Simulation of an Indoor Autonomous Drone for Assisting Police in Mass Shootings

Massachusetts Academy of Math and Science

Rumaisa Abdulhai
rabdulhai@wpi.edu

Table of Contents

1. Abstract

Mass shootings have claimed thousands of lives in the United States as there are no effective mechanisms to prevent them. After the 2018 Parkland shooting, I was thinking about how I could use technology to protect my classmates if a shooting were to happen at my school. This research investigated how autonomous drones could assist law enforcement during school shootings. An autonomous drone application was developed using the Robot Operating System, Gazebo, and Python that enabled a quadcopter to navigate to a precise location inside a school where the perpetrator and victims of a mass shooting are likely to be found. The application consists of four modules: design, mapping, navigation, and live streaming. First, a 2D occupancy grid-map of a middle school was achieved using a quadcopter, 2D Hokuyo LiDAR, and a SLAM algorithm. With a teleoperated drone, the mapping component took 7 hours with GMapping, and 2.5 hours with Google Cartographer for a complex school. With frontier exploration and GMapping, the drone mapped the school autonomously in 3.5 hours. Next, using Adaptive Monte-Carlo Localization with Dijkstra's global and Dynamic Window Approach local path planning algorithms, the drone autonomously navigated while avoiding obstacles from the courtyard to a classroom 43 meters away in 171 seconds at an average speed of 0.25 m/s. The live-streaming component used an onboard camera to transmit live video to a web server that police could monitor. This autonomous drone application demonstrates that indoor drones could inform police of real-time dangers during school shootings, enabling them to take timely actions to save precious lives.

2. Literature Review: Autonomous Robotics Systems

2.1. Introduction

The field of robotics consists of computer-controlled electromechanical systems using various sensors to navigate the environment and manipulate objects around them. These robotic systems are currently utilized in a variety of fields, from assembly lines and medical surgery to space exploration and self-driving cars. As automation becomes more entrenched in humans' lives, the perceived potential of these robotics systems has shifted. Assembly lines, once thought to be a radical invention, are much more predictable compared to the current applications of robotics systems such as self-driving cars or fighting fires, where they are expected to perform actions in highly uncertain and uncontrolled environments. Sensor measurements become critical in such situations and must be utilized to take decisive action under uncertainty in uncontrolled environments.

With the advent of autonomous drones or unmanned aerial vehicles (UAVs), there have been various outdoor applications in fields such as agriculture, cinematography, and delivery. Drones have reduced the time for crop and soil inspection and have made the process of inspection more consistent. One drone, namely the Agras DJI octocopter, can spray fertilizer with its four nozzles on up to 6000 square meters of land in 10 minutes (Puri, Nayyar, & Raja, 2017). Drones are also being used for delivering medical aid. The Zipline Company uses drones to deliver blood packages to hospitals in Rwanda. In addition to cutting standard delivery time from hours to minutes using GPS technology, the zipline drones have also saved the lives of patients who suffer from blood loss during surgery (Ackerman & Strickland, 2018). The true feat of autonomous drone technology can be seen in the new Skydio 2, which can follow an individual participating in extreme outdoor sports at high speeds while avoiding obstacles and recording high-quality footage at the same time (Skydio Inc, 2019). Autonomous drones are making an enormous impact in the agriculture and the cinematography industry and are expected to contribute to other domains, including fighting urban and forest fires, law enforcement, and safety inspection of national infrastructures such as roads, dams, and bridges.

While there have been many outdoor drone applications, building an indoor drone application has unique challenges: the drone must navigate through narrow and tight spaces while avoiding obstacles and fly in a

GPS-denied environment (Khosiawan & Nielsen, 2016). Therefore, there must be an alternative to GPS. SLAM, or Simultaneous Localization & Mapping, is essential for indoor navigation. SLAM is the methodology of estimating the location of a robot in an environment while trying to form a map of the environment, using sensors such as lasers, monocular cameras, or binocular cameras (Li, Bi, Lan, Qin, Shan, Lin, & Chen, 2016). Once a map is established and the drone is aware of its position in the environment, a path planning algorithm is necessary for the drone to efficiently navigate from a source point to a destination point while avoiding any obstacles in its path.

This project focuses on building an autonomous indoor application to assist police during mass shootings. The project will explore the use of autonomous indoor drones in providing effective mechanisms to collect real-time information about the precise location and perpetrator(s) of a mass shooting to reduce the loss of life significantly. The goal of this project is to simulate mapping, path planning, and navigation of an indoor drone to the point of emergency, collect real-time information about the location of the perpetrators involved and relay it to a trusted group of individuals such as law enforcement who then can take appropriate action to save precious human lives. The literature review will primarily cover the development of localization, mapping, and path planning methods and algorithms to date in the field of autonomous robots and drones. It will also cover the usage of various sensors used in terrestrial and aerial robots. The following provides an organization of the literature review. Section 2.2: Interfaces and Programs provides a brief survey and overview of the software components that could potentially be used in this project. Section 2.3: Sensors, surveys the various sensor devices that are used in mobile robotics currently. Section 2.4: Probabilistic Robot Interaction reviews the fundamental principles and filters used in robotics that are relevant to this project. Section 2.5 surveys the path planning algorithms used in the mobile robotics arena. Section 2.6 finally provides conclusions and pointers for undertaking the project after the literature review.

2.2. Interface and Programs

This section gives an overview of the types of drones and the software platforms commonly used in the research of autonomous vehicles that could potentially be used in this project.

2.2.1. Research vs. Commercial Drones

Drones can be classified into two broad categories: commercial and research drones. Commercial drones are off-the-shelf drones that are to be used for the purpose they were built, such as aerial photography or spraying fertilizer. Commercial drones lack the processing power or the software development platform to modify them for other purposes. In addition, they have sensors limited to the application for which they are built. On the other hand, research drones have considerable computing power, a wide variety of sensors, and a rich development environment that allows developers to develop and test many different autonomous drone applications. The other significant difference between a commercial drone and a research drone is that a research drone has an onboard computer that makes decisions during navigation in real-time, while most commercial drones are remote-controlled. Research drones can process computation-heavy algorithms on board and utilize the information gained to perform complex tasks. Research drones are also customizable with various sensors such as stereo cameras, laser sensors, or ultrasonic sensors. Hence, research drones are better suited for developing indoor drone applications than commercial drones (Guermonprez, 2017). A research drone model from the hector_quadrotor package provided by the Robot Operating System (ROS) will be used in the first phase of this project to test the flight of an autonomous drone.

2.2.2. Gazebo Simulator

The Gazebo Simulator is a well-known industrial software platform for developing realistic 3D simulations of robots in indoor and outdoor environments. It supports real sensors on the market, such as monocular or stereo cameras and 2D or 3D lasers. It also includes a visualization environment to observe the navigation of an autonomous robot (OSRF).

The gazebo simulator allows one to create virtual custom objects, robots, drones, and environments in which they interact. Various indoor environments such as a school hallway or a hotel lobby can be constructed using the primary cylinder, sphere, and cube shapes provided. One can also choose to import standard models of environments or robots provided by Gazebo rather than build everything from scratch. Some examples of these models include gas stations, walls, warehouse robots, and apartment buildings. The visual appearance, size, and orientation of objects can be changed using the link inspector in the model editor. The object can appear to be large, but only a small part of it is real in the environment.

Gazebo environments are known as worlds. One can choose how to view the world by changing the perspective of the camera. There is also a cube icon that allows one to view their world from common perspectives such as top view, front view, right view, and side view. Meshes are used to insert custom 3D objects with complex size measurements and are not necessarily defined by a certain width, length, and height. Joints are used to connect objects in a custom model. There are various joints provided in the Gazebo simulator, including fixed, revolute, and screw joints. An object can turn into a real sensor using plugins. There are plugins for specific cameras and laser models available online. For example, one can use a Kinect depth sensor plugin to record depth information about its surroundings (OSRF).

