2018

INTERACTIVE MAP MAKING FOR ROUTE PLANNING AND OBSTACLE AVOIDANCE IN AN UNSTRUCTURED OUTDOOR ENVIRONMENT

Audette, Matthew R.

Monterey, CA; Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

INTERACTIVE MAP MAKING FOR ROUTE PLANNING AND OBSTACLE AVOIDANCE IN AN UNSTRUCTURED OUTDOOR ENVIRONMENT

by

Matthew R. Audette

September 2018

Thesis Advisor: Xiaoping Yun
Co-Advisor: James Calusdian

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			*Form Approved OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** September 2018	**3. REPORT TYPE AND DATES COVERED** Master's thesis

4. TITLE AND SUBTITLE INTERACTIVE MAP MAKING FOR ROUTE PLANNING AND OBSTACLE AVOIDANCE IN AN UNSTRUCTURED OUTDOOR ENVIRONMENT	**5. FUNDING NUMBERS**
6. AUTHOR(S) Matthew R. Audette	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A	**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release. Distribution is unlimited.	**12b. DISTRIBUTION CODE** A

13. ABSTRACT (maximum 200 words)

As autonomous ground robots fulfill greater roles within the military there is a requirement for an operator to be able to quickly give minimal route-planning guidance in support of an autonomous mission. The objective of this thesis is to develop a route-planning algorithm that uses open source satellite imagery to allow a user to plot a start point, a goal point, and identify large-scale obstacles within the robot's operating area. In this thesis, we build on previous work that developed a potential field obstacle avoidance algorithm. We advance the development of the autonomous mission capability by creating a global path-planning algorithm. The algorithm uses the visibility graph and A* search method to produce the optimal path from the given start point to the goal. The navigation algorithm developed allows users to generate imagery-based obstacle maps in Google Earth Pro and successfully produces an optimal path in the form of global positioning satellite coordinates via extensive MATLAB code development. The method was evaluated on a ground robot navigating in an outdoor environment using the waypoints generated. The path-planning algorithm was successfully implemented, but due to difficulties encountered with the navigation node of the mobile robot, a complete verification was not possible. Improvements to the robot's ability to traverse over rugged terrain will make this solution more viable for a wider range of outdoor environments.

14. SUBJECT TERMS visibility matrix, visibility graph, A*, optimal route, path planning, unmanned ground robot, autonomous ground robot, autonomous ground vehicle, robot	**15. NUMBER OF PAGES** 101
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UU

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89) Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release. Distribution is unlimited.

INTERACTIVE MAP MAKING FOR ROUTE PLANNING AND OBSTACLE AVOIDANCE IN AN UNSTRUCTURED OUTDOOR ENVIRONMENT

Matthew R. Audette
Captain, United States Marine Corps
BS, Southern Polytechnic State University, 2011

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
September 2018

Approved by: Xiaoping Yun
Advisor

James Calusdian
Co-Advisor

Clark Robertson
Chair, Department of Electrical and Computer Engineering

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

As autonomous ground robots fulfill greater roles within the military, there is a requirement for an operator to be able to quickly give minimal route-planning guidance in support of an autonomous mission. The objective of this thesis is to develop a route-planning algorithm that uses open-source satellite imagery to allow a user to plot a start point, a goal point, and identify large-scale obstacles within the robot's operating area. In this thesis, we build on previous work that developed a potential field obstacle avoidance algorithm. We advance the development of the autonomous mission capability by creating a global path-planning algorithm. The algorithm uses the visibility graph and A* search method to produce the optimal path from the given start point to the goal. The navigation algorithm developed allows users to generate imagery-based obstacle maps in Google Earth Pro and successfully produces an optimal path in the form of global positioning satellite coordinates via extensive MATLAB code development. The method was evaluated on a ground robot navigating in an outdoor environment using the waypoints generated. The path-planning algorithm was successfully implemented, but due to difficulties encountered with the navigation node of the mobile robot, a complete verification was not possible. Improvements to the robot's ability to traverse over rugged terrain will make this solution more viable for a wider range of outdoor environments.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ACRONYMS AND ABBREVIATIONS

2D	two-dimensional
DOF	degrees-of-freedom
EKF	extended Kalman filter
EOD	explosive ordnance disposal
GLONASS	globalnaya navigatsionnaya sputnikovaya sistema (global navigation satellite system)
GNSS	global navigation satellite system
GPS	global positioning system
IED	improvised explosive device
INS	inertial navigation system
LIDAR	light direction and ranging
MUM-T	manned-unmanned teaming
P3-AT	pioneer 3-all terrain
PC	personal computer
POI	point-of-interest
RAS	robotic autonomous systems
ROS	robot operating system
RST	robotics system toolbox
SSH	secure shell
UGV	unmanned ground vehicle
USB	universal serial bus
UXO	unexploded ordnance

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I would like to first offer my thanks to Dr. Xiaoping Yun and Dr. James Calusdian. Dr. Yun, thank you for your guidance in the research, theory, and applications used throughout this thesis. Dr. Calusdian, thank you for all the assistance in the lab, during experimentation, and troubleshooting. I appreciate you being a constant sounding board, source of knowledge, and your unwavering enthusiasm for the work. Additionally, I would like to thank Dr. Brian Bingham for the introduction to ROS, your mentorship, and exposure to the broader Naval Postgraduate School robotics and autonomy community.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. PURPOSE AND GOALS OF THIS THESIS

The purpose of this thesis is to develop the capability for an operator to generate missions for an unmanned ground vehicle while providing as little route-planning guidance as possible to the system. The tools used for this purpose should be as intuitive and non-technical as possible so as to decrease training time and increase ease of use.

The most common mission planning tools within the military are based on satellite imagery. Satellite imagery is easy to comprehend and more up to date than traditional paper maps. It is readily available and used within the military for training and operations. When a mission planner or operator is provided with a satellite imagery-based map of their area of responsibility, they generate a mission by determining a start point, an end goal, and intermediate waypoints. Areas that cannot be traversed, such as buildings, previously known obstacles, or areas that are off limits, are identified and removed from possible routes. This same method should be used when planning a mission for an autonomous ground vehicle. If an operator were provided an imagery-based map, he should be able to identify for the robot a start point, a goal, and known large-scale obstacles in the area of responsibility, and the robot should be able to plan its own route and successfully navigate to the goal.

In this research, we seek to develop a platform-agnostic set of tools that are capable of identifying waypoints determined by an optimal route-planning algorithm that a robot can then use to navigate to its destination. A "global map" of a pre-determined area of responsibility containing large-scale fixed obstacles that are unlikely to change, such as buildings, unnavigable terrain, and areas that are off limits for autonomous vehicles should be simple and quick to generate. When paired with a route-planning algorithm and a local obstacle avoidance algorithm, this combination will provide a robust navigation capability for an autonomous mobile robot. The robot should be able to navigate the given mission in an unstructured outdoor environment by performing global path planning as well as local

obstacle avoidance. Possible use cases for this capability would be autonomous patrolling, logistical support, or missions in support of humanitarian aid/disaster relief type operations.

B. MOTIVATION

The future battlefield is increasingly autonomous. Increased technological capability, decreases in the cost of fielding autonomous systems, and an increasingly complex battlefield supports the transition of certain dangerous, dirty, or dull tasks from human to machine. Emphasis is being placed on this by the highest levels of the Marine Corps. The Marine Corps Operational Concept calls for leaders to "refine the concept of manned-unmanned teaming (MUM-T) to integrate robotic autonomous systems (RAS) with manned platforms and Marines" [1].

The predominance of unmanned ground vehicles (UGVs) has increased exponentially. Technology journalist P. W. Singer explains that, "When U.S. forces went into Iraq, the original invasion had zero robotic systems on the ground. By the end of 2004, the number was up to 150. By the end of 2005, it was up to 2,400. By the end of 2006, it had reached the 5,000 mark and growing. It was projected to reach as high as 12,000 by the end of 2008" [2]. These unmanned systems were primarily found in the most dangerous of tasks: countering improvised explosive devices (IED) and unexploded ordnance (UXO). These systems, however, lacked autonomous capabilities, and while a typical soldier refers to them as a "robot," the engineers producing them ensured that the nomenclature attached was "remotely operated vehicle."

The Marine Corps is placing greater emphasis on autonomous ground platforms to get away from the "one operator, one robot" model to "one operator, many robots" in order to increase the lethality of the individual warfighter [3]. The 2018 Marine Corps Science and Technology Strategic Plan calls for the

> [development of] affordable technologies to enhance effective and efficient employment of ground robotics. Focus on improving capabilities while reducing training and operating requirements of user Marines. Fully autonomous vehicles are not necessarily the goal. Technologies that enable effective "supervised autonomy" by the Marine user, to include teleoperation, machine vision, perception, obstacle avoidance, convoy

following, and the ability to self-navigate pre-planned routes are desired capabilities. [4]

The tactical level Marine Corps is the target audience for manned-unmanned teaming, as can be seen the artist's rendition in Figure 1. Autonomous systems are moving out of the peripheral roles of Explosive Ordnance Disposal (EOD) and route clearance support and into the building blocks of the Marine combat capability. The Marine Corps Warfighting Laboratory has conducted MUM-T tests with 3rd Battalion, 5th Marine Regiment with several autonomous platforms filling various roles within the infantry squad, a test of which can be seen in Figure 2 [5].

Figure 1. The Marine Corp's Strategic Vision for Manned-Unmanned Teaming. Source: [4].

Figure 2. Marines with 3/5 Experiment with MUM-T.
Source: [5].

C. PREVIOUS WORK

This thesis research falls into a line of research conducted in the Naval Postgraduate School Electrical and Computer Engineering's robotics laboratory. The ultimate goal is to produce a robot that can travel from any one location within an area of responsibility to another while traversing both indoor and outdoor environments. The Naval Postgraduate School campus serves as a testbed for experimentation and evaluation. An example of the desired navigation capability is a robot traveling from Building 436, the Police Service building, to the fourth floor of Spanagel Hall. In order for a robot to do this autonomously, it has to navigate a variety of environments. The work done in the course of this thesis builds directly off earlier research that developed an obstacle avoidance algorithm for this robotic platform [6]. The previously developed artificial potential field-based obstacle avoidance algorithm successfully navigates around local obstructions within the path of the robot. In this thesis, we extend the capability of the same robotic platform by providing a higher-level path-planning algorithm that advances the existing capabilities.

II. DESCRIPTION OF HARDWARE AND SOFTWARE SYSTEMS

In the introduction of his benchmark book on robot motion planning, Stanford roboticist Jean-Claude Latombe describes a robot as "a mechanical device … equipped with actuators and sensors under the control of a computing system" [7]. By its very nature, a robot is a system that crosses the disciplines of mechanical engineering, electrical engineering, and computer science. What follows is a brief description of the robot's hardware and software in order to describe the system's capabilities.

A. HARDWARE

The hardware in this robotic system was selected based upon an ability to function in the outdoor environment. Such a system requires a rugged chassis; a light weight, low power computer; and appropriate sensors to navigate. The subsystems also require robust software support.

1. Omron Adept MobileRobots Pioneer 3-All Terrain

The mobile robot platform used in this research was the Pioneer 3-All Terrain, or P3-AT. The base platform is shown in Figure 3. Adept MobileRobots describes it as "a small four-wheel, four-motor skid-steer robot ideal for all-terrain or laboratory experimentation" [8]. The reinforced pneumatic tires give it the ability to traverse flooring, asphalt, sand, gravel, and dirt. The platform uses a skid steering drive style that gives a zero-turn radius and a swing radius of 34 cm. It is powered by up to three 12-V lead acid batteries that are hot-swappable to allow for continuous operation. The onboard microcontroller has three serial expansion ports that allow for additional sensors, processors, and computers to be added to the system to increase the robot's capabilities. Also included are a sonar array consisting of 16 front and rear mounted sonar sensors, front and rear segmented bumper arrays, and an emergency stop button that disables the drive motors [9].

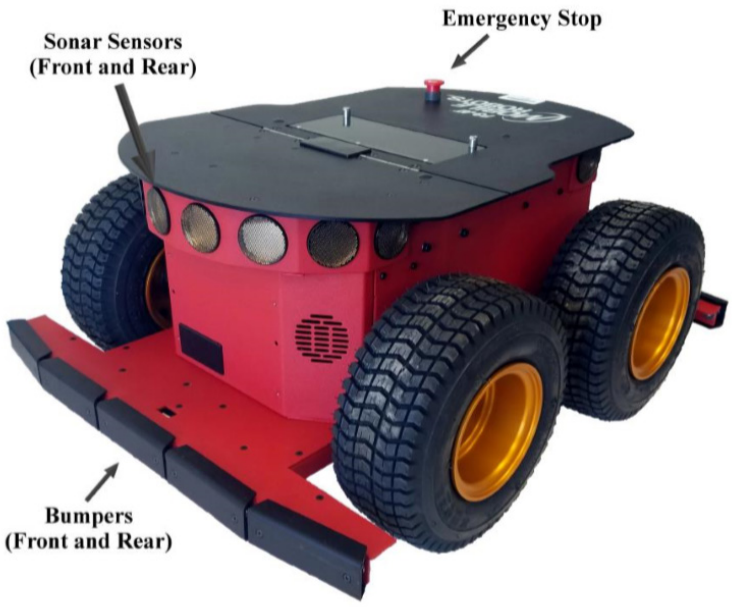

Figure 3. The P3-AT Robot. Source: [6].

2. SlimPRO SP675P Mini PC

The entirety of the processing on this mobile robot was done on the onboard SlimPRO Mini Personal Computer (PC) running the Ubuntu 14.04 Long Term Support (LTS) operating system. Details of the computer were sourced from the vendor's website [10]. According to the site, the computer has a small form factor, just 14.6 × 25.4 × 4.2 cm, and ample processing power with an Intel Pentium Central Processing Unit (CPU). The computer runs off of 12-V direct current and has a low power consumption of 30 W at normal use. The computer has six universal serial bus (USB) 2.0 ports, as seen in Figure 4, that allow for peripheral sensors and interfaces to be added. These features make it ideal for mobile robotics applications. The SlimPRO PC also has Wi-Fi capabilities which allows for remote interfacing while testing. Typically, the SlimPRO's command line interface was accessed by secure shell (SSH) with a remote laptop via a Wi-Fi router.

Figure 4. The SlimPRO SP675P Mini PC. Source: [10].

3. Sensor Suite

A thorough breakdown of the individual sensors installed on this robotic platform can be found in [6]. A brief survey of the specific hardware used in path planning and navigation follows.

The primary sensor used in the obstacle avoidance algorithm is the Hokuyo UTM-30LX Scanning Laser Rangefinder shown in Figure 5. This light imaging, detection, and ranging (LIDAR) module has a detection range from 0.1 to 30 m. The detection envelope is 270° with an angular resolution of 0.25° per step with a scan time of 25 ms per scan. This LIDAR module is a two-dimensional (2D) sensor. The LIDAR interfaces with the SlimPRO via USB2.0 [11].

Figure 5. The Hokuyo UTM-30LX Scanning Laser Rangefinder.
Source: [11].

The primary means of localization for the mobile robot is the LORD MicroStrain 3DM-GX5-45 global navigation satellite system (GNSS) / inertial navigation system (INS). The MicroStrain GNSS/INS is detailed by its datasheet as a high-performance, industrial grade sensor that is capable of utilizing either the GPS, GLONASS, BeiDou, or Galileo navigation satellite constellations [12]. In addition to the high performance GNSS capability, the MicroStrain provides the robot with nine degrees-of-freedom (DOF) inertial measurements. It provides triaxial magnetometer, gyroscope, and accelerometer data that can be used by software aboard the robot to provide localization calculations. The MicroStrain, seen in Figure 6, has two on-board processors that run an Extended Kalman Filter (EKF) in order to automatically provide more accurate estimations of the system's position, velocity, and attitude [12]. The outputs of the EKF are then transmitted to the onboard computer via USB serial interface.

Figure 6. The LORD MicroStrain GNSS Aided INS. Source: [12].

The robot chassis has built-in sonar and collision detection bumpers. Also mounted on top of the chassis is a webcam used solely for the purposes of recording experiments. These sensors were not used in the implementation of the navigation algorithm. The entire ground robot system as used throughout this research can be seen in Figure 7.

Figure 7. P3-AT with Sensor Suite as Used in Experimentation. Source: [6].

B. SOFTWARE

When selecting the software tools to use, the primary goal was to choose non-proprietary software in order to allow easer implementation of the code on other robot platforms. The second consideration was that any software used should be robust with a wide user base within the scientific and engineering community in order to ensure that the tools developed continue to function for years to come.

The software piece of this mobile robot application seeks to use readily available software suites such as MATLAB, Robotics Operating System (ROS), and free to use global imagery data with embedded GPS coordinates with Google Earth Pro.

1. MATLAB

MATLAB is a programming platform whose primary users are engineers and scientists. It is a robust programming tool that is capable of data analysis, algorithm development, and creating mathematical models [13]. It has a multitude of developer toolboxes that allow it to interface with third-party software. MATLAB was chosen as the primary development tool for designing the route-planning algorithm because of its ease of use, ability to iteratively troubleshoot code before implementation aboard the robot, and its ability to interface with third-party software through toolkits.

Throughout the course of this research the Robotic Systems Toolkit (RST) and the `read_kml` toolbox were used. The RST is a "system toolbox [that] provides an interface between MATLAB® and Simulink® and the Robot Operating System (ROS) that enables you to test and verify applications on ROS-enabled robots" [14]. It provides the interface between the route-planning algorithm and the obstacle avoidance algorithm developed in MATLAB and the Robot Operating System network which controls the sensors and actuators of the robot. The `read_kml` toolbox is a MATLAB script developed by Amy Farris of the United States Geological Survey. It reads in `.kml` files as a `string` and parses out the stored GPS coordinates [15].

