waypoint priority and obstacle configuration changes within the environment. These scenarios are implemented in a multi-robot system such that a waypoint-based Traveling Salesman Problem (TSP) is solved by a simulated Unmanned Aerial Vehicle (UAV) which is then passed to an Unmanned Ground Vehicle (UGV) for point-to-point path planning and local navigation. This model provides a basis for the setup of a disaster-relief or natural disaster scenario such as ensuring houses have been evacuated due to an incoming wildfire or other priority-based search and rescue operations.

# Project Description

In order to simulate the UGV and UAV systems the problem was broken into three main stages: global path planning, point-to-point path planning, and local navigation. The global path planning problem can best be simplified to the traditional Travelling Salesman Problem (TSP). A popular methodology for solving TSPs, Self-Organizing Maps, was selected for implementation. In order to introduce point-to-point route planning, a recently proposed path planning algorithm called Graft-RRT (Rapidly Exploring Random Trees), which is based off the more well-researched Bi-Directional RRT\* algorithm, was implemented. Finally, the VFH+ algorithm was applied to facilitate local navigation to and from goal points on the path planning route. These systems are integrated tightly with a novel Digital Twin system which pre-computes researcher-specified obstacle and waypoint priority modifications to the initial map.

# Global Path Planning: Self-Organizing Map

Self-Organizing Maps (SOMs) were originally introduced in the late 1990s by T. Kohonen for the purpose of reducing the dimensionality of high-dimension data in an unsupervised manner [1]. This is done by mapping data points into a representative lower dimension dataset, such as modeling three-dimensional data in a two-dimension map, by clustering values which share similar properties together via neurons. These neurons (or nodes) represent grid points within the two-dimensional map and are trained on the higher-dimensionality dataset such that a component called the weight vector applies an appropriate vector to input data such that similar data points are appropriately clustered together and can be classified uniquely. In the application of the SOM methodology to the TSP, a proposed solution written in Python by D. Vicente in [2] is leveraged. D. Vicente modifies the SOM structure from a typical grid-like structure to a circular ring. Each node in this ring is then only cognizant of its neighboring nodes to the left and right in this ring. This methodology allows for an effect akin to having a rubber band wrapped around a grouping of thumb tacks. Applying this modification to the algorithm allows it to converge neurons/nodes towards the city locations while minimizing the perimeter of the polygon it generates. Additional modifications were also introduced by D. Vicente which introduce a learning rate model to encourage initial exploration and then transition to favoring exploitation as convergence occurs. A related change can also be seen in relation to the regression and weight update formulation. These convergence methodologies were modeled after Q-Learning algorithms.

The SOM TSP algorithm provided by D. Vicente, while effective in providing near-optimal solutions, does so at an impractical rate for real-time operations, therefore, a mechanism was introduced into the algorithm which allows for the prioritization of returning a slightly less optimal solution once the system begins to enter a plateau-like state. This is achieved by looking at the mean error between the previous iteration and the current iteration and determining if it is less than a given threshold. In order to avoid undesirable terminations, where the general error in the solution is undesirable regardless of the plateau state, the error of the current state is insured to be within a given threshold. A very high-level algorithm representing this system is shown in Algorithm 1.

Table

Description automatically generated

Algorithm 1: Real-time Priority SOM Algorithm

The high-level overview in Algorithm 1 focuses on the “isPlatue” implementation as it is the only significant point of change in the algorithm from D. Vicente. An additional change to better support this training regimen was made such that the specific waypoint being trained on is no longer random and instead training is done in the order they occur in the array. Finally, the code-set provided by D. Vicente is written in Python and was converted to work within MATLAB. These algorithm modifications were tested and compared to the original algorithm on two TSP datasets available in Appendix I and II, respectively. Of particular interest is the effects of these modifications on execution time versus total path distance. An average was taken on five runs against these datasets and placed into Table 1 for analysis.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Data Set | Cities | Algorithm | Calculation Time | Route Distance |
| 1 | 225 | Original | 217.6181 seconds | 4071.0225 units |
| 1 | 225 | Modified | 107.6577 seconds | 4137.0858 units |
| 2 | 45 | Original | 31.5373 seconds | 667.3191 units |
| 2 | 45 | Modified | 13.5929 seconds | 684.8095 units |

Table 1: SOM TSP Performance

Table 1 demonstrates the ability of the early termination code to drastically reduce calculation time by over half in most instances. The impact to route cost can be denoted as incurring an average approximate increase of 2.1% to the total route length. This means that in larger city sets, significantly larger route deviation may be introduced. In the interest of maintaining closer to real-time execution for the purpose of this project’s scenario, the modified algorithm provides sufficient, actionable, and desirable results. Table 1’s results were recorded with the number of iterations set to 100000 and the learning rate at 0.8. Additional tuning is a crucial component of the original algorithm, as such, a variety of test parameters were applied to Data Set 2 using the modified algorithm in order to determine the effects modifications of these parameters have on computation time and route length. Parameter changes were applied independently of each other, that is only one parameter was changed while the other remained at its default value. Each run represents the average values of five executions.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Run # | Iterations | Learning Rate | Calculation Time | Route Distance |
| 1 | 100,000 | 0.8 | 13.25 seconds | 689.3819 units |
| 2 | 90,000 | 0.8 | 13.75 seconds | 686.8465 units |
| 3 | 80,000 | 0.8 | 13.39 seconds | 689.0832 units |
| 4 | 70,000 | 0.8 | 13.27 seconds | 679.3534 units |
| 5 | 60,000 | 0.8 | 13.34 seconds | 680.2939 units |
| 6 | 50,000 | 0.8 | 13.44 seconds | 678.8881 units |
| 7 | 40,000 | 0.8 | 13.39 seconds | 684.3415 units |
| 8 | 30,000 | 0.8 | 13.31 seconds | 684.98 units |
| 9 | 20,000 | 0.8 | 13.34 seconds | 686.05 units |
| 10 | 10,000 | 0.8 | 13.43 seconds | 679.7005 units |
| 11 | 5,000 | 0.8 | 8.27 seconds | 684.9789 units |
| 12 | 2,500 | 0.8 | 4.06 seconds | 723.5293 units |
| 13 | 1,250 | 0.8 | 2.05 seconds | 768.0403 units |
| 14 | 625 | 0.8 | 1.06 seconds | 778.3201 units |
| 15 | 312 | 0.8 | 0.58 seconds | 769.1385 units |

Table 2: Effects of Iterations

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Run # | Iterations | Learning Rate | Calculation Time | Route Distance |
| 1 | 100,000 | 1.0 | 0.29 seconds | 819.9979 units |
| 2 | 100,000 | 0.9 | 13.19 seconds | 695.1229 units |
| 3 | 100,000 | 0.8 | 13.61 seconds | 689.6837 units |
| 4 | 100,000 | 0.7 | 13.45 seconds | 680.1638 units |
| 5 | 100,000 | 0.6 | 13.53 seconds | 696.9797 units |
| 6 | 100,000 | 0.5 | 13.79 seconds | 697.3463 units |
| 7 | 100,000 | 0.4 | 15.18 seconds | 684.4828 units |
| 8 | 100,000 | 0.3 | 14.66 seconds | 691.1515 units |
| 9 | 100,000 | 0.2 | 15.44 seconds | 698.5763 units |
| 10 | 100,000 | 0.1 | 19.79 seconds | 695.0364 units |

Table 3: Effects of Learning Rate

Tables 2 and 3 show the effect that the new early termination mechanism has upon parameter tuning. The number of iterations becomes nearly irrelevant to the outcome of both calculation time and route distance, only showing noticeable effects once the number of iterations falls below a threshold value likely determined by the number of waypoints present within a map. This demonstrates that it may in-fact be advantageous to change the iteration mechanism to either be infinite until the mean-error or learning-rate termination clauses execute or convert the iteration mechanism to represent a minimum iteration requirement. Future research would be prudent to determine which methodology would be better suited for this use-case. Table 3 illustrates a handful of interesting outcomes, particularly at learning rates 1.0, 0.4, 0.3, 0.2, and 0.1. The extremely fast calculation time and poor route distance calculation time at learning rate 1.0 is nearly equivalent to the execution of the algorithm at very few iterations. This is likely occurring due to the random distribution of points aligning to the data set well, resulting in a mean-error termination earlier than intended. Learning Rate values of 0.4, 0.3, 0.2, and 0.1, however, demonstrate that the learning rate is capable of effecting computation time of the algorithm typically requiring the algorithm to execute more iterations in order to compose a route distance of similar quality to that of higher rates. Using this analysis, it was noted that appropriate learning rates tend to be around 0.8 and 0.7. These two values were tested upon the test scenario maps, and the value 0.8 was selected. Additionally, due to the necessity that iterations remain high to combat the effect of insufficient iterations, the value also remained at its default of 100,000. Example outputs of the Algorithm on the projects test scenario maps are available in Figures 1, 2, and 3.

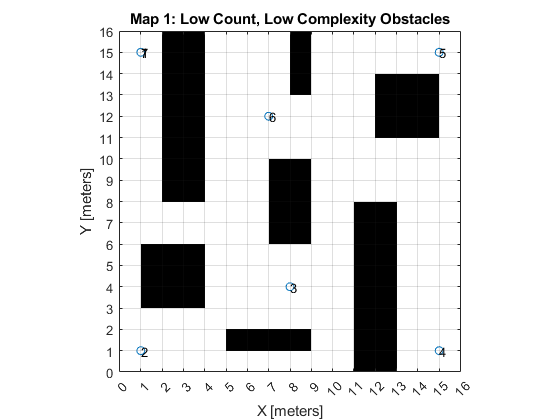


Figure 1: Map 1 Example TSP Solution

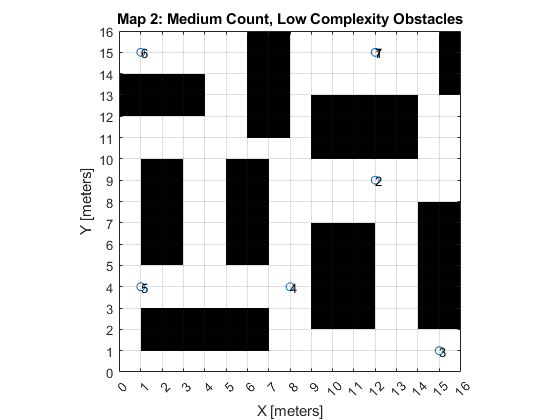


Figure 2: Map 2 Example TSP Solution

A screenshot of a computer

Description automatically generated with low confidence

Figure 3: Map 3 Example TSP Solution

# Point-to-Point Path Planning: Graft-RRT

Rapidly Exploring Random Trees (RRTs) are a well-researched methodology for the calculation of valid solutions in the motion planning task. Initially introduced by S. LaValle in [3], the core concept of RRT-based algorithms is to create a tree-like structure rooted at a starting point which rapidly expands by connecting randomly selected points from the environment’s free-space to the closest points within the tree, as long as the line connecting these points does not traverse the environment’s obstacle space. Given infinite iterations, this process creates a powerful path-planning algorithm that is able to produce optimal navigation solutions in complex environments. However, due to the random nature of the RRT the convergence-rate towards an optimal solution correlates with a trend towards infinite iterations of the algorithm. This has resulted in a variety of research efforts to advance the RRT algorithm in order to remove the convergence-rate problem. One such algorithm is the Graft-RRT produced by Z. Luo, Q. Li, J. Qiu, and C. Zhu in [4]. Proposed in February 2022, the Graft-RRT algorithm builds upon two foundational modifications of the RRT known to greatly increase the convergence rates: RRT\* and Bi-Directional RRT.

RRT\* improves upon the traditional RRT by implementing a system of re-parenting of the tree as the structure grows [5]. This process is typically referred to as re-wiring. Generally, in RRT\* algorithms the nearby neighbors of the new point are examined in attempt to identify points within the neighborhood where a connection would minimize the cost to the new point from the start point when travelling along the tree structure. Doing so minimizes the path taken within the final solution. In addition to this modification, the inverse operation also occurs where the neighboring points are also examined to determine if their connection to the new point would reduce their overall cost from the starting point.

Bi-Directional RRT is an umbrella name for many algorithms which introduce a dual-tree structure that grows simultaneously from the start point and a selected goal point. The Bi-Directional RRT algorithm used in the case of Graft-RRT, is B-RRT which is based off of another algorithm referred to as RRT-Connect [6]. RRT-Connect works by generating points in turn for each tree and connecting them to the respective target tree. Each point is then analyzed to determine if it is within some distance threshold to nodes of both trees. When this occurs, the trees are connected to form one uniform tree which is expanded in traditional RRT fashion. B-RRT builds upon this idea by integrating RRT\* into RRT-Connect.

Leveraging the optimality of B-RRT, Graft-RRT attempts to better exploit the phenomenon of multi-tree expansion by adding a third tree, the Graft Tree, within the free-space region between the start and goal trees in order to promote expansion and convergence in the direction of the goal region. The addition of this tertiary tree is also coupled with a compression and expansion stage which targets the simplification of path geometry resulting in additional optimization to the global path cost identified by the RRT stage of the algorithm. The implemented Graft-RRT algorithm is a modified version of a codebase found on GitHub [7]. While a valid implementation is provided by this code, the generation of suitable Graft-Tree root nodes in the pre-processing stage is non-deterministic and prone to obstacle occlusion, therefore, an alternative methodology for the identification of root nodes was created. This methodology is denoted in Algorithm 2.

Text

Description automatically generated

Algorithm 2: Random Center Selection

Generally, this algorithm identifies the points closest to the midpoint between the start and goal point that are not within the obstacle region of the environment. It then takes 25% of the points from the start to the midpoint and 25% of the points from the end to the midpoint. In both cases, the points composing the selected 25% are taken by their proximity to the original midpoint. After this process, a random point is selected from the combined pools (to avoid selection bias towards point in the start or end point) to act as the root node in the Graft Tree. This process is illustrated in Figure 4.

Chart, line chart

Description automatically generated

Figure 4: Graft Tree Root Node Selection

Algorithm 3, presented below, represents the algorithm as described by [4]. The modified “rand\_center” function described in Algorithm 2 replaces the GitHub implementation of the “GetTree” function presented within Algorithm 3.

Text, letter

Description automatically generated

Algorithm 3: Graft-RRT [4]

Additional modifications were also made to the algorithm to better support floating point inputs and outputs in order to enable higher resolution route solutions. It is important to remember that RRT algorithms require tuning on a per scenario basis. For this instance Graft-RRT supports the tuning of three parameters: epsilon (maximum travel distance allowed when connecting a node), threshold (how close to a target does a point need to be to consider it reached), and iterations (the number of attempts to connect additional points). These values should be tuned by the researcher as required for any additional maps added to the scenario set. Tables 4, 5, and 6 demonstrate the tuning of these parameters on test scenario maps 1, 2, and 3. Each row represents the mean of a data collection of five runs. Each parameter was tuned independently of other parameters in order to determine its effects on total path distance and total calculation time.

| Map # | Epsilon | Threshold | Iterations | Calculation Time | Path Cost |
| --- | --- | --- | --- | --- | --- |
| **1** | **1** | **0.5** | **10000** | **0.10966 seconds** | **74.4687 units** |
| **2** | **1** | **0.5** | **10000** | **0.11913 seconds** | **65.2126 units** |
| **3** | **1** | **0.5** | **10000** | **0.15792 seconds** | **73.1195 units** |
| 1 | 0.9 | 0.5 | 10000 | 0.1252 seconds | 75.2525 units |
| 2 | 0.9 | 0.5 | 10000 | 0.16047 seconds | 66.4005 units |
| 3 | 0.9 | 0.5 | 10000 | 0.22634 seconds | 75.7128 units |
| 1 | 0.8 | 0.5 | 10000 | 0.23065 seconds | 75.8059 units |
| 2 | 0.8 | 0.5 | 10000 | 0.23737 seconds | 68.2997 units |
| 3 | 0.8 | 0.5 | 10000 | 0.28348 seconds | 77.6648 units |
| 1 | 0.7 | 0.5 | 10000 | 0.27875 seconds | 75.4374 units |
| 2 | 0.7 | 0.5 | 10000 | 0.37712 seconds | 68.1328 units |
| 3 | 0.7 | 0.5 | 10000 | 0.43949 seconds | 73.3747 units |
| 1 | 0.6 | 0.5 | 10000 | 0.51724 seconds | 73.4757 units |
| 2 | 0.6 | 0.5 | 10000 | 0.57477 seconds | 67.4011 units |
| 3 | 0.6 | 0.5 | 10000 | 0.63619 seconds | 70.6152 units |
| 1 | 0.5 | 0.5 | 10000 | 1.7348 seconds | 75.5845 units |
| 2 | 0.5 | 0.5 | 10000 | 1.8122 seconds | 70.0957 units |
| 3 | 0.5 | 0.5 | 10000 | 2.2698 seconds | 74.552 units |