These features of Gazebo help developers build indoor and outdoor environments such as an office building or an outdoor amusement park. This project will use Gazebo to construct an indoor school environment to simulate and test an autonomous drone application to help police during mass shootings.

2.2.3. Robot Operating System

The Gazebo Simulator is integrated with another software platform called the Robot Operating System (ROS). ROS, supported by the Open Source Robotics Foundation, provides the infrastructure for building autonomous robots. It is a powerful tool for autonomous robot programming as it is open source and applies to multiple programming languages, including Python, Java, and C++. ROS also has various libraries for building and writing code. With a subscriber/publisher system, multiple parts of the robot can listen to sensor output by subscribing to topics where the information is published and can take appropriate action based on these measurements (OSRF). This project will utilize ROS as the primary development environment to build an autonomous drone application.

2.3. Sensors

This section provides a survey of the most important sensor devices used in the field of autonomous robots. Sensors are devices that enable a robot or drone to figure out the state of itself and the objects around it using processes such as mapping and obstacle avoidance. In a way, sensors are the eyes, ears, and the skin of the robot or drone that provide the vital capabilities of sight, hearing, and touch to discover the objects in the environment.

Sensors could be as simple as a touch switch that senses whether a door is open or closed or a more complex device such as LiDARs (Light Detection and Ranging) used by self-driving cars to know the exact shape and size of the objects on the road. Some sensors, such as cameras, are passive, which means they do not emit signals but rather take screenshots of the environment. Active sensors such as Sonars (Sound Navigation and Ranging) emit sound pulses and then detect its echo to locate the distance to obstacles. Sensors can be broadly classified based on the signals they use: ultrasound, electromagnetic, or vision (Matarić 2008).

2.3.1. Ultrasound sensors

Ultrasound sensors, otherwise known as sonar sensors, use sound waves in excess of human audible frequencies (above 40KHz) to detect objects that could be at a distance of about 1-100 feet using the principles of echo and reflection of sound. Sonars have found use in medical imaging as well as mobile robotics to detect obstacles. Although ultrasound sensors are relatively inexpensive, the distances calculated are not very accurate as the sound waves could suffer specular reflection from smooth surfaces, and the emitted sound pulses may not return to the robot. This drawback is avoided by modifying the surfaces of objects in the environment and making them rough to cause proper bounceback of sound signals. The other approach called active perception to make ultrasound sensors more accurate is for the robot to move around, make several measurements at different angles, and take the average distances. Despite their disadvantages, ultrasound sensors have found successful applications in outdoor robots to map complex surroundings as they are affordable. Their applications in indoor drone applications are somewhat limited due to specular reflection (Matarić 2008).

2.3.2. Electromagnetic sensors

Electromagnetic sensors use radio, infrared, or visible electromagnetic waves for object detection, navigation, and mapping. When used in the visible spectrum, they are called lasers. Unlike ultrasound sensors, lasers do not suffer from the drawback of specular reflection. They can map their surroundings and detect obstacles to a very high degree of accuracy down to millimeters. However, lasers cannot detect very close objects as it cannot distinguish between emitted and reflected waves due to the high speed of light. Phase shift measurements are used to overcome this deficiency to estimate the distance to the obstacle instead of using the speed of light. Lasers are relatively expensive when compared with ultrasound sensors. LiDARs, which also use light waves and follow

similar principles as lasers, are even more expensive and could also be bulky. These sensors are used in high demanding applications such as terrestrial robots and to a limited extent in drones because of their very high accuracy and range of perception of several hundred meters. In the preliminary phase of this project, a drone model with a LiDAR sensor is used in Gazebo and ROS to map out the initial environment as it is readily available as a software package (Matarić 2008).

2.3.3. Visual sensors

Cameras are passive sensors that fall into the category of visual sensors as they do not emit a signal but take pictures of the environment from different angles to estimate the edges of obstacles to avoid them. Camera-based sensors could be monocular with one camera or binocular with two cameras. Binocular vision sensors are also called stereo cameras. While monocular camera sensors are planar images, stereo cameras provide depth and thus can construct 3D images. The basic principle of camera sensors is to take several pictures or frames successively at different angles and reconstruct a 3D picture with edges of objects in the environment. In addition to the objects in the field of vision, the images are also corrupted with usual noise and shadows. Special processing is needed to remove this corruption of images by the process of convolution. Convolution is applying filters to images to remove noise or other corruption such as shadows. Camera-based sensors are relatively inexpensive similar to ultrasound sensors, but the downside is they require very high signal processing as they have to process potentially hundreds or thousands of frames of images in real-time. So the robots using them must be equipped with high computing power such as a special purpose GPU (Graphics Processing Unit) from either Intel or Nvidia. Although LiDAR sensors are suitable to use in the initial phase of this project, stereo-based camera sensors are more practical to employ in drones due to their ubiquitous availability and cost (Matarić 2008).

2.4. Probabilistic Robot Interaction

This section reviews the fundamental principles, notation, and filters used in robotics that are relevant to this project. Sensor measurements, control actions, and states of the robot have unavoidable uncertainty in them, and therefore the mathematics of probability plays a significant role in robotics. Sensor measurements, control actions,

and states of the robot are all modeled as random variables in robotics and have probability density functions that allow one to estimate the probable pose of the robot or drone.

2.4.1. State Estimation and Control

The state is a collection of information about the robot as well as the environment. The dynamic state includes the parameters that change as the robot moves around. State at a certain time t can be noted with the symbol x_t. The state includes everything from the robot's location and orientation, often referred to as a robot's pose, to the velocities and locations of moving objects in the surrounding environment. The state of the robot is updated at discrete time steps using an important principle known as the 'Markov property.' The Markov property states that given the current state x_t of a robot and all its previous states $x_{t-1}, x_{t-2 \ldots}$ the next state x_{t+1} depends only on the current state x_t and is independent of all the previous states. This simple principle allows us to calculate the future states of a robot rather accurately using the theory of conditional probability.

The first type of interaction a robot can have with the environment is to take a sensor measurement, such as a laser scan. Measurement data at a particular time t can be noted with the symbol z_t. On the other hand, control actions involve the robot using its actuators and motors to move or move other objects in the environment. Control data can be noted with the symbol u_t. Even if the robot does not move any of its motors or actuators, the mere fact that time has elapsed is considered a control action, and the robot state has to be updated. An odometry sensor can be used to collect control data about the revolution of a robot's wheels, resulting in the state being updated. The state (x_t), control action (u_t), and sensor measurement (z_t) are the primary random variables that are used in the algorithms of probabilistic robotics.

2.4.2. Bayes Filter

The autonomous robot or drone needs to estimate its position and orientation in an environment to execute its task successfully. The Bayes filter is a method or algorithm used to compute the position and orientation of the robot or other objects in its environment. The belief of a robot describes the set of probabilities about its current state in an environment. The belief at time t can be noted as bel(x_t). At the beginning of time step t, the robot is in state x_{t-1}. During time step t, the robot takes an action u_t and moves to x_t. The robot then takes a measurement z_t and updates its state to x_t (Thrun, Burgard, & Fox, 2010). This basic sequence of taking first an action followed by a

sensor measurement at every time step is how the robot updates its own state and the state of its environment. This sequence of steps is combined in a basic algorithm known as the Bayes Filter given in Figure 2.1. The Bayes Filter is one of the essential concepts that form the basis for many other algorithms in the field of robotics.