2. Robot Operating System

The Robot Operating System (ROS) is "a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behavior across a wide variety of robotic platforms" [16]. It is free to use and has a large amount of third-party support. Throughout the course of this research the version known as ROS Indigo was used. ROS functions as a type of conceptual plumbing for different pieces of code associated with different sensors and processing capabilities. ROS has a very specific nomenclature for its network. Nodes are processes within ROS that perform computations and communicate messages. A message is a data structure that is transmitted from a node to a topic. ROS topics are repositories for messages where nodes can either subscribe to receive information from the node or publish messages to it. A visual representation of an example ROS network can be seen in Figure 8. ROS topics are represented by green rectangles, ROS nodes are shown in blue ovals, and how the messages publish and subscribe to topics is shown by the direction of the arrows.

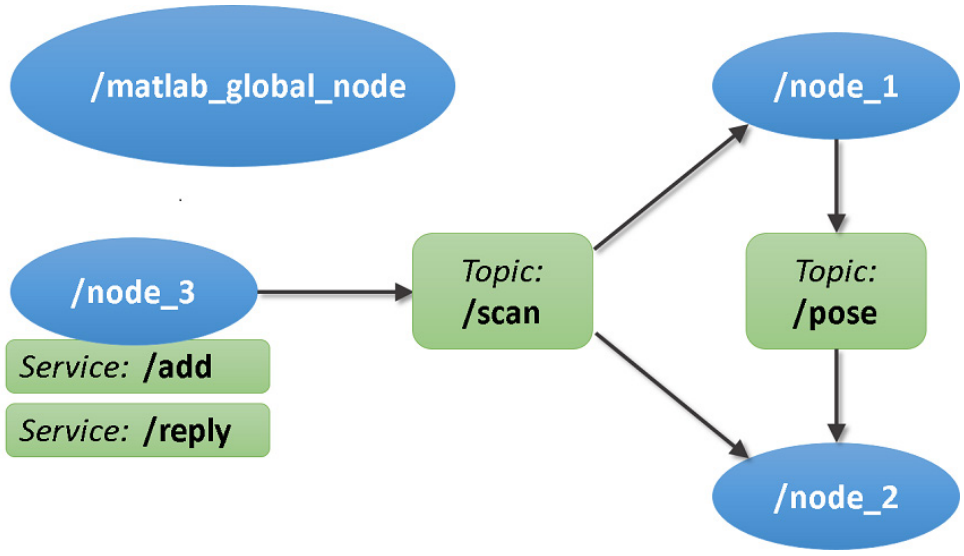

Figure 8. An Example ROS Network. Source: [17].

ROS has several powerful data processing and visualization tools built in. A ROS `.bag` file stores the message data published to topics so that the data can be played back or analyzed at a later date [18].

3. Google Earth Pro

Google Earth Pro is a free repository of imagery that allows users to "to view and use a variety of content, including map and terrain data" [19]. It was selected as the primary source of mapping imagery for this research because of its embedded GPS coordinates and intuitive ability to draw polygons, waypoints, and save map data as a `.kml` file format. Google Earth Pro is also compatible with both ROS and MATLAB through one of several free third-party software toolkits. The exact release used in the course of this research was Google Earth Pro 7.1.8.3036 (32 bit). Most users are familiar with its interface. Google Earth Pro has the built-in ability for the user to create map overlays of polygons, lines, and waypoints with annotations and save them in the common `.kml` file format. This ability to create overlays can be leveraged to easily map obstacles and port them into another program for route planning.

C. SUMMARY AND INTEGRATION

All processing took place on the P3-AT's onboard computer. In order to issue commands to the robot to run programs a secure shell script (SSH) was used. Access to the SlimPRO's command line was accessed through a laptop via a shared Wi-Fi network, as shown in Figure 9.

Figure 9. Robotic System Used in Experimentation

THIS PAGE INTENTIONALLY LEFT BLANK

III. DESCRIPTION OF ROADMAP DEVELOPMENT

Mobile robot navigation is broken down into two constituent parts: global path-planning and local obstacle avoidance. Global path planning consists of identifying a clear path that allows the robot to reach the assigned goal. Local obstacle avoidance plays the role of modifying the path of the robot so as to avoid collisions [20]. While a priori map-making is common with robots in structured lab environments and has a long history in mobile robotics research, it is an unrealistic assumption for a real-world environment that all obstacles will be known beforehand to make an accurate map. The environment that an outdoor robot operates in will be largely unstructured and changing. There are, however, some large-scale constants in an area that a robot will operate in. Buildings, terrain that is unnavigable due to the robot's specific platform such as stairs, or off-limits areas, are unlikely to change on a day-to-day basis. By mapping these large obstacles a path-planning algorithm can provide a generalized optimal solution for a long distance path and rely on a local obstacle avoidance algorithm to navigate around small scale, moving, or unmapped obstacles.

A. CONFIGURATION SPACE

In order to express the relationship between a robot and its environment, a standard naming convention has been utilized to describe specific configurations and orientations of objects within a space.

The standard notation for the configuration space is detailed in [7]. The representation for a robot is A. The environment that the robot operates in is called the workspace and is denoted by W. The workspace W is a Euclidean space that in this instance of research is a two-dimensional space. Workspaces for air or undersea systems are generally three-dimensional spaces. The coordinate frame attached to W is fixed and denoted as F_W. The robot A has attached to it a moving coordinate frame F_A that is used to describe the location of the robot within W. Obstacles within the workspace are denoted and numbered as B_i. The configuration q of robot A is a specific position (x, y) and

heading θ within W and is written $A(q)$. The configuration $A(q)$ has a specific position and orientation of F_A within F_W. The configuration space of robot A is the space C which contains all possible configurations of A. A visual representation of the workspace, coordinate frames, and a robot configuration is shown in Figure 10.

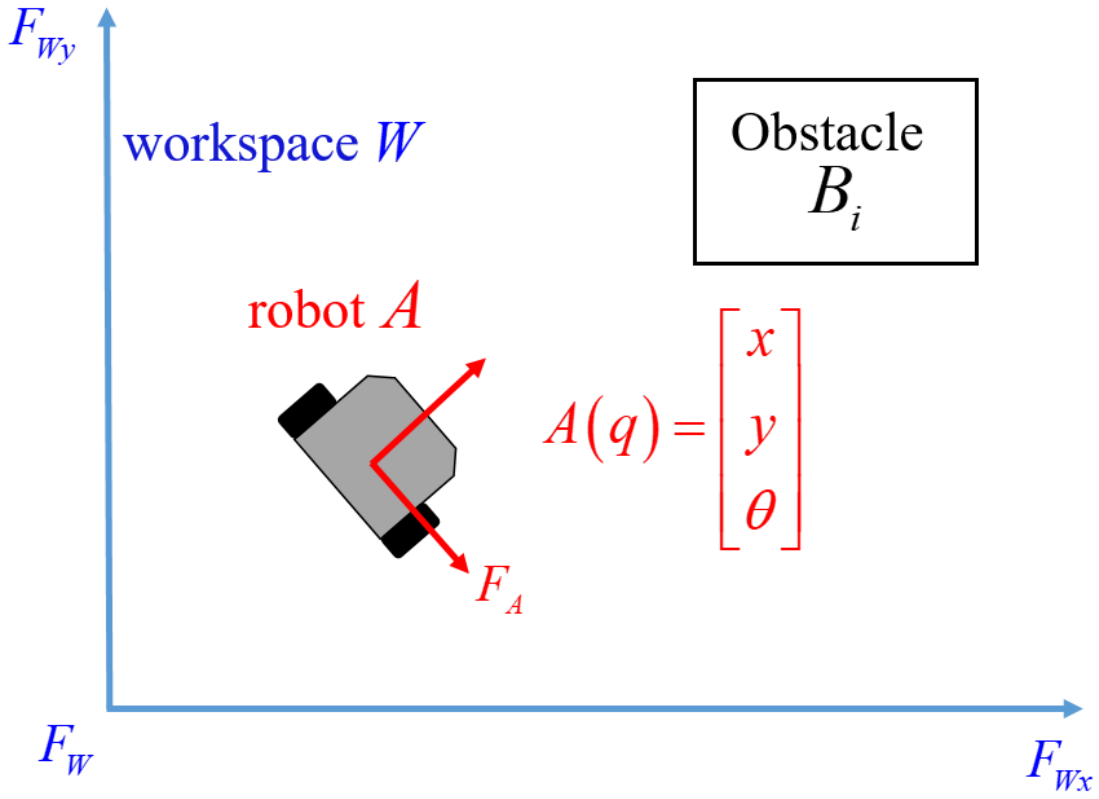

Figure 10. Representation of Workspace and the Configuration of Robot A

As explained in [7], obstacles within the configuration space occupy regions in which the robot A cannot occupy. The space occupied by the individual obstacles can be described as $B_i, i = 1,\ldots,n$ within the workspace W by mapping them to the space C to form a region

$$CB_i = \{q \in C \mid A(q) \cap B_i \neq 0\}. \quad (1)$$

The union of all the spaces in which an obstacle occupies is called the $C_{obstacle}$ region or an "obstacle field." It is mathematically defined as

$$C_{obstacle} = \bigcup_{i=1}^{n} CB_i. \tag{2}$$

The space within C that is free of obstacles and is navigable by robot A is called the free space. Free space is defined as

$$C_{free} = C - \bigcup_{i=1}^{n} CB_i. \tag{3}$$

Any configuration of the robot within free space is called a free configuration [7].

B. PATH-PLANNING APPROACHES

Path planning is the task of finding a path τ from an initial configuration q_{init} to a goal configuration q_{goal} that is within C_{free} [7]. There are several methods for finding a path within a workspace W in which $C_{obstacle}$ is defined. The three most common path-planning algorithms are the visibility graph, Voroni diagrams, and cell decomposition. Each has their own strengths and weaknesses.

The visibility graph is one of the earliest path-planning methods. In their 1979 article detailing the visibility graph method, Lozano-Perez and Wesley attribute the method to the legendary roboticist and computer scientist Nils Nilsson [21]. An example of a visibility graph can be seen in Figure 11. The method consists of making a graph of nodes that consist of all vertices within $C_{obstacle}$ as well as q_{init} and q_{goal}. Any two nodes that have unobstructed paths between them that lie within C_{free} are joined by a line called an edge.

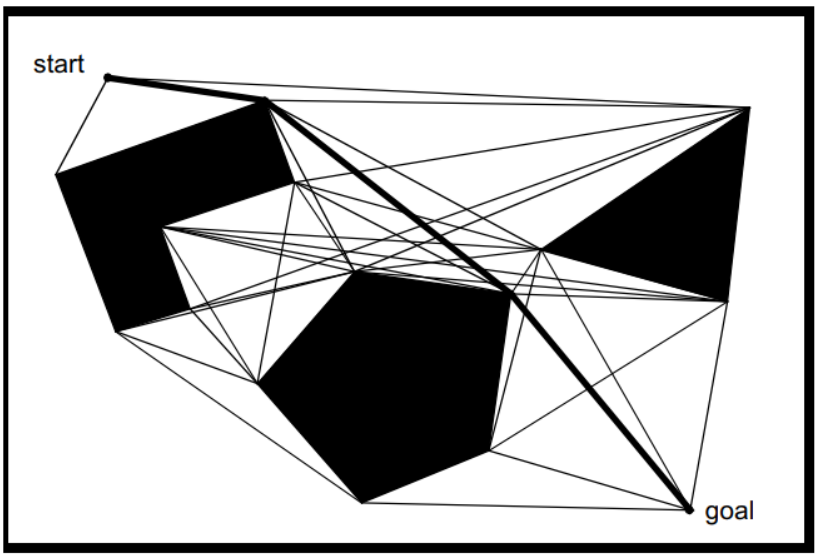

Figure 11. Visibility Graph Example. Source: [20].

Adjacent edges of an obstacle are unobstructed, so these are also considered edges within the visibility graph. The edges between the nodes constitute the shortest distance between those two points. As such, when a visibility graph is paired with an optimal path finding method, if a path is found between q_{init} and q_{goal}, the resulting path is the most efficient.

The drawback of the visibility graph method is that as more obstacles are introduced to the environment, the graph becomes more populated with edges. The path search method, therefore, becomes slower due to the increased computational effort required. Another more serious shortcoming of this method is that the planned paths, while efficient, take the robot very close to the obstacles in W. A common workaround for this is to artificially increase the size of the obstacle by the largest radius of the robot. This method of artificial expansion is unnecessary if the mobile robot is capable of local obstacle avoidance.

The second method, known as the Voroni diagram, takes the opposite approach of the visibility graph. It seeks to maximize the distance between the robot and the obstacles by finding nodes that are equidistant from all obstacles. The paths that are generated are either straight lines or parabolic in shape, as shown in Figure 12. Computing these

equidistant trajectories is more computationally intensive than visibility graphs. The tradeoff is that the robot minimizes chances of collision with an obstacle if all obstacles are known and plotted beforehand.

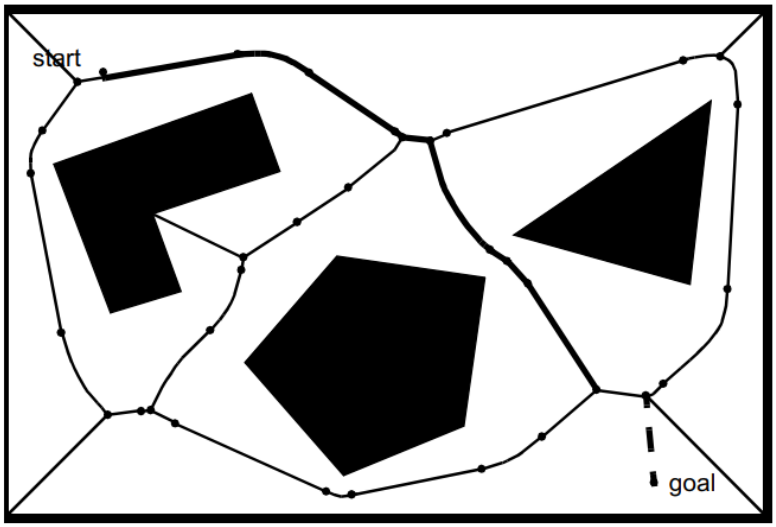

Figure 12. Voroni Diagram Example. Source: [21].

The third approach is known as cell decomposition. Cell decomposition consists of breaking the space C_{free} into simple regions called cells and determining which cells are adjacent to one another [7]. Adjacent cells are linked to form a "connectivity graph," which can be seen in Figure 13. A search method is then used to plan the path between the cells containing q_{init} and q_{goal}. There are several methods used to determine how to build the cells, what points within the cells to connect, and how to construct the connectivity graph. The advantage of cell decomposition is that, if desired, they achieve coverage of the space C_{free} in that a path can be planned so that a robot will pass through all the established sectors in the graph. Certain subtypes of cell decomposition, such as variable-size approximate cell decomposition, vary their complexity levels to match their environment [20].

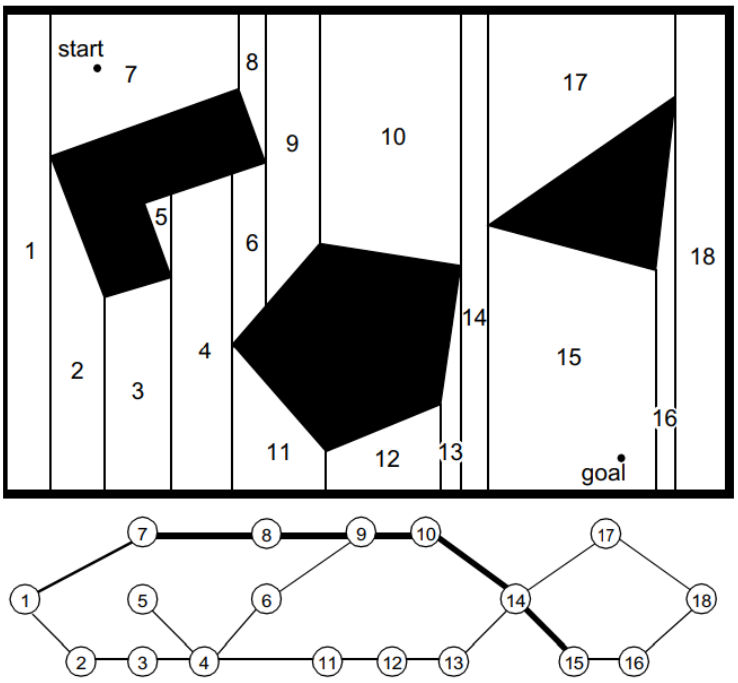

Figure 13. Cell Decomposition and Connectivity Graph Example. Source: [21].

Of the three path-planning methods considered, the visibility graph method was selected for this research because of its ease of implementation. Additionally, it was well suited for integration with the existing obstacle avoidance algorithm developed previously for local maneuvering. This already established work paired with the efficiency and ease of implementation of the visibility graph led to that method being selected for the path-planning algorithm. The Voroni diagram's advantage of keeping a robot away from obstacles comes at the price of an inefficient, non-optimal path and provides a duplicate capability to the local obstacle avoidance algorithm. Cell decomposition's advantage of providing absolute coverage of an area at the expense of providing a suboptimal path was not a requirement for this application.

C. VISIBILITY GRAPH METHOD

The visibility graph planning method consists of three steps. First, a visibility graph must be constructed from a provided field of obstacles, a q_{init}, and a q_{goal}. Next, a search

method must be used to find a path between q_{init} and a q_{goal}. Last, if a path is not found, the algorithm must indicate a failure [7].

Obstacles within C are represented as polygons [22]. Each polygon is represented by its vertices in the form of *x* and *y* coordinates, as shown in Figure 14.