Table 4: Epsilon Tuning

| Map # | Epsilon | Threshold | Iterations | Calculation Time | Path Cost |
| --- | --- | --- | --- | --- | --- |
| 1 | 1 | 0.5 | 10000 | 0.11484 seconds | 74.3047 units |
| 2 | 1 | 0.5 | 10000 | 0.11694 seconds | 69.0216 units |
| **3** | **1** | **0.5** | **10000** | **0.15493 seconds** | **76.0375 units** |
| **1** | **1** | **0.5** | **15000** | **0.10403 seconds** | **73.228 units** |
| 2 | 1 | 0.5 | 15000 | 0.12312 seconds | 69.0272 units |
| 3 | 1 | 0.5 | 15000 | 0.16136 seconds | 79.7237 units |
| 1 | 1 | 0.5 | 20000 | 0.1143 seconds | 73.4964 units |
| **2** | **1** | **0.5** | **20000** | **0.12306 seconds** | **66.4089 units** |
| 3 | 1 | 0.5 | 20000 | 0.17578 seconds | 77.4425 units |
| 1 | 1 | 0.5 | 2500 | 0.10592 seconds | 74.9157 units |
| 2 | 1 | 0.5 | 2500 | 0.1538 seconds | 71.9022 units |
| 3 | 1 | 0.5 | 2500 | 0.19847 seconds | 76.8308 units |
| 1 | 1 | 0.5 | 5000 | 0.10584 seconds | 75.3541 units |
| 2 | 1 | 0.5 | 5000 | 0.11668 seconds | 67.8624 units |
| 3 | 1 | 0.5 | 5000 | 0.16529 seconds | 79.6429 units |
| 1 | 1 | 0.5 | 7500 | 0.12859 seconds | 75.2374 units |
| 2 | 1 | 0.5 | 7500 | 0.12812 seconds | 72.9277 units |
| 3 | 1 | 0.5 | 7500 | 0.17285 seconds | 74.2926 units |

Table 5: Iteration Tuning

| Map # | Epsilon | Threshold | Iterations | Calculation Time | Path Cost |
| --- | --- | --- | --- | --- | --- |
| 1 | 1 | 0.5 | 10000 | 0.10575 seconds | 76.6066 units |
| 2 | 1 | 0.5 | 10000 | 0.1153 seconds | 67.5758 units |
| **3** | **1** | **0.5** | **10000** | **0.16259 seconds** | **74.7469 units** |
| 1 | 1 | 0.6 | 10000 | 0.13275 seconds | 76.9053 units |
| 2 | 1 | 0.6 | 10000 | 0.11813 seconds | 67.5556 units |
| 3 | 1 | 0.6 | 10000 | 0.19526 seconds | 80.1852 units |
| 1 | 1 | 0.7 | 10000 | 0.11686 seconds | 75.2364 units |
| 2 | 1 | 0.7 | 10000 | 0.15219 seconds | 68.5781 units |
| 3 | 1 | 0.7 | 10000 | 0.19922 seconds | 76.0459 units |
| 1 | 1 | 0.8 | 10000 | 0.14531 seconds | 76.4877 units |
| 2 | 1 | 0.8 | 10000 | 0.14884 seconds | 72.5529 units |
| 3 | 1 | 0.8 | 10000 | 0.24326 seconds | 82.0528 units |
| 1 | 1 | 0.9 | 10000 | 0.14173 seconds | 76.5527 units |
| 2 | 1 | 0.9 | 10000 | 0.16964 seconds | 68.2774 units |
| 3 | 1 | 0.9 | 10000 | 0.24073 seconds | 85.0704 units |
| 1 | 1 | 1.0 | 10000 | 0.284 seconds | 72.5577 units |
| 2 | 1 | 1.0 | 10000 | 0.34606 seconds | 68.7201 units |
| 3 | 1 | 1.0 | 10000 | 0.43871 seconds | 76.9608 units |
| 1 | 1 | 0.4 | 10000 | 0.13432 seconds | 75.5864 units |
| 2 | 1 | 0.4 | 10000 | 0.10822 seconds | 70.719 units |
| 3 | 1 | 0.4 | 10000 | 0.71639 seconds | 77.3984 units |
| **1** | **1** | **0.3** | **10000** | **0.12539 seconds** | **75.6978 units** |
| **2** | **1** | **0.3** | **10000** | **0.12782 seconds** | **67.1574 units** |
| 3 | 1 | 0.3 | 10000 | UNSTABLE | UNSTABLE |
| 1 | 1 | 0.2 | 10000 | 0.1211 seconds | 75.475 units |
| 2 | 1 | 0.2 | 10000 | 0.11848 seconds | 71.011 units |
| 3 | 1 | 0.2 | 10000 | 0.1617 seconds | 83.5143 units |
| 1 | 1 | 0.1 | 10000 | 0.13939 seconds | 75.1707 units |
| 2 | 1 | 0.1 | 10000 | 0.15661 seconds | 67.7729 units |
| 3 | 1 | 0.1 | 10000 | 0.19201 seconds | 83.9193 units |

Table 6: Threshold Tuning

Tuning of the parameters demonstrates that the step size, epsilon, increased the computation time while producing no desirable improvements to the overall route distance. This resulted in the maximum allowed step size for all maps being set to one grid unit. In testing iterations, it was shown that iterations generally do not result in increased computation time. In terms of iterations, it was denoted that lowering of the number of allowed iterations resulted in increased computation time in some instances. These were determined to be within the margin of error for the algorithm. As such, the iteration times which produced the lowest route distance were selected for usage. Threshold, while a tunable variable, is likely not one which should be modified frequently. This determines the connection range of the RRT to the goal and is often tied to a reasonable value considering all parameter selections. In this case, the values determined in the tuning process were considered reasonable given the selected values and were accepted as the tuned values for the map. Example execution results of the Graft RRT algorithm with their determined optimal parameters on the test scenario maps are available in Figures 5, 6, and 7.

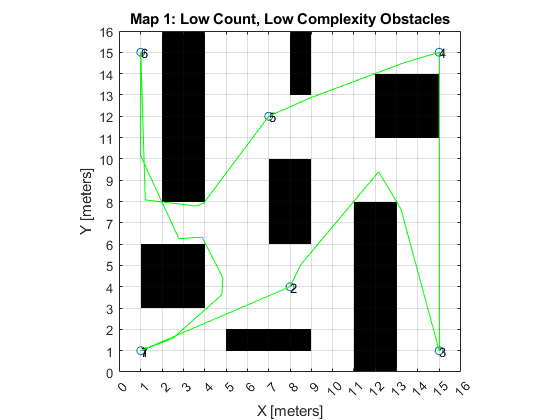


Figure 5: Map 1 Graft-RRT Solution

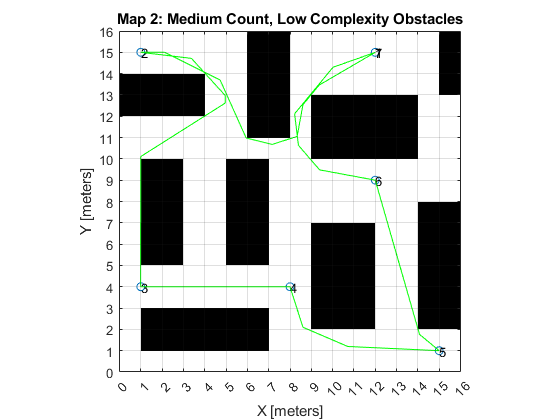


Figure 6: Map 2 Graft-RRT Solution

A picture containing text, crossword puzzle

Description automatically generated

Figure 7: Map 3 Graft-RRT Solution

# Local Navigation: VFH+

VFHs (Vector Field Histograms) are a method of real-time local navigation initially proposed by J. Borestein and Y. Koren, in which the environment is represented by a histogram grid (occupancy grid in this case) [8]. This grid is measured using ranging data from a variety of sensors. This data can then be placed into polar histogram bins based off where they fall within a polar coordinate system determined by the robot’s center. This allows the creation of a polar histogram which contains a statistical representation of the obstacles near the robot. Using the polar histogram, the robot can then determine open sectors of space that it can travel through and calculate a steering direction. This direction can then be biased, to a degree, towards a particular navigation target. VFH+ is a modification of this original algorithm proposed by I. Ulrich and J. Bornstein which improves upon the standard VFH in terms of its ability to produce smooth robot trajectories, its ability to handle highly volumes of clustered obstacles, and overall improves the algorithm’s reliability [9]. The first significant change to the VFH algorithm presented by VFH+ is the inclusion of a feature extraction phase which clusters and weights returned points by feature sets (obstacles) allowing for higher resolution depictions of obstacles within the environment. In the traditional VFH, the raw sensor return data is used in the construction of the polar histogram. The VFH+ algorithm also includes obstacle orientation in its consideration of the binning process, allowing for better obstacle discrimination. It also accounts for the velocity and orientation of the robot itself in order to make predictive adjustments to its steering direction to prevent collisions and maximize pathing efficiency. The VFH+ algorithm was implemented using a combination of the Mobile Robotic Simulation Toolkit (MRS) and the MATLAB Navigation Toolkits [10, 11]. The MRS Toolkit is used to allow for the simulation of LiDAR data on the occupancy grids created by the scenario test maps, while the MATLAB Robotics Toolkits provides native implementations for a waypoint follow controller and VFH+ controller which can work in unison to avoid obstacles while pathing towards a particular navigation point. Using these three components together, a simulated robot can be created and placed within our test scenario maps. Importantly, there are a handful of parameters which require the attention of researchers implementing these tools. In the MRS Toolbox, the “LidarSensor” object contains three important parameters: Sensor Offset, Scan Angles, and Max Range that are subject to tuning. In the case of this works test scenarios, Sensor Offset was set [0, 0] and not considered in the tuning process. Scan Angles represent the HFOV (Horizontal Field of View) of the LiDAR sensor, and in this case was set to be 360 degrees. Finally, Max Range represents how far the LiDAR should return point data. Due to the standardized map size of 16 units, it was determined a Max Range of 2 was reasonable. The MATLAB “controllerPurePursuit” object provides three adjustable parameters: LookaheadDistance, DesiredLinearVelocity, and MaxAngularVelocity. Due to the nature of VFH, the tuning process for these parameters for one map (Map 3) is indicated in Tables 7, 8, and 9 and is based off qualitative observations of the outputs. Importantly, each parameter was tuned using the previously determined best setting for the other parameters or an unchanging parameter value, this allowed for the appropriate tuning to carry throughout testing, minimizing sudden changes in performance.

| Run # | Look Ahead | Linear Velocity | Angular Velocity | Analysis |
| --- | --- | --- | --- | --- |
| 1 | 0.10 | 0.20 | 0.5 | Oscillations noted in trajectory when in between two obstacles. Collisions occurred with obstacles. | |
| 2 | 0.20 | 0.20 | 0.5 | Some oscillations noted, some struggling with cornering of obstacles. | |
| 3 | 0.30 | 0.20 | 0.5 | Good cornering of obstacles, minor oscillations noted | |
| **4** | **0.40** | **0.20** | **0.5** | **Great cornering of obstacles, little to no oscillations noted** | |
| 5 | 0.50 | 0.20 | 0.5 | No better/worse than 0.40 | |

Table 7: Look Ahead Tuning

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Run # | Look Ahead | Linear Velocity | Angular Velocity | Analysis |
| 1 | 0.40 | 0.10 | 0.5 | Good pathing, however, the simulation runs too unreasonable slow given the scale. | |
| **2** | **0.40** | **0.20** | **0.5** | **Good pathing** | |
| 3 | 0.40 | 0.30 | 0.5 | Collision with obstacles occurs on tight cornering | |
| 4 | 0.40 | 0.40 | 0.5 | Collision with obstacles occurs on most turns | |
| 5 | 0.40 | 0.50 | 0.5 | Fails to adhere to map bounds on tight turns, collides with obstacles | |

Table 8: Linear Velocity Tuning

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Run # | Look Ahead | Linear Velocity | Angular Velocity | Analysis |
| 1 | 0.40 | 0.20 | 0.5 | Good Pathing | |
| 0.2 | 0.40 | 0.20 | 0.6 | Good Pathing, one instance of double looping to make a single point turn | |
| 3 | 0.40 | 0.20 | 0.7 | Good pathing, nice tight transitions | |
| 4 | 0.40 | 0.20 | 0.8 | Very tight and good RRT line following, however, some of the transitions in 0.7 were more desirable | |
| **5** | **0.40** | **0.20** | **0.9** | **Generally, the best pathing, allowed for the best matching of designated RRT path without collisions** | |

Table 9: Angular Velocity Tuning

The final tunable portion of the Local Navigation system is the MATLAB “controllerVFH” object, which contains parameter options for the DistanceLimits, NumAngularSectors, HistogramThresholds, RobotRadius, and SafetyDistance. The DistanceLimits parameter controls the range readings considered from the LiDAR. This is represented as an array similar to [0 10] where 0 represents the lower bound and 10 represents the upper bound. In this case, we have locked the lower bound to 0.05 and will only work to tune the upper bound. The number of angular sectors to use in the binning process is controlled by NumAngularSectors. HistogramThresholds controls how dense a particular area needs to be considered occupied or free space, this is done with an array similar to [3 10] where 3 is the lower bound (free space) and 10 is the upper bound (occupied space). Densities which fall within this range take on the occupied value of the previous histogram. In our case, we have locked this value to [3 8]. Robot Radius and Safety Distance control how wide a particular section of free space needs to be in order to be considered navigable and have a scalar value. In this instance Robot Radius has been locked to 0.1 and only safety distance is tuned. These parameters went through a tuning process equivalent to the previously discussed controller parameters and observations are denoted in Tables 10-14.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Run # | Upper Distance Limit | Number of Sectors | Safety Distance | Analysis |
| 1 | 2 | 180 | 0.1 | Collision with obstacles occurred. |
| **2** | **1** | **180** | **0.1** | **Good Pathing, few deviations from intended path** |
| 3 | 0.75 | 180 | 0.1 | Good Pathing, even less unnecessary deviations from intended path |
| 4 | 0.50 | 180 | 0.1 | Good pathing, similar trend in better deviation tendency. However, questionable ability to account for obstacles given look ahead and velocity. |
| 5 | 0.25 | 180 | 0.1 | Collision with obstacles occurred. |

Table 10: Upper Distance Limit Tuning

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Run # | Upper Distance Limit | Number of Sectors | Safety Distance | Analysis |
| 1 | 1.0 | 180 | 0.1 | Good Pathing |
| 2 | 1.0 | 210 | 0.1 | Good Pathing, little to no unnecessary deviations from path |
| 3 | 1.0 | 280 | 0.1 | Some increased oscillations around certain obstacles, however, path deviations are now practically minimal |
| 4 | 1.0 | 310 | 0.1 | Decreased oscillations, even less path deviation from 280. |
| **5** | **1.0** | **360** | **0.1** | **No unnecessary deviations and minimal oscillations** |

Table 11: Number of Sectors Tuning

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Run # | Upper Distance Limit | Number of Sectors | Safety Distance | Analysis |
| 1 | 1.0 | 360 | 0.1 | Good Pathing, however, gets unreasonably close to obstacles in some instances |
| **2** | **1.0** | **360** | **0.2** | **Good pathing, reasonable far away from obstacles at all times, however, some instances of over and under corrections were present** |
| 3 | 1.0 | 360 | 0.3 | Multiple occasions of over correction and under correction, some resulting in near misses with obstacles |
| 4 | 1.0 | 360 | 0.4 | Maintaining safety distance at 0.4 allows for smooth turns into prime transition positions for Map 3. This may not be an ideal general value though. |
| 5 | 1.0 | 360 | 0.5 | Overcorrections again present, resulting in sub-optimal values. Additionally, this value may not be ideal for all maps. |

Table 12: Safety Distance Tuning

Example outputs of the tuned VFH are provided for the test scenario maps in Figures 8, 9, and 10. Researchers may desire to tweak the determined values as they see fit based off the data recorded in Tables 10-12.