1: **Algorithm Bayes_filter(**$bel(x_{t-1}), u_t, z_t$**):**
2: for all x_t do
3: $\overline{bel}(x_t) = \int p(x_t \mid u_t, x_{t-1})\ bel(x_{t-1})\ dx$
4: $bel(x_t) = \eta\ p(z_t \mid x_t)\ \overline{bel}(x_t)$
5: endfor
6: return $bel(x_t)$

Figure 2.1: Pseudocode for Bayes Filter (Thrun, Burgard & Fox, 2010).

The goal of the Bayes filter is to calculate the current belief of an autonomous robot/drone based on the previous belief and current measurement and control data. The steps in Figure 2.1 are briefly explained below for any time step *t*:

1. This step just defines inputs used by the Bayes filter. The previous belief bel(x_{t-1}), control action u_t the robot intends to take, and the acquired measurement z_t are used as inputs.
2. This step initiates a for-loop for all possible values of the state x_t.
3. The robot takes action u_t at x_t. The input's prior belief, bel(x_t - 1) and control action u_t are used to calculate the intermediate belief.
4. After taking the sensor measurement z_t, the robot re-calculates the final belief at x_t.
5. This step ends the for-loop.
6. The new calculated belief becomes the previous belief for the next time step t+1 and the algorithm continues until the robot reaches its intended goal.

The Bayes filter forms the fundamental basis for all other filter approaches and is therefore applicable broadly. The Kalman filter is an extension of the Bayes filter for continuous state spaces with the assumption that the state space is normally or Gaussian distributed.

2.4.3. Simultaneous Localization and Mapping

Simultaneous Localization and Mapping (SLAM) is the problem of building a map of an indoor environment while tracking the robot's pose in real-time using sensors such as lasers or cameras. Real-time processing allows for information to be collected from camera or laser positions and estimate the pose rather accurately. The first research on SLAM was conducted in 1986 on terrestrial robots with lasers by Smith and

Cheeseman (Kudan 2016). In Visual SLAM, stereo cameras or a combination of monocular cameras are used to map the environment. Visual SLAM works by tracking a set of features or points in the environment through successive camera frames and then trying to find their 3D location while at the same time using that information to calculate the camera poses at those times. SLAM can be seen as an optimization problem to minimize the error between the tracked and expected locations, which is called bundle adjustment. Multi-core processor machines are fundamental to being able to run this algorithm called bundle adjustment. Using a multi-core processor, the localization part can separately run in real-time on one thread while the mapping thread runs bundle adjustment on the map on the other thread. This project will use the gmapping package provided by ROS, which implements SLAM.

2.5. Path Planning

This section reviews the various path planning algorithms used in the field of robotics. The general robotics path planning problem consists of a robot trying to find the optimal or fastest path between two points given a map of the environment and the robot's localization. Path planning has various applications in autonomous mobile robots as well as network routing, video games, and gene sequencing. The same method is also used to find the shortest path for data packets on the internet.

For path planning algorithms to execute, the environment's map must be interpreted properly. The most common method used to build a map and interpret it is called discrete approximation. In this method, the whole environment is divided into a square grid. Each square within the grid is called a node that is connected to other nodes or squares by edges. The resulting map is known as an occupancy grid map where obstacles, free space, and unexplored space are discretely marked with separate squares. The free space is designated with white squares, the obstacles space is designated with black squares, and the unexplored space is marked with gray squares.

There are two broad categories of path planning algorithms; grid or graph-based planning and sample-based planning algorithms. Algorithms such as Dijkstra's, A Star, and D Star fall into the grid-based category, while Rapidly Exploring Random Tree (RRT) and its variations fall into the sample-based category. Dijkstra's Algorithm is one of the earliest path planning algorithms (Correll, 2014). Invented by Edsger W. Dijkstra, it finds the shortest path between any two nodes on a graph of nodes connected by weighted edges (Yan, 2014).

2.5.1. A Star Algorithm

A Star also finds the shortest path, but usually between two points on a coordinate grid. The algorithm uses two lists to keep track of visited and unvisited nodes, with the starting node in the visited list. Adjacent obstacle-free nodes are added to the unvisited list. For each adjacent or child node of the current node, the sum of the g and h values are calculated to return the f value. The g value is the distance from the starting to the current position. The h value is the heuristic, which is the estimated cost from the current to the goal position (Wenderlich, 2011). The node with the minimum f value becomes the next current node and is added to the visited list. The algorithm continues until the goal node is added to the visited list. The path is generated by referencing the parent of the current node until the starting node is reached (Swift, 2017). D Star extends on A Star in that it allows part of the path to be replanned around an obstacle. However, if the graph is extremely large, A Star and D Star take much longer due to the high calculations required. This weakness in grid-based planning is addressed in sample-based planning algorithms such as RRT (Correll, 2014).

2.5.2. Rapidly-Exploring Random Tree Algorithms

The Rapidly-Exploring Random Trees (RRT) Algorithm, invented in 1998 by LaValle & Kuffner, is an online method of building a tree to explore a region of free space (Williams & Frazzoli, 2010). The RRT Algorithm works by randomly generating points in the free space and trying to connect each point to its closest neighbor point. When points are chained into a tree, it is made sure that the connections do not pass through obstacles and avoid obstacles instead. When a randomly generated point lies in a certain radius of the goal, the algorithm finishes execution. Although the algorithm is simple to implement and takes a very short amount of time to execute, RRT does not find the shortest path. Its graph is very cubic and makes irregular paths (Chin, 2019).

The RRT Star algorithm was made to fix this problem. RRT Star, developed by Dr. Karaman and Dr. Frazzoli, builds on top of RRT to find the shortest path. The RRT Star Algorithm has two additions to the RRT Algorithm. The first is that RRT Star finds the costs of the vertices, which are the distances between the points/vertices and their parent vertex. Nodes with cheaper costs replace the nearest nodes. The second addition is that RRT Star rewires the tree every time a new vertex is connected to its nearest neighbor. This rewiring ensures

that the cost of each node is minimized and makes the graph smoother than the RRT algorithm's graph (Chin, 2019). Figure 2.2 below demonstrates the RRT algorithm in action in the presence of obstacles in a python simulator. For this sample implementation, the original open-source code, which simply generated random points in the sample space, was used and modified to work with obstacles and a node class that draws out the final path. The expanding tree of nodes is shown in white, the obstacles are shown in red, and the highlighted blue line is the path found in the presence of obstacles.

In the first phase of this project, Dijkstra's algorithm is expected to be used for path planning. In a later phase, other specialized path planning algorithms like RRT Star or A Star are expected to be used.

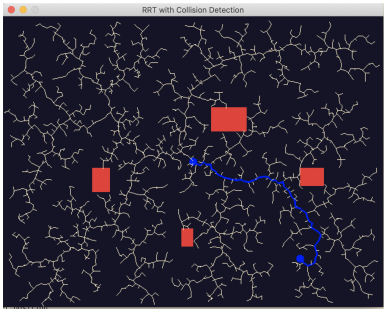

Figure 2.2: RRT Algorithm in action with obstacle avoidance (Abdulhai, 2019).