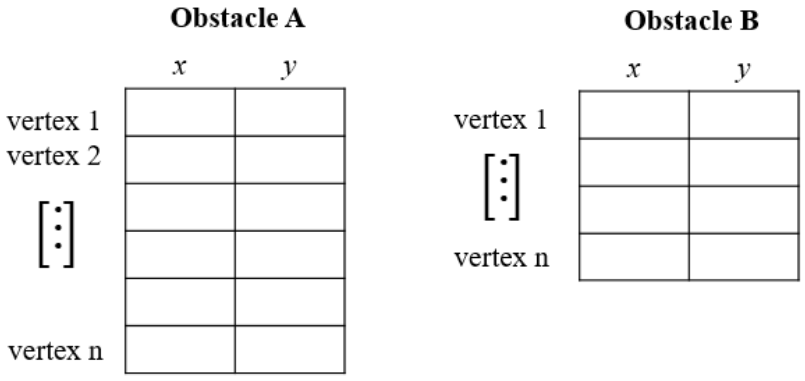

Figure 14. Obstacles Represented as an Array of Vertices

When each obstacle is described by an array of its vertices, two nodes n_1 and n_2 are selected. These two nodes are checked to see if they are connectable by an edge that exists entirely within C_{free}. This is done by checking if the line segment between n_1 and n_2 runs through the interior of any of the obstacles, as shown in Figure 15.

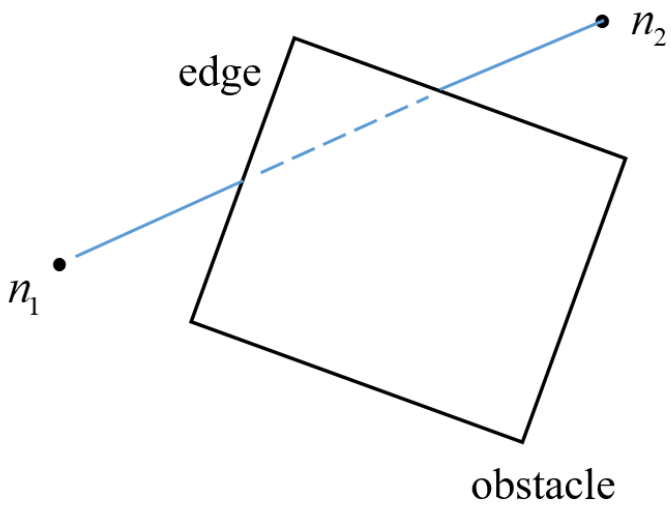

Figure 15. Checking for Visibility Between n_1 and n_2

If the line segment runs through the interior of any obstacle, n_1 and n_2 are not connected. The line segment between n_1 and n_2 is checked for intersection against every edge within the obstacle field $C_{obstacle}$. When n_1 and n_2 have been checked against every edge within $C_{obstacle}$ and deemed that the edge between them is visible or not, n_2 is replaced with n_3 and the process is repeated.

The method to mathematically determine the intersection between two line segments is explained in [22]. The two end points of each line are represented as two-dimensional vectors, as shown in Figure 16.

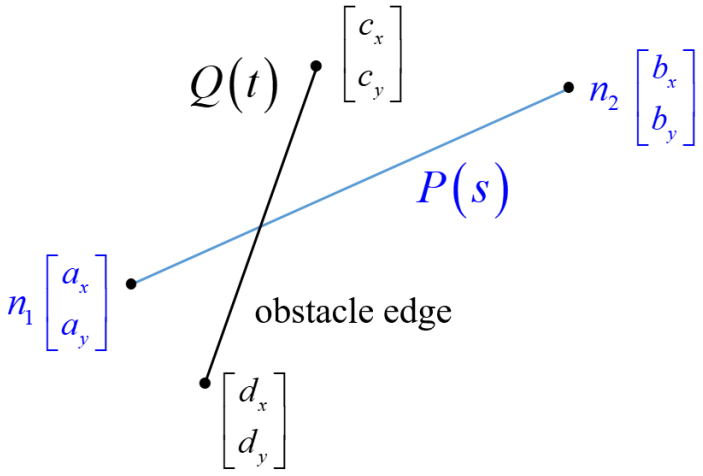

Figure 16. The Representation of Two Lines as Four Vectors

The first line containing points a and b is defined by the function

$$P(s)=\begin{bmatrix} p_x(s) \\ p_y(s) \end{bmatrix}=(1-s)\begin{bmatrix} a_x \\ a_y \end{bmatrix}+s\begin{bmatrix} b_x \\ b_y \end{bmatrix}, \tag{4}$$

where the parameter s is $0 \le s \le 1$. The line containing the points c and d is defined by the function

$$Q(t)=\begin{bmatrix} q_x(t) \\ q_y(t) \end{bmatrix}=(1-t)\begin{bmatrix} c_x \\ c_y \end{bmatrix}+t\begin{bmatrix} d_x \\ d_y \end{bmatrix}, \tag{5}$$

where the parameter t is $0 \le t \le 1$. The two line segments intersect if there are values of s and t such that $P(s)=Q(t)$; i.e.,

$$(1-s)\begin{bmatrix} a_x \\ a_y \end{bmatrix}+s\begin{bmatrix} b_x \\ b_y \end{bmatrix}=(1-t)\begin{bmatrix} c_x \\ c_y \end{bmatrix}+t\begin{bmatrix} d_x \\ d_y \end{bmatrix}. \tag{6}$$

Equation (6) can be arranged such that

$$\begin{bmatrix} b_x-a_x & c_x-d_x \\ b_y-a_y & c_y-d_y \end{bmatrix}\begin{bmatrix} s \\ t \end{bmatrix}=\begin{bmatrix} c_x-a_x \\ c_y-a_y \end{bmatrix}. \tag{7}$$

The matrix and vector in Equation (7) can be substituted with the symbols Φ and ϕ, respectively, to give

$$\Phi\begin{bmatrix} s \\ t \end{bmatrix} = \phi. \tag{8}$$

Solving Equation (8) for s and t will tell if an intersection exists if both s and t are between 0 and 1.

There exist special cases for this rule. If the matrix Φ is singular, the line segments $P(s)$ and $Q(s)$ are parallel [22]. An additional special case is that if the vector ϕ is in the column space of Φ, meaning that $\phi \in sp\{\Phi\}$, the two line segments belong to the same line, as shown in Figure 17. In cases like this, the returned values of s and t are infinity, which represents that the lines overlap at an infinite number of points.

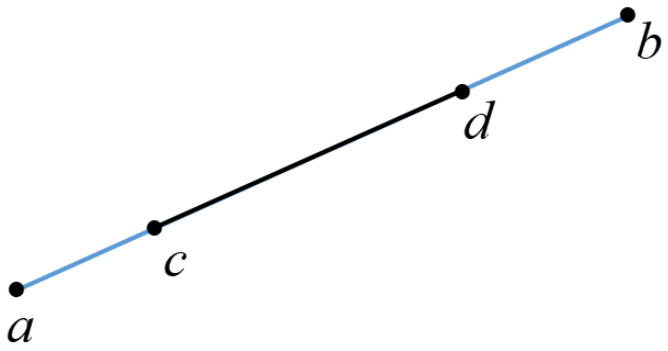

Figure 17. Lines $P(s)$ and $Q(t)$ Overlap When ϕ is in the Column Space of Φ

The last of the special cases occurs if the vector ϕ is not in the column space of Φ, or $\phi \notin sp\{\Phi\}$. This means two line segments are parallel without intersection, as shown in Figure 18. In cases like this, the calculated values of s and t are infinity despite the two lines not overlapping.

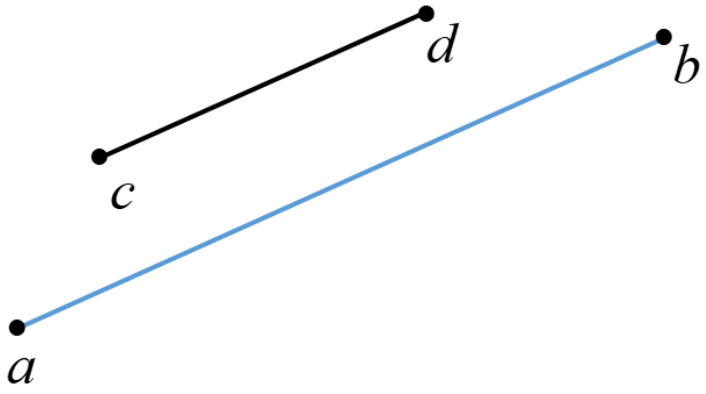

Figure 18. Lines $P(s)$ and $Q(t)$ Are Parallel When ϕ Is Not Contained Within the Column Space of Φ

The issue with this is that these lines do not intersect at any points, but mathematically the same results are returned as if they intersected at an infinite number of points. There must be an additional check within the code to verify if s and t are infinity, which of the two special cases the lines fall into. This is done by checking that column space of the vector ϕ is not in the column space of Φ [22]. Without this check, paths that are free of obstacles but are parallel with edges of obstacles elsewhere in the visibility graph are labeled as obstructed. A side-by-side comparison of a visibility matrix can be seen in Figure 19. The visibility graph on the left does not include this check of column spaces and leaves out several key edges. The visibility graph on the right includes this check and returns that these parallel paths are in fact free of obstacles.

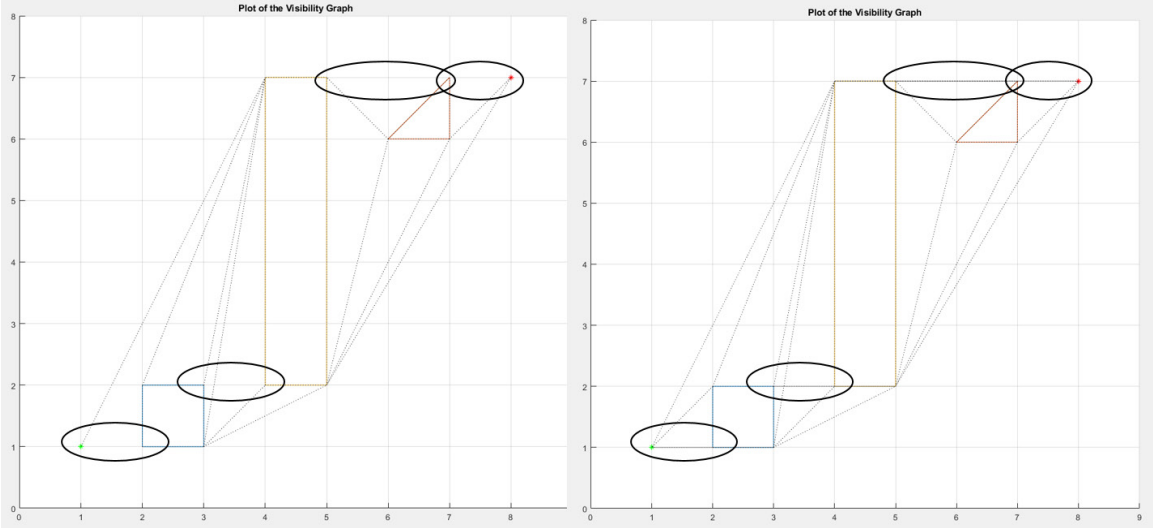

Figure 19. Visibility Graphs Without and With Checking for Column Space

A visibility graph is a tool that is easily translatable by a human. This kind of representation, however, is not easily searchable or addressable by a computer. In order to represent a visibility graph in a searchable way a visibility matrix is created. A visibility matrix is a symmetric matrix in which each row and column index represent an individual node within the obstacle field $C_{obstacle}$, as illustrated in Figure 20. Each space containing a "0" denotes that between the node in row *m* and column *n* there is no visibility. Conversely, a "1" in the space indicates that a clear path between row *m* and column *n* exists. The diagonal of the matrix is null values as it would relate a vertex of an obstacle to itself. A demonstration of how to read a visibility matrix is provided in Figure 21. Following the highlighted rows and columns of the matrix shows that node two is visible from nodes one, k, q_{init}, and q_{goal}.

	Node 1	Node 2			Node k	q_{init}	q_{goal}
Node 1	X	1	0	1	0	1	0
Node 2	1	X	0	0	1	1	0
	0	0	X	1	0	0	0
	1	0	1	X	1	0	1
Node k	0	1	0	1	X	0	1
q_{init}	1	1	0	0	0	X	0
q_{goal}	0	0	0	1	1	0	X

Figure 20. An Example of a Visibility Matrix

	Node 1	Node 2			Node k	q_{init}	q_{goal}
Node 1	X	1	0	1	0	1	0
Node 2	1	X	0	0	1	1	0
	0	0	X	1	0	0	0
	1	0	1	X	1	0	1
Node k	0	1	0	1	X	0	1
q_{init}	1	1	0	0	0	X	0
q_{goal}	0	0	0	1	1	0	X

Figure 21. Reading a Visibility Matrix

D. A* SEARCH METHOD

For all path-planning methods that produce roadmaps or decision trees for a robot to navigate, a way to determine which discrete path to take is needed. There are path-planning methods that return a result quickly, but the result may or may not be optimal. There are search methods that are guaranteed to return the optimal path, but they do so by performing an exhaustive search which is time intensive [20]. The A* search method, however, returns an optimal path without performing an exhaustive search. It does so by assigning a cost to each node that considers both the distance needed to travel to a node and the distance remaining to the goal and prioritizing the search by the lowest cost path.

The A* search method was developed by Hart, Nilsson, and Raphael [23]. It produces a path by taking all the nodes within a roadmap and assigning them a value called a *cost.* This cost is calculated by a *cost function* $f(q)$ and is used in determining the optimal path. Each node within an A* search has certain parameters. Each node has a list of neighbors called "successors," a "parent node" that is the parent node with the lowest cost, a cost $f(q)$, a movement cost $g(q)$, and a heuristic cost $h(q)$. An example roadmap is shown in Figure 22 with movement costs in blue and heuristic costs in red.

The movement cost $g(q)$ is associated with the path traveled to get from one node to another. It can be calculated by the length of the path, the difficulty of traveling it, or both. In the course of this research the distance from one node to another was used for the movement cost. The most commonly used heuristic cost $h(q)$ is the straight line distance from the node to the goal. In this research, the straight line distance for testing (x, y) coordinates was the Euclidean distance. When using latitude and longitudinal coordinates generated by GPS, the straight line distance was calculated using the haversine great circle distance.

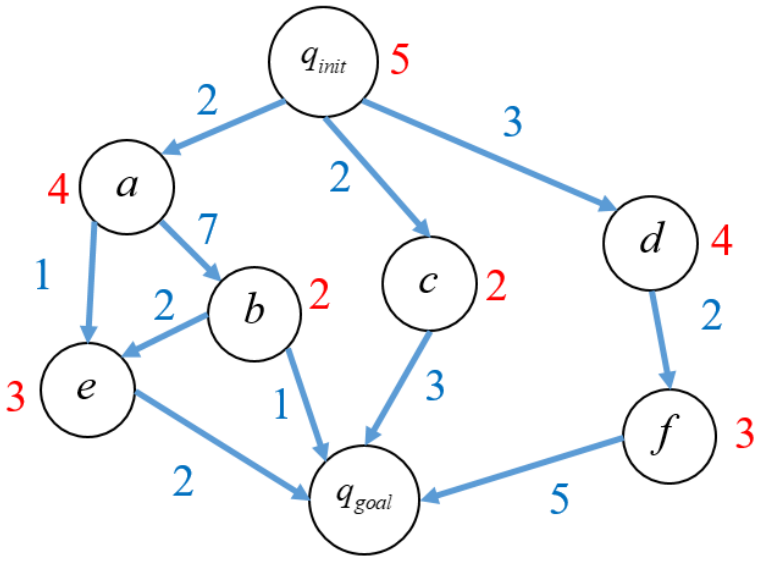

Figure 22. Example Roadmap

Latombe explains the order of operations for an A* search in [7]. At the start of the search method all the nodes are organized into two lists. The *open list* stores all nodes prior to them being sorted and assigned a cost $f(q)$. The closed list starts empty and is populated one node at a time at the end of each iteration of the search method.

At the start of the function, q_{init} is assigned a cost of zero, all other nodes are assigned a cost of infinity, and all nodes are placed on the open list. At the top of each iteration, the open list is sorted in ascending order by cost value. On the first iteration this places q_{init} at the top. The node at the top of the open list is referred to as q. At the end of each iteration q is removed and placed onto the closed list. The function then repeats itself until q_{goal} is at the top of the open list.

Each iteration of the A* search method starts with sorting the open list by the cost value assigned to each node. The top node on the list is q. Each of $q's$ successors have their parent value set to q. This parent value is used at the end of the search function to trace the shortest path from q_{goal} backwards to q_{init}. Then each of $q's$ successors has its cost calculated. The cost function is

$$f(q) = g(q) + h(q). \tag{9}$$

The movement cost $g(q)$ is calculated by summing the shortest total path cost from the start node to the successor. In the case of *node e* shown in Figure 22, the movement cost is three via *node a* and q_{init} instead of 11 via *nodes b, a*, and q_{init}.

When the cost functions for the successor nodes have been calculated, each successor is checked to see if it is already on the closed list. If the successor is on the closed list and the current calculated cost function is lower than the cost currently saved on the closed list, the closed list value is replaced with the lower cost function and corresponding parent node. This ensures that if a new, lower cost path is found to a node, the optimal path reflects this. Then each node is checked to see if it is on the open list. If the value the node holds on the open list is higher than the current iteration of the cost function, the node's cost is changed to the lower calculation, and the parent node is changed to reflect the new

lower cost. This ensures that as the open list is sorted, the nodes with two possible paths leading to them only have the shortest path considered.

After this is done for all of $q's$ successor nodes, the calculations are done for this iteration of the search. The current node q is removed from the open list and placed on the closed list. The function starts again by sorting the open list in ascending order by cost value. The search function ends when q_{goal} is at the top of the open list. When the goal node has been reached, the fastest route is traced backwards from q_{goal} to q_{init} by following each node's saved parent node.