Chart, line chart

Description automatically generated

Figure 8: Map 1 VFH Solution

A picture containing text, crossword puzzle

Description automatically generated

Figure 9: Map 2 VFH Solution

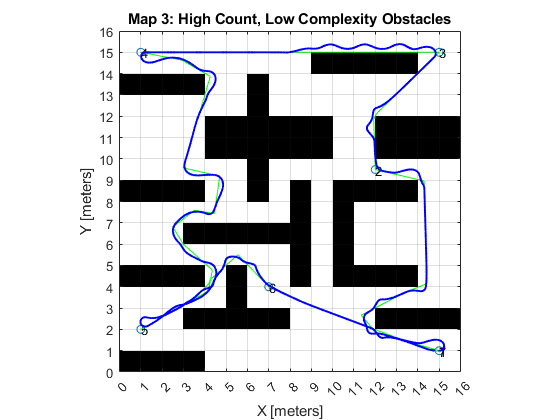


Figure 10: Map 3 VFH Solution

# Digital Twin Implementation

Implementations of Digital Twin technologies are unique to the researcher’s desires, requirements, scenarios, and a multitude of various factors that the researcher needs to consider. In this work, we are considering and simulating the use of a UAV and UGV vehicle in a disaster relief scenario. It was considered that emergency response officials may desire to instigate a sudden change to waypoint order provided by the TSP as to respond to the dynamics of a disaster such as wildfires or flooding. Situations such as these are considered as branch scenarios by this work. Branch Scenarios involve any change to the environment in which the Digital Twin should simulate a potential response to the change, these branch decisions will typically be represented in a new figure with the response visualized as dashed lines. In some other scenarios, it may be desired to reverse this and have the robot, not the Digital Twin, implement these branch decisions while the Digital Twin continues the previously defined path. This configuration is not considered in this work, however, one may simply view the provided Digital Twin maps as the real robot’s new path. An additional branch decision scenario considered by the Digital Twin implementation is hidden obstacle response, which is a sudden obstacle that is “found” post path-planning which interferes with the intended path. These Digital Twin scenarios are implemented in a pre-processing manner; however, modifications to the pathing and routing details occur as though the transitions were found during simulation execution. This Digital Twin system allows researchers to visualize and understand the effects that potential changes will have within the system on a purely qualitative and visual level since distance/pathing changes and optimality of those changes is not considered by this research.

## Priority Waypoint Transition

The Priority Waypoint Transition (PWT) handler works as a flag enabled system that passes in transition information to the Priority Override function. This function is provided the transition waypoint, the priority waypoint, current optimal route, all map waypoints, and the map size in order to rapidly recompute the TSP problem for the modified circumstance. This function removes the waypoints which have already been travelled to in reaching the transition waypoint from consideration in the TSP process. It then places them in a new route called the priority route, which initially contains all points up to the transition way point with the priority waypoint next in line. After this stage, the SOM algorithm is re-executed on the priority waypoint plus the remaining waypoints which have not been travelled to. In this way the system optimizes for the remaining transition points. It then stitches the SOM computed route with the priority route in order to provide a uniform priority route. This, in effect, provides a semi-optimal waypoint transition to the priority waypoint while completing the remaining components of the route. This process only occurs when the transition point selected occurs before the priority waypoint, if this does not occur no Digital Twin simulation will be performed as no branch decision occurs. Figure 11 demonstrates the PWT process on an example input, example outputs of this system are visualized in Figures 12, 13, and 14.

Text

Description automatically generated

Figure 11: PWT Process Numerical Example

A picture containing text, crossword puzzle

Description automatically generatedA diagram of a house

Description automatically generated with low confidence

Figure 12: Map 1 – Full Route Override Example

A picture containing text, crossword puzzle

Description automatically generatedA picture containing graphical user interface

Description automatically generated

Figure 13: Map 2 – Partial Route Transition Example

Shape, arrow

Description automatically generatedA picture containing text, crossword puzzle

Description automatically generated

Figure 14: Map 3 – Partial Route Transition Example

## Hidden Obstacle Transition

Hidden Obstacle Transitions (HOTs) are implemented in similar fashion to the PWT handler, a flag enables a system to execute the creation of an additional obstacle defined by the researcher as “hiddenObstaclePolygon”. This polygon is used to generate a secondary occupancy grid which considers both this new obstacle and the previous map obstacles. This occupancy grid is then used within the RRT-process in a live-like fashion that initially creates a path considering only the original obstacle map, however, in the event of a path collisions with the new obstacle, it is recomputed with the newly created occupancy grid. This allows the Digital Twin to create a branch decision based off the effects the obstacle would have on the traditional path. In the visualization, path changes are visualized as a solid red line in the original path with a dotted green line representing the branch decision path. A basic diagram demonstrating this technique is shown Figure 15, and example outputs of this system are provided in Figures 16, 17, 18, 19, 20, and 21.

Diagram

Description automatically generated

Figure 15: Hidden Obstacle RRT-Process Example

A picture containing text, crossword puzzle

Description automatically generated

Figure 16: Map 1 – Original Path

A picture containing text, crossword puzzle

Description automatically generated

Figure 17: Map 1 – Digital Twin Path

A picture containing text, crossword puzzle

Description automatically generated

Figure 18: Map 2 – Original Path

A picture containing text, crossword puzzle

Description automatically generated

Figure 19: Map 2 – Digital Twin Path

A picture containing text, crossword puzzle

Description automatically generated

Figure 20: Map 3 – Original Path

A picture containing text, crossword puzzle

Description automatically generated

Figure 21: Map 3 – Digital Twin Path

# System Flow Overview

The Global Path Planning, Point-to-Point Path Planning, Local Navigation, and Digital Twin systems have now been described in their implementation circumstances. It is important for the researcher to understand how these systems are interconnected to fully understand their results. First a map is selected within MATLAB by the researcher, these are typically stored within the “Maps” directory and have simple names such as “Map1.m”, “Map2.m”, or “Map3.m”. These maps define the various properties of the map such as obstacles (not including hidden obstacles), waypoints (not including priority transitions), and the map title and size desired. The selected map is then called in the main execution file “Main.m” in via a command such as run(“Maps/Map1.m”). This is done to propagate map information into the workspace. This map data is then passed to Stage 1 of the simulation, Global Path Planning. This executes the SOM algorithm on the map’s waypoints producing an optimal route. This information is then displayed on the map figure produced by running the map. After this stage, the first Digital Twin, PWT, is checked for its execution flag. If this flag is set to true, then the PWT executes resulting in the creation of both a priority route and a new figure (assuming the transition is valid and should occur). This is performed in this manner to isolate the two systems as much as theoretically possible. Immediately following the PWT system, the HOT flag is checked. If this flag is true, then an additional occupancy map is created which considers the hidden obstacle polygon encoded by the researcher in combination with the map obstacles and an additional map figure is created. It should be noted that this simulation system only supports the execution of one Digital Twin modality at a time and Obstacle Overrides will be disabled if PWT is enabled. The next system to execute is the Point-to-Point Path Planner, Graft-RRT. This system’s responses deviate based off the flag states of the PWT and HOT. In the event either PWT or HOT is enabled the creation of two semi-unique solutions occurs and are combined later for a final solution route in the navigation phase. These routes are then plotted on their respective figures such that the researcher can see the intended execution routes. Finally, the Local Navigation controller, VFH+, handles routing along the originally planned, PWT route, or HOT route as needed. In the HOT route, the original path is taken unless the path deviates into a recalculated segment. This code execution flow is shown in the flowchart provided in Figure 22.

Diagram

Description automatically generated

Figure 22: Code Flow Diagram

# Results

The HOT and PWT mechanism were tested on three separate test scenario maps. These maps in their unexecuted form are available for review in Appendices 3-5. Each Map’s results are shown and discussed in order to determine the significance and variety of the Digital Twin’s results. In Figures 23 and 24, we can see and verify that the Digital Twin appropriately handles the priority waypoint transition and executes a different path plan once the transition is intended to occur. In the case of the hidden obstacle transition (HOT) found in Figures 25 and 26, both the original path and digital twin path, even though the digital twin recomputed a more optimal path, remain similar and that the VFH could handle the potential collision optimally.

Chart

Description automatically generated

Figure 23: Map 1 (PWT) – Original Path

Chart, line chart

Description automatically generated

Figure 24: Map 1 (PWT) – Digital Twin Path (Waypoint 3 to Waypoint 6)

Chart, line chart

Description automatically generated

Figure 25: Map 1 (HOT) – Original Path

Chart, line chart

Description automatically generated

Figure 26: Map 1 (HOT) – Digital Twin Path

Figures 27 and 28, demonstrate that an equivalent waypoint transition as that found in Map 1 can be implemented and executed as intended on Map 2. It also shows the negative effects a large-scale re-prioritization can have on the path, greatly decreasing the efficiency of the pathing for the Digital Twin. Figures 29 and 30, show the ability of the Digital Twin to properly replan and avoid situations that the VFH+ in the original path was not capable of escaping. This demonstrates the importance of using Digital Twin technologies to investigate alternative path solutions.

Graphical user interface

Description automatically generated

Figure 27: Map 2 (PWT) – Original Path

Graphical user interface

Description automatically generated with low confidence

Figure 28: Map 2 (PWT) – Digital Twin Path (Waypoint 2 to Waypoint 5)

A picture containing text, crossword puzzle

Description automatically generated

Figure 29: Map 2 (HOT) – Original Path

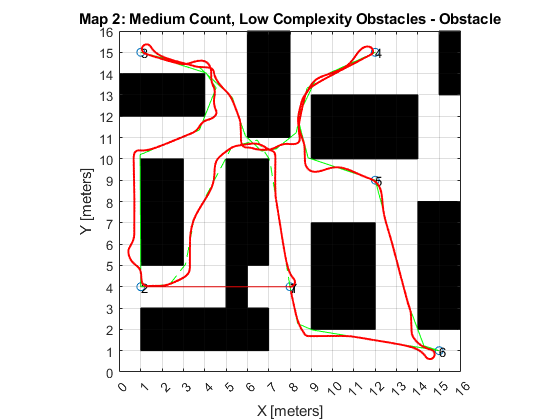


Figure 30: Map 2 (HOT) – Digital Twin Path

In Map 3, Figures 31 and 32, again demonstrate, the later in the route the priority waypoint is intended to occur, the ability of the Digital Twin to account for waypoint transition occurs at the cost of path optimality. This allows for some semi-efficiency to be noted in the left-side of the map with a break-down in optimality on the right-hand side. Figures 33 and 34 demonstrate the intended functionality of the Digital Twin technology, in the computation of the new path no branching decision was required to handle the new obstacle. As such, the VFH mirrors (nearly exactly) the original path discovered by the initial execution.

A picture containing diagram

Description automatically generated

Figure 31: Map 3 (PWT) – Original Path

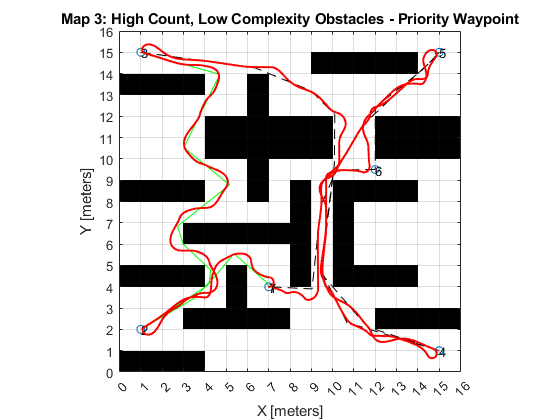


Figure 32: Map 3 (PWT) – Digital Twin Path (Waypoint 3 to Waypoint 6)

A picture containing graphical user interface

Description automatically generated

Figure 33: Map 3 (HOT) – Original Path

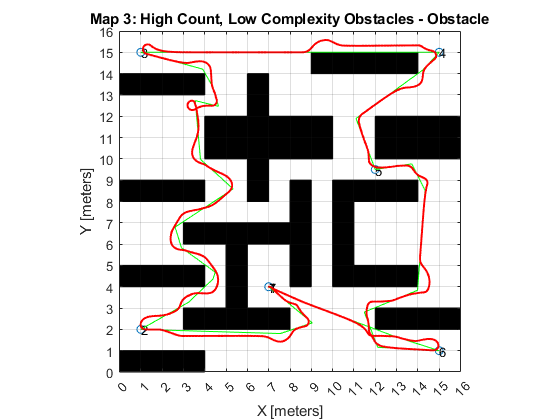


Figure 34: Map 4 (HOT) – Digital Twin Path

# Conclusions and Future Work

This work demonstrates the effective implementation and integration of a functional robot simulation stack for a UAV/UGV system with a secondary Digital Twin simulation for the purposes of natural disaster relief. This allowed for a thorough visualization and examination into the usefulness and capabilities of Digital Twin simulations in determining effectiveness and performance of the simulated UGV system on path branch decisions which may affect the robot’s path in real-world scenarios. It was shown, that while effective in making and executing these decisions, they make be deficient in proper optimality. In future work, a goal should be to leverage the Digital Twin simulation to automatically transition its path plan to the real robot on successful execution, thereby, minimizing computation cycles and path planning requirements to overcome pit falls such as the stuck robot scenario discovered on Map 2. Additionally, it was noted that not all simulations will require Digital Twin intervention such as those scenarios where transitions have already occurred or are otherwise already accounted for. In these events, future work should explore ways to execute additional optimization algorithms through Digital Twin to cohere to a better end path. As a final point for future work, it is desirable to implement these systems on a real-world test bench so that a better understanding of the downsides and upsides of the simulation in its current state can be acquired and to also expand upon its functionalities such that it can be more applicable and useful in real-world scenarios.

# References

[1] Kohonen, T. Self-organized formation of topologically correct feature maps. Biol. Cybern. 43, 59–69 (1982). <https://doi.org/10.1007/BF00337288>

[2] D. Vicente, “Using Self-Organizing Maps to solve the Traveling Salesman Problem,” *Diego Codes*, Feb. 01, 2018. https://diego.codes/post/som-tsp/ (accessed May 10, 2023).

‌[3] S. LaValle, “Rapidly-Exploring Random Trees: A New Tool for Path Planning,” Iowa State University, Oct. 1998. Accessed: May 10, 2023. [Online]. Available: <http://msl.cs.uiuc.edu/~lavalle/papers/Lav98c.pdf>

[4] Z. Luo, Q. Li, J. Qiu and C. Zhu, "Graft-RRT: An efficient method to find and converge global paths through intermediate points," 2022 International Conference on Machine Learning and Knowledge Engineering (MLKE), Guilin, China, 2022, pp. 256-260, doi: 10.1109/MLKE55170.2022.00056.

[5] S. Karaman and E. Frazzoli, ‘Incremental Sampling-based Algorithms for Optimal Motion Planning’, arXiv [cs.RO]. 2010.

‌[6] J. J. Kuffner and S. M. LaValle, "RRT-connect: An efficient approach to single-query path planning," Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No.00CH37065), San Francisco, CA, USA, 2000, pp. 995-1001 vol.2, doi: 10.1109/ROBOT.2000.844730.

[7] luoer, “global-plan,” *GitHub*, Sep. 21, 2022. <https://github.com/robot-rolle/path-plan>

[8] J. Borenstein and Y. Koren, "The vector field histogram-fast obstacle avoidance for mobile robots," in IEEE Transactions on Robotics and Automation, vol. 7, no. 3, pp. 278-288, June 1991, doi: 10.1109/70.88137.

[9] I. Ulrich and J. Borenstein, "VFH+: reliable obstacle avoidance for fast mobile robots," Proceedings. 1998 IEEE International Conference on Robotics and Automation (Cat. No.98CH36146), Leuven, Belgium, 1998, pp. 1572-1577 vol.2, doi: 10.1109/ROBOT.1998.677362.