2.6. Conclusion

This literature review has covered the software modules, sensors, localization, mapping, and path planning methods and algorithms that are used currently in autonomous robots and drones with a goal to develop a new autonomous drone application that can help police during mass shootings. Various sensors such as ultrasound, electromagnetic, and vision sensors have been surveyed along with their pros and cons. In the field of mapping, the fundamental filter called the Bayes Filter had been introduced. Other filters, such as Kalman filters, which are extensions of Bayes filters, are also planned to be investigated for use in this project. The most common path planning algorithms such as Dijkstra, A Star, and RRT have been surveyed. A high-level plan for the development of the autonomous drone application to fight mass school shootings is as follows. In the first phase of the project, an initial layout of a simple indoor environment consisting of a couple of school hallways will be designed in Gazebo. The SLAM gmapping package will be used to localize the robot and map the environment using a laser sensor. A

model drone is planned to be imported into Gazebo and then used to navigate from one point to another in the environment autonomously using Dijkstra's path planning algorithm. This will serve as the first proof of concept application with working mapping and navigation software modules. In the second phase, a realistic and complex middle school environment is planned to be created consisting of several school hallways and classrooms. The mapping and navigation modules developed during the first phase will be tested on the complex indoor school environment. Fine-tuning of the mapping and navigation software modules will be performed to achieve the correctness and efficiency of the autonomous indoor drone application. Finally, recording of the scene will be attempted as the drone navigates inside the school. If time permits, a more complex path planning algorithm such as A Star or RRT will be implemented and will be used to replace Djiskstra's global path planning algorithm for efficiently navigating the robot autonomously between two points inside the school environment.

3. Introduction

During the last several years, school shootings have claimed hundreds of precious student lives in the United States. One can never tell when and where the next mass shooting will happen, inciting fear among parents, students, and school staff as well as worshippers in churches, synagogues, and mosques. With lawmakers sharply divided on gun control, there are no effective laws or procedures to prevent mass shootings. After the Parkland High School shooting in Florida where 17 students were fatally shot by a 21-year-old gunman, I had been thinking about how technology can be used to protect my school if a shooting were to happen there. The first potential solution is airport-style checkpoints consisting of metal detectors and scanners at school entrances to deter potential shooters. However, these checkpoints are prohibitively expensive in the equipment, operations, and time delays that they would cause, especially during the opening hours of the school. They would also negatively impact the open character of a typical American public school. Terrestrial or ground robots are a second alternative solution that could be used once a shooting has taken place. Terrestrial robots can approach the shooter without fear of losing a human life. But terrestrial robots are slow, expensive, can be tumbled over by a potential shooter, and can block potential victims from escaping. It is also difficult for terrestrial robots to move in school hallways or climb stairs efficiently. Aerial robots such as drones are a third alternative that could be used in an indoor environment to deter potential shooters as they are fast, relatively inexpensive, and are increasingly being used in outdoor applications such as agriculture and cinematography. They can fly in hallways or stairs with adequate clearance for ground traffic in times of emergency. Therefore, this project investigates the usage of indoor drones to fight mass shootings and develops an autonomous indoor drone application to assist law enforcement during such critical emergencies.

An autonomous drone is a highly complex aerial robot also known as an Unmanned Aerial Vehicle (UAV) that has found applications in agriculture, delivery, and cinematography. Drones have reduced the time for crop and soil inspection. One drone, namely the Agras DJI octocopter, can spray fertilizer with its four nozzles on up to 6000 square meters of land in 10 minutes (Puri, Nayyar, & Raja, 2017). The Zipline Company uses drones to deliver blood packages to hospitals in Rwanda, cutting normal delivery time from hours to minutes, thereby saving the lives of patients who suffer from blood loss during surgery (Ackerman & Strickland, 2018). The true feat of autonomous drone technology can be seen in the new Skydio 2, which can follow an individual participating in extreme outdoor

sports at high speeds while avoiding obstacles and recording high-quality footage at the same time (Skydio Inc, 2019). Outdoor drone applications have been relatively easier to design and test due to the availability of GPS technology that works quite well in outdoor environments. Outdoor drones can localize themselves relatively easily and navigate at very fast speeds to a specified point using GPS technology.

While there have been outdoor drone applications in agriculture, cinematography, and delivery outlined above, building an indoor drone application has unique challenges: the drone must navigate through narrow and tight spaces while avoiding obstacles and fly in a GPS-denied environment (Khosiawan & Nielsen, 2016). An alternative to GPS is therefore required. Simultaneous Localization & Mapping (SLAM) is one such alternative that has been used for indoor drone navigation. SLAM is the methodology of estimating the location of a robot in an environment while trying to form a map of the environment, using sensors such as lasers, monocular cameras, or binocular cameras (Li et al., 2016). Once a map is established and the drone is aware of its position in the environment, a path planning algorithm is necessary for the drone to efficiently navigate autonomously from a source point to a destination point while avoiding any obstacles in its path. The tight spaces of hallways, classrooms, and stairs reduce the speed at which drones can travel within an indoor environment.

The following provides my vision of how autonomous drones or UAVs could be used to fight shooters and protect potential victims during mass school shootings. When an intrusion is detected or a gunshot is fired in a school, one or more drones stationed indoors could be activated by a sensor to fly to the scene within minutes or even seconds. One drone could track the perpetrator, another could inform potential victims the safest route to exit the building and yet another could comb the school to capture real-time video feed. The drone's onboard cameras can transmit the video in real-time to law enforcement. This way, police are well aware of the scene before they arrive, which will help them greatly in taking appropriate action. It is okay to lose a few inexpensive drones rather than losing precious human lives. Advanced features on drones could possibly be used to disarm the shooter.

Although indoor drone mapping and navigation are much more technically challenging than outdoor, indoor drones hold great promise in applications such as warehousing, hotels, arenas, hospitals, and schools for delivery and security as they are compact, relatively fast, and are able to record and transmit video from scenes. This project, therefore, investigates how an autonomous indoor drone application could be developed and used to solve the urgent social need: assist law enforcement to fight mass shootings and save precious human lives. It will

investigate the use of an autonomous indoor drone to collect real-time information about the precise location and perpetrator(s) of a mass shooting in schools or places of worship. The specific goals of this project are to simulate mapping, path planning, and navigation of an indoor drone in a school environment to a point of emergency, collect real-time information about the location of the perpetrators involved, and relay it to a trusted group of individuals such as law enforcement who then can take appropriate action to save precious human lives. When a gunshot is fired or an intruder is detected, the drone will autonomously navigate to the point of disturbance and serve as a first responder to the scene. The drone's onboard cameras will be able to capture and relay live footage and provide a valuable window to critical pieces of information during a mass shooting to law enforcement, fulfilling the needs of emergency responders. This is expected to reduce setup time and increase communication between the victims and responders thereby saving precious human lives.

The following provides an organization of this project report. *Section 4: Materials and Methods* describes the software packages, methods, and algorithms used in this project. It also covers the ROS mapping and navigation phases, and key parameters that are used in the development of the autonomous drone application. *Section 5: Results* describe the key results obtained during the testing of the developed autonomous drone application including the successful mapping of the model school, navigation of the drone between any two points and streaming of the video from the scene. *Section 6: Discussion and Conclusion* analyzes the results achieved critically and places them into the context of the application in real life. *Section 7: Improvements* details the improvements made on the autonomous drone application over the summer and fall of 2020. *Section 8: References* maintains a list of the sources used throughout this project. *Section 9: Acknowledgements* highlights the key mentors and professors that have guided this project. Finally, *Section 10: Appendices* outlines the limitations, assumptions, challenges, and improvements for this project. It also contains information about the ROS architecture. The project GitHub page with all the code and documentation for the autonomous drone application can be viewed here.

4. Materials & Methods

This section covers the software tools and procedures used in this project to build an autonomous indoor drone application that allows the drone to map an indoor environment, navigate from a source to a destination point, and transmit video from the scene to law enforcement.