E. AN OVERVIEW OF THE OBSTACLE AVOIDANCE ALGORITHM

The existing local obstacle avoidance capability for the P3-AT robot was developed in a previous thesis by LT Calvin Hargadine [6], which covers in depth the artificial potential-field function. A rudimentary knowledge of the function is helpful in understanding the navigation algorithm. The potential field function creates a gradient across the robot's map. It treats the robot as a point under the influence of the gradient. The robot seeks to roll downhill, as depicted in Figure 23. The goal is placed at the lowest potential gradient on the map. All obstacles exhibit "repulsive forces" that appear as uphill gradients that force the robot away.

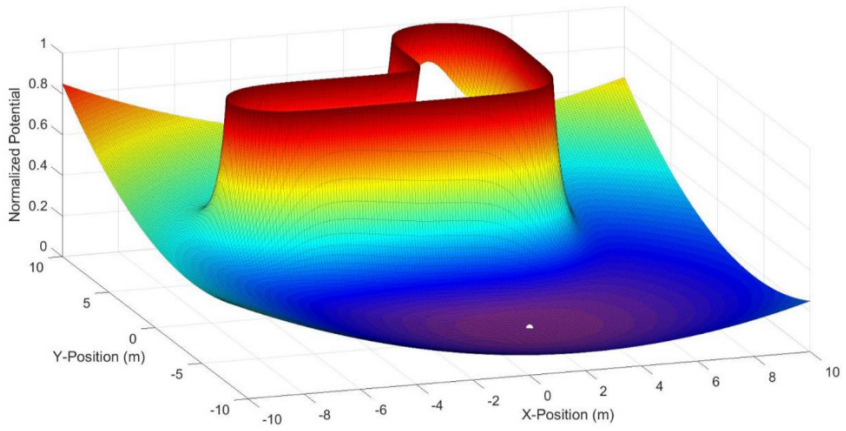

Figure 23. Example of an Artificial Potential Field with Obstacle. Source: [6].

The mathematics used to establish the gradient is explained in [7]. The force acting on the robot at position $q=(x,y)$ is $F(q)$. The artificial potential field $U(q)$ is related to $F(q)$ by

$$F(q)=-\nabla U(q) \tag{10}$$

with $\nabla U(q)$ being the gradient vector at position q. The potential field $U(q)$ is comprised of the attractive field of the goal $U_{att}(q)$ and the repulsive field of any obstacle present $U_{rep}(q)$. The individual potential fields can be related to the forces by

$$F(q)=F_{att}(q)+F_{rep}(q)=-\nabla U_{att}(q)-\nabla U_{rep}(q). \tag{11}$$

The attractive potential field is a parabolic function that converges toward zero as the robot approaches the goal. It is modified by a gain k_{att} that can be adjusted to modify the robot's performance. The attractive force is defined as

$$F_{att}=-\nabla U_{att}(q)=-k_{att}(q-q_{goal}). \tag{12}$$

The repulsive force should be a strong force when the robot is close to an obstacle but has no influence the robot's trajectory if the obstacle is sufficiently far away. The repulsive field $U_{rep}(q)$ has a minimum distance of influence ρ_0. Outside of the distance ρ_0, the repulsive field drops to zero. The distance of the robot to the obstacle at point q is denoted as $\rho(q)$. The repulsive field, like the attractive field, also has an adjustable gain k_{rep}. The repulsive force is defined as

$$F_{rep}(q)=-\nabla U_{rep}(q)=\begin{cases} k_{rep}\left(\dfrac{1}{\rho(q)}-\dfrac{1}{\rho_0}\right)\dfrac{1}{\rho^2(q)}\dfrac{q-q_{obstalce}}{\rho(q)} & \text{if } \rho(q)\le\rho_0 \\ 0 & \text{if } \rho(q)\ge\rho_0 \end{cases} \tag{13}$$

and is combined with the attractive force in order to generate the total force acting on the robot.

A significant limitation of the artificial potential-field function is the problem of local minimums that appear if obstacles have certain shapes [6]. A local minimum "traps" a robot in a position that is not the goal, as denoted by that asterisk at $(-1, 6)$ in Figure 24.

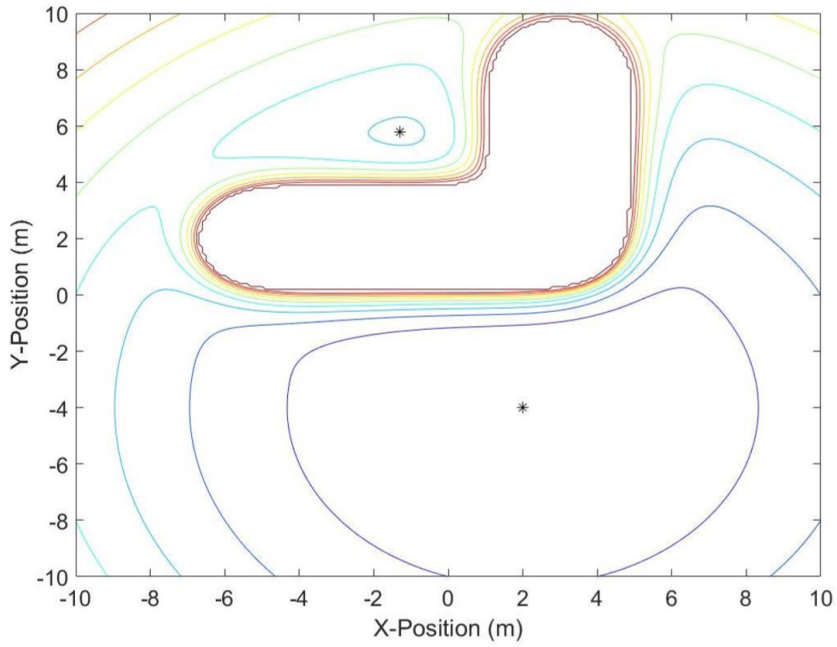

Figure 24. An Artificial Potential Field with a Local Minima. Source: [6]

LT Hargadine's thesis involved a method to escape local minimum. When a local minimum was reached, but it was not the assigned goal position, the robot entered a wall following mode. The robot traced the outline of the obstacle until the local minima was escaped, and the robot continued to q_{goal}. The algorithm that LT Hargadine produced also included two "emergency" modes. If an obstacle were to find its way within a pre-set safe distance, the robot stopped, waited five seconds, and then reversed. The four states of the robot are shown in Figure 25.

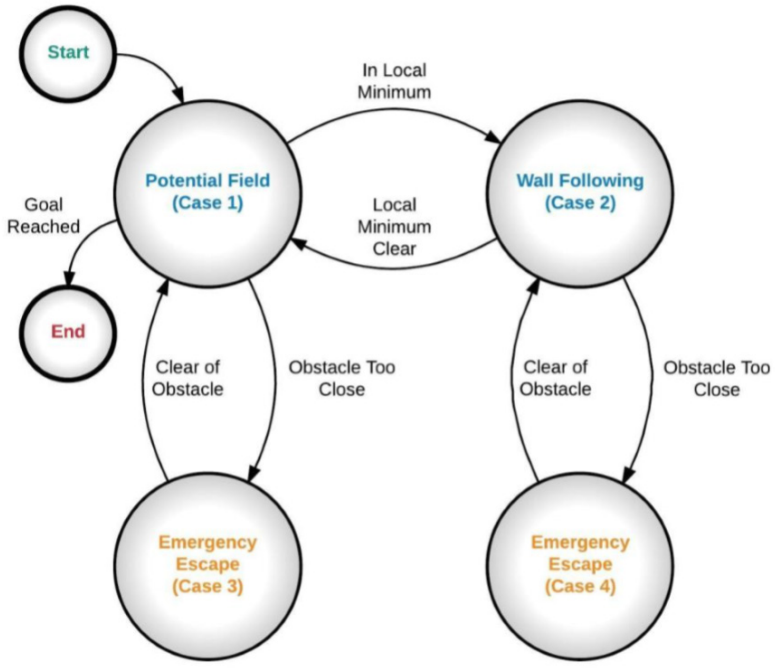

Figure 25. State Diagram of Obstacle Avoidance Control Logic. Source: [6].

F. INTEGRATION OF THE OBSTACLE AVOIDANCE ALGORITHM AND PATH-PLANNING ALGORITHM

The obstacle avoidance algorithm that LT Hargadine had developed for his thesis pulled pre-selected GPS waypoints from a text document. The user was prompted to select one of the ten preset waypoints, and the robot navigated to it before prompting the user to select another. This algorithm is in a stand-alone MATLAB script that runs through a single iteration before prompting the user for another waypoint to be selected.

In this research, the A* search method outputs a list of nodes leading from q_{init} to q_{goal}. If the nodes are output in the same latitude, longitude format that the waypoints navigated to by LT Hargadine's code, the obstacle avoidance algorithm can be looped while each new node in the optimal path is fed to it. To facilitate this the stand-alone obstacle avoidance script that ran through once was modified into a MATLAB function. This new function, `potentialFieldToWaypoint.m` , took the input of a GPS coordinate and navigated to it. The full code for the modified function is found in Appendix

F. With this newly reformatted code, the obstacle avoidance algorithm is treated as a stand-alone "black box" that is looped while being fed desired coordinates.

A flow diagram of the navigation algorithm is depicted in Figure 26. Following this flow chart, the "mission commands" box is the user generated obstacle map. The "cognition planning" is the path-planning algorithm that creates the visibility matrix and uses the A* search method to generate the optimal path in the form of GPS coordinates. The "path execution" box is the obstacle avoidance MATLAB function that generates the "actuator commands." These come in the form of ROS messages published to the ROS ARIA node that controls the robot in the "acting" box. On the left-hand side, the "perception" group is the data read in by the LIDAR and GNSS/IMU that is fed back into the navigation algorithm to monitor progress to the next waypoint while avoiding obstacles.

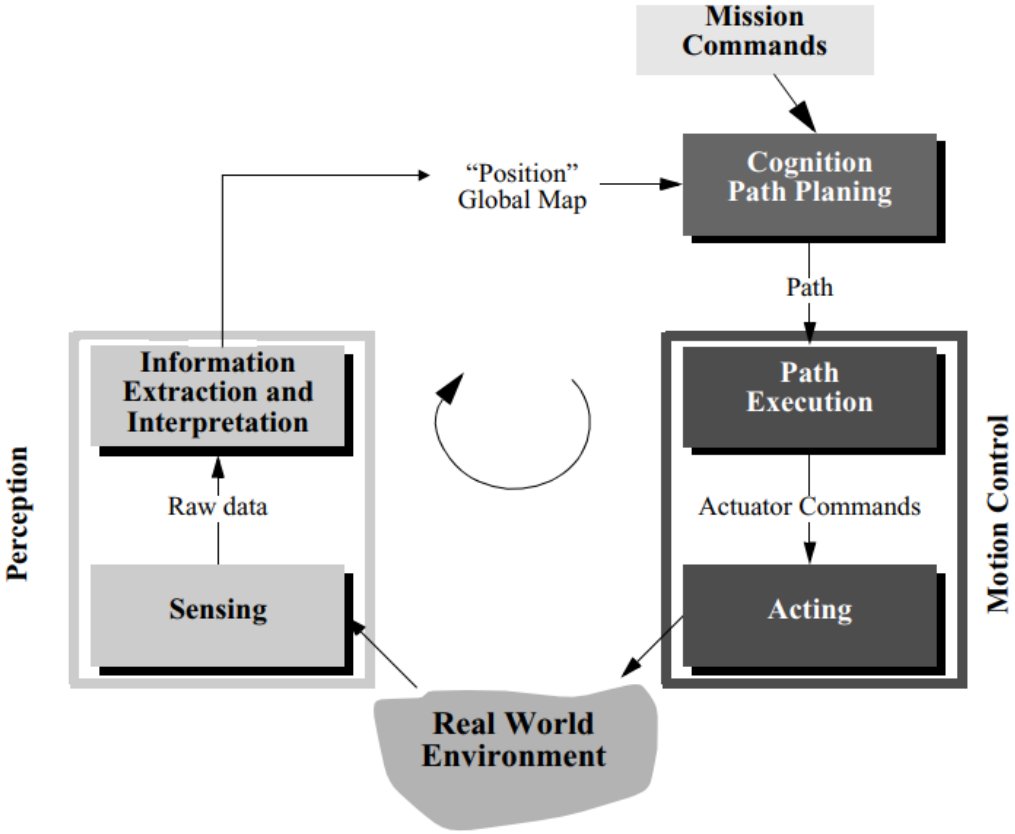

Figure 26. Example Control Scheme for Autonomous Navigation. Adapted from [20].

IV. MATLAB CLASS DEVELOPMENT

During the execution of the path-planning algorithm, large numbers of uniform objects are created using MATLAB. Specific functions are performed on each one repeatedly. To facilitate this, custom MATLAB object classes were developed in order to create uniform instances of obstacles, obstacle fields, nodes, and A* searches.

Object oriented programming has the advantage that it "encapsulates data and operations in objects that interact with each other via the object's interface," [24]. In a custom MATLAB class, an object can be created and assigned fixed properties, and the operations that will be performed can be stored under the methods. Full code for these functions can be found on the author's GitHub page at [25]. The following material includes a brief discussion of each class, its development, and its main properties and methods, and how they interact.

A. OBSTACLE AND OBSTACLE FIELD CLASSES

The two main classes created for the visibility mapping part of the research are the `obstacle` and `obstacleField` classes. The `obstacle` class represents individual obstacles defined by the user and nests into the `obstacleField` class. The `obstacleField` class stores multiple `obstacle` objects and constructs the visibility graph and visibility matrix.

The methods within the `obstacle` class are used to store characteristics of an obstacle object. The major properties and methods of the class can be seen in Figure 27. The obstacle avoidance function reads in GPS coordinates in the form of [latitude, longitude] arrays. Google Earth Pro `.kml` files store polygons as arrays of their vertices. This established that the MATLAB obstacle class stores the location of the obstacle in the form of an array of vertices. The coordinates of the vertices are stored in the property `vertices`. The `numVertices` is a quick reference of how many vertices are in an obstacle and is used in the obstacle field class. When a visibility graph is created, the lines that form $Q(t)$ are taken from the edges of all other obstacles within the field. A quick

reference of numbering the individual obstacles makes pulling $Q(t)$ from all other obstacles easier. As such, each obstacle is given the property of `ObstacleNumber` which is blank until a value is assigned by the obstacle field class.

```
%% Obstacle Class
%  By Matt Audette

Classdef obstacle
    properties
        Vertices       % Stores the vertices of an obstacle
        NumVertices    % Stores the number of vertices of an obstacle
        ObstacleNumber % Will be assigned by the Field Class.

    methods
      function obstacle = createObstacle(inputArray)
            % Creates an obstacle from the [x,y] coordinates of the vertices
            % Auto populates the NumVertices property
            % ObstacleNumber is filled by the obstacleField class
```

Figure 27. Pseudocode Showing the Major Properties and Functions of the Obstacle Class

The `obstacle` class nests into the `obstacleField` class, which contains the majority of the path-planning code. It is used to store and index obstacles, the initial point, and the goal point. The `obstacleField` class also creates the visibility graphs and matrices that are fed into the A* search method. Pseudocode showing the major properties and methods for the `obstacleField` class can be seen in Figure 28. The `Field` property stores the individual obstacle objects in an array. The initial and goal points are stored in the `qinit` and `qgoal` properties. When the visibility matrix is created, it is stored in a corresponding property `VisibilityMatrix`. The `PointIndex` is a three dimensional array. An example of the information it stores can be seen in Figure 29. The first layer stores the vertices of the obstacle of the field as well as `qinit` and `qgoal`. The second layer stores each point's obstacle number and the number of which vertex within that obstacle that specific point is. These parameters are used heavily in the `VisibilityMatrix` function, which is the main function of the class. The other functions within the `obstacleField` class are constructor functions for both the class

and each property, plotting functions, and reference functions used within the creation of the visibility graph and plot.

```
%% Obstacle Field Class
%  By Matt Audette

classdef obstacleField
    properties
        Field;            % an array of obstacle objects
        NumObstacles;     % Number of obstacles stored in the field
        qinit;            % Stores the start point
        qgoal;            % Stores the goal point
        PointIndex        % an array of all the points in the obstacle field
        VisibilityMatrix

    methods
        function obstacleField(obstacles) %Constructor Method

        function addObstacle(obstacle) % Add an obstacle to an existing field
        function initPoint(point)      % create qinit
        function goalPoint(point)      % create qgoal
        function plotField(o_field)    % plot the field

        function of = constructPointIndex(o_field)
        function visibilityMatrix(o_field)    % construct visibility matrix
        function plotVisibilityGraph(o_field) % construct visibility graph
```

Figure 28. Pseudocode Showing the Major Properties and Methods of the Obstacle Field Class

Obstacle 1

Page 1: (:, :, 1)

x	*y*
1	1
1	2
2	2
2	1

Page 2: (:, :, 2)

Obst #	*Point #*
1	1
1	2
1	3
1	4

Figure 29. Example of the Information Stored in the Point Index Three Dimensional Array

The MATLAB script `TestBed.m` can be found in Appendix B and gives sample uses of calling all the methods within both classes. It is intended to provide users with the ability to follow and create their own obstacle field instances by hand.

B. NODE AND A* SEARCH CLASSES

The A* search method was created as its own separate class from the obstacle field. The primary reason was to facilitate the use of waypoints. Creating a new visibility graph for each waypoint is a time-consuming process. Instead of creating a new visibility graph for each new instance of a waypoint, one visibility graph can be used with multiple A* search instances. This is done by setting `qgoal` to waypoint 1 and finding the optimal path to this waypoint. When this is completed, waypoint 1 is set to `qinit`, waypoint 2 is set to `qgoal`, and a new A* search is run. This is repeated until the `qgoal` is reached. The optimal path from `qinit` to `qgoal` through all waypoints can now be identified by stringing together the optimal paths for each waypoint.