[10] “Mobile Robotics Simulation Toolbox,” www.mathworks.com. https://www.mathworks.com/matlabcentral/fileexchange/66586-mobile-robotics-simulation-toolbox

[11] “Navigation Toolbox,” *www.mathworks.com*. <https://www.mathworks.com/products/navigation.html>

# Appendices

## Appendix 1 – TSP Dataset 1

1 170.0 85.0

2 166.0 88.0

3 133.0 73.0

4 140.0 70.0

5 142.0 55.0

6 126.0 53.0

7 125.0 60.0

8 119.0 68.0

9 117.0 74.0

10 99.0 83.0

11 73.0 79.0

12 72.0 91.0

13 37.0 94.0

14 6.0 106.0

15 3.0 97.0

16 21.0 82.0

17 33.0 67.0

18 4.0 66.0

19 3.0 42.0

20 27.0 33.0

21 52.0 41.0

22 57.0 59.0

23 58.0 66.0

24 88.0 65.0

25 99.0 67.0

26 95.0 55.0

27 89.0 55.0

28 83.0 38.0

29 85.0 25.0

30 104.0 35.0

31 112.0 37.0

32 112.0 24.0

33 113.0 13.0

34 125.0 30.0

35 135.0 32.0

36 147.0 18.0

37 147.5 36.0

38 154.5 45.0

39 157.0 54.0

40 158.0 61.0

41 172.0 82.0

42 174.0 87.0

## Appendix 2 – TSP Dataset 2

1 155.42 150.65

2 375.92 164.65

3 183.92 150.65

4 205.42 150.65

5 205.42 171.65

6 226.42 171.65

7 226.42 186.15

8 226.42 207.15

9 226.42 235.65

10 226.42 264.15

11 226.42 292.65

12 226.42 314.15

13 226.42 335.65

14 205.42 335.65

15 190.92 335.65

16 190.92 328.15

17 176.92 328.15

18 176.92 299.65

19 155.42 299.65

20 155.42 328.15

21 155.42 356.65

22 183.92 356.65

23 219.42 356.65

24 240.92 356.65

25 269.42 356.65

26 290.42 356.65

27 387.42 136.15

28 318.92 356.65

29 318.92 335.65

30 318.92 328.15

31 318.92 299.65

32 297.92 299.65

33 290.42 328.15

34 290.42 335.65

35 297.92 328.15

36 254.92 335.65

37 254.92 314.15

38 254.92 292.65

39 254.92 271.65

40 254.92 243.15

41 254.92 221.65

42 254.92 193.15

43 254.92 171.65

44 276.42 171.65

45 296.42 150.65

46 276.42 150.65

47 375.92 150.65

48 308.92 150.65

49 354.92 164.65

50 338.42 174.65

51 354.92 174.65

52 338.42 200.15

53 338.42 221.65

54 354.92 221.65

55 354.92 200.15

56 361.92 200.15

57 361.92 186.15

58 383.42 186.15

59 383.42 179.15

60 404.42 179.15

61 404.42 186.15

62 418.92 186.15

63 418.92 200.15

64 432.92 200.15

65 432.92 221.65

66 418.92 221.65

67 418.92 235.65

68 397.42 235.65

69 397.42 243.15

70 375.92 243.15

71 375.92 257.15

72 368.92 257.15

73 368.92 264.15

74 347.42 264.15

75 347.42 278.65

76 336.42 278.65

77 336.42 328.15

78 347.42 328.15

79 347.42 342.65

80 368.92 342.65

81 368.92 353.65

82 418.92 353.65

83 418.92 342.65

84 432.92 342.65

85 432.92 356.65

86 447.42 356.65

87 447.42 321.15

88 447.42 292.65

89 432.92 292.65

90 432.92 314.15

91 418.92 314.15

92 418.92 321.15

93 397.42 321.15

94 397.42 333.65

95 375.92 333.65

96 375.92 321.15

97 361.92 321.15

98 361.92 299.65

99 375.92 299.65

100 375.92 285.65

101 397.42 285.65

102 397.42 271.65

103 418.92 271.65

104 418.92 264.15

105 439.92 264.15

106 439.92 250.15

107 454.42 250.15

108 454.42 243.15

109 461.42 243.15

110 461.42 214.65

111 461.42 193.15

112 447.42 193.15

113 447.42 179.15

114 439.92 179.15

115 439.92 167.65

116 419.92 167.65

117 419.92 150.65

118 439.92 150.65

119 454.42 150.65

120 475.92 150.65

121 475.92 171.65

122 496.92 171.65

123 496.92 193.15

124 496.92 214.65

125 496.92 243.15

126 496.92 271.65

127 496.92 292.65

128 496.92 317.15

129 496.92 335.65

130 470.42 335.65

131 470.42 356.65

132 496.92 356.65

133 347.42 150.65

134 539.92 356.65

135 560.92 356.65

136 589.42 356.65

137 589.42 342.65

138 603.92 342.65

139 610.92 342.65

140 610.92 335.65

141 610.92 321.15

142 624.92 321.15

143 624.92 278.65

144 610.92 278.65

145 610.92 257.15

146 589.42 257.15

147 589.42 250.15

148 575.42 250.15

149 560.92 250.15

150 542.92 250.15

151 542.92 264.15

152 560.92 264.15

153 575.42 264.15

154 575.42 271.65

155 582.42 271.65

156 582.42 285.65

157 596.42 285.65

158 560.92 335.65

159 596.42 314.15

160 582.42 314.15

161 582.42 321.15

162 575.42 321.15

163 575.42 335.65

164 525.42 335.65

165 525.42 314.15

166 525.42 299.65

167 525.42 281.65

168 525.42 233.15

169 525.42 214.65

170 525.42 193.15

171 525.42 171.65

172 546.92 171.65

173 546.92 150.65

174 568.42 150.65

175 475.92 160.65

176 603.92 150.65

177 624.92 150.65

178 624.92 136.15

179 596.42 136.15

180 575.42 136.15

181 553.92 136.15

182 532.42 136.15

183 575.42 356.65

184 489.92 136.15

185 468.42 136.15

186 447.42 136.15

187 425.92 136.15

188 404.42 136.15

189 370.42 136.15

190 361.92 150.65

191 340.42 136.15

192 326.42 136.15

193 301.92 136.15

194 276.42 136.15

195 254.92 136.15

196 315.92 136.15

197 212.42 136.15

198 190.92 136.15

199 338.92 150.65

200 155.42 136.15

201 624.92 299.65

202 318.92 321.65

203 155.42 314.15

204 311.92 356.65

205 355.42 136.15

206 318.92 314.15

207 362.92 164.65

208 254.92 356.65

209 383.42 333.65

210 447.42 335.65

211 470.42 345.65

212 525.42 250.15

213 546.92 335.65

214 525.42 261.15

215 525.42 356.65

216 336.42 298.65

217 336.42 313.15

218 293.42 136.15

219 336.42 306.15

220 425.92 264.15

221 391.42 353.65

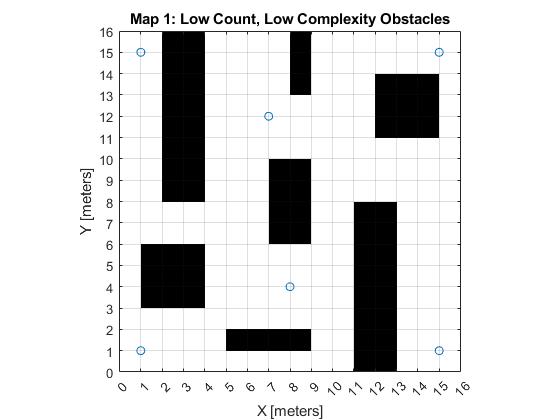
222 482.92 335.65

223 429.92 167.65

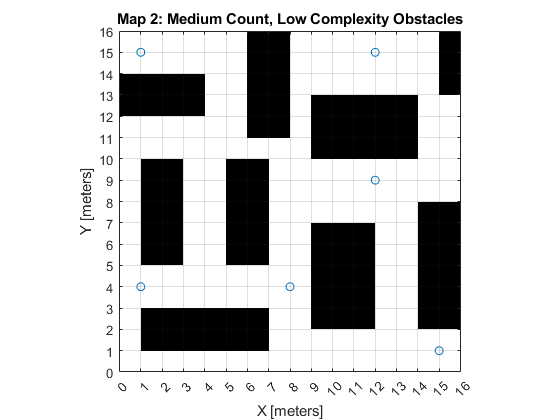
224 330.92 150.65

225 368.42 150.65

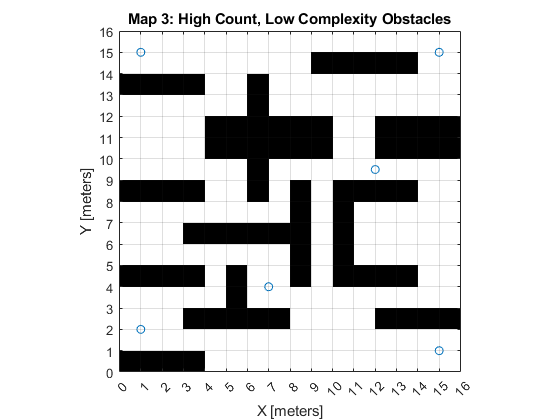
## Appendix 3 – Map 1

****

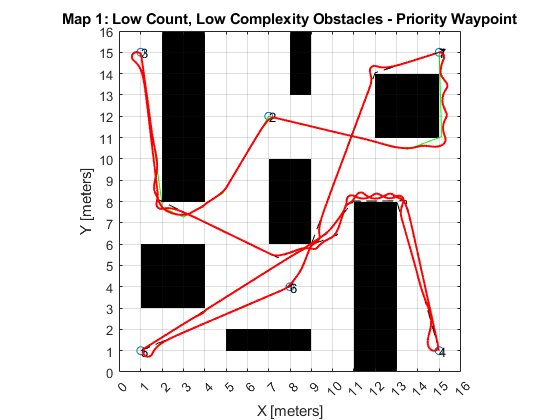
## Appendix 4 – Map 2

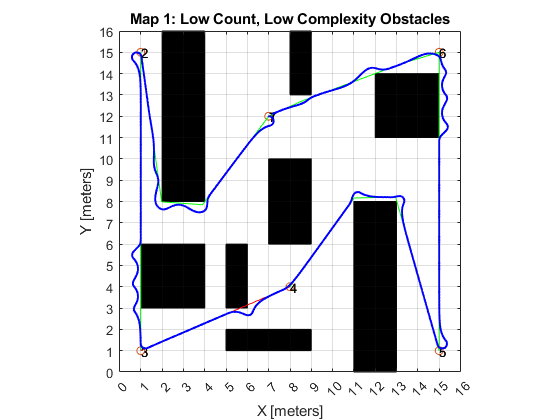
****

## Appendix 5 – Map 3

****

## Appendix 6 – Full Output Map 1

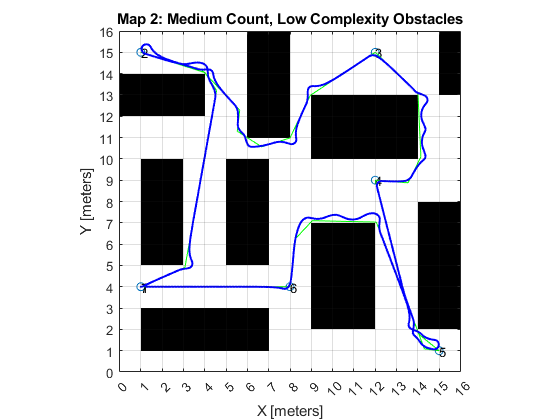
****

****

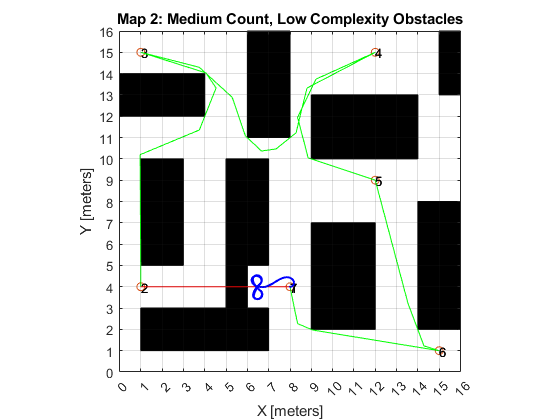
**A picture containing text, diagram, line, plot

Description automatically generated**

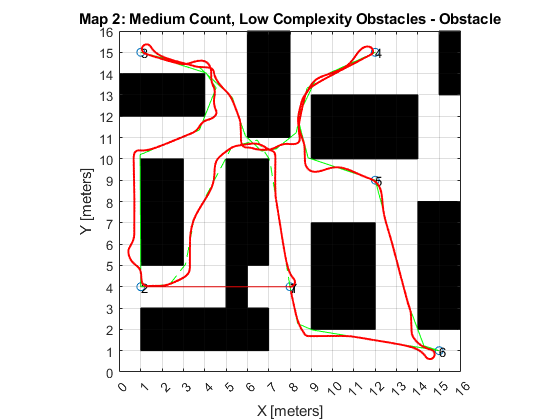
## Appendix 7 – Full Output Map 2

****

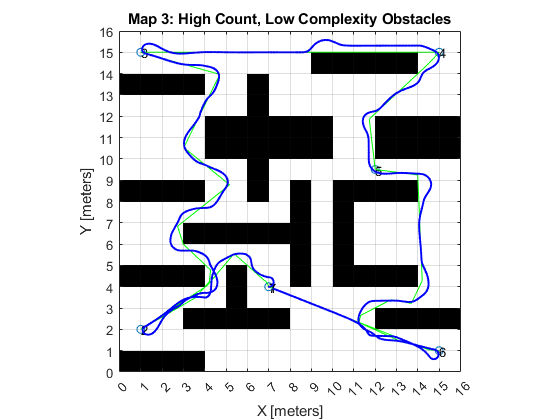
****

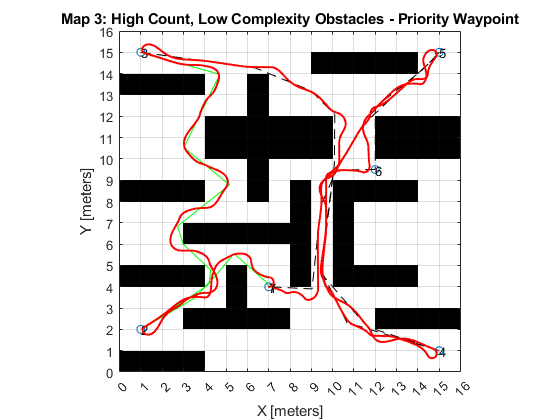
****

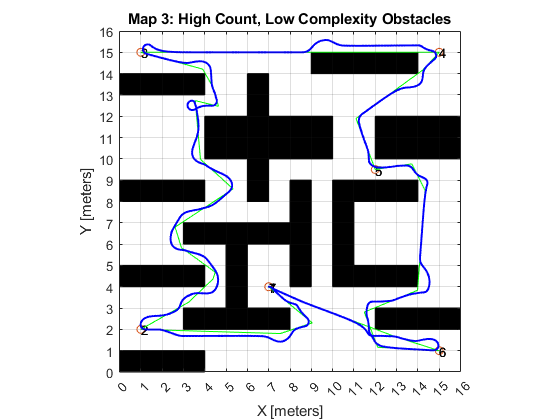
**Point of Interest: Map 2 Demonstrates a fail-state for Real Robot Path**

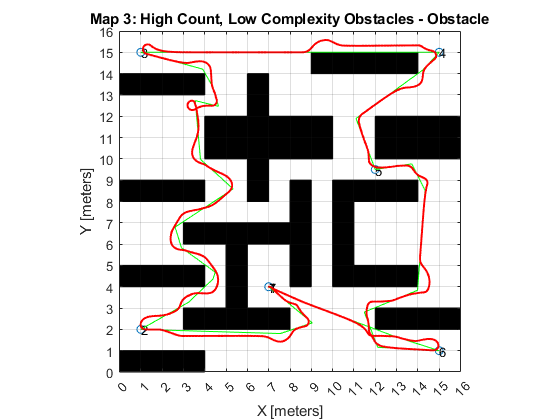
****

## Appendix 8 – Full Output Map 3

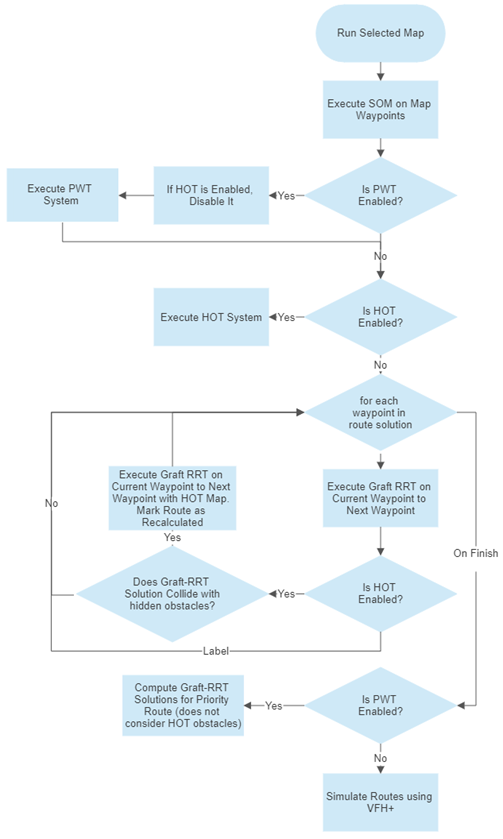
****

****

****

****

## Appendix 9 – Block Diagram



## Appendix 10 – Main.m Code

%% Initialization

% Clear Previous Workspace Information

clc;

clear;

close all;

% Setup Paths and add them to the environment

addpath(genpath("Algorithms"));

addpath(genpath("Digital Twin"));

addpath(genpath("Maps"));

addpath(genpath("Toolboxes"));

addpath(genpath('Toolboxes/MRS Toolbox/examples'));

addpath(genpath('Toolboxes/MRS Toolbox/src'));

addpath(genpath('Toolboxes/MRS Toolbox/doc'));