4.1. Software Tools

The three key software components used on this project are a Linux machine, the Robot Operating System (ROS), and the Gazebo simulator. A Linux machine running Ubuntu 16.04 provides the basic software foundation on which all the other components reside. ROS provides the infrastructure for building autonomous robots. It is open-source software available in multiple programming languages and contains libraries for building and writing code for autonomous robots. With a subscriber/publisher system, multiple parts of the robot can listen to sensor output by subscribing to topics where the information is published and take appropriate action based on these measurements. There are various ROS versions, but the most commonly used version is ROS Kinetic Kame. Python was used with ROS Kinetic to write the code along with an Integrated Development Environment (IDE) called Visual Studio Code.

Along with a full installation of ROS, the Gazebo simulator is included and used to develop the autonomous drone application. Gazebo is a well-known industrial platform for developing and testing realistic 3D simulations of autonomous robots in both indoor and outdoor environments. It provides support for real sensors on the market such as monocular or stereo cameras, and 2D or 3D LiDARs.

4.2. Installation

The first step in the project was to set up the development environment and download the necessary packages. A Lenovo laptop running Windows was set up with a Linux machine. Windows was uninstalled and Ubuntu 16.04 was installed as it is compatible with ROS Kinetic. Next, a full version of ROS was installed on the Linux machine. A workspace called *catkin_ws* was created with an *src* subfolder. The *hector_quadrotor* stack was cloned into the *~/catkin_ws/src* directory. The *hector_quadrotor* stack provides the drone model used for this project

which is a quadcopter with the Hokuyo 2D LiDAR sensor. The specific package within the stack that provides the drone model is called *hector_quadrotor_description*. In addition, the *gmapping*, *amcl*, and *move_base* packages were installed on the Linux machine. These three packages are used for mapping, localization, and navigation, respectively. The role of these packages will be explained later on. Additional details on installation can be found in Appendix E.

4.3. ROS Packages

The second step in the project was developing packages for the application. The 5 ROS packages created within the *quadcopter_sim* folder were the *quadcopter_gazebo*, *quadcopter_takeoff_land*, *quadcopter_mapping*, *quadcopter_teleop*, and *quadopter_navigation* packages. The first package created in this package was the *quadcopter_gazebo* package. This package launches the indoor 3D environment and instantiates the quadcopter at the origin. The second package created was the *quadcopter_takeoff_land* package. The purpose of this package is to take off, hover, and land the drone appropriately. The *quadcopter_mapping* package creates a 2D occupancy grid map of the 3D indoor environment using laser scans from the 2D LiDAR. The *quadcopter_teleop* package allows keyboard control of the quadcopter in the x-y coordinate plane to map the environment. Finally, the *quadcopter_navigation* package allows the quadcopter to navigate from the source point to the destination point using the constructed map and current laser scans. The following Table 4.1 shows the 5 ROS packages and a brief summary of their roles in the project.

Table 4.1: The 5 ROS Packages of quadcopter_sim.

ROS Package	Description
quadcopter_gazebo	Responsible for the indoor environment and quadcopter model instantiation.
quadcopter_takeoff_land	Responsible for safely taking off, hovering, and landing the quadcopter.
quadcopter_mapping	Responsible for producing a 2D occupancy grid map of the indoor environment using the quadcopter.
quadcopter_teleop	Responsible for controlling the quadcopter in the x-y coordinate plane.

quadcopter_navigation	Responsible for the autonomous navigation of the quadcopter from a source to a destination position.

4.4. Mapping Phase

The third step in the project was the mapping of the indoor environment. The instructions for mapping are summarized in Table 4.2 below.

Table 4.2: Summarized Instructions for Mapping Phase (Abdulhai, 2020).

Step	Command	Description
1	`roslaunch quadcopter_gazebo quadcopter.launch`	Launches the indoor environment and instantiates the drone.
2	`roslaunch quadcopter_mapping quadcopter_mapping.launch`	Launches the slam_gmapping node to collect LaserScans and publish a map.
3	`roslaunch quadcopter_takeoff_land quadcopter_takeoff_land.launch`	Launches the /quadcopter_takeoff_land node to take off the quadcopter.
4	`roslaunch quadcopter_teleop quadcopter_teleop.launch`	Launches the /quadcopter_teleop node to control the quadcopter in the x-y coordinate plane.
5	`rosrun map_server mapsaver -f /home/<username>/catkin_ws/src/quadcopter_navigation/maps/new_map`	Once the mapping is complete, it saves the map to the desired directory.
6	`rostopic pub /quadcopter_land -r 5 std_msgs/Empty "{}"`	Lands the drone safely by publishing an Empty message at a rate of 5.

4.5. Navigation Phase

The fourth step in the project was to allow autonomous navigation of the drone in the indoor environment. The instructions for navigation are summarized in Table 4.3 below.

Table 4.3: Summarized Instructions for Navigation Phase (Abdulhai, 2020).

Step	Command	Description
1	`roslaunch quadcopter_gazebo quadcopter.launch`	Launches the indoor environment and instantiates the drone.
2	`roslaunch quadcopter_navigation quadcopter_move_base.launch`	Launches the move_base node to perform a 2D navigation goal and autonomously navigate the quadcopter to the goal position.
3	`roslaunch quadcopter_takeoff_land quadcopter_takeoff_land.launch`	Launches the /quadcopter_takeoff_land node to takeoff the quadcopter.
4	`rostopic pub /quadcopter_land -r 5 std_msgs/Empty "{}"`	Lands the drone safely by publishing an Empty message at a rate of 5.

4.6. Video Streaming

The fifth step in the project was to transmit live video footage to law enforcement to enable them to take appropriate action to save human lives. While the navigation is running, the live streaming process should be executed. In order to create the server for the stream, the Gazebo environment must at least be running in one Terminal Tab. In another terminal tab, the following line was executed to start the server:

```
rosrun web_video_server web_video_server
```

The following site was visited on the browser to view the server: http://0.0.0.0:8080. On the page, the rostopic for the front camera image can be seen. The first link from the left was clicked to view the video stream of the drone from the front camera. If a navigation goal is specified, the quadcopter will navigate towards the destination while live streaming video footage to the HTTP server.

5. Results

This section presents the results obtained using the developed autonomous indoor drone application for two indoor environments: Prototype 1 and Prototype 2. Prototype 1 is a simple indoor environment consisting of two connected school hallways. Prototype 1 was used to test the key software modules of mapping and navigation of the autonomous drone without focusing on the complexities of the indoor environment. Once the mapping and navigation modules worked well for the simple indoor environment, Prototype 2 was designed. Prototype 2 is a complex and realistic middle school environment consisting of the school main office, media center, auditorium, cafeteria, courtyard, 4 hallways, and 12 classrooms. The mapping and navigation modules were further optimized and tested on the complex indoor environment and additionally, the video streaming module was added. The results are presented below for both prototypes. The results for each prototype are divided into three parts: Creation of the Indoor Environment, Mapping the Environment, and Navigation of the environment. Finally, there are separate results for the live streaming of the scene from the front camera of the quadcopter for the complex environment.

5.1. Prototype 1: Simple Indoor Environment

5.1.1. Environment Creation Phase

One of the first steps of this project involved using the Gazebo Simulator to construct a simple indoor environment with walls as obstacles. Several of these wall objects were used and placed in proper orientations to produce the sample indoor environment consisting of two school hallways shown in Figure 5.1.