The input of the `AStarSearch` class is an `obstacleField` object. The primary tool used in the `AStarSearch` class to generate an optimal path is the obstacle field's visibility matrix. If the inputted obstacle field does not have a visibility matrix, the `AStarSearch` class generates one automatically.

The building block for the `AStarSearch` class is the `node` object, much like the `obstacle` object is for the `obstacleField`. The `node` class creates objects with parameters that are either indexed or set by the `AStarSearch` class. Pseudocode for the `node` class can be seen in Figure 30. The main properties of the class are the `location` of the point that forms the node, the `index`, which directly correlates to the `PointIndex` of the `obstacleField` class, and the parameters `f`, `g`, and `h`, which are set by the `AStarSearch` class. Each `node` also has the `neighbors` parameter, which is filled by pulling from the visibility matrix. There are also `initFlag` and `goalFlag` Boolean variables, which are set to one (true) if the node is the q_{init} or q_{goal} and left as zero (false) otherwise. Last is the parameter `cameFrom`, which is set by the `AStarSearch` class's `optimalPath` method. When performing the search method, each of $q's$ successors

have their parent value set to q. The `cameFrom` parameter is where that information is stored.

```
%% Node Class
%  By Matt Audette

classdef node
    properties
        index     % Ties directly into the PointIndex of the obst field
        location  % Node location- will be pulled from the obst field
        f         % Cost value for A*; f = g + h
        g         % The movement cost
        h         % The heuristic cost
        neighbors % Will store nodes that are visible- will be pulled from
                  % 1's in the visibility matrix of the obstacle field obj.
        cameFrom  % Will store what node it came from- will be chained to
                  % produce the optimal path
        goalFlag  % 1 if the goal, 0 if not
        initFlag  % 1 if start point, 0 if not

    methods
        function nd = node(xy_point_array) % constructor function
        function nd = setIndex(index)
        function nd = setF(cost)
        function nd = setG(cost)
        function nd = setH(cost)

        function nd = setNeighbors(inputArray) % saves the neighbor's indices

        function nd = setCameFrom(index)  % index of the parent node

        function nd = setInitFlag(input)  % 1 if start point, 0 otherwise
        function nd = setGoalFlag(input)  % 1 if goal point, 0 otherwise
```

Figure 30. Pseudocode Showing the Major Properties and Methods of the Node Class

The `AStarSearch` class takes as an input an `obstacleField` object. Pseudocode for the class can be seen in Figure 31. When an instance of an `AStarSearch` object has an `obstacleField` loaded into it, the `AStarSearch` object stores the visibility matrix and automatically converts the vertices of the obstacles, the `qinit`, and `qgoal` of the `obstacleField` into node objects. Each node's neighbors are auto populated by searching the visibility graph for instances of "1" between nodes. The `AStarSearch` class has a `costFlag` property. The cost between two nodes is calculated as a straight line distance. When obstacles are stored in an [x, y] coordinate

system, Euclidian distance is used and the `costFlag` is set to "0." When obstacles are stored in a [latitude, longitude] coordinate system, the distance is calculated by using the haversine great circle distance and the `costFlag` is set to "1." The primary method in the `AStarSearch` class is the `findOptimalPath` function. It performs the A* search in the manner described in Chapter III.D. The open list, closed list, and optimal path are all saved in corresponding properties for later review by the user.

The MATLAB script `AStarTestBed.m` can be found in Appendix E and gives sample of calling all the methods within both `node` and `AStarSearch` classes. It is intended to provide users with the ability to follow and create their own `AStarSearch` instances by hand.

```
%% A Star Search Class for a Visibility Matrix Function
%  By Matt Audette

classdef AStarSearch
    properties
        obstacleField    % Primary input is an obstacleField object
        VisibilityMatrix % Stores the vis matrix of the obstacle field

        nodeIndex   % An index that will store the node objects from the vis
                    % matrix. Primary method of referencing nodes
        openList    % Standard A* search
        closedList  % Standard A* search
        costFlag    % Which distnace function to use: 0= Euclidean for [x,y]
                    % coordinates, 1= Harversine great circle for [lat,long]
        optimalPath % Stores the formatted coordinates

    methods
        function astar = AStarSearch(o_field) % Constructor method
        function astar = pointsToNodes(astar) % Converts each obstacle in the
                                % obstacle field's vertices into a node object
        function astar = setCostFlag(flag) % 0= Euclidian dist, 1= haversine
        function astar = findOptimalPath(astar)
        function plotOptimalPath(astar)
```

Figure 31. Pseudocode Showing the Major Properties and Methods of the AStarSearch Class

C. INTEGRATION WITH GOOGLE MAPS

A conversion tool was needed to interface Google Earth Pro's `.kml` files with MATLAB. Several `.kml` reading and writing toolkits were found using MATLAB's built-in toolkit locator. After examining several options, the `read_kml` toolbox by Amy Farris of the United States Geological Service was chosen [15]. The toolbox consists of a single script that reads in `.kml` files as a `string` data type and parses them into `double` arrays of [*x,y,z*] coordinates. Other toolboxes considered had robust abilities to read and write `.kml` files but provided numerous capabilities that were not needed for this research. In the `read_kml` script's original format, the MATLAB function used to change the `string` to `double` data types truncates the decimal places at four digits. For regular mathematical purposes this suffices, but for reading GPS coordinates that are saved to the thirteenth decimal place, this represents a significant loss of fidelity. The exact loss varies with latitude, but at Monterey, California's location this represented a 7.0-m difference in reading in the coordinates 36.5964472992381, −121.8764444525143 versus 36.5964, −121.8764. In order to fix this, code was added to keep the previously truncated decimal places. Exact changes in the code can be seen in Appendix G.

The `read_kml` toolbox was the backbone for the three functions used to read the `.kml` files created by the user and convert them to the proper format for the `obstacle`, `obstacleField`, and `AStarSearch` classes. Each of the three user generated `.kml` file types got its own corresponding `kmlTo-` function. The goal point and initial points were read into MATLAB and converted using the `kmlToGoal` and `kmlToInit` functions, respectively. The obstacle map was read in by the `kmlToObstacle` function where the GPS coordinates for each polygon were converted to an obstacle, and then all the obstacles were loaded into an `obstacleField` type object. The `.kml` file interfacing functions are found in Appendix C. The MATLAB script `Google_Earth_test.m` can be found in Appendix D and gives sample uses of calling each of the `kmlTo-` functions. It is intended to provide users with the ability to follow and create their own obstacle field instances by hand.

D. GLOBAL NAVIGATION ALGORITHM SUMMARY

What follows is a brief description of how the map making, MATLAB code, and robot navigation are all tied together. In Appendix A, the code `DemoFile.m` shows a demonstration of the navigation algorithm reading in `.kml` files, determining the optimal path, and navigating the robot to it. The full code, example `.kml` files for obstacle maps, start points, and end points, can be found on the author's GitHub page at [25].

The user first opens Google Maps Pro. Using the "draw polygons" tool in the top tool bar, the user draws bounding boxes around large-scale obstacles within their area of operations. Buildings, terrain unnavigable by the robot, or off limits areas are included. Users do not need to include small, movable obstacles in the imagery such as cars, tables, or trees, as the local obstacle avoidance algorithm navigates the robot around these. When the obstacle field is complete, the user saves it as a `.kml` file. A start point is placed by using the "Add Placemark" tool, and it is saved as its own `.kml` file. The process is repeated for the goal point.

The pseudocode laid out in Figure 32 shows the steps taken in the MATLAB `DemoFile.m` script. The code reads in the three `.kml` files, converts them to an `obstacleField`, builds the visibility matrix, feeds the `obstacleField` to an `AStarSearch` instance, and then determines the optimal path. The optimal path is plotted for the user to see. The coordinates from the optimal path are then given to the `potentialFieldtoWaypoint` function that navigates the robot to the waypoints. A video demonstration of this process can be found at [26].

```
%% Roadmap Planning Algorithm
% In Google Earth:

create in Google Earth a .kml file outline the obstacles
create in Google Earth a .kml file for the initial point
create in Google Earth a .kml file for the goal point

% In MATLAB:
create in empty obstacle field
load it with the obstacles from the .kml file
load its initial and goal points from the .kml file

create the visibility matrix
plot the visibility graph

create an empty A* search object
load it with the obstacle field
find the optimal path and its coordinate list
plot the optimal path

for the length of the optimal path
    run the potential field navigation algorithm to the next point
end
```

Figure 32. Pseudocode for the Order of Operations in Using the Navigation Algorithm

THIS PAGE INTENTIONALLY LEFT BLANK

V. EXPERIMENTS AND RESULTS

With the completion and verification of the path-planning algorithm, which was implemented in MATLAB, it was time to evaluate the navigation algorithm's ability to operate in a real-world environment. An obstacle map of NPS, seen in Figure 33, was created in Google Earth Pro. Bounding boxes were drawn around obstacles there are visible within the imagery that the robot cannot navigate through: buildings, stair cases, and obstacles that the 2D LIDAR is not able to detect such as curbs. The two main areas around which experiments occurred are labeled as Building 1 and Courtyard.

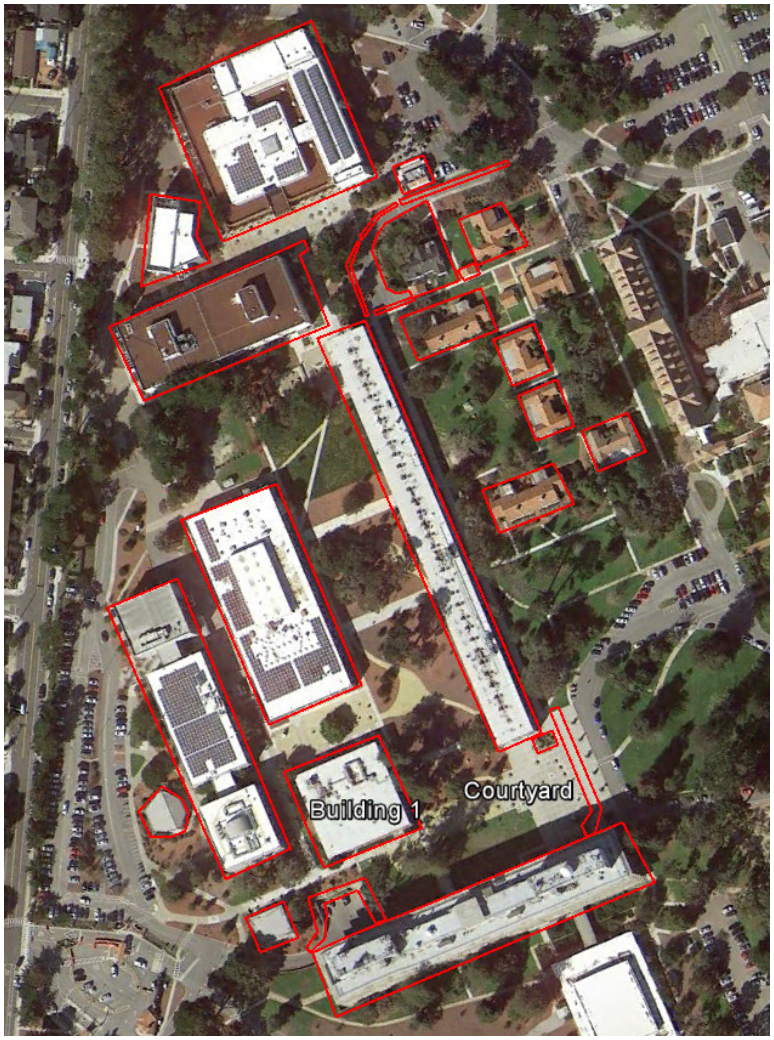

Figure 33. Obstacle Map of the Naval Postgraduate School Campus

This map was then given to the path-planning algorithm, which in turn produced the visibility graph seen in Figure 34. The visibility graph consists of 26 obstacles comprising 130 nodes. The polygons present in the visibility graph correspond to the bounding boxes drawn around the known obstacles in the map. The clear lines-of-sight in-between nodes are shown by the dotted lines and indicate the 964 possible paths that the robot may travel. In the last step of the path-planning algorithm, the visibility graph is used by the `AStarSearch` class to find the optimal path between a user given `qinit` and `qgoal`.

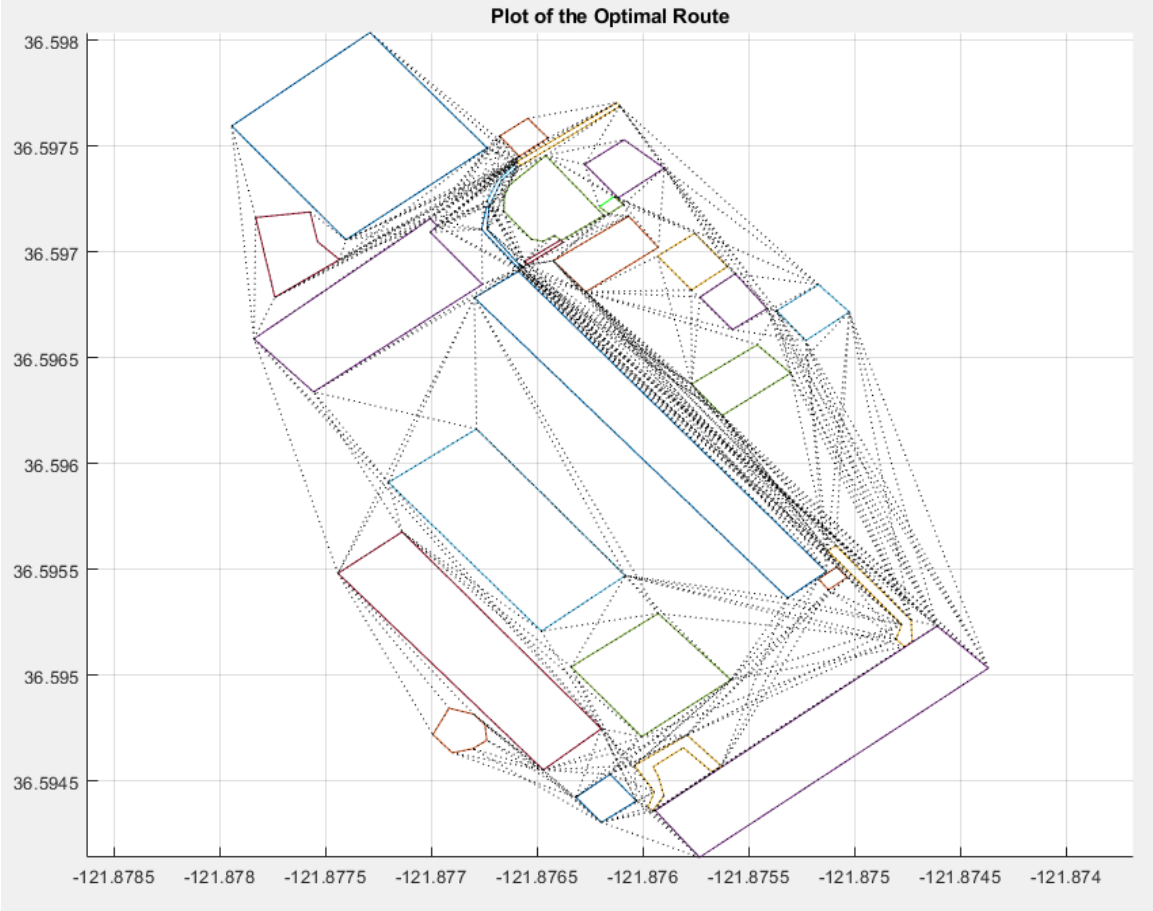

Figure 34. Visibility Graph of the NPS Campus Generated by the Path-Planning Algorithm

A. EXPERIMENT 1: GPS WAYPOINT NAVIGATION USING THE PATH-PLANNING ALGORITHM

Experiment 1 was developed to assess the ability of the robot to navigate to GPS waypoints generated by the path-planning algorithm without deviating heavily from the desired path. The courtyard was selected to eliminate the problems of rough terrain and GPS loss due to proximity to buildings. A five-sided polygon was drawn in the courtyard, which is shown in Figure 35. The numbered pins in the figure correspond to the five waypoints that the robot will navigate to in the experiment.

Figure 35. Experiment 1's Polygon and Numbered Waypoints

The results are shown in Figure 36. The path traveled by the robot, shown in blue, shows that each waypoint generated by the path-planning algorithm was successfully navigated to. The trajectory of the robot does appear to deviate from the straight line path between the waypoints shown in green. This is due to the goal radius placed around each waypoint. A navigation waypoint for a robot is surrounded by a goal radius that artificially

increases the size of a goal to an acceptable tolerance. Without this goal radius surrounding a waypoint, the robot will most likely never reach the exact assigned goal despite being within an acceptable distance. In the robot's configuration file, the parameter that corresponds to the distance for the goal radius was set to 2.0 m. When the robot enters this radius, it terminates navigation to the current waypoint. It then recalculates the path to the next waypoint from its current position rather than from the waypoint's programmed position. This accounts for the deviations in the path.