%% Parameter Selection

% Initialize new workspace by running a map (Valid Options: Map1.m, Map2.m, and Map 3.m)

run("Maps/Map1.m");

% Digital Twin - Priority Override

doPriorityOverride = false;

transitionWaypoint = [1 15];

priorityWaypoint = [15 1];

% Digital Twin - Obstacle Override

doObstacleOverride = true;

hiddenObstaclePolygon = [5 3; 5 6; 6 6; 6 3;];

%% Stage 1: Solve Travelling Salesman Problem (Self-Organizing Maps)

tic;

[network, optimal\_route, route\_distance] = self\_organizing\_map(map\_waypoints, map\_size, 100000, 0.8);

optimal\_route(end+1) = optimal\_route(1); % We want the robot to loop back around

for i=1:height(optimal\_route)

text(map\_waypoints(optimal\_route(i), 1), map\_waypoints(optimal\_route(i), 2), string(i));

end

disp("Solved TSP in " + toc + " seconds");

%% Digital Twin - Waypoint Priority Override

if (doPriorityOverride)

% Generate Priority Route

[priorityTransitionOccured, newRoute, priorityTransitionPoint] = PriorityOverride(transitionWaypoint, priorityWaypoint, optimal\_route, map\_waypoints, map\_size);

% This block handles painting a second figure for the Digital Twin to

% graph on, it only opens if the transition produces actionable results

if (priorityTransitionOccured)

mapPrio = figure;

show(occupancyMap);

title(map\_title + " - Priority Waypoint");

axis ([0 map\_size 0 map\_size]);

yticks(0:1:map\_size);

xticks(0:1:map\_size);

axis square;

grid on;

hold on;

plot(map\_waypoints(:,1), map\_waypoints(:,2), "o");

% Overlay the new route on the map

for i=1:height(newRoute)

text(map\_waypoints(newRoute(i), 1), map\_waypoints(newRoute(i), 2), string(i));

end

end

end

% In the event we are doing a priority override, prevent the obstacle

% override

if (doPriorityOverride && doObstacleOverride)

doObstacleOverride = false;

disp("Only one DT simulation can be run at once. Obstacle Override Ignored.");

end

%% Digital Twin - Obstacle Override

if (doObstacleOverride && ~doPriorityOverride)

[mapObst, occupancyMapObst, occupancy\_grid\_obst] = ObstacleOverride(map\_waypoints, ...

hiddenObstaclePolygon, occupancyMap, optimal\_route, map, map\_size, ...

map\_obstacles, occupancy\_grid, map\_title);

end

%% Stage 3: Find Point-To-Point Path for Each Waypoint in Order

% This stage is responsible for executing the Graft-RRT\* function on the

% waypoints in the order they were provided in Stage 2 or by Digital Twin

% Importantly the function takes in the following parameters:

% Start Waypoint

% Target Waypoint

% Map Size

% Obstacle Buffers

% Epsilon (Step Size)

% Threshold (Threshold to Goal Point)

% Number of Iterations

% The start waypoint and target/goal waypoint are determined by looping over

% the length/height of the produced best route in Stage 2. Epsilon, Threshold, and Iterations are researcher and

% map dependent and once an ideal set of parameter values has been identified it is recommend to save

% them into the MapX.m file.

% The outputs of the function are:

% Route

% Route Calculation Time

% The output route is not the entire solution tree, but the singular best

% route produced by the Graft-RRT algorithm. A line is then drawn from point

% to point in order to visualize the path taken from waypoint to waypoint on

% the map.

%% Find Navigation Solution via PTP Algorithm (Graft RRT)

route\_solutions = struct(); % This pre-allocates an array with the number of routes we will generate (number of waypoints in optimal route - 1)

obstacle\_route\_solutions = struct(); % Same as route\_solutions but is used in the event the Digital Twin for Obstacles is used

solution\_time = 0;

tic

for i=1:height(optimal\_route)-1

% Creates the route solution for the original map

[route, route\_calculation\_time] = graft\_rrtstar(map\_waypoints(optimal\_route(i), :), map\_waypoints(optimal\_route(i+1), :), map\_size, occupancy\_grid,map\_rrt\_epsilon, map\_rrt\_threshold, map\_rrt\_iterations);

solution\_time = solution\_time + route\_calculation\_time;

route\_solutions(i).data = route;

route\_solutions(i).recalculated = false;

% If we are doing obstacle override, we check the generated path and

% see if we need to re-perform the RRT on the specific waypoint

% transition (i.e it collides with the new obstacles)

if (doObstacleOverride)

for j=1:height(route)-1

if (~ObstacleFree(route(j, 1:2), route(j+1, 1:2), map\_size, occupancy\_grid\_obst))

[new\_route, new\_route\_calculation\_time] = graft\_rrtstar(map\_waypoints(optimal\_route(i), :), map\_waypoints(optimal\_route(i+1), :), map\_size, occupancy\_grid\_obst,map\_rrt\_epsilon, map\_rrt\_threshold, map\_rrt\_iterations);

solution\_time = solution\_time + route\_calculation\_time;

obstacle\_route\_solutions(i).data = new\_route;

route\_solutions(i).recalculated = true;

break;

end

end

end

end

% This handles creating an entirely new path for the remaing route

% waypoints

if (doPriorityOverride && priorityTransitionOccured)

priority\_route\_solutions = struct();

for i=1:height(newRoute)-1

[prio\_route, prio\_route\_calculation\_time] = graft\_rrtstar(map\_waypoints(newRoute(i), :), map\_waypoints(newRoute(i+1), :), map\_size, occupancy\_grid,map\_rrt\_epsilon, map\_rrt\_threshold, map\_rrt\_iterations);

solution\_time = solution\_time + prio\_route\_calculation\_time;

priority\_route\_solutions(i).data = prio\_route;

end

end

disp("Path Plan Generated in " + solution\_time + " seconds")

% Plot Navigation Solution

for i=1:height(optimal\_route)-1

% This block handles plotting data in the event priority override was

% enabled

if (doPriorityOverride && priorityTransitionOccured && i >= priorityTransitionPoint)

figure(mapPrio);

solutionPath = priority\_route\_solutions(i).data;

line(solutionPath(:, 1), solutionPath(:, 2), "Color", "k", "LineStyle", "--")

elseif (doPriorityOverride && priorityTransitionOccured)

figure(mapPrio);

solutionPath = route\_solutions(i).data;

priority\_route\_solutions(i).data = solutionPath;

line(solutionPath(:, 1), solutionPath(:, 2), "Color", "g", "LineStyle", "-")

end

% This block handles plotting in the event that the obstale override

% was enabled, recalculated is only set to true in the event the path

% is recalculated, and is all false when override is disabled. This

% means we don't need a specific if check for the doObstacleOverride

solutionPath = route\_solutions(i).data;

if (route\_solutions(i).recalculated)

figure(map)

line(solutionPath(:, 1), solutionPath(:, 2), "Color", "r", "LineStyle", "-")

figure(mapObst)

line(solutionPath(:, 1), solutionPath(:, 2), "Color", "r", "LineStyle", "-")

solutionPath = obstacle\_route\_solutions(i).data;

figure(mapObst)

line(solutionPath(:, 1), solutionPath(:, 2), "Color", "g", "LineStyle", "--")

else

figure(map)

line(solutionPath(:, 1), solutionPath(:, 2), "Color", "g", "LineStyle", "-")

if (doObstacleOverride)

figure(mapObst)

line(solutionPath(:, 1), solutionPath(:, 2), "Color", "g", "LineStyle", "-")

end

end

end

% This simply handles executing the VFH based off the execution parameters

if (doPriorityOverride && priorityTransitionOccured)

% Priority Waypoint VFH Case

VFHOnPath(priority\_route\_solutions, mapPrio, true);

VFHOnPath(route\_solutions, map, false);

elseif (doObstacleOverride)

% Obstacle Override VFH Case

solutionPath = route\_solutions;

% This regenerates a combined solution since the the obstacle route

% simply uses recalculated values

for i=1:height(optimal\_route)-1

if (route\_solutions(i).recalculated)

solutionPath(i).data = obstacle\_route\_solutions(i).data;

route\_solutions(i).data = [map\_waypoints(optimal\_route(i), :); map\_waypoints(optimal\_route(i+1), :)]; % This allows the VFH to "postulate" on how to solve invalid paths

end

end

VFHOnPath(solutionPath, mapObst, true);

VFHOnPath(route\_solutions, map, false);

else

% Normal Run Case - No Digital Twin Case

VFHOnPath(route\_solutions, map, false);

End

## Appendix 11 – Map(s) Code

### Map1.m

%% Map 1: Low Obstacle Count, Low Complexity

% This file configures Map 1 which is intended to be used in main.m as a "demonstration" of the system. It

% features a low count, low complexity set of obstacles. The remainder of this file demonstrates a

% semi-standardized way of creating Map files.

%% Map Configuration

map\_title = "Map 1: Low Count, Low Complexity Obstacles";

map\_size = 16;

occupancyMap = binaryOccupancyMap(map\_size, map\_size, 1, "grid");

occupancy\_grid = zeros(map\_size, map\_size);

% Map RRT Parameters

map\_rrt\_epsilon = 1;

map\_rrt\_iterations = 15000;

map\_rrt\_threshold = 0.3;

%% Create Map Waypoints

% 1st Parameter: Input Waypoint Array</li>

% 2nd Parameter: New Waypoint X,Y</li>

% Input Waypoint Array is either a new array, [], or the output array

% provided by the previous CreateWaypoint call (in the case of the code

% below map\_waypoints)<br>New Waypoint is a simple array containing a

% singular set of X,Y coordinates. As an example, if I wanted a waypoint on

% the point 10, 4 then the waypoint would look like [10 4].

map\_waypoints = CreateWaypoint([], [1 1]);

map\_waypoints = CreateWaypoint(map\_waypoints, [1 15]);

map\_waypoints = CreateWaypoint(map\_waypoints, [8 4]);

map\_waypoints = CreateWaypoint(map\_waypoints, [7 12]);

map\_waypoints = CreateWaypoint(map\_waypoints, [15 15]);

map\_waypoints = CreateWaypoint(map\_waypoints, [15 1]);

%% Create Map Obstacles

% If you wish tosimulate an obstacle proceeding partially in and off the map, simply clip

% the obstacle vertices to be directly on the upper or lower bound of the

% map size.

% The CreateObstacle function takes in the following parameters:</p>

% 1st Parameter: Input Obstacle Array

% 2nd Parameter: Obstacle Vertices in Clockwise Order

%

% Input Obstacle Array is either a new array, [], or the output array

% provided by the previous CreateObstacle call (in the case of the code

% below map\_obstacles)

%

% Obstacle Vertices is a column array with X values in column 1 and Y values in column 2

map\_obstacles = CreateObstacle([], [2 8; 2 16; 4 16; 4 8;]);

map\_obstacles = CreateObstacle(map\_obstacles, [1 3; 1 6; 4 6; 4 3;]);

map\_obstacles = CreateObstacle(map\_obstacles, [5 1; 5 2; 9 2; 9 1;]);

map\_obstacles = CreateObstacle(map\_obstacles, [7 6; 7 10; 9 10; 9 6;]);

map\_obstacles = CreateObstacle(map\_obstacles, [8 13; 8 16; 9 16; 9 13;]);

map\_obstacles = CreateObstacle(map\_obstacles, [11 0; 11 8; 13 8; 13 0;]);

map\_obstacles = CreateObstacle(map\_obstacles, [12 11; 12 14; 15 14; 15 11;]);

% Convert polyshape obstacles to occupancy grid

for i = 1:map\_size

for j = 1:map\_size

x = map\_size - i - 0.5 + 1;

y = j - 0.5;

isOccupied = false;

for k = 1:numel(map\_obstacles)

isOccupied = inpolygon(x, y, map\_obstacles(k).Vertices(:, 2), map\_obstacles(k).Vertices(:, 1));

if (isOccupied)

occupancy\_grid(i, j) = 1;

break

end

end

end

end

%% Update Map with Generated Occupancy Map

setOccupancy(occupancyMap, occupancy\_grid);

map = figure;

show(occupancyMap);

%% Configure Map Display

title(map\_title);

axis ([0 map\_size 0 map\_size]);

yticks(0:1:map\_size);

xticks(0:1:map\_size);

axis square;

grid on;

hold on;

% Plot Waypoint Configuration onto the Map Figure

plot(map\_waypoints(:,1), map\_waypoints(:,2), "o");

grid on;

% We do some cleanup of unneeded workspace variables

clear i

clear j

clear k

clear x

clear y

### Map2.m

%% Map 2: Medium Obstacle Count, Low Complexity

% This file configures Map 2 which is intended to be used in main.m as a "demonstration" of the system. It

% features a medium count, low complexity set of obstacles. The remainder of this file demonstrates a

% semi-standardized way of creating Map files.

%% Map Configuration

map\_title = "Map 2: Medium Count, Low Complexity Obstacles";

map\_size = 16;

occupancyMap = binaryOccupancyMap(map\_size, map\_size, 1, "grid");

occupancy\_grid = zeros(map\_size, map\_size);

% Map RRT Parameters

map\_rrt\_epsilon = 1;

map\_rrt\_iterations = 20000;

map\_rrt\_threshold = 0.3;

%% Create Map Waypoints

% 1st Parameter: Input Waypoint Array</li>

% 2nd Parameter: New Waypoint X,Y</li>

% Input Waypoint Array is either a new array, [], or the output array

% provided by the previous CreateWaypoint call (in the case of the code

% below map\_waypoints)<br>New Waypoint is a simple array containing a

% singular set of X,Y coordinates. As an example, if I wanted a waypoint on

% the point 10, 4 then the waypoint would look like [10 4].

map\_waypoints = CreateWaypoint([], [1 4]);

map\_waypoints = CreateWaypoint(map\_waypoints, [1 15]);

map\_waypoints = CreateWaypoint(map\_waypoints, [8 4]);

map\_waypoints = CreateWaypoint(map\_waypoints, [12 9]);

map\_waypoints = CreateWaypoint(map\_waypoints, [12 15]);

map\_waypoints = CreateWaypoint(map\_waypoints, [15 1]);

%% Create Map Obstacles

% If you wish tosimulate an obstacle proceeding partially in and off the map, simply clip

% the obstacle vertices to be directly on the upper or lower bound of the

% map size.

% The CreateObstacle function takes in the following parameters:</p>

% 1st Parameter: Input Obstacle Array

% 2nd Parameter: Obstacle Vertices in Clockwise Order

%

% Input Obstacle Array is either a new array, [], or the output array

% provided by the previous CreateObstacle call (in the case of the code

% below map\_obstacles)

%

% Obstacle Vertices is a column array with X values in column 1 and Y values in column 2

[map\_obstacles] = CreateObstacle([], [0 12; 0 14; 4 14; 4 12]);

[map\_obstacles] = CreateObstacle(map\_obstacles, [1 5; 1 10; 3 10; 3 5]);

[map\_obstacles] = CreateObstacle(map\_obstacles, [1 1; 1 3; 7 3; 7 1]);

[map\_obstacles] = CreateObstacle(map\_obstacles, [5 5; 5 10; 7 10; 7 5]);

[map\_obstacles] = CreateObstacle(map\_obstacles, [6 11; 6 16; 8 16; 8 11]);

[map\_obstacles] = CreateObstacle(map\_obstacles, [9 10; 9 13; 14 13; 14 10]);

[map\_obstacles] = CreateObstacle(map\_obstacles, [15 13; 15 16; 16 16; 16 13]);

[map\_obstacles] = CreateObstacle(map\_obstacles, [9 2; 9 7; 12 7; 12 2]);

[map\_obstacles] = CreateObstacle(map\_obstacles, [14 2; 14 8; 16 8; 16 2]);

% Convert polyshape obstacles to occupancy grid

for i = 1:map\_size

for j = 1:map\_size

x = map\_size - i - 0.5 + 1;

y = j - 0.5;

isOccupied = false;

for k = 1:numel(map\_obstacles)

isOccupied = inpolygon(x, y, map\_obstacles(k).Vertices(:, 2), map\_obstacles(k).Vertices(:, 1));

if (isOccupied)

occupancy\_grid(i, j) = 1;

break

end

end

end

end

%% Update Map with Generated Occupancy Map

setOccupancy(occupancyMap, occupancy\_grid);

map = figure;

show(occupancyMap);

%% Configure Map Display

title(map\_title);

axis ([0 map\_size 0 map\_size]);

yticks(0:1:map\_size);

xticks(0:1:map\_size);

axis square;

grid on;

hold on;

% Plot Waypoint Configuration onto the Map Figure

plot(map\_waypoints(:,1), map\_waypoints(:,2), "o");

grid on;

% We do some cleanup of unneeded workspace variables

clear i

clear j

clear k

clear x

clear y

### Map3.m

%% Map 1: High Obstacle Count, Low Complexity

% This file configures Map 3 which is intended to be used in main.m as a "demonstration" of the system. It

% features a high count, low complexity set of obstacles. The remainder of this file demonstrates a

% semi-standardized way of creating Map files.