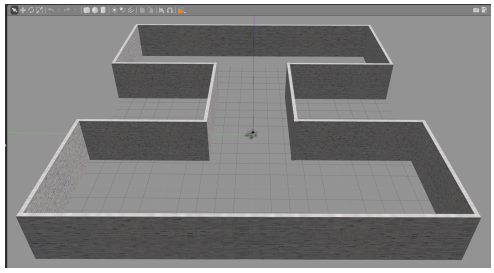

Figure 5.1: The Simple Gazebo Indoor Environment

5.1.2. Mapping Phase

After building the simple indoor environment, a quadrotor with a Hokuyo 2D LiDAR was instantiated at the center of the environment. The quadrotor is shown in Figure 5.2 below. Next, the quadrotor was commanded to take off and hover at a specified height using the created takeoff_land node. The teleoperated node for controlling the drone around the indoor environment in the x and y plane using keyboard keys was initiated. Next, the gmapping node for 2D mapping of the indoor environment was initiated and the drone began collecting the LaserScan data. The node's input is the laser scan data from the Hokuyo 2D LiDAR sensor and the output is a 2D probabilistic occupancy grid map of the environment. The occupancy grid map continuously updates as the teleoperated drone explores the environment. A snapshot of the mapping phase and an intermediate image of the resulting probabilistic map is shown in Figure 5.3 below. The white space on the map means free space, the black area means obstacles, and the grey area means the area is unexplored. The final completed map is shown in Figure 5.4. It took about 45 minutes to map the prototype 1 environment. The complete process of mapping was recorded and can be found here.

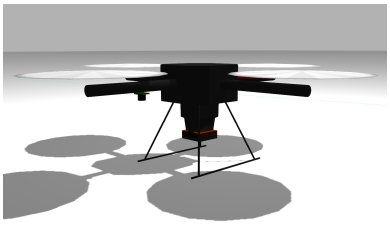

Figure 5.2: Instantiated quadrotor with a Hokuyo 2D LiDAR.

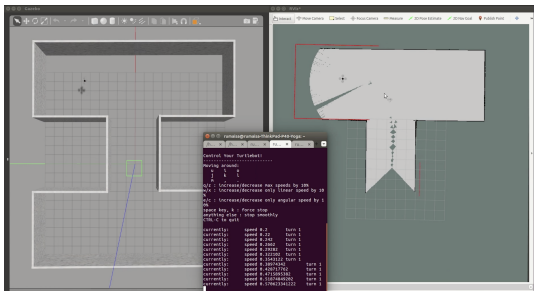

Figure 5.3: Snapshot of the Mapping Phase of the Environment.

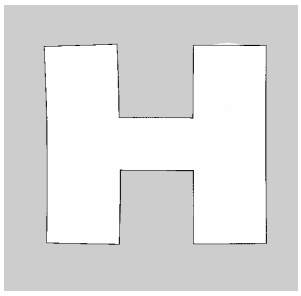

Figure 5.4: The Completed Probabilistic Map.

5.1.3. Navigation Phase

Once a successful map of the simple indoor environment was obtained, a drone was instantiated at the center of the indoor environment and hovered to a specified height using the takeoff_node. The Rviz visualization tool was then launched for commanding the drone to a certain destination point using the 2D navigation goal. The move_base and amcl nodes for path planning and localization were launched. The 2D navigation goal was specified, and the drone was able to successfully and autonomously navigate from the source point to the specified destination point using global path planning Dijkstra's algorithm and the local path planning DWA Local Planner avoiding known obstacles in the path. The time for the drone to complete each 2D navigation goal was dependent on the distance between the source and destination points and the specified speed of the drone. The time it took for the drone was not recorded as the working of the drone application was a priority rather than efficiency during testing of prototype 1. A snapshot of the navigation process in the environment is shown in Figure 5.5 below. The entire navigation process was recorded and can be found here.

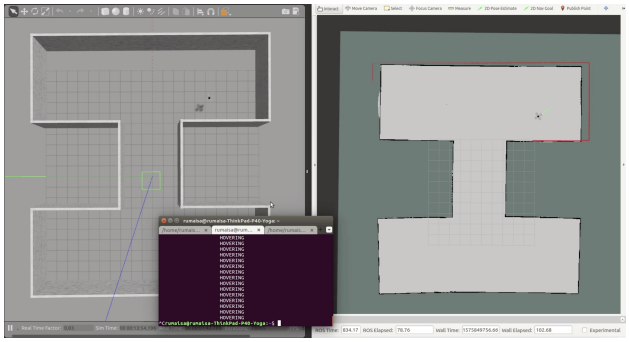

Figure 5.5: Snapshot of the Navigation Process using Dijkstra's Algorithm and DWA.

5.2. Prototype 2: Complex Indoor Environment

5.2.1. Environment Creation Phase

Once the mapping and navigation modules were tested successfully on the simple indoor environment, prototype 2 representing a complex school environment was designed that measured about 70m x 70m x 2.8m. Figures 6.6 and 6.7 show the top-down view and bird's view of the complex school environment respectively consisting of the school main office, media center, auditorium, cafeteria, courtyard, 4 hallways, and 12 classrooms.

The complex school environment represents a typical middle school in the US and it is based on Oak Middle School in Shrewsbury, MA that I attended as a middle schooler. The design of the complex environment took about 3 hours over a span of two weeks. The environment is sufficiently complex to test the autonomous drone application as it contains several hallways, classrooms and large spaces such as cafeteria, auditorium, courtyard, etc and shooting could occur in any of these spaces. The drone stationed in a central place then has to fly to any spot of emergency in the school. The designed school environment represents just one floor of the building, however.

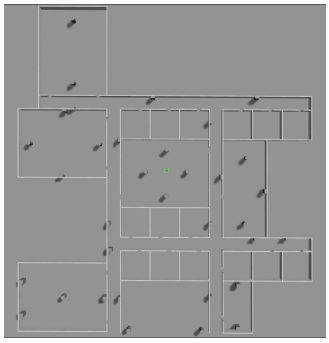

Figure 5.6: Top-Down View of the School.

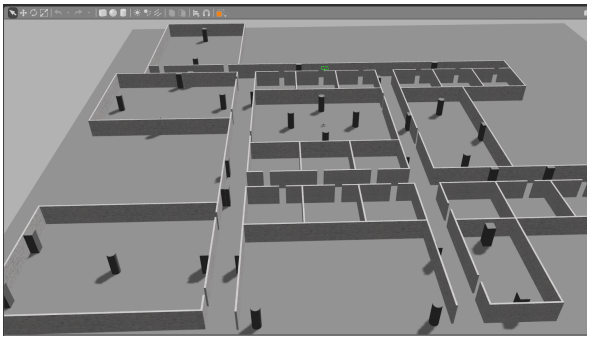

Figure 5.7: Birds-Eye View of the School.

5.2.2. Mapping Phase

After building the complex indoor school environment, a quadrotor with a Hokuyo 2D LiDAR shown in Figure 5.2 was instantiated at the center of the courtyard of the school. Next, the quadrotor was commanded to take off and hover at a specified height using the created takeoff_land node. The teleoperated node for controlling the drone around the indoor environment in the x and y plane using keyboard keys was initiated. Next, the gmapping node for 2D mapping of the indoor environment was initiated and the drone began collecting the LaserScan data. The node's input is the laser scan data from the Hokuyo 2D LiDAR sensor, and the output is a 2D probabilistic occupancy grid map of the environment. The occupancy grid map continuously updates as the drone explores the environment (teleoperated). In the beginning, there were no poles or pillars in the courtyard and it was one large empty space. Due to this absence of poles or pillars, the drone was unable to get an accurate map as it references its position based on obstacles. The mapping process had to be aborted and restarted after poles were inserted in the courtyard, auditorium, and cafeteria as shown in Figure 5.7. A snapshot of the mapping phase and an intermediate image of the resulting probabilistic map is shown in Figure 5.8 below. The white space on the map means free space,

the black area means obstacles, and the grey area means the area is unexplored. The final completed map is shown in Figure 5.9 below.