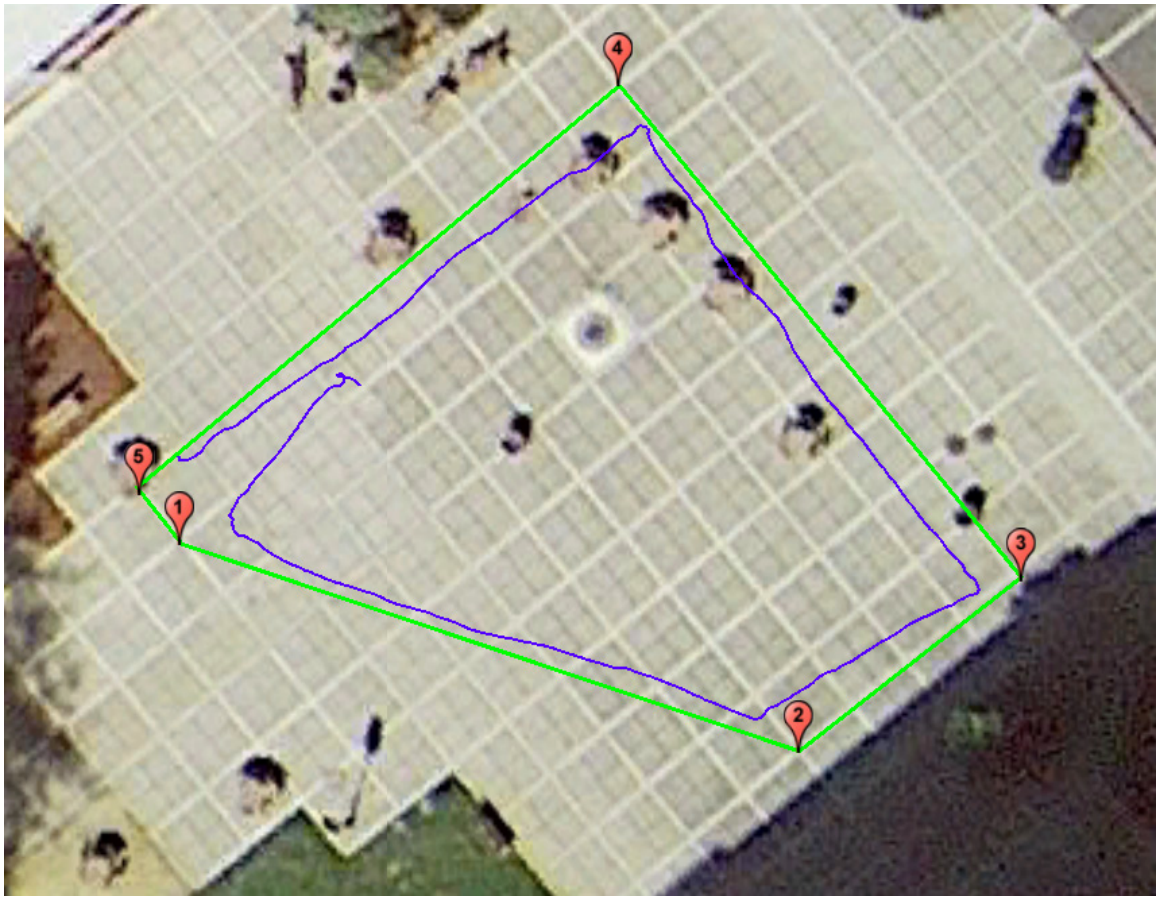

Figure 36. Path Traveled by the P3-AT in Experiment 1

The next test was an assessment of how the navigation algorithm performed when avoiding obstacles. The robot was run through the same course, and periodically an obstacle was placed in front of the robot's path. The resulting trajectories can be seen in

Figure 37. The yellow pins indicate points-of-interest (POIs) that warrant discussion. At the indicated POIs an obstacle was placed in front of the robot. When the robot encountered an obstacle, the artificial potential field function performed its role obstacle avoidance role as desired. It is noted that when the robot was forced to deviate heavily from the straight line path, as seen at POI 4, it made no attempt to regain the straight-line path between points four and five but instead went straight to the waypoint. These results indicate that the path-planning algorithm was integrated successfully with the obstacle avoidance algorithm.

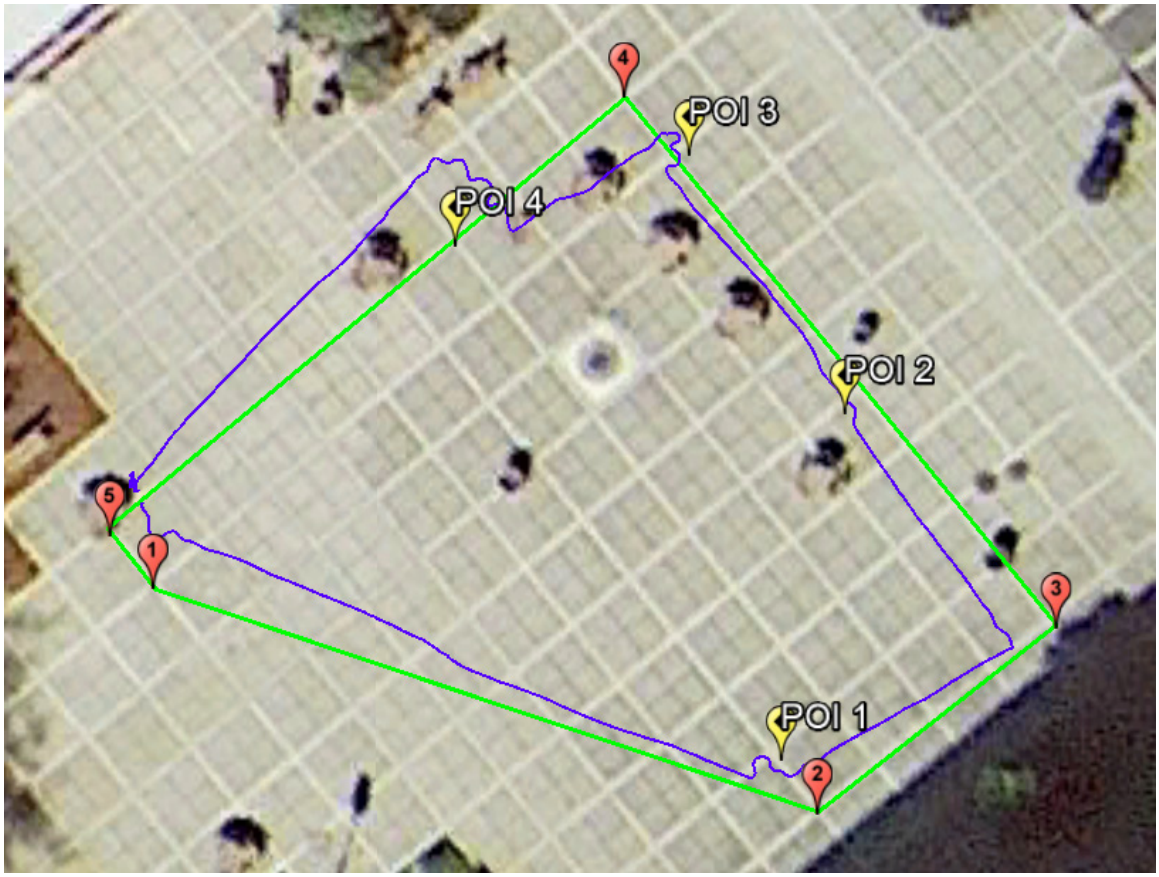

Figure 37. Evaluating Obstacle Avoidance While Navigating to Waypoints

B. EXPERIMENT 2: MEDIUM DISTANCE NAVIGATION

With the successes of the performance of the navigation algorithm on flat terrain, it was determined that a medium distance test over rugged terrain needed to be run.

Experiment 2 was developed to evaluate this capability. A start point on the west side of the courtyard was chosen as well as a goal north of Building 1, as seen in Figure 38. These new points and the previously made obstacle map there ran through the path-planning algorithm and an optimal path was produced. The optimal path generated is shown in Figure 39 and highlighted in green.

Figure 38. Start and Goal Points for Experiment 2

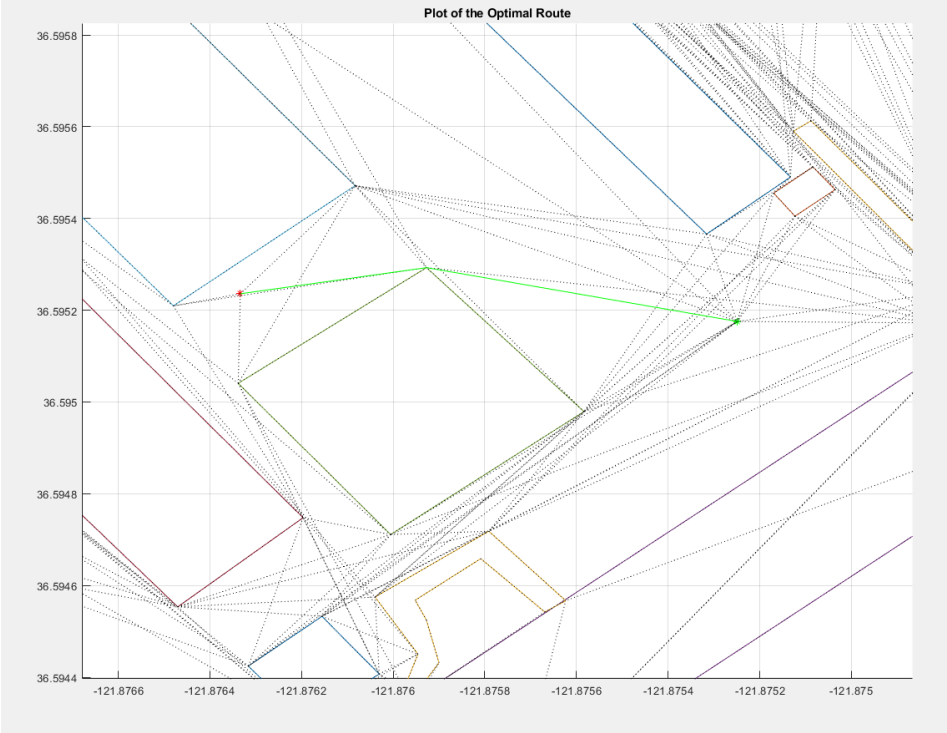

Figure 39. The Optimal Path Generated by the Path-Planning Algorithm for Experiment 2

The results of the experiment are shown in Figure 40. The robot moved from its starting position and picked up the first waypoint without issue. When moving to the next waypoint, the robot entered the loose terrain. At POI 1, shown by the yellow pin, the robot encountered bushes, and successfully navigated around them. As the robot continued to travel around the bushes it encountered larger piles of woodchips. At certain points the wheels slipped and on two occasions the chassis pushed through piles of woodchips. At POI 2, the robot found itself under tall trees and near Building 1. The navigation outputs indicated that the robot had lost GPS connectivity. In this condition, paired with the loose terrain and several small bushes it was attempting to navigate around, the robot's odometry measurements became unusable. At this point, the robot was not traveling more than a meter in either direction as it had become mired in woodchips. The odometry measurements-based trajectory line, however, indicates that the robot traveled into Building 1 and further under the tress. This discrepancy demonstrates that the odometry

measurements were adversely affected by the terrain. After several minutes of the robot being unable to free itself the experiment was ended.

Figure 40. The Optimal Route and Route Traversed for Experiment 2

VI. CONCLUSIONS

A. ASSESSMENT OF GOALS

The purpose of this thesis was to facilitate the integration of unmanned ground vehicles into lower levels of the Marine Corps by providing operators non-technical tools to generate mission guidance for a robot. The goal of developing the capability for an operator to quickly generate an obstacle map and the robot to perform optimal path planning was successful. A path-planning algorithm was developed using the visibility graph and A* search methods. It was then successfully integrated with an existing obstacle avoidance algorithm. While the path-planning algorithm yielded the desired results, is was not completely integrated with the P3-AT because of the limitations with the capabilities of the robot. On several occasions the ability to navigate to waypoints was successful, but when the robot traveled over loose terrain, near buildings, and under trees, the robot lost the ability to localize itself.

B. SYSTEM IMPROVEMENTS AND AREAS OF FUTURE WORK

Further work could incorporate into the A* search's cost function the ability to account for rough terrain. A bounding box could be drawn around loose terrain, and a penalty for any route traveling through it could be imposed via the movement cost. Using this method will still consider paths through rough terrain but favor paths less likely to cause issues with the chassis.

It would be possible to use post processing of the LIDAR readings in order to update the obstacle map. The LIDAR returns obstacle positions as measurements of distance and bearing. These measurements could be used to plot detected obstacles and update the user-made obstacle map. Future missions could base the visibility graph on these more up-to-date, closer to ground truth maps.

The capabilities of the sensor suite could be increased by using the onboard webcam for computer vision. Computer vision could be used for obstacle detection, classification, and navigation. When fused with the data from the LIDAR, the localization and navigation abilities of the robot will be greatly increased.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A. DEMONSTRATION.M

```
%% Demonstration File

% This script takes the three test bed scripts-TestBed.m,
% GoogleEarthToolboxTest.m, and AStarTestBed.m, and concatonates all
% them into one script with a demonstration file.

% Instructions:
% Go Into GoogleEarth, pick a location, and create an obstacle field by
% drawing squares around visible obstacles. Save is as
'demoObstacles.kml'

% Then pick a single point to be the initial position. Save it as
% "demoInit.kml." Do the same with a goal pos, save as "demoGoal.kml"

% Then run this script in MATLAB.
% Demonstration files can be found at https://github.com/MattAud/Thesis
% A video demonstration can be watched at
% https://www.youtube.com/watch?v=Y0Jkl2ozzEE

% Matt Audette
% 20180618

clear all
clc

%% Load the .kml files:

obstacleFieldName = 'demoObstacles.kml'
initPointName = 'demoInit.kml'
goalPointName = 'demoGoal.kml'

%% Create the Obstacle field from the .kml:
% initiate an empty field
demoField = obstacleField();
% load it with our obstacles
demoField = kmlToObstacleField(demoField, obstacleFieldName)

%% Load in the initial and goal points:
% And the start point:
demoField = kmlToInit(demoField, initPointName);
% And the goal point:
demoField = kmlToGoal(demoField, goalPointName);

%% Plot it:
plotField(demoField)

%% Build the Visibility Matrix:
demoField = visibilityMatrix(demoField)
plotVisibilityGraph(demoField)
```

```
%% Find and plot the Optimal Path:
%First, create an AStarSearch Object:
aStarDemo = AStarSearch(demoField)
%Find and plot the optimal path:
aStarDemo = findOptimalPath(aStarDemo)
plotOptimalPath(aStarDemo)

%% Produce the optimal path:
coordinateList = coordsFromOptimalPath(aStarDemo.optimalPath)

%% Loop the Robot Navigation Function:
for i = 1:size(coordinateList, 1)
    %Send the coordinate and waypoint #
    %and wait for the robot to go to that point:
    potentialFieldToWaypoint( coordinateList(i, :), i)

end
```

APPENDIX B. TESTBED.M

```
%% Obstacle Field Test Bed
% By Matt Audette
% Last Update: 20180121
% Remarks: This is a script that will be used to test the obstacle and
% obstacle field classes.

clc
clear all
close all
format compact

%% Create Obstacles:
disp(' ---------- Create three obstacles ------- ')
obstA = obstacle([2,1; 2,2; 3,2; 3,1]);
obstB = obstacle([6,6; 7,6; 7,7]);
obstC = obstacle([4,2; 5,2; 5,7; 4,7]);

%% Create the obstacle fields:
disp(' ----------Create Two Fields ------- ')
emptyField = obstacleField() %an empty test field
testField1 = obstacleField(obstA, obstB) %use the constructor and add 2
% obstacles. Ensure that the NumObstacles goes up:

% Now test the addObstacle function and ensure that the counts are
% correct and do no overwrite one another:
disp(' ---------- Expand the Fields ------- ')

%Note: in this script, the call is:
% emptyField = addObstacle(emptyField, obst1, obst2)
% In the command window, it would be typed:
% addObstacle(emptyField, obst1, obst2)
% addObstacle(testField1, obst3)

emptyField = addObstacle(emptyField, obstA, obstB)
testField1 = addObstacle(testField1, obstC)

disp(' ---------- Set an Init Point ------- ')
testField1 = initPoint(testField1, [1,1])
% In the command window: initPoint(testField1, [1,1])

disp('----- Create a waypoint load it as a point ----')
qstart = Waypoint( [3,4], 0);
qend = Waypoint( [8,8] );
qend = setClass(qend, 1);
emptyField = initPoint(emptyField, qstart.Location)
emptyField = goalPoint(emptyField, qend.Location)
% In the command window: initPoint(testField1, [1,1])

disp(' ---------- Set a Goal Point ------- ')
testField1 = goalPoint(testField1, [8,7])
```

```
% In the command window: goalPoint(testField1, [8,7])

disp(' ---------- Plot the Field ------- ')
plotField(testField1)
% In the command window: goalPoint(testField1, [8,7])

% Test the function of the empty qinit and qgoal functions:
plotField(emptyField)

disp(' ---------- Test the Field Size Change ------- ')
emptyField = setFieldSize(emptyField, 15, 1)
plotField(emptyField)

disp(' --------- Auto Size Feature ----------')
emptyField = autoFieldSize(emptyField)
plotField(emptyField)

disp(' --------- Built Point Index Feature----------')
% the point index array is a single array that will contain all the
% points in each obstacle, the waypoints, and the qinit and goal. It
% will be the primary way to index points in the visibility graph
% and plot functions.

% To access a point index number C, call the command
% obstacleField.PointIndex(C,:) and it will return the point.
testField1 = constructPointIndex(testField1)
disp(' Test calling a singe point by its index:')
testField1.PointIndex(3,:, 1) %where '3' is the index

disp(' --------- Get Point From Index Feature----------')
% This fuction works backwards from the point index array: it is
% fed an index number and returns the point. This will be used to
% map the completed visibility matrix to the points in the field.
pointTest = getPointFromIndex(testField1, 3)

%This function will also check if the point index is empty, and if so,
%build a point index. Test this with emptyField:
pointTest2 = getPointFromIndex(emptyField, 3)

disp(' --------- Get Obstacle Info From Index Feature----------')
% This function gives an index number from the PointIndex and returns
the
% second array's infor. It will be in the form of [obstacle#, point#].
% Test is out for the same two points as before:
obstacleTest = getObstacleFromIndex(testField1, 3)
obstacleTest2 = getObstacleFromIndex(emptyField, 3)

%% test the Index feature

index = min( find(testField1.PointIndex(:,1,2) == 2))
testOutPoint = getObstacleFromIndex(testField1, index)

%% Build the visibility Matrix:
```

```
testField1 = visibilityMatrix(testField1) %in the command prompt, use
    % visibilityMatrix(testFeild1)
%emptyField = visibilityMatrix(emptyField)

% return the point from the point index:
points1 = recallPointsFromMatrix(testField1, 1, 4)
% This line below throws an error message to see if it works.
%points2 = recallPointsFromMatrix(emptyField, 1, 4)

disp('Outline how we will plot the visibility array:')
% call the plot function:
plotVisibilityGraph(testField1)
%plotVisibilityGraph(emptyField)
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX C. KML TO OBSTACLE SCRIPTS