%% Map Configuration

map\_title = "Map 3: High Count, Low Complexity Obstacles";

map\_size = 16;

occupancyMap = binaryOccupancyMap(map\_size, map\_size, 1, "grid");

occupancy\_grid = zeros(map\_size, map\_size);

% Map RRT Parameters

map\_rrt\_epsilon = 1;

map\_rrt\_iterations = 10000;

map\_rrt\_threshold = 0.5;

%% Create Map Waypoints

% 1st Parameter: Input Waypoint Array</li>

% 2nd Parameter: New Waypoint X,Y</li>

% Input Waypoint Array is either a new array, [], or the output array

% provided by the previous CreateWaypoint call (in the case of the code

% below map\_waypoints)<br>New Waypoint is a simple array containing a

% singular set of X,Y coordinates. As an example, if I wanted a waypoint on

% the point 10, 4 then the waypoint would look like [10 4].

map\_waypoints = CreateWaypoint([], [1 2]);

map\_waypoints = CreateWaypoint(map\_waypoints, [1 15]);

map\_waypoints = CreateWaypoint(map\_waypoints, [7 4]);

map\_waypoints = CreateWaypoint(map\_waypoints, [12 9.5]);

map\_waypoints = CreateWaypoint(map\_waypoints, [15 15]);

map\_waypoints = CreateWaypoint(map\_waypoints, [15 1]);

%% Create Map Obstacles

% If you wish tosimulate an obstacle proceeding partially in and off the map, simply clip

% the obstacle vertices to be directly on the upper or lower bound of the

% map size.

% The CreateObstacle function takes in the following parameters:</p>

% 1st Parameter: Input Obstacle Array

% 2nd Parameter: Obstacle Vertices in Clockwise Order

%

% Input Obstacle Array is either a new array, [], or the output array

% provided by the previous CreateObstacle call (in the case of the code

% below map\_obstacles)

%

% Obstacle Vertices is a column array with X values in column 1 and Y values in column 2

[map\_obstacles] = CreateObstacle([], [0 0; 0 1; 4 1; 4 0]);

[map\_obstacles] = CreateObstacle(map\_obstacles, [0 4; 0 5; 4 5; 4 4]);

[map\_obstacles] = CreateObstacle(map\_obstacles, [0 8; 0 9; 4 9; 4 8]);

[map\_obstacles] = CreateObstacle(map\_obstacles, [0 13; 0 14; 4 14; 4 13]);

[map\_obstacles] = CreateObstacle(map\_obstacles, [3 2; 3 3; 5 3; 5 5; 6 5; 6 3; 8 3; 8 2]);

[map\_obstacles] = CreateObstacle(map\_obstacles, [3 6; 3 7; 8 7; 8 9; 9 9; 9 4; 8 4; 8 6;]);

[map\_obstacles] = CreateObstacle(map\_obstacles, [6 8; 6 10; 4 10; 4 12; 6 12; 6 14; 7 14; 7 12; 10 12; 10 10; 7 10; 7 8]);

[map\_obstacles] = CreateObstacle(map\_obstacles, [9 14; 9 15; 14 15; 14 14;]);

[map\_obstacles] = CreateObstacle(map\_obstacles, [12 10; 12 12; 16 12; 16 10;]);

[map\_obstacles] = CreateObstacle(map\_obstacles, [10 4; 10 9; 14 9; 14 8; 11 8; 11 5; 14 5; 14 4]);

[map\_obstacles] = CreateObstacle(map\_obstacles, [12 2; 12 3; 16 3; 16 2]);

% Convert polyshape obstacles to occupancy grid

for i = 1:map\_size

for j = 1:map\_size

x = map\_size - i - 0.5 + 1;

y = j - 0.5;

isOccupied = false;

for k = 1:numel(map\_obstacles)

isOccupied = inpolygon(x, y, map\_obstacles(k).Vertices(:, 2), map\_obstacles(k).Vertices(:, 1));

if (isOccupied)

occupancy\_grid(i, j) = 1;

break

end

end

end

end

%% Update Map with Generated Occupancy Map

setOccupancy(occupancyMap, occupancy\_grid);

map = figure;

show(occupancyMap);

%% Configure Map Display

title(map\_title);

axis ([0 map\_size 0 map\_size]);

yticks(0:1:map\_size);

xticks(0:1:map\_size);

axis square;

grid on;

hold on;

% Plot Waypoint Configuration onto the Map Figure

plot(map\_waypoints(:,1), map\_waypoints(:,2), "o");

grid on;

% We do some cleanup of unneeded workspace variables

clear i

clear j

clear k

clear x

clear y

### Helpers/CreateObstacle.m

function [outObstacles, occupancy\_map] = CreateObstacle(inObstacles, inVertices, occupancy\_map)

%CREATEOBSTACLE Create a new obstacle which will be inserted into a

%previously existing obstacle array.

% Takes in a map's obstacle array and merges in a new polyshape given the

% obstacle's verticies

% Create new Polyshape

newObstacle = polyshape(inVertices);

% Merge into Array

outObstacles = [inObstacles; newObstacle];

end

### Helpers/CreateWaypoint.m

function [outWaypoints] = CreateWaypoint(inWaypoints, newWaypoint)

%CREATEWAYPOINT Create a new waypoint which will be inserted into a

%previously existing waypoint array.

% Takes in a map's waypoint array and merges in a new waypoint given the

% waypoint's coordinates

% Merge into Array

outWaypoints = [inWaypoints; newWaypoint];

end

## Appendix 12 – Digital Twin Code

### ObstacleOverride.m

function [mapObst, occupancyMapObst, occupancy\_grid\_obst] = ObstacleOverride(map\_waypoints, hiddenObstaclePolygon, occupancyMap, optimal\_route, map, map\_size, map\_obstacles, occupancy\_grid, map\_title)

% Create new Binary Grid

occupancyMapObst = binaryOccupancyMap(map\_size, map\_size, 1, "grid");

occupancy\_grid\_obst = zeros(map\_size, map\_size);

setOccupancy(occupancyMapObst, occupancy\_grid\_obst);

% This mimics the system in Maps/MapX.m

% We are just doing it to get the new obstacle associated with the

% previous map obstacles and add them to the occupancy map as necessary

digitalTwinObst = CreateObstacle(map\_obstacles, hiddenObstaclePolygon, occupancy\_grid);

for i = 1:map\_size

for j = 1:map\_size

x = map\_size - i - 0.5 + 1;

y = j - 0.5;

for k = 1:numel(digitalTwinObst)

isOccupied = inpolygon(x, y, digitalTwinObst(k).Vertices(:, 2), digitalTwinObst(k).Vertices(:, 1));

if (isOccupied)

occupancy\_grid\_obst(i, j) = 1;

break

end

end

end

end

% Update Occupancy Map for new Figure

setOccupancy(occupancyMapObst, occupancy\_grid\_obst);

mapObst = figure;

show(occupancyMapObst);

% Setup Map Parameters

title(map\_title + " - Obstacle");

axis ([0 map\_size 0 map\_size]);

yticks(0:1:map\_size);

xticks(0:1:map\_size);

axis square;

grid on;

hold on;

% Plots waypoints

plot(map\_waypoints(:,1), map\_waypoints(:,2), "o");

% Re-Label Waypoints

for i=1:height(optimal\_route)

text(map\_waypoints(optimal\_route(i), 1), map\_waypoints(optimal\_route(i), 2), string(i));

end

% This re-updates the current map with the new occupancy map

setOccupancy(occupancyMap, occupancy\_grid\_obst);

figure(map);

show(occupancyMap)

title(map\_title);

axis ([0 map\_size 0 map\_size]);

yticks(0:1:map\_size);

xticks(0:1:map\_size);

axis square;

grid on;

hold on;

% Plots Waypoints

plot(map\_waypoints(:,1), map\_waypoints(:,2), "o");

% Re-Labels Waypoints

for i=1:height(optimal\_route)

text(map\_waypoints(optimal\_route(i), 1), map\_waypoints(optimal\_route(i), 2), string(i));

end

end

### PriorityOverride.m

function [isNeeded, newRoute, transitionPoint] = PriorityOverride(transitionWaypoint,priorityWaypoint,optimalRoute,mapWaypoints,mapSize)

newRoute = zeros(height(optimalRoute), 1);

% The user has indicated they would like a specific translation on two

% waypoints. They should be specified as the actual waypoint

% coordinates as we do not want to rely on the "current" ordering. That

% is, we will force a "re-wire" at a "current position" to go towards the "priority" waypoint.

% Get Map Waypoint Index (ID)

waypoint\_curr\_id = find(ismember(mapWaypoints, transitionWaypoint,'rows'));

% Get Map Waypoint Location in SOM-route

waypoint\_route\_index = find(optimalRoute == waypoint\_curr\_id, 1);

transitionPoint = waypoint\_route\_index;

% Determine already travelled waypoints

newRoute(1:waypoint\_route\_index) = optimalRoute(1:waypoint\_route\_index);

route\_to\_travel = optimalRoute(waypoint\_route\_index+1:end);

% We now circumvent SOM-ordering by merging the next-waypoint into the

% next position of route\_traveled.

% Get Map Waypoint Index (ID)

waypoint\_prio\_id = find(ismember(mapWaypoints, priorityWaypoint,'rows'));

waypoint\_prio\_route\_index = find(optimalRoute == waypoint\_prio\_id, 1);

% We check to see if the swap makes sense, aka, have we already

% "visited" the priority waypoint by the time the waypoint transition

% would occur

isNeeded = true;

if (waypoint\_prio\_route\_index < waypoint\_route\_index)

disp("Priority Override not needed")

isNeeded = false;

newRoute = NaN;

end

if (isNeeded)

% Add Priority Waypoint next in line in priorityRoute

newRoute(waypoint\_route\_index + 1) = waypoint\_prio\_id;

% Re-calculate Route-To-Travel (we must first map the ids back to

% their actual waypoints)

neededWaypoints = mapWaypoints(route\_to\_travel, :);

neededWaypoints(end, :) = [];

% We now have the order in which they should appear

[~, priority\_optimalRoute, ~] = self\_organizing\_map(neededWaypoints, mapSize, 100000, 0.8);

% Re-map priority\_optimalRoute back to neededWaypoints

neededWaypointsOrdered = neededWaypoints(priority\_optimalRoute, :);

neededWaypointsOriginalReferences = zeros(height(neededWaypointsOrdered), 1);

for i=1:height(neededWaypointsOrdered)

for j=1:height(mapWaypoints)

if (mapWaypoints(j, :) == neededWaypointsOrdered(i, :))

neededWaypointsOriginalReferences(i) = j;

break;

end

end

end

% We circular shift the waypoints such that they appear in the correct

% order

shiftAmount = find(neededWaypointsOriginalReferences == waypoint\_prio\_id);

if (shiftAmount ~= 1)

neededWaypointsOriginalReferences = circshift(neededWaypointsOriginalReferences, 1-shiftAmount);

end

% The route should always loop back to the beginning

newRoute(waypoint\_route\_index+1:end-1) = neededWaypointsOriginalReferences(1:end);

newRoute(end, :) = newRoute(1, :);

end

## Appendix 13 – Self Organizing Map Code

function [network, optimalRoute, routeDistance] = self\_organizing\_map(map\_waypoints, map\_size, iterations, learning\_rate)

%% Stage 1: Generate the Initial Network</h2>

% Neurons are dynamically scaled with the number of waypoints present on the

% map. The constant multiplier can be adjusted to fine-tune the network.

activeNeurons = height(map\_waypoints) \* 8;

networkNeurons = rand(activeNeurons, 2);

% In order to early terminate for dwindling improvements, a relationship must be

% made between the closest neuron to a waypoint and its distance. This in

% effect should cause the program to terminate when the rate of improvement

% begins to dwindle.

error\_array = zeros(height(map\_waypoints), 2);

error\_array(:, 2) = map\_size\*map\_size;

%% Stage 2: Train the Network

for i = 1:iterations

error\_array(:, 1) = error\_array(:, 2);

% Stage 2a: Sample In Waypoint Order

waypoint\_index = mod(i-1, height(map\_waypoints)) + 1;

waypoint = map\_waypoints(waypoint\_index, :);

% Stage 2b: Find Closest ("Winning") Neuron

% Its index and value are saved to be used in the gaussian calculation,

% the winning neuron defaults to being the first element in the

% generated network (networkNeurons). This can potentially be

% functionalized/optimized by moving it to a function and finding a way

% to apply pdist to two arrays.

closestNeuron\_Index = 1;

closestNeuron\_Distance = pdist([waypoint; networkNeurons(1, :)]);

for j = 2:height(networkNeurons)

distance = pdist([waypoint; networkNeurons(j, :)]);

if (distance < closestNeuron\_Distance)

closestNeuron\_Index = j;

closestNeuron\_Distance = distance;

end

end

% Stage 2c: Apply Gaussian Regression

% Calculate a gaussian based off closest neuron, this will be applied to

% the rest of the network as a way to update their weights. This

% methodology allows for a greater effect on points closer to the

% closest point with dwindling affects as the distance to the closest

% point increases. Radius (radix) is determined by the

% "activeNeurons", and is labeled this way primarily since the number

% is based off the number of neurons at start and it decays to 0. In

% many ways this resembles "drop out" seen in many Neural Networks such

% as CNNs.

center = closestNeuron\_Index;

domain = height(networkNeurons);

radix = floor(activeNeurons/10);

% This is used to prevent NaNs from populating into the Network

if (radix < 1)

radix = 1;

end

deltas = abs(center - (0:1:domain-1));

distances = min(deltas, (domain-deltas));

% Calculate Guassian

gaussian = exp(-(distances.\*distances) / (2\*(radix\*radix)));

% Apply Gaussian Filter to the Network

networkNeurons = networkNeurons + ((gaussian(:) .\* ([(waypoint(:, 1) - networkNeurons(:, 1)) (waypoint(:, 2) - networkNeurons(:, 2))])) \* learning\_rate);

% Stage 2d: Decay Learning Parameters

learning\_rate = learning\_rate \* 0.99997;

activeNeurons = activeNeurons \* 0.9997;

% If the radius (or amount of activeNeurons) has completely decayed we

% can go ahead and exit out as we can no longer learn.

if (activeNeurons < 1)

break;

end

% If the learning rate has completely decayed we can go ahead and exit

% out as we can no longer learn.

if (learning\_rate < 0.001)

break;

end

% Stage 2e: Early Termination Clause for Real-time Priortization

error\_array(waypoint\_index, 2) = pdist([waypoint; networkNeurons(closestNeuron\_Index, :)]);

if (((mean(error\_array(:,1) - mean(error\_array(:, 2)))) < 0.001) && mean(error\_array(:, 2) < 0.001))

break;

end

end

%% Stage 3: Build Calculated Route

% Unlike in a normal SOM, the output of our training is the data we want. We

% identify the waypoint order by finding the "winning" neurons which are

% the neurons closest to each waypoint. This list of winning neurons can be

% found in order and the order in which they appear is the "optimal"

% waypoint route found by the SOM.<br>This portion of the code leverages

% the GetClosestPoint helper function to quickly calculate the closest

% point in the network neurons to the specific map waypoint provided.