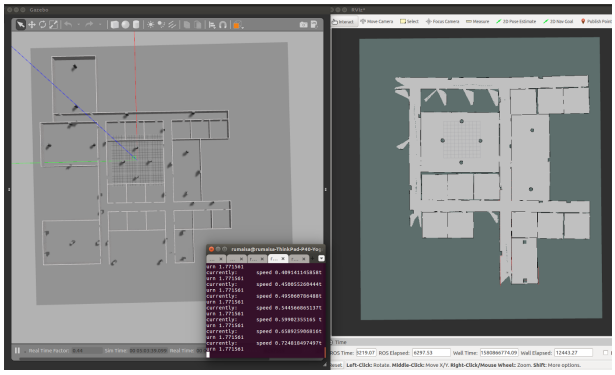

Figure 5.8: Mapping of the school environment.

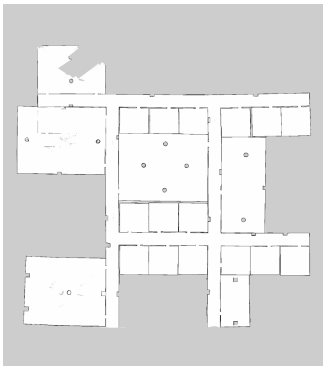

Figure 5.9: The Completed Probabilistic Map.

5.2.3. Navigation Phase

Once a successful map of the complex indoor environment was obtained, the quadcopter was instantiated again at the center of the courtyard and hovered to a specified height using the takeoff_node. The Rviz visualization tool was then launched for commanding the drone to a certain destination point using the 2D navigation goal. The move_base and amcl nodes for path planning and localization were launched. The 2D navigation goal was specified, and the quadcopter was able to successfully and autonomously navigate from the source point to the specified destination point. The time for the drone to complete each 2D navigation goal was dependent on the distance between the source and destination points and the speed of the drone. The source point, destination point, distance traveled, elapsed time, and average speed for five different destination goals at various points in the school was recorded and shown in Table 5.1. A snapshot of the navigation process in the environment is shown in Figure 6.10 below. The entire navigation process was recorded and can be found here.

Table 5.1: The Navigation results for various destination goals.

Goal	Initial Position	Final Position (m)	Distance Traveled (m)	Elapsed Time (s)	Average Speed (m/s)
Classroom 1	[0.0,0.0]	[-15.13,-5.82]	40.81	294	0.139
Classroom 2	[0.0,0.0]	[-14.98,8.58]	35.7	234	0.153
Classroom 3	[0.0,0.0]	[14,4,-32,71]	43.37	171	0.254
Library	[0.0,0.0]	[-6.63,-15.0]	25.77	165	0.156
Main Office	[0.0,0.0]	[-35.58, -14,44]	54.62	476	0.115

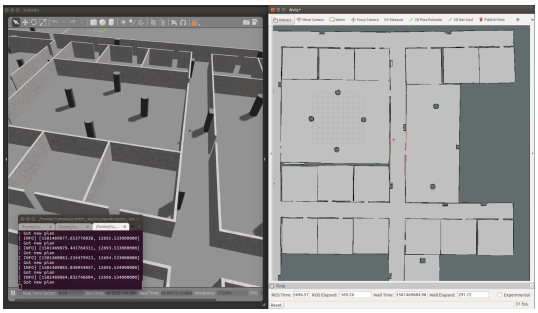

Figure 5.10: Snapshot of the Navigation Process in the school environment.

5.3. Live Video Streaming

This section entails the results achieved for the live video streaming aspect of the drone application. The purpose of the live streaming is to inform law enforcement of the potential dangers before they arrive at the scene of the mass shooting. While the navigation process was running and the quadcopter navigated from its starting position to the goal position, a web server was instantiated and listened to the camera topic of the quadcopter. This raw live video footage was captured and transmitted to the web server. The live streaming demo can be viewed here. Figure 5.11 below shows the quadcopter navigating to a classroom while livestreaming video footage from its front camera to a web server.

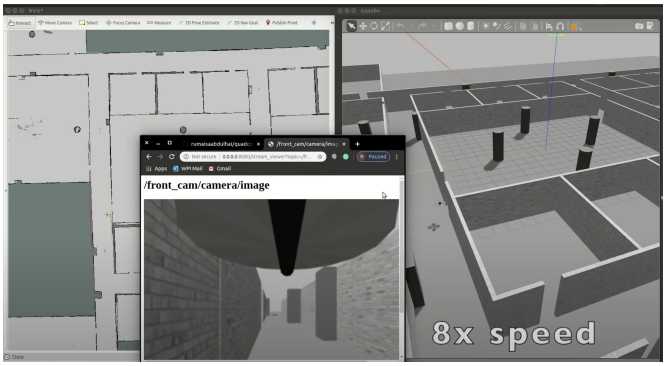

Figure 5.11 Snapshot of the navigation with live streaming to a web server.

6. Discussion & Conclusion

This section discusses how well the application results obtained on this project meet success criteria and their overall significance in assisting law enforcement during mass school shootings. In addition, it outlines the steps required to translate this project into a real-world application. This project set out to investigate how an autonomous indoor drone application could be developed to solve an urgent social need: assist law enforcement to fight mass shootings and save precious human lives. The specific goals of this project were to simulate mapping, path planning, and autonomous navigation of an indoor drone in a school environment to a point of emergency, collect real-time information about scene and the perpetrators involved, and relay it to a trusted group of individuals such as law enforcement who then can take appropriate action to significantly reduce loss of life.

The key goals laid out above on this project were modestly achieved. During the course of 7 months, an autonomous indoor drone application was developed using ROS, the Gazebo Simulator, and Python that can assist law enforcement. First, the Gazebo Simulator was used to build an indoor environment of a US middle school consisting of the school main office, media center, auditorium, cafeteria, courtyard, 4 hallways, and 12 classrooms. The building process took approximately 4 hours for a moderately sized middle school. Second, a teleoperated quadcopter with a Hokuyo 2D LiDAR and the gmapping algorithm were used to successfully build a 2D occupancy grid-map of the designed school environment. The resulting map captured the free and obstacle spaces of the school's indoor environment. The mapping process took approximately 7 hours to build the 2D occupancy grid map. Third, Adaptive Monte Carlo Localization (AMCL) was used with Dijkstra's global path planning and the

Dynamic-Window Approach (DWA) local planning algorithms to enable the drone to successfully navigate from a starting position to any chosen destination position while avoiding obstacles. The time to navigate from the center of the courtyard to various destinations inside the school spanned from 2.5 to 8 minutes. Finally, the onboard camera on the quadcopter successfully captured and transmitted video from the scene in real-time to a web server. This real-time video could be used by law enforcement to acquire critical information from the shooting in order to take immediate steps to reduce the loss of precious lives. The developed autonomous drone application and the results not only serves as a proof of concept but also could be replicated for any elementary, middle, or high school in the United States to assist law enforcement during the time of a mass shooting.

6.1. Success Criteria for Navigation

The first and most important criterion of success was that the drone must be able to navigate to its destination in a time-critical manner. Typically, the maximum loss of life is inflicted in the first 5 to 30 minutes of a mass school shooting. The drone should require a maximum of 2 minutes to arrive at the scene of the shooting so that it is able to relay video information to law enforcement to allow them to take appropriate action to prevent further harm. The school designed in this application was 70 x 70 m. For the quadcopter to be able to traverse 140 m from one end of the school to the other in 2 minutes, it must move with an average speed of 1.17 m/s. The maximum average speed achieved for the quadcopter, according to Table 6.1, is 0.254 m/s to navigate from the center of the courtyard to the top right classroom in 171 seconds, which is about 2 minutes and 51 seconds. Although the time taken for the quadcopter to navigate to the classroom is close to the 2 minutes proposed earlier, the quadcopter has shown to be slower at other times, indicating that more work must be done to produce a consistent time under 2 minutes.