```
%% KML to Init Point function
% Matt Audette
%
% Reads in a .kml file and spits out an obstacle field term of "end
% point" that can then be loaded into the Obstacle Field function.
% Uses the modified read_kml function by Amy Farris available on the
% Mathworks online app repository.

function oField = kmlToInit(oField, fileNameString)
    % Read in the .kml file, store it in an array
    [stoX, stoY, stoZ] = read_kml(fileNameString)
    point = [stoX, stoY];
    % Load it:
    oField = initPoint(oField, point);
end


%% KML to Goal function
% Matt Audette
%
% Reads in a .kml file and spits out an obstacle field term of "end
% point" that can then be loaded into the Obstacle Field function.
% Uses the modified read_kml function by Amy Farris available on the
% Mathworks online app repository.

function oField = kmlToGoal(oField, fileNameString)
    % read in the .kml file, store it in an array
    [stoX, stoY, stoZ] = read_kml(fileNameString)
    point = [stoX, stoY];
    % Load it:
    oField = goalPoint(oField, point);
end



%% KML to Obstacle Field function
% Matt Audette
%
% Reads in a .kml file and spits out in obstacle field.
% Uses the modified read_kml function by Amy Farris available on the
% Mathworks online app repository.

function oField = kmlToObstacleField(oField, fileNameString)
    % Read in the .kml file, store it in an array
    [stoX, stoY, stoZ] = read_kml(fileNameString)
    pointCounter = length(stoX)

    %% Sort the coordinates into obstacles:
    % The .kml file is just a series of points. If you look at the GPS
    % coordinates, the stored points repeat the first/last point,
    % square bounding box will read points 1, 2, 3, 4, 1, then the next
```

```
    % meaning the a coordinates. We will use that to identify where one
    % obstacle begins and ends.
    temp = [];
    for i = 1:pointCounter
        % We need to take the first coordinate to be loaded into the
        % temporary holder so that we can use it as a reference to look
        % for the repeat value. We'll call the variable "checker."
        if length(temp) == 0
            checker = stoX(i)
        end
        temp = [temp; [stoX(i), stoY(i)]];

        %if we check the first coordinate against itself, it'll be
        %true. So skip the first element.
        if length(temp) > 2
            %if the current value matches the checker reference value,
            %trim the last coordinate off (it's a repeat) and then
            % clear the temporary storage matrix.
            if stoX(i) == checker
                %disp('FOUND A MATCH!')
                trimNumber = length(temp)-1;
                %temp
                holder = temp(1:trimNumber, :)
                %The obstacleField class takes in obstacles,
                %so turn holder into an obstacle:
                holderObstacle = obstacle(holder);
                oField = addObstacle(oField, holderObstacle);
                temp = [];
            end
        end
    end

end
```

APPENDIX D. GOOGLE_EARTH_TEST.M

```
%% Matlab / Google Earth toolbox test kit:
% Used the modified add on "read_kml" (a stand alone function)
% to read .kml files. Tests the kmlTo____ functions

% Matt Audette
% 20180430

%% Test Reading in a .kml file
% In my Matlab main folder, I have a .kml file that I saved.
% The read_kml produces an three matrices: [x, y, z]

%[x, y, z] = read_kml('test_polygon.kml')

% Reading in from a specific file path
readInStr = 'C:\Users\audet\OneDrive\Documents\MATLAB\
multi_point_test.kml';
[A,B,C] = read_kml(readInStr)

%% Test bed for the KML to obstacle function:

%% Read in the .kml file:
% Store it in an array
[stoX, stoY, stoZ] = read_kml(readInStr)
pointCounter = length(stoX)

%% Sort the coordinates into obstacles:
% The .kml file is just a series of points. If you look at the GPS
% the a coordinates, the stored points repeat the first/last point,
% meaning square bounding box will read points 1, 2, 3, 4, 1, then the
% next coordinates. We will use that to identify where one obstacle and
% begins at he ends.
temp = [];
for i = 1:pointCounter
    % We need to take the first coordinate to be loaded into the
    % temporary holder so that we can use it as a reference to look for
    % the repeat value. We'll call the variable "checker."
    if length(temp) == 0
        checker = stoX(i)
    end
    temp = [temp; [stoX(i), stoY(i)]];

    %if we check the first coordinate against itself, it'll be
    %true. So skip the first element.
    if length(temp) > 2
        %if the current value matches the checker reference value,
        %trim the last coordinate off (it's a repeat) and then clear
        % the temporary storage matrix.
        if stoX(i) == checker
            disp('FOUND A MATCH!')
            trimNumber = length(temp)-1;
```

```
            temp
            holder = temp(1:trimNumber, :)
            testObstacle = obstacle(holder)
            temp = [];
        end
    end

end

%% Test out the kmlToObstacleField:
testField2 = obstacleField();
testField2 = kmlToObstacleField(testField2, readInStr)

% Load in the goal point:
goalString = 'EndPointTest.kml';
testField2 = kmlToGoal(testField2, goalString);

% And the start point:
initString = 'StartPointTest.kml';
testField2 = kmlToInit(testField2, initString);

%testField2 = autoFieldSize(testField2);
plotField(testField2)
testField2 = visibilityMatrix(testField2)
plotVisibilityGraph(testField2)

%% Look at a KML with points:
fileNameStr3 = 'testKmlWithPoints.kml';
[D,E,F] = read_kml(fileNameStr3);
G = [D,E]
```

APPENDIX E. ASTAR_TESTBED.M

```
%% A* Search Test Bed
% by Matt Audette
% last update: 20180530

% This is the test script for kicking the tires on my node/A* search
% object class.

% RUN THE SCRIPT TestBed BEFORE HAND!
% RUN THE SCRIPT GoogleEarthToolboxTest BEFORE HAND, TOO!
% This script will read in the obstacleField objects from both.
clc
clear all
format compact

TestBedMkI
GoogleEarthToolboxTest

% Build a test node:
node1 = node([1,2])

% Fill out the other properties:
% If you do this in the command line, it's just 'setIndex(node1, 1)'
node1 = setIndex(node1, 1)

node1 = setF(node1, 0);
% an f of '0' indicates that it's the start point, so we should set the
% initFlag property to 1. And it is therefore NOT the goal node, so...
node1 = setInitFlag(node1, 1);
node1 = setGoalFlag(node1, 0);

% And fill out the rest of the properties in order to test them:
node1 = setG(node1, 0);
node1 = setH(node1, 20);
node1 = setNeighbors(node1, [2,3,4,5])
node1 = setCameFrom(node1, 7)

%% This completes the node functions. Move into the A* Search Object:
% Create an A* search object:
aStarTest = AStarSearch(testField1)

% Load of the node index:
aStarTest = pointsToNodes(aStarTest)

% Now do it with the GoogleMapsToolboxTest
aStarFromKML = AStarSearch(testField2)
%aStarFromKML = pointsToNodes(aStarFromKML)

aStarTest = nodeIndextoCellArray(aStarTest)
%aStarFromKML = nodeIndextoCellArray(aStarFromKML)
```

```
% set the cost function flagL
aStarTest = setCostFlag(aStarTest, 0)
aStarFromKML = setCostFlag(aStarFromKML, 1)

% Testing the optimal path function:
aStarTest = findOptimalPath(aStarTest)
aStarFromKML = findOptimalPath(aStarFromKML)

% Plot them:
plotOptimalPath(aStarTest)
plotOptimalPath(aStarFromKML)

%% Test getting the coordinate list:
coordList = coordsFromOptimalPath(aStarFromKML.optimalPath)
```

APPENDIX F. POTENTIALFIELDTOWAYPOINT.M

```
%% Potential Field to Waypoint Function
% Matt Audette

% This is a function based off of Calvin Hargadine's thesis script,
% 'p3ATnavigation.m'. The goal of this is to treat his code like a
% "black box"that mine feeds formatted GPS coordinates in to.

% His code initiates the robot and takes in a single waypoint from a
% user prompt. I'm going to make a change that the input comes from a
% pair of points passed from this function.

function potentialFieldToWaypoint( coordinates, goalnum )
%'coordinates' is the [x,y] values that will be the goal.

    % Pioneer 3-AT Localization and Navigation Script
    % Incorporating Potential Field function for navigation and
    % GPS/IMU through Kalman Filter for localization

    %%%% ENSURE ROS MASTER NODE IS STARTED AND MATLAB NODE GENERATED
    %%%% PRIOR TO RUNNING THIS SCRIPT -- USE rosinit

    %% Setup and parameter initialization
    % Create global variables for use in communicating with ROS system
    global Pose
    global Laser
    global Goal
    global NavStatus
    global GPSFix

    % Create ROS publishers, subscribers, and service client
    poseSub = rossubscriber('/geonav_p3odom',@p3atPoseCallback)
    laserSub = rossubscriber('/scan',@p3atLaserCallback)
    cmdPub = rospublisher('/RosAria_Node/cmd_vel','geometry_msgs/
Twist')
    goalPub = rospublisher('/nav/goal_odom','nav_msgs/Odometry')
    casePub = rospublisher('/current_case','std_msgs/String')
    goalSub = rossubscriber('/geonav_goalodom',@p3atGoalCallback)
    navstatusSub = rossubscriber('/nav/status',@p3atNavStatusCallback)
    fixSub = rossubscriber('/gps/fix',@p3atGPSFixCallback)
    client = rossvcclient('/reset_kf')

    % Pause for publisher/subscriber registration
    pause(2)

    % Create empty messages for publication
    caseMsg = rosmessage(casePub)
    cmdMsg = rosmessage(cmdPub)
    goalMsg = rosmessage(goalPub)
```

```
% Get parameters and goal information the robot
[param, sto_goals] = robotConfigReader_multigoal;

% Ask user for desired goal number
%goalnum = input('Enter desired WP number (from 1 to 10):');
%current_goal = goal(goalnum,:);
current_goal = coordinates;

% Publish initial goal message for ROS system transform
for k = 1:5
    goalMsg.Pose.Pose.Position.X = current_goal(-5);
    goalMsg.Pose.Pose.Position.Y = current_goal(2);
    goalMsg.Pose.Pose.Orientation.X = 0;
    goalMsg.Pose.Pose.Orientation.Y = 0;
    goalMsg.Pose.Pose.Orientation.Z = 0;
    goalMsg.Pose.Pose.Orientation.W = 1;
    send(goalPub,goalMsg);
    pause(0.1)
end

% Get current NavStatus message
navstatus = NavStatus.Data';

% Ensure NavStatus is good (2) and if not, reset KF
if navstatus(1) ~= 2
    call(client)
else
end

% Define parameters for navigation algorithm
K1 = param(3);              % forward velocity gain
K2 = param(2);              % turning velocity gain
maxvel = 3;                 % maximum velocity of robot
laser_max = 20;             % robot laser view horizon
goaldist = 0.5;             % distance metric for reaching goal
goali = 1;                  % current goal index
xi = param(5);              % attractive force gain
eta = param(4);             % repulsive force gain
d = param(1);               % distance above which robot velocity
                            % is constant
rho0 = param(6);            % offset from obstacle to ignore
                            % repulsive term
c = 1;                      % initial case variable
navrun = 0;                 % navigation fix status variable

% Define parameters for wall-following algorithm
angK = 1;                   % turning velocity gain for WF
                            % algorithm
linK = 1;                   % forward velocity gain for WF
                            % algorithm
g_dist = [];                % initialize goal distance
g_dist0 = [];               % initialize initial goal distance
Dcount = 0;                 % goal distance counter
N_Buffer = 20;              % number of measurements used to
```

```
                                % average repulsive force
    Frep_Buffer = zeros(N_Buffer,1);    % initialize repulsive force
                                % buffer

    % Output velocity filter parameters
    Kfilterold = 0.6;           % percentage of old velocity used
    Kfilternew = 0.4;           % percentage of new velocity used
    LinearVel_old = 0.0;        % initialize linear velocity
    AngularVel_old = 0.0;       % initialize angular velocity

    %% Potential Field Algorithm
    while 1                     % Infinite loop until goal is reached
        % publish goal coordinates
        goalMsg.Pose.Pose.Position.X = current_goal(2);
        goalMsg.Pose.Pose.Position.Y = current_goal(1);
        goalMsg.Pose.Pose.Orientation.X = 0;
        goalMsg.Pose.Pose.Orientation.Y = 0;
        goalMsg.Pose.Pose.Orientation.Z = 0;
        goalMsg.Pose.Pose.Orientation.W = 1;
        send(goalPub,goalMsg);

        % get the laser ranges
        laser_range = Laser.Ranges;

        % angular resolution vector
        laser_angle =
(Laser.AngleMin:Laser.AngleIncrement:Laser.AngleMax)';

        % get goal coordinates in XY world frame
        q_goal = [Goal.Pose.Pose.Position.X,
Goal.Pose.Pose.Position.Y];

        % get current GPS fix
        gpsfix = [GPSFix.Status.Service,GPSFix.Status.Status]

        % get current nav status
        navstatus = NavStatus.Data'

        % if good nav status, set nav status variable
        if navstatus(1) == 2
            navrun = 1;
        else
        end

        % if bad nav status with previous good fix and good GPS fix,
        % reset KF
        if navstatus(1) == 3 && navrun == 1 && gpsfix(2) == 30
            call(client)
            navrun = 0;
        else
        end

        % switch/case for algorithm decision logic
```

```
        switch c
            case 1              % Potential Field Algorithm
                fprintf('Potential Field\n')
                caseMsg.Data = 'Potential Field';    % publish current
                  % case to ROS
                send(casePub,caseMsg)

                % get X, Y and Theta
                pose = Pose.Pose.Pose;
                quat = pose.Orientation;
                angles = quat2eul([quat.W quat.X quat.Y quat.Z]);
                yaw = angles(1);
                x = pose.Position.X;
                y = pose.Position.Y;
                th = yaw;

                fprintf('X: %f, Y: %f, Theta: %f \n',x,y,th);

                % call the attractive force function
                wp_x = q_goal(goali,1);
                wp_y = q_goal(goali,2);
                [dist, angvel, linvel] = attforcepot(x,y,th,wp_x,wp_y);

                % evaluate what to do next based on the distance to the
                % waypoint.
                if (dist <= goaldist)
                    % if you have reached the goal
                    if (goali < size(q_goal,1))
                        % if there are multiple goals
                        disp('Going to next waypoint!');
                        goali = goali+1;
                    else
                        % if there is a single goal
                        fprintf('WP #%d at x: %f, y: %f, Distance: %f\
n',goalnum,wp_x,wp_y,dist);
                        cmdMsg.Linear.X = 0.0;
                        cmdMsg.Angular.Z = 0.0;
                        fprintf('Publishing cmd_vel with lin. vel: %f,
ang. vel.: %f\n', ...
                            0.0,0.0);
                        send(cmdPub,cmdMsg);
                        disp('Done!')
                        break;      % exit while loop as final goal is
                                    % reached
                    end
                else
                    % goal not yet reached
                    fprintf('WP #%d at x: %f, y: %f, Distance: %f\
n',goalnum,wp_x,wp_y,dist);
                    if (dist <= d)
                        goalvelx = linvel;
                        goalvelw = angvel;
                    else
                        goalvelx = maxvel;
                        goalvelw = angvel;
```

```
                end
            end

            pause(0.1)          % pause for ROS system

            Frept = [0;0];      % initialize repulsive force

            for i = 1:1032
                if laser_range(i) <= laser_max
                    % object position in the laser i coordinate in
                    % meters
                    p_laser = [laser_range(i) 0 0 1]';
                    Xobj = cos(laser_angle(i))*p_laser(1);
                    Yobj = sin(laser_angle(i))*p_laser(1);
                    rho = sqrt(Xobj^2+Yobj^2);
                    if rho < rho0
                        Frep = eta*(1/p_laser(1)-1/
rho0)*(1/(p_laser(1)^2))*[-cos(laser_angle(i)) -sin(laser_angle(i))]';
                    else
                        Frep = [0;0];
                    end
                    Frept = Frept+Frep;
                else
                end
            end

            Frep_Buffer = [Frept(2); Frep_Buffer(2:N_Buffer-1)];
            MeanBuffer = mean(Frep_Buffer);

            % calculate total force and build velocity terms
            Fatt = [goalvelx;goalvelw];
            Ftot = xi*Fatt + eta*Frept;
            fprintf('\n\nNorm of Ftot: %f\n',norm(Ftot));
            LinearVel = K1*Ftot(1);
            AngularVel = K2*Ftot(2);

            % determine which case to enter next
            if min(laser_range) < 0.5
                c = 3;
            elseif norm(Ftot) < 0.5 && dist > 1
                c = 2;
                g_dist0 = dist;
                g_dist = dist;
            else
                c = 1;
            end

        case 2              % Wall-Following Algorithm
            fprintf('\nWall Following\n\n')
            caseMsg.Data = 'Wall Following';    % publish current
                           % case to ROS
            send(casePub,caseMsg)

            % get X, Y and Theta
```

```
                pose = Pose.Pose.Pose;
                quat = pose.Orientation;
                angles = quat2eul([quat.W quat.X quat.Y quat.Z]);
                yaw = angles(1);
                x = pose.Position.X;
                y = pose.Position.Y;
                th = yaw;

                fprintf('X: %f, Y: %f, Theta: %f \n',x,y,th);