% For each city, find the closest neuron, and mark it as the winning neuron

optimalRoute = [];

for i = 1:height(map\_waypoints)

[bestRoutePoint, ~] = GetClosestPoint(map\_waypoints(i,:), networkNeurons);

optimalRoute = [optimalRoute; [i bestRoutePoint]];

end

optimalRoute = sortrows(optimalRoute, 2);

optimalRoute = optimalRoute(:, 1);

%% Stage 4: Calculate Path Cost (Optional)

routeDistance = 0;

for i = 1:height(optimalRoute)-1

[~, distance] = GetClosestPoint([map\_waypoints(optimalRoute(i), 1), map\_waypoints(optimalRoute(i), 2)], [map\_waypoints(optimalRoute(i+1), 1), map\_waypoints(optimalRoute(i+1), 2)]);

routeDistance = routeDistance + distance;

end

network = networkNeurons;

end

### Helpers/GetClosestPoint.m

function [closestPoint,closestDistance] = GetClosestPoint(origin,canidates)

% GetClosestPoint takes in a point (origin) and an array of other points

% (canidates) and returns the index of the canidate that is closest to origin in terms

% of eucledian distance.

closestPoint = 1;

closestDistance = pdist([origin; canidates(1, :)]);

for i = 1:height(canidates)

distance = pdist([origin; canidates(i, :)]);

if (distance < closestDistance)

closestPoint = i;

closestDistance = distance;

end

end

end

## Appendix 14 – Graft-RRT Code

function [path, calculation\_time] = graft\_rrtstar(source, goal, map\_size, occupancy\_grid, epsilon, threshold, iterations)

stepsize=epsilon;

disTh=threshold;

maxFailedAttempts = iterations;

rrstar\_area = stepsize .\* 2;

center\_point = rand\_center(source, goal, map\_size, occupancy\_grid);

% Safety Checks on the algorithm

if ~CheckCollision(source, occupancy\_grid, map\_size)

error('Source Waypoint is occluded by an Obstacle');

end

if ~CheckCollision(goal, occupancy\_grid, map\_size)

error('Goal Waypoint is occluded by an Obstacle');

end

static\_param\_data =[stepsize disTh maxFailedAttempts rrstar\_area; source 0 0 ; center\_point 0 0 ; goal 0 0];

% static\_param\_data = stepsize disTh maxFailedAttempts rrstar\_area

% source(1,1) source(1,2) point\_connect(1,1) point\_connect(1,2)

% center\_point(1.1) center\_point(1,2) prev\_ahead prev\_behind

% goal(1,2) goal(1,2) prev\_behind\_ahead prev\_behing\_behind

% point\_connect2(1,1) point\_connect2(1,2)

% Initialize Base Values

global\_length = 0;

local\_length = 0;

% Initalize the Three Trees

RRTree\_main=double([source -1 global\_length local\_length ]);

RRTree\_goal=double([goal -1 global\_length local\_length]);

GraftTree=double([center\_point -1]);

% These help manage state during the while loop (are placed into the

% combined donytic\_param\_data) for easier reference

Graft\_ahead=false;

Graft\_behind=false;

ahead\_found = false;

behind\_found = false;

errtree\_found = false;

donytic\_param\_data =[Graft\_ahead,Graft\_behind,ahead\_found,behind\_found,errtree\_found];

tic;

% Verify the two points cannot be visibly connected

if (ObstacleFree(source, goal, map\_size, occupancy\_grid))

% They can be, manually create path and return early

distance = distanceCost(source, goal);

path = [source(1), source(2), 0, 0];

path = [path; goal(1), goal(2), distance, distance];

calculation\_time = toc;

return;

end

% We loop while we haven't found/connected all trees

while ~donytic\_param\_data(3) || ~donytic\_param\_data(4)

% If we have not connected graft to the goal tree and the goal tree

% hasn't been found yet

if ~donytic\_param\_data(1) && ~donytic\_param\_data(3)

% Perform RRT from the Start Tree

[RRTree\_main,static\_param\_data,donytic\_param\_data]=g\_rstara(RRTree\_main,GraftTree,RRTree\_goal,static\_param\_data,donytic\_param\_data,map\_size, occupancy\_grid);

end

% If we have not connected graft to the start tree and the start

% tree hasn't been found

if ~donytic\_param\_data(2) && ~donytic\_param\_data(4)

% Perform RRT from the Goal Tree

[RRTree\_goal,static\_param\_data,donytic\_param\_data]=g\_rstarb(RRTree\_main,GraftTree,RRTree\_goal,static\_param\_data,donytic\_param\_data,map\_size, occupancy\_grid);

end

% If we haven't found the goal tree or start tree

if ~donytic\_param\_data(3) || ~donytic\_param\_data(4)

% Perform RRT from the Graft Tree

[GraftTree,static\_param\_data,donytic\_param\_data]=graft\_extend(RRTree\_main,GraftTree,RRTree\_goal,static\_param\_data,donytic\_param\_data,map\_size, occupancy\_grid);

end

% If we have found goal tree and behind tree

if donytic\_param\_data(3) && donytic\_param\_data(4)

% Generate Path and exit loop

in\_path = re\_path(RRTree\_main,GraftTree,RRTree\_goal,static\_param\_data,donytic\_param\_data);

break;

end

end

% This handles compression and expansion phase of Graft-RRT

in\_path=check\_globalvalue(in\_path,map\_size, occupancy\_grid);

out\_path=check\_globalvalueII(in\_path,20,map\_size,occupancy\_grid);

in\_path=check\_globalvalue(out\_path,map\_size, occupancy\_grid);

% Set output Path to the path from the compression/expansion phase

path=in\_path;

calculation\_time = toc;

end

### Helpers/g\_rstara.m

function [RRTree\_main,static\_param\_data,donytic\_param\_data] = g\_rstara(RRTree\_main,GraftTree,RRTree\_goal,static\_param\_data,donytic\_param\_data,map,obstacles)

% creat param talbe for input data

% static\_param\_data= stepsize disTh maxFailedAttempts rrstar\_area

% source(1,1) source(1,2) point\_connect(1,1) point\_connect(1,2)

% center\_point(1.1) center\_point(1,2) prev\_ahead prev-graft-ahead

% goal(1,2) goal(1,2) prev\_behing prev-graft\_behind

% point\_connectmain(1,1) point\_connectmain(1,2) point\_connect2(1,1) point\_connect2(1,2)

% donytic\_param\_data =[Graft\_ahead,Graft\_behind,ahead\_found,behind\_found];

faildcount = 0;

% Loop until iteration count is reached

while (faildcount <= static\_param\_data(1,3))

% Get the sample point based off rules specified by Graft-RRT

% algorithm

if rand < 0.5

sample = RandomPoint(0, map);

elseif rand < 0.75

sample = [static\_param\_data(3,1) static\_param\_data(3,2)];

else

sample = [static\_param\_data(4,1) static\_param\_data(4,2)];

end

buffer\_distance = in\_distance(RRTree\_main(:,1:2),sample);

% This is a method for acquiring Q\_new that is faster than the for loop method

I(1) = check\_loacalvalue(buffer\_distance,RRTree\_main,static\_param\_data(1,4));

% 0 is not allowed so we just quickly set it to 1 (indexing)

if (I(1) == 0)

I(1) =1;

end

% We now know the closest node in the start tree

closestNode = RRTree\_main(I(1), 1:2);

% Point the growth direction twoards the newPoint

theta = atan2(sample(1)-closestNode(1),sample(2)-closestNode(2));

% Generate Start Tree point from found trajectory (remember this is bound

% by epsilon)

newPoint = double(closestNode(1:2) + static\_param\_data(1,1) \* [sin(theta) cos(theta)]);

% If it's not obstacle free, we skip this iteration and move on

if (~ObstacleFree(closestNode(1:2),newPoint,map,obstacles))

faildcount = faildcount+1;

continue;

end

% Grab's the closest point in entire Graft Tree by index to the new

% point

[~,Index\_graft] = min(in\_distance(GraftTree(:,1:2),newPoint),[],1);% Whether it is optimized here depends on the phenomenon, the memory usage is too high

% This is intended to connect to the two trees (Graft and Start)

if distanceCost(GraftTree(Index\_graft(1),1:2),newPoint)<static\_param\_data(1,2) && ObstacleFree(GraftTree(Index\_graft(1),1:2),newPoint,map,obstacles)

donytic\_param\_data(1,1) = false;

donytic\_param\_data(1,3) = true;

static\_param\_data(2,3) = newPoint(1,1);

static\_param\_data(2,4) = newPoint(1,2);

static\_param\_data(3,3) = I(1);

static\_param\_data(3,4) = Index\_graft(1);

break;

end

% check the start tree with the goal tree state;

[~,Index\_goal] = min(in\_distance(RRTree\_goal(:,1:2),newPoint),[],1);% Whether it is optimized here depends on the phenomenon, the memory usage is too high

% Same as above but for Goal Tree

if distanceCost(RRTree\_goal(Index\_goal(1),1:2),newPoint)<static\_param\_data(1,2) && ObstacleFree(RRTree\_goal(Index\_goal(1),1:2),newPoint,map,obstacles)

donytic\_param\_data(1,1) = false;

donytic\_param\_data(1,3) = true;

donytic\_param\_data(1,4) = true;

donytic\_param\_data(1,5) = true;

static\_param\_data(5,1) = newPoint(1,1);

static\_param\_data(5,2) = newPoint(1,2);

static\_param\_data(3,3) = I(1);

static\_param\_data(4,3) = Index\_goal(1);

break;

end

[Distance\_main,Index\_main] = min( in\_distance(RRTree\_main(:,1:2),newPoint),[], 1); % Whether it is optimized here depends on the phenomenon, the memory usage is too high

% We continue the loop if we are not within the distance threshold

if in\_distance(newPoint, RRTree\_main(Index\_main(1),1:2)) < static\_param\_data(1,2)

faildcount = faildcount+1;

continue;

end

buffer\_local = Distance\_main;

buffer\_global = RRTree\_main(I(1),4) + buffer\_local;

RRTree\_main = [RRTree\_main;newPoint I(1) buffer\_global buffer\_local];

donytic\_param\_data(1,1) = false;

break;

end

end

### Helpers/g\_rstarb.m

function [RRTree\_goal,static\_param\_data,donytic\_param\_data]=g\_rstarb(RRTree\_main,GraftTree,RRTree\_goal,static\_param\_data,donytic\_param\_data,map,obstacles)

% creat param talbe for input data

% static\_param\_data= stepsize disTh maxFailedAttempts rrstar\_area

% source(1,1) source(1,2) point\_connect(1,1) point\_connect(1,2)

% center\_point(1.1) center\_point(1,2) prev\_ahead prev-graft-ahead

% goal(1,2) goal(1,2) prev\_behing prev-graft\_behind

% point\_connectmain(1,1) point\_connectmain(1,2) point\_connect2(1,1) point\_connect2(1,2)

% donytic\_param\_data =[Graft\_ahead,Graft\_behind,ahead\_found,behind\_found];

faildcount =0;

% Loop until iteration count is reached

while faildcount <= static\_param\_data(1,3)

% Get the sample point based off rules specified by Graft-RRT

% algorithm

if rand < 0.5

sample = RandomPoint(0, map);

elseif rand < 0.75

sample = [static\_param\_data(3,1) static\_param\_data(3,2)];

else

sample = [static\_param\_data(2,1) static\_param\_data(2,2)];

end

% This is a method for acquiring Q\_new that is faster than the for loop method

buffer\_distance = in\_distance(RRTree\_goal(:,1:2),sample);

I(1) = check\_loacalvalue(buffer\_distance,RRTree\_goal,static\_param\_data(1,4));

% 0 is not allowed so we just quickly set it to 1

if I(1) == 0

I(1) =1;

end

% We now know the closest node in the goal tree

closestNode = RRTree\_goal(I(1), 1:2);

% Point the growth direction twoards the goal tree

theta = atan2(sample(1)-closestNode(1),sample(2)-closestNode(2));

% Generate Goal Tree from found trajectory (remember this is bound

% by epsilon)

newPoint = double(closestNode(1:2) + static\_param\_data(1,1) \* [sin(theta) cos(theta)]);

% If it's not obstacle free, we skip this iteration and move on

if ~ObstacleFree(closestNode(1:2),newPoint,map,obstacles)

faildcount = faildcount+1;

continue

end

% Grab's the closest point in entire Graft Tree by index to the new

% point

[~,Index\_graft] = min(in\_distance(GraftTree(:,1:2),newPoint),[],1);

% This is intended to connect to the two trees (Graft and Goal)

if distanceCost(GraftTree(Index\_graft(1),1:2),newPoint)<static\_param\_data(1,2) && ObstacleFree(GraftTree(Index\_graft(1),1:2),newPoint,map,obstacles)

donytic\_param\_data(1,2) = false;

donytic\_param\_data(1,4) = true;

static\_param\_data(5,3) = newPoint(1,1);

static\_param\_data(5,4) = newPoint(1,2);

static\_param\_data(4,3) = I(1);

static\_param\_data(4,4) = Index\_graft(1);

break;

end

% check the start tree with the goal tree state;

[~,Index\_goal] = min(in\_distance(RRTree\_main(:,1:2),newPoint),[],1); % Whether it is optimized here depends on the phenomenon, the memory usage is too high

% Same as above but for Start Tree

if distanceCost(RRTree\_main(Index\_goal(1),1:2),newPoint)<static\_param\_data(1,2) && ObstacleFree(RRTree\_main(Index\_goal(1),1:2),newPoint,map,obstacles)

donytic\_param\_data(1,2) = false;

donytic\_param\_data(1,3) = true;

donytic\_param\_data(1,4) = true;

donytic\_param\_data(1,5) = true;

static\_param\_data(5,1) = newPoint(1,1);

static\_param\_data(5,2) = newPoint(1,2);

static\_param\_data(4,3) = I(1);

static\_param\_data(3,3) = Index\_goal(1);

break;

end

[Distance\_goal,DIndex\_goal] = min( in\_distance(RRTree\_goal(:,1:2),newPoint),[], 1);

% We continue the loop if we are not within the distance threshold

if in\_distance(newPoint, RRTree\_goal(DIndex\_goal(1),1:2)) < static\_param\_data(1,2)

faildcount = faildcount+1;

continue;

end

buffer\_local = Distance\_goal;

buffer\_global = RRTree\_goal(I(1),4) + buffer\_local;

RRTree\_goal = [RRTree\_goal;newPoint I(1) buffer\_global buffer\_local];

donytic\_param\_data(1,2) = false;

break;

end

end

### Helpers/check\_globalvalue.m

function in\_path = check\_globalvalue(path,map,obstacles)

stack\_path = [path(1,1:2),0,0]; % Stored in the cache tree

tree\_length= 0;

stack\_length=1;

queue\_path =0; % Queue Head

step\_length=0;

i\_path =1;

while i\_path < length(path)

for j\_path = i\_path:length(path)

if ObstacleFree(path(i\_path,1:2),path(j\_path,1:2),map,obstacles)

queue\_path =j\_path; % Put it in the queue and delete the previous one directly

end

end

step\_length =distanceCost(path(i\_path,1:2),path(queue\_path,1:2));

% It is possible for this function to enter an indefinite loop, this

% catches that case and prevents it by forcing a queue\_path increment

if (step\_length == 0)

warning("Degraded Path - Escaping Loop Entrapment")

queue\_path = queue\_path+1;

end

tree\_length =step\_length+stack\_path(stack\_length,3);% One round of looping down to detect the length of the stack and the head of the queue

i\_path=queue\_path; % Put into the queue to directly replace the scan of the queue

stack\_path=[stack\_path;[path(i\_path,1:2),tree\_length,step\_length]];

stack\_length=stack\_length+1;

end

in\_path =stack\_path;

### Helpers/check\_globalvalueII.m

function out\_path= check\_globalvalueII(in\_path,step\_size,map,obstacles)

buffer\_tree =in\_path(:,1:2); % input the path of the final path

buffer\_tree(:,3:6)=0;

%set two tree which is the old path's children ,

%the left children is buffer\_tree(:,3:4) . the right children is buffer\_tree(:,5:6)

% what we want found path is the riht children and left children

%

buffer\_tree(1,3:4)=buffer\_tree(1,1:2);

% buffer\_tree(length(buffer\_tree)-1,5:6)=buffer\_tree(length(buffer\_tree),1:2);

i\_path = 2;

buffer\_point1=[0,0];

buffer\_point2=[0,0];

new\_length=0;

[l,c]=size(buffer\_tree);

while i\_path <=(l-1),

% Generates new point

[L,R] = line\_check(buffer\_tree((i\_path-1),3:4),buffer\_tree(i\_path,1:2),buffer\_tree((i\_path+1),1:2),map,step\_size,obstacles);

buffer\_tree(i\_path-1,5:6)=L;

buffer\_tree(i\_path,3:4)=R;

i\_path = i\_path+1;

end

buffer\_tree(l-1,5:6)=buffer\_tree(l,1:2);

new\_tree=[ buffer\_tree(1,3:4);buffer\_tree(1,5:6)];

for index = 2:l-1

new\_tree = [new\_tree;buffer\_tree(index,3:4);buffer\_tree(index,5:6)];

end

out\_path = new\_tree;

### Helpers/check\_loacalvalue.m

% buffer\_distance :the distance of every point in the tree get to the new point(sample)

% RRTree :the main tree

% area : the area of the local point

% exit\_data : return the index of the tree of the shortest distance point

function exit\_data= check\_loacalvalue(buffer\_distance,RRTree,area)

local\_father\_length=0;

local\_value=[0,0];

for i=1:length(buffer\_distance),

if buffer\_distance(i,1) <area ,

local\_father\_length = buffer\_distance(i,1)+RRTree(i,4);

local\_value=[[i,local\_father\_length];local\_value];

end

end

local\_value=sortrows(local\_value,2);

exit\_data=local\_value(2);

end

### Helpers/CheckCollision.m

function [containsValid, validPoints] = CheckCollision(points, occupancy\_grid, map\_size)

% CheckCollision The purpose function is dual purpose. It is used within

% pre-processing to determine what points (if any) in the array of points

% are valid. The returns are containsValid and validPoints. containsValid

% is a quick check (implemented primarily for ease of use when checking a

% singular point) and validPoints is an array of all points (inside of the passed in

% points) that are not in an obstacle or outside the map.