                % call the attractive force function
                wp_x = q_goal(goali,1);
                wp_y = q_goal(goali,2);
                [dist, angvel, linvel] = attforcepot(x,y,th,wp_x,wp_y);
                pause(0.1)

                % if closer to the goal than last time, increment DD
                if dist < g_dist
                    Dcount = Dcount + 1
                else
                end

                g_dist = dist;

                Frept = [0;0];      % initialize repulsive force

                for i = 1:1032
                    if laser_range(i) <= laser_max
                        % object position in the laser i coordinate in
                        % meters
                        p_laser = [laser_range(i) 0 0 1]';
                        Xobj = cos(laser_angle(i))*p_laser(1);
                        Yobj = sin(laser_angle(i))*p_laser(1);
                        rho = sqrt(Xobj^2+Yobj^2);
                        if rho < rho0
                            Frep = eta*(1/p_laser(1)-1/
rho0)*(1/(p_laser(1)^2))*[-cos(laser_angle(i)) -sin(laser_angle(i))]';
                        else
                            Frep = [0;0];
                        end
                        Frept = Frept+Frep;
                    else
                    end
                end

                % determine angle to the repulsive force vector
                objang = atan2(Frept(2),Frept(1));
                if objang < 0
                    objang = objang + 2*pi;
                else
                end

                objangdeg = objang*180/pi
```

```
                % determine which way to turn and keep repulsive force
                % vector perpendicular with robot heading
                if MeanBuffer > 0
                    if objangdeg >= 100
                        angvel = angK*0.4;
                        linvel = linK*0.05;
                    elseif objangdeg < 80
                        angvel = -angK*0.4;
                        linvel = linK*0.05;
                    else
                        angvel = 0.0;
                        linvel = 0.3;
                    end
                elseif MeanBuffer < 0
                    if objangdeg < 260
                        angvel = -angK*0.4;
                        linvel = linK*0.05;
                    elseif objangdeg > 280
                        angvel = angK*0.4;
                        linvel = linK*0.05;
                    else
                        angvel = 0.0;
                        linvel = 0.3;
                    end
                end

                % develop output velocities
                LinearVel = linvel;
                AngularVel = angvel;

                % determine which case to enter next
                if min(laser_range) < 0.5
                    c = 4;
                elseif Dcount == 70
                    c = 1;
                    g_dist = [];
                    Dcount = 0;
                    Frep_Buffer = zeros(N_Buffer,1);
                else
                    c = 2;
                end

            case 3 % Emergency Avoidance Alg (From Potential Field)
                ii = 0;
                while ii < 5
                    % stop immediately for 5 seconds
                    fprintf('Emergency Avoidance\n')
                    caseMsg.Data = 'Emergency Avoidance (PF)';
                    send(casePub,caseMsg)
                    % populate the message
                    fprintf('WP #%d at x: %f, y: %f, Distance: %f\
n',goalnum,wp_x,wp_y,dist);
                    cmdMsg.Linear.X = 0.0;
                    cmdMsg.Angular.Z = 0.0;
                    % publish message
```

```
                    fprintf('Publishing cmd_vel with lin. vel: %f, ang.
vel.: %f\n', ...
                        0.0,0.0);
                    send(cmdPub,cmdMsg);
                    pause(0.2)
                    ii = ii + 0.2;
                end
                jj = 0;
                while jj < 4
                    % backup for 4 seconds to make enough room to
                    % maneuver around obstacle
                    caseMsg.Data = 'Emergency Avoidance (PF)';
                    send(casePub,caseMsg)
                    fprintf('WP #%d at x: %f, y: %f, Distance: %f\
n',goalnum,wp_x,wp_y,dist);
                    cmdMsg.Linear.X = -0.2;
                    cmdMsg.Angular.Z = 0.0;
                    % publish
                    fprintf('Publishing cmd_vel with lin. vel: %f, ang.
vel.: %f\n', ...
                        0.0,0.0);
                    send(cmdPub,cmdMsg);
                    pause(0.2);
                    jj = jj + 0.2;
                end

                % get the laser ranges
                laser_range = Laser.Ranges;
                % determine if obstacle is out of min range parameter
                if min(laser_range) < 0.5
                    c = 3;
                else
                    c = 1;
                end

            case 4 % Emergency Avoidance Alg (From Wall Following)
                ii = 0;
                while ii < 5
                    % stop immediately for 5 seconds
                    fprintf('Emergency Avoidance\n')
                    caseMsg.Data = 'Emergency Avoidance (WF)';
                    send(casePub,caseMsg)
                    % populate the twist message
                    fprintf('WP #%d at x: %f, y: %f, Distance: %f\
n',goalnum,wp_x,wp_y,dist);
                    cmdMsg.Linear.X = 0.0;
                    cmdMsg.Angular.Z = 0.0;
                    % publish
                    fprintf('Publishing cmd_vel with lin. vel: %f, ang.
vel.: %f\n', ...
                        0.0,0.0);
                    send(cmdPub,cmdMsg);
                    pause(0.2)
                    ii = ii + 0.2;
                end
```

```
                jj = 0;
                while jj < 4
                    % backup for 4 seconds to make enough room to
                    % maneuver around obstacle
                    caseMsg.Data = 'Emergency Avoidance (WF)';
                    send(casePub,caseMsg)
                    fprintf('WP #%d at x: %f, y: %f, Distance: %f\
n',goalnum,wp_x,wp_y,dist);
                    cmdMsg.Linear.X = -0.2;
                    cmdMsg.Angular.Z = 0.0;
                    % publish
                    fprintf('Publishing cmd_vel with lin. vel: %f, ang.
vel.: %f\n', ...
                        0.0,0.0);
                    send(cmdPub,cmdMsg);
                    pause(0.2);
                    jj = jj + 0.2;
                end

                % get the laser ranges
                laser_range = Laser.Ranges;

                % determine if obstacle is out of min range parameter
                if min(laser_range) < 0.5
                    c = 3;
                else
                    c = 2;
                end

            otherwise
        end

        % build filtered output velocity parameters
        cmdMsg.Linear.X = Kfilternew*LinearVel +
Kfilterold*LinearVel_old;
        cmdMsg.Angular.Z = Kfilternew*AngularVel +
Kfilterold*AngularVel_old;

        % publish on cmd_vel topic
        fprintf('Publishing cmd_vel with lin. vel: %f, ang. vel.: %f\
n', ...
            cmdMsg.Linear.X,cmdMsg.Angular.Z);
        send(cmdPub,cmdMsg);

        LinearVel_old = cmdMsg.Linear.X;
        AngularVel_old = cmdMsg.Angular.Z;
    End

end
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX G. MODIFIED READ_KML.M

```
function [x,y,z] = read_kml(fileName)
% READ_KML Reads in (x,y,z) from a GoogleEarth kml file.
%
%  I have tried to make this code as robust as possible, but it may
%  crash or give unexpected resutls if the file is not formatted
%  exactly as expected.
%
% Example:
%   [x,y,z] = read_kml('test.kml');
%
% where test.kml looks like:
% <?xml version="1.0" encoding="UTF-8"?>
% <kml xmlns="http://earth.google.com/kml/2.1">
% <Placemark>
%   <name>test_length</name>
%   <description>junk</description>
%   <LineString>
%       <tessellate>1</tessellate>
%       <coordinates>
% -73.65138440596144,40.45517368645169,0 -
73.39056199144957,40.52146569128411,0 -
73.05890757388369,40.59561213913959,0 -
72.80519929505505,40.66961872411046,0 -
72.61180114704385,40.72997510603909,0 -
72.43718187249095,40.77509309196679,0 </coordinates>
%   </LineString>
% </Placemark>
% </kml>
%
% afarris@usgs.gov 2016March09, now can read mulitple sets of
% coordinates
% afarris@usgs.gov 2006November

%% open the data file and find the beginning of the data
fid=fopen(fileName);
if fid < 0
    error('could not find file')
end

% ADDED BY MATT AUDETTE:
% Without this line, the str2double() function in the very last lines
% truncates the incoming GPS coordinates to 4 decimal places, which
% translates to 11.7 meters N/S and 7.8 meters E/W.
format long
% End Matt Audette's modifications

% This loop reads the data file one line at a time. If if finds the
% word <coordinates>, it knows there is data until it reads the word
% </coordinates>. After loading this data, it keeps reading the file,
% looking for another instance of <coordinates> until it finds the word
% </kml> which signals that the end of the file has been reached.
```

```
% Some files have all the data on one line, others have newline
% characters in various points in the file. I hope this code that
% works in all cases.

done=0;
endoffile = 0;
ar = 1;

while endoffile == 0
    while done == 0
        junk = fgetl(fid);
        f = strfind(junk,'<coordinates>');
        ff = strfind(junk,'</kml>');
        if ~isempty(f)
            done = 1;
        elseif  ~isempty(ff)
            endoffile = 1;
            done = 1;
        end
    end
    if endoffile
        break
    end
    % 'junk' either ends with the word '<coordinates>' OR
    % some data follows the word '<coordinates>'
    if (f + 13) >= length(junk)
        % no data on this line
        % done2 is set to zero so the next loop will read the data
        done2 = 0;
    else
        % there is some data in this line following '<coordinates>'
        clear f2
        f2 = strfind(junk,'</coordinates>');
        if ~isempty(f2)
            %all data is on this line
            % there may be multiple sets of data on this one line
            % I read them all
            for i = 1 : size(f2,2)
                alldata{ar} = junk(f(i)+13:f2(i)-1);
                % I add in whitespace b/c sometimes it is missing
                alldata{ar+1} = ' ';
                ar = ar+2;
            end
            % done2 is set to one because the next loop does not need
            % to run
            done2 = 1;
        else
            % only some data is on this line
            alldata{ar} = junk(f+13:end);
            % I add in whitespace b/c sometimes it is missing
            alldata{ar+1} = ' ';
            ar = ar+2;
            % done2 is set to zero so the next loop will read the
            % rest of the data
            done2 = 0;
```

```
        end
        % check to see if at end of the file
        ff = strfind(junk,'</kml>');
        if  ~isempty(ff)
            % no more data
            endoffile = 1;
            break
        else
            % need to keep looking for more data
            done = 0;
        end
    end

    % If not all the data was on the line with the word <coordiate>,
    % read in the data
    while done2 == 0
        % read in line from data file
        junk = fgetl(fid);
        f = strfind(junk,'</coordinates>');
        if isempty(f) == 1
            % no ending signal, just add this data to the rest
            alldata{ar} = junk;
            ar = ar + 1;
        else
            % ending signal is present
            done = 0;
            if f < 20
                % </coordinates> is in the begining of the line, ergo
                % no data on this line; just end the loop
                done2 = 1;
            else
                % the ending signal (</coordinates>) is present: remove
                % it, add data to the rest and signal the end
                % of the loop
                f2 = strfind(junk,'</coordinates>');
                alldata{ar} = junk(1:f2-1);
                ar = ar + 1;
                done2 = 1;
                disp('done with line')
            end
        end
        % check to see if at end of the file
        ff = strfind(junk,'</kml>');
        if  ~isempty(ff)
            % no more data
            endoffile = 1;
            break
        else
            % need to keep looking for more data
            done = 0;
        end
    end
end
fclose(fid);
```

```
%% get the data into neat vectors
%  I have to divide the string into X, Y and Z values.
%
% This is hard b/c there is no comma between points
% (just commans between x and y, and between
% y and z)  ie;  -70.0000,42.0000,0 -70.1000,40.10000,0 -70.2,....
%
% I used to do this by finding commas and spaces, now I use
% 'strsplit'!  Thank you Matlab!

% 'alldata' is one huge cell
% turn alldata into regular vector so it is easier to work with
data = cell2mat(alldata);
% data is one huge string, split it so there is seperate element for
each number
C = strsplit(data,{',',' '});
% sometimes first and/or last element in C is empty, this causes
problems
len = size(C,2);
if isempty(C{1}) && isempty(C{end})
    D = C(2:len-1);
elseif isempty(C{1}) && ~isempty(C{end})
     D = C(2:end);
elseif isempty(C{end}) && ~isempty(C{1})
     D = C(1:len-1);
end

% There has GOT to be a better way to split C into 3 variables!
a = 1;
for i = 1 : 3: length(D)-2
    x(a,1) = str2double(D{i});
    a=a+1;
end
a=1;
for i = 2 : 3: length(D)-1
    y(a,1) = str2double(D{i});
    a=a+1;
end
a=1;
for i = 3 : 3: length(D)
    z(a,1) = str2double(D{i});
    a=a+1;
end
```

LIST OF REFERENCES

[1] Department of the Navy. "Marine Corps operating concept," Washington, DC, USA, 2016. [Online]. Available: https://www.mccdc.marines.mil/MOC/

[2] P. W. Singer, *Wired for War: The Robotics Revolution and Conflict in the Twenty-first Century*. New York, NY USA: Penguin Press, 2009.

[3] *2017 Marine Corps Warfighting Labs Futures Directorate Initiative Portfolio,* Marine Corps Warfighting Lab Futures Directorate, Quantico, Virginia, 2016. [Online]. Available: https://www.mcwl.marines.mil/Portals/34/Documents/Portfolio/2017_MCWL_FD_InitiativePortfolio_small.pdf

[4] *2018 U.S. Marine Corps S&T Strategic Plan*, Marine Corps Warfighting Lab Futures Directorate, Quantico, Virginia, 2016. [Online]. Available: https://www.onr.navy.mil/-/media/Files/About-ONR/2018-USMC-S-and-T-StrategicPlan.ashx?la=en&hash=73B2574A13A8EC6AAE60CF4670E05C6F97309B8F

[5] DVIDS. "Marines test new futuristic equipment, capabilities [Image 9 of 10]," July 13, 2016. [Online]. Available: https://www.dvidshub.net/image/2726228/marines-test-new-futuristic-equipment-capabilities

[6] C. S. Hargadine, "Mobile robot navigation and obstacle avoidance in unstructured outdoor environments," M.S. thesis, Dept. of Electrical and Computer Engineering, NPS, Monterey, California, 2017. [Online]. Available: https://calhoun.nps.edu/handle/10945/56937

[7] J.-C. Latombe, *Robot motion planning*, 2nd ed. Boston: Kluwer, 1991.

[8] Omron Adept MobileRobots. "Pioneer 3-AT Specification Sheet," 2011. [Online]. Available: http://www.mobilerobots.com/Libraries/Downloads/Pioneer3AT-P3AT-RevA.sflb.ashx

[9] Omron Adept MobileRobots. "MobileRobots Research Mobile Robot Platforms Compare Technical Specifications," Accessed July 8, 2018. [Online]. Available: http://www.mobilerobots.com/ResearchRobots/ResearchMatrix.aspx

[10] CappuccinoPC.com. "SlimPRO SP675P Mini PC," Accessed July 8 2018. [Online]. Available: http://www.cappuccinopc.com/slimpro-sp675p.asp

[11] Hokuyo Automatic Co. "Scanning Rangefinder Distance Data Output/UTM-30LX Product Details," Accessed June 14 2018. [Online]. Available: https://www.hokuyo-aut.jp/search/single.php?serial=169

[12] LORD Corporation, *LORD MicroStrain 3DM-GX5-45 GNSS-Aided Inertial Navigation System Datasheet,* 8400–0091, 2018. [Online]. Available: http://www.microstrain.com/sites/default/files/3dm-gx5-45_datasheet_8400-0091.pdf

[13] MathWorks. "What is MATLAB?" Accessed July 11, 2018. [Online]. Available: https://www.mathworks.com/discovery/what-is-matlab.html

[14] MathWorks. "Robotics System Toolbox," Accessed July 11, 2018. [Online]. Available: https://www.mathworks.com/products/robotics.html

[15] A. Farris, United States Geological Service, 2016. read_kml, ver 2. [Online]. Available: www.mathworks.com/matlabcentral/fileexchange/13026-read_kml

[16] Robot Operating System. "About ROS." Accessed April 10, 2018. [Online]. Available: http://www.ros.org/about-ros/

[17] MathWorks. "Get Started with ROS - MATLAB & Simulink." Accessed May 5, 2018. [Online]. Available: https://www.mathworks.com/help/robotics/examples/get-started-with-ros.html

[18] Robot Operating System. "Recording and playing back data." Accessed April 15, 2018. [Online]. Available: http://wiki.ros.org/ROS/Tutorials/Recording%20and%20playing%20back%20data

[19] Google. "Download Google Earth Pro." Accessed July 11, 2018. [Online]. Available: https://www.google.com/earth/download/gep/agree.html

[20] R. Siegwart and I. R. Nourbakhsh, *Introduction to autonomous mobile robots*. Cambridge, MA: MIT Press, 2004.

[21] Tomas Lozand-Perez and M. A. Wesley, "An Algorithm for Planning Collision-Free Paths Among Polyhedral Obstacles," *Communications of the ACM*, vol. 22, no. 10, pp. 560–570, Oct. 1979.

[22] "Establishing Connectivity in a Visibility Graph," class notes for EC4310 Fundamentals of Robotics, Dept. of Electrical and Computer Engineering, Naval Postgraduate School, Monterey, CA, USA, Winter 2018. .

[23] P. E. Hart, N. J. Nilsson, and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," *IEEE Trans. Syst. Sci. Cybern.*, vol. 4, no. 2, pp. 100–107, Jul. 1968.

[24] MathWorks. "Why Use Object-Oriented Design." Accessed January 20, 2018. [Online]. Available: https://www.mathworks.com/help/matlab/matlab_oop/why-use-object-oriented-design.html

[25] M. Audette, “GitHub Repository: Thesis,” GitHub, February 4, 2018. [Online]. Available: https://github.com/MattAud/Thesis

[26] M. Audette, “Obstacle Map to Optimal Path,” YouTube, August 20, 2018. [Online]. Available: https://www.youtube.com/watch?v=Y0Jkl2ozzEE

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California