% Remove Points Exceeding X Bounds ( > map\_size)

invalidMapBoundsX = points(:, 1) >= map\_size;

points(invalidMapBoundsX, :) = [];

% Removing Points Exceeding X Bounds (< 0)

invalidMapBoundsX = points(:, 1) <= 0;

points(invalidMapBoundsX, :) = [];

% Remvoing Points Exceeding Y Bounds (> map\_size)

invalidMapBoundsY = points(:, 2) >= map\_size;

points(invalidMapBoundsY, :) = [];

% Removing Points Exceeding X Bounds (< 0)

invalidMapBoundsY = points(:, 2) <= 0;

points(invalidMapBoundsY, :) = [];

% Remove Points within Obstacles

invalidPoints = zeros(height(points), 1);

for i=1:height(points)

pointX = floor(points(i, 1) + 1);

pointY = map\_size - floor(points(i, 2));

if (occupancy\_grid(pointY, pointX) == 1)

invalidPoints(i) = 1;

end

end

points = points(~invalidPoints, :);

containsValid = height(points)>=1;

validPoints = points;

end

### Helpers/DistanceCost.m

function h=distanceCost(a,b)

h = sqrt((a(:,1)-b(:,1)).^2 + (a(:,2)-b(:,2)).^2 );

### Helpers/graft\_extend.m

function [GraftTree,static\_param\_data,donytic\_param\_data]=graft\_extend(RRTree\_main,GraftTree,RRTree\_goal,static\_param\_data,donytic\_param\_data,map,obstacles)

% creat param talbe for input data

% static\_param\_data= stepsize disTh maxFailedAttempts rrstar\_area

% source(1,1) source(1,2) point\_connect(1,1) point\_connect(1,2)

% center\_point(1.1) center\_point(1,2) prev\_ahead prev-graft-ahead

% goal(1,2) goal(1,2) prev\_behing prev-graft\_behind

% point\_connectmain(1,1) point\_connectmain(1,2) point\_connect2(1,1) point\_connect2(1,2)

% donytic\_param\_data =[Graft\_ahead,Graft\_behind,ahead\_found,behind\_found];

faildcount =0;

while faildcount <= static\_param\_data(1,3)

% Sampling Type is handled based off the currently missing portions of

% the tree determined by donytic\_param\_data as a set of states

if ~donytic\_param\_data(1,3) && ~donytic\_param\_data(1,4)

if rand < 0.5

sample = RandomPoint(0, map);

elseif rand < 0.75

sample = [static\_param\_data(2,1) static\_param\_data(2,2)];

else

sample = [static\_param\_data(4,1) static\_param\_data(4,2)];

end

end

if donytic\_param\_data(1,4)

if rand <0.5

sample = RandomPoint(0, map);

else

sample = [static\_param\_data(2,1) static\_param\_data(2,2)];

end

end

if donytic\_param\_data(1,3)

if rand <0.5

sample = RandomPoint(0, map);

else

sample = [static\_param\_data(4,1) static\_param\_data(4,2)];

end

end

% Find closest point in graft tree to sample point

[~,I] = min(in\_distance(GraftTree(:,1:2),sample),[],1);

% We now know the closest node in the graft tree

closestNode = GraftTree(I(1), 1:2);

% Point the growth direction twoards the newPoint

theta = atan2(sample(1)-closestNode(1),sample(2)-closestNode(2));

% Generate Graft Tree point from found trajectory (remember this is bound

% by epsilon)

newPoint = double(closestNode(1:2) + static\_param\_data(1,1) \* [sin(theta) cos(theta)]);

% If it's not obstacle free, we skip this iteration and move on

if ~ObstacleFree(closestNode(1:2),newPoint,map,obstacles)

faildcount = faildcount+1;

continue;

end

% Grab's the closest point in entire Start Tree by index to the new

% point

[~,Index\_main] = min(in\_distance(RRTree\_main(:,1:2),newPoint),[],1);

% check two tree whether link in the map

% first judge is for the ahead

% the second judge is for the behind t

% This is intended to connect to the two trees (Graft and Start)

if distanceCost(RRTree\_main(Index\_main(1),1:2),newPoint)<static\_param\_data(1,2) && ObstacleFree(RRTree\_main(Index\_main(1),1:2),newPoint,map, obstacles)

donytic\_param\_data(1,2) = false;

donytic\_param\_data(1,3) = true;

static\_param\_data(2,3) = newPoint(1,1);

static\_param\_data(2,4) = newPoint(1,2);

static\_param\_data(3,3) = Index\_main(1);

static\_param\_data(3,4) = I(1);

break;

end

% Grab's the closest point in entire Goal Tree by index to the new

% point

[~,Index\_goal] = min(in\_distance(RRTree\_goal(:,1:2),newPoint),[],1);

% check two tree whether link in the map

% first judge is for the ahead

% the second judge is for the behind t

% This is intended to connect to the two trees (Graft and Goal)

if distanceCost(RRTree\_goal(Index\_goal(1),1:2),newPoint)<static\_param\_data(1,2) && ObstacleFree(RRTree\_goal(Index\_goal(1),1:2),newPoint,map, obstacles)

donytic\_param\_data(1,1) = false;

donytic\_param\_data(1,4) = true;

static\_param\_data(5,3) = newPoint(1,1);

static\_param\_data(5,4) = newPoint(1,2);

static\_param\_data(4,3) = Index\_goal(1);

static\_param\_data(4,4) = I(1);

break;

end

[~,Index\_graft] = min(in\_distance(GraftTree(:,1:2),newPoint),[], 1);

% We continue the loop if we are not within the distance threshold

if in\_distance(newPoint, GraftTree(Index\_graft(1),1:2)) < static\_param\_data(1,2)

faildcount = faildcount+1;

continue;

end

GraftTree = [GraftTree; newPoint I(1)];

break;

end

end

### Helpers/in\_distance.m

function d = in\_distance(a,b)

% Eucledian Distance

d = sqrt( ( a(:,1) - b (:,1) ) .^ 2 + ( a(:,2) - b(:,2) ) .^ 2 );

end

### Helpers/line\_check.m

% input:first point center point and the last point , map , scan size;

% output: new point 1 2;

function [new\_point1,new\_point2]=line\_check(front\_point,center\_point,back\_point,map,step\_size,obstacles)

front\_step = 0;

back\_step =0;

step\_size\_count =1; %count the number of the step

% Using the center point we attempt to gradually check the line between

% two points in order to simplify its geometry if possible

new\_pointL =center\_point;

new\_pointR =center\_point;

test\_pointL =center\_point;

test\_pointR =center\_point;

% Determine distances from each side of the center point

line\_front= distanceCost(front\_point,center\_point);

line\_back= distanceCost(center\_point,back\_point);

% Identify the shortest and the longest lengths

short\_len =min(line\_front,line\_back);

long\_len = max(line\_front,line\_back);

% We have k steps to work with in the simplification process

k =long\_len/short\_len;

short\_step = short\_len/step\_size;

long\_step = short\_step.\*k;

% Point thetaI in direction to front point

% Point thetaII in direction to back point

thetaI =atan2(front\_point(1)-center\_point(1),front\_point(2)-center\_point(2));

thetaII = atan2(back\_point(1)-center\_point(1),back\_point(2)-center\_point(2));

% This determines step direction for long and short step

if line\_front == long\_len

front\_step = long\_step;

back\_step = short\_step;

elseif line\_front == short\_len

front\_step = short\_step;

back\_step = long\_step;

end

% Loop over gradually testing new points and verifying they're obstacle

% free. The simplified points are returned once we run out of steps to

% check

while step\_size\_count < step\_size ,

test\_pointL = double(center\_point(1:2)+(front\_step .\* step\_size\_count) \* [sin(thetaI) cos(thetaI)]);

test\_pointR = double(center\_point(1:2)+(back\_step .\* step\_size\_count) \* [sin(thetaII) cos(thetaII)]);

if ObstacleFree(test\_pointL,test\_pointR,map,obstacles),

new\_pointL = test\_pointL;

new\_pointR = test\_pointR;

end

step\_size\_count = step\_size\_count+1;

end

new\_point1 = new\_pointL;

new\_point2 = new\_pointR;

### Helpers/ObstacleFree.m

function [inFreeSpace] = ObstacleFree(pointOne,pointTwo, map\_size, occupancy\_grid)

%OBSTACLEFREE Takes in two arguments: Point 1 and Point 2. It generates an

% evenly space distribution of points along the line created by Point 1 and

% Point 2. The resolution of this distribution is determined by max(100,

% map\_size) where it will distribute either 100 points or map\_size points

% along the line depending on whichever is greatest. These points are then

% compared with the obstacle occupancy grid to verify the lines do not

% collide. ObstacleFree simply returns a true or false value.

lineXPoints = linspace(pointOne(1, 1), pointTwo(1, 1), max(100, map\_size));

lineYPoints = linspace(pointOne(1, 2), pointTwo(1, 2), max(100, map\_size));

line = [lineXPoints', lineYPoints'];

[containsValid, validPoints] = CheckCollision(line, occupancy\_grid, map\_size);

inFreeSpace = height(validPoints) == height(line);

end

### Helpers/rand\_center.m

function [point] = rand\_center(qsource, qgoal, map\_size, occupancy\_grid)

%GETTREE Returns the Tgraft tree discussed in the Graft-RRT algorithm (Map

% Size is used as a resolution constraint on generating center point)

% Another implementation of this process is available at: https://github.com/robot-rolle/path-plan/blob/Graft\_RRT/rand\_center.m

% We do not attempt to calculate direction and process points along that

% line. Instead we generate a series of points from start to goal and

% filter out any points within an obstacle or outside the map.

% Straight Line of Points from Start\_Waypoint to Goal\_Waypoint

% We calculate midpoint and generate two line segments based off that

% (before and after the midpoint) such that we have points closer to the

% end\_point and points closer to the start\_point. Selecting the N closest

% points to the midpoint out of those points on either segment we can then

% randomly sample for a "center" point. In the terms of the paper, our

% distance parameters would be related to maximizing the distance of a

% point from both the start and end point.

straightLineMidpoint = [((qsource(1) + qgoal(1))/2) ((qsource(2) + qgoal(2))/2)];

% By default linspace uses 100 points (maybe this should be investigated at

% lower resolutions such as map\_size or map\_size \* 2). This currently

% chooses the highest between the two in order to maintain some form of

% "scalability" on maps larger than 100 units though resolution will be 1:1

% rather than many points per unit

straightLineX = linspace(qsource(1), straightLineMidpoint(1), max(100, map\_size));

straightLineY = linspace(qsource(2), straightLineMidpoint(2), max(100, map\_size));

straightLineFromStart = [straightLineX', straightLineY'];

straightLineX = linspace(qgoal(1), straightLineMidpoint(1), max(100, map\_size));

straightLineY = linspace(qgoal(2), straightLineMidpoint(2), max(100, map\_size));

straightLineFromEnd = [straightLineX', straightLineY'];

% Remove Points within an Obstacle

[~, validStraightLineFromStart] = CheckCollision(straightLineFromStart, occupancy\_grid, map\_size);

[~, validStraightLineFromEnd] = CheckCollision(straightLineFromEnd, occupancy\_grid, map\_size);

% Calculate 25% of the number of total valid points for each segment and

% allocate that to our random sampling set

numberCenterPointsFromStart = floor(0.25\*(height(validStraightLineFromStart)));

numberCenterPointsFromEnd = floor(0.25\*(height(validStraightLineFromEnd)));

% Gets the desired points from the start points (takes them from closest to

% furthest from mid point)

centerStartPoints = zeros(numberCenterPointsFromStart, 2);

for i=1:numberCenterPointsFromStart

centerStartPoints(i, :) = validStraightLineFromStart(height(validStraightLineFromStart)-i, :);

end

% Gets the desired points from the end points (takes them from closest to

% furthest from mid point)

centerEndPoints = zeros(numberCenterPointsFromEnd, 2);

for i=1:numberCenterPointsFromEnd

centerEndPoints(i, :) = validStraightLineFromEnd(height(validStraightLineFromEnd)-i, :);

end

% Chooses a random point from the calculated portions of the segments and

% uses that as the root node of the graft tree

try

sampleSet = [centerStartPoints;centerEndPoints];

point = sampleSet(randi(height(sampleSet), 1), :);

catch

disp("Sample Set issue");

end

end

### Helpers/RandomPoint.m

function [point] = RandomPoint(a,b)

%RANDOMPOINT Generates a random (floating point) number between a and b.

% Derived from: https://www.mathworks.com/help/matlab/ref/rand.html

point = a + (b - a).\*rand(1, 2);

end

### Helpers/re\_path.m

function new\_path = re\_path(RRTree\_main,GraftTree,RRTree\_goal,static\_param\_data,donytic\_param\_data)

% creat param talbe for input data

% static\_param\_data= stepsize disTh maxFailedAttempts rrstar\_area

% source(1,1) source(1,2) point\_connect(1,1) point\_connect(1,2)

% center\_point(1.1) center\_point(1,2) prev\_ahead prev-graft-ahead

% goal(1,2) goal(1,2) prev\_behing prev-graft\_behind

% donytic\_param\_data point\_connectmain(1,1) point\_connectmain(1,2) point\_connect2(1,1) point\_connect2(1,2)

% donytic\_param\_data =[Graft\_ahead,Graft\_behind,ahead\_found,behind\_found,errtree\_found];

main\_tree=[];

graft\_tree\_ahead=[];

graft\_tree\_behind = [];

goal\_tree=[];

% If errtree\_found is true generate the tree path using the main and

% goal trees based off the prev\_ahead and prev\_graft\_behind point

if donytic\_param\_data(5),

prev = static\_param\_data(3,3);

main\_tree = [static\_param\_data(5,1) static\_param\_data(5,2)]; % Connection Point

while prev > 0,

main\_tree = [RRTree\_main(prev,1:2);main\_tree]; % Grab path to Connection Point

prev = RRTree\_main(prev,3);

end

prev =static\_param\_data(4,4);

goal\_tree = [static\_param\_data(5,1) static\_param\_data(5,2)]; % Connection Point

while prev > 0,

goal\_tree = [goal\_tree;RRTree\_goal(prev,1:2)]; % Grab path to connection point

prev = RRTree\_goal(prev,3);

end

main\_tree(end,:)=[]; % we don't want the end since it is just connection point and that is already in goal tree

new\_path=[main\_tree;goal\_tree ]; % Combine the paths

else

% If errtree\_found is not true then the tree path is generated from

% the connect points

prev = static\_param\_data(3,3);

main\_tree =[static\_param\_data(2,3) static\_param\_data(2,4)];

% Recursively build\_main tree from prev to get path from connect

% point

while prev > 0,

main\_tree = [RRTree\_main(prev,1:2);main\_tree];

prev = RRTree\_main(prev,3);

end

% Repeat the same process for the goal tree

prev = static\_param\_data(4,3);

goal\_tree =[static\_param\_data(5,3) static\_param\_data(5,4)];

while prev > 0,

goal\_tree = [goal\_tree;RRTree\_goal(prev,1:2)];

prev = RRTree\_goal(prev,3);

end

% Repeat the same process for the graft tree (we have to do it in

% two places though since it is connected on both sides of the

% tree)

graft\_path\_ahead = [static\_param\_data(2,3) static\_param\_data(2,4)];

prev = static\_param\_data(3,4);

while prev >0,

graft\_path\_ahead = [graft\_path\_ahead;GraftTree(prev,1:2)];

prev = GraftTree(prev,3);

end

graft\_path\_behind = [static\_param\_data(5,3) static\_param\_data(5,4)];

prev = static\_param\_data(4,4);

while prev >0,

graft\_path\_behind = [GraftTree(prev,1:2);graft\_path\_behind];

prev = GraftTree(prev,3);

end

% Connect Start Tree, Graft Tree, and Goal Tree form one full

% uniform path

main\_tree(end,:)=[];

% Remove connect points on graft tree as the Start and Goal trees

% will contain those for us

graft\_path\_ahead(end,:)=[];

graft\_path\_behind(end,:)=[];

new\_path = [main\_tree; graft\_path\_ahead;graft\_path\_behind;goal\_tree];

end

## Appendix 15 - README