Reasons that may have contributed to the navigation results include the computational load of the navigation algorithms on the Linux machine, accuracy of the occupancy grid mapping, and inadequate localization of the quadcopter. First, the quadcopter must process laser scans and run multiple computationally intensive algorithms in simultaneous threads, including the AMCL and the DWA algorithms. This computational load may result in delayed navigation and in real life, it can drain the battery of the quadcopter as well. Second, the quadcopter may not have produced a completely accurate map of the school due to there not being enough features for the

quadcopter to track. This mapping inaccuracy resulted in certain rooms being shorter in the map than their true length in the indoor environment. An example of where this inaccuracy may have affected the navigation of the quadcopter is when the quadcopter may think it is farther from a wall when it is actually closer, resulting in the quadcopter bumping into walls, delaying the time to reach the destination.

6.2. Success Criteria for Mapping

The second criterion of success was that the drone must efficiently generate a quality occupancy grid map of the indoor environment of the school. The mapping of the middle school for this project took approximately 7 hours with certain parts of the map being distorted. Ideally, the mapping process should take 3 to 4 hours without any distortions. The reason for the longer time to generate the occupancy grid maps was due to the manual navigation of the drone around the obstacles. This manual teleoperation of the drone was purposely slower as moving the drone faster would result in the quadcopter not being able to recognize the full dimensions of the features of the environment. It is expected that if this application were applied in real school environments, the occupancy grid map would not become distorted as schools have plenty of obstacles and the drone is able to move faster as well as localize itself accurately. In addition processing, large amounts of LiDAR scans are computationally intensive and this could also have contributed to the large time required for generating occupancy grid maps. An alternative to 3D occupancy maps would be 3D occupancy maps that will not only detect obstacles in a plane at a certain height but also above and below the hovering height of the drone. But 3D maps are even more computationally intensive and would require an even more powerful processor onboard the drone. In addition, computationally intensive algorithms would require a larger battery on the drone.

6.3. Success Criteria for Live Streaming

The third criterion of success is that the live-streamed videos from the drone must be clear and provide enough detail for the law enforcement to identify the location inside the school where the drone is navigating. The police must also be able to assess the condition of students and the location of the perpetrator from these videos. This was reasonably achieved as the drone captured physical scenes during its navigation and the physical features of the school environment could be identified. However, the objects appeared gray as the built environment was gray

in color. This uniformity of color would not happen in a real school as the objects inside a school have plenty of colors and the drone will transmit scene videos in their original form without any modification.

6.4. Success Criteria for Quadcopter

Other success criteria include the size of the drone and the power of the onboard computer. In order for the drone to navigate through the doors it should not be as big as the width of the door as then it would easily crash into the sides. Ideally, the drone width must be less than half the width of the doors. In this project, the drone was about 0.7 m in diameter and the door width was about 0.9 m. The quadcopter must have a powerful enough onboard computer to run the computationally intensive mapping and navigation algorithms without relying on external computing power in the cloud. The quadcopter in Gazebo met this criterion as it did have an onboard computer running a Linux machine. However, the Linux machine had to run the Gazebo platform itself which shall not be needed in the real drone. This would free up some computing power. More powerful and compact Nvidia GPUs are available on the market that could be used in real-life drones. The developed application met all other basic success criteria such as the drone hovered autonomously once taking off, moved towards the goal autonomously once the goal position was identified, the drone passed through classroom doors without crashing and made proper turns towards the goal.

The results achieved on this project are significant and have a high impact as they can save precious lives of students and teachers during mass school shootings. The police take an average of 15 to 20 minutes to arrive on the scene and are usually unaware of the precise location of the shooter. This application enhances the communication between the potential victims and the police, enabling police to take effective action against the perpetrator. Immense chaos and confusion occur during the first 20 minutes of the shooting, and it is during this time the autonomous drone application could prove enormously useful by tracking down the shooter, distracting the shooter, capturing video from the scene, and transmitting it to law enforcement who then can take the most appropriate actions to save precious lives. If one or more drones using the developed indoor drone application are allowed to autonomously navigate to the point of shooting within minutes, they could deter the shooter from inflicting more harm, and more importantly give the police critical information to act effectively. The autonomous drone application developed can be expanded with identified improvements, and eventually translate into a

real-world application that can save the lives of students and teachers during mass shootings. Although the autonomous indoor drone proof of concept application was specifically built for an indoor school environment, the software application could be generalized and adapted for a variety of indoor environments such as a church, synagogue, mosque, hotel, or convention center.

7. Improvements

Over the summer of 2020, improvements were made to the mapping aspect of my autonomous drone application which reduced the mapping time from 7 hours to 2.5 hours. Instead of teleoperation, a technique called frontier exploration was used alongside gmapping to achieve an autonomous mapping of the indoor school environment. The name of the package used is called rrt_exploration provided by the Open Source Robotics Foundation. The method works by identifying 4 boundary points and one initial point where the algorithm, based on the Rapidly Exploring Random Tree Star (RRT*) algorithm, looks for the nearest frontier points, or the location where the unknown territory meets the explored territory, to explore using automatically set 2D navigation goals. By the end of July, frontier exploration and gmapping worked for a simple environment. The mapping demo can be viewed here. The navigation with this map worked successfully. However, when tried with a complex environment, the quadcopter would get stuck at the edge of a classroom door. The quadcopter size had to be reduced. Using gmapping, frontier exploration, and the smaller quadcopter, most of the environment was able to be mapped. Figure 6.1 shows the occupancy grid map below. The quadcopter eventually got stuck in one corner and the mapping process had to be aborted. It took about 3 hours to achieve the map, which cuts down the regular mapping time by approximately 57%. This work was done with the same setup as before, with a laptop running Ubuntu 16.04 with ROS Kinetic.

In August, the gmapping SLAM algorithm was replaced with google cartographer based SLAM algorithm along with ROS Kinetic. It did not produce a usable map. The problem with this map is that there are no black lines, which are supposed to signify obstacles. When navigation was tried with this map, it did not work. In early November, Google cartographer was attempted again, this time with a laptop running Ubuntu 18.04 with ROS Melodic. A 2D occupancy map with google cartographer and teleop was produced as seen below in Figure 6.2. The

time to map the complex school environment was approximately 2.5 hours compared to 7 hours, a reduction of about 65% from gmapping. The mapping demo can be viewed here. The navigation with this map worked as well. The navigation demo can be viewed here. The various combinations that have been tried are summarized in Table 6.1 below. This year as part of my Senior Independent Project, I hope to experiment with Visual SLAM Algorithms using monocular and stereo cameras. I also hope to integrate Frontier Exploration and Google Cartographer, integrate the PX4 Flight Controller into the overall code base and test the application in real life.

Table 6.1: All the combinations for the various mapping methods.

Teleop or Frontier Exploration	**Gmapping or Cartographer**	**Simple or Complex Environment**	**Working Map? (Yes/No)**	**Working Navigation? (Yes/No)**
Teleop	Gmapping	Simple	Yes	Yes
Teleop	Gmapping	Complex	Yes	Yes
Frontier exploration	Gmapping	Simple	Yes	Yes
Frontier exploration	Gmapping	Complex	No (75% complete)	No (not tried, but expected to work)
Teleop	Cartographer	Simple	Yes	Yes
Teleop	Cartographer	Complex	Yes	Yes
Frontier exploration	Cartographer	Simple	No	No (unable to achieve a successful map)
Frontier exploration	Cartographer	Complex	No	No (unable to achieve a successful map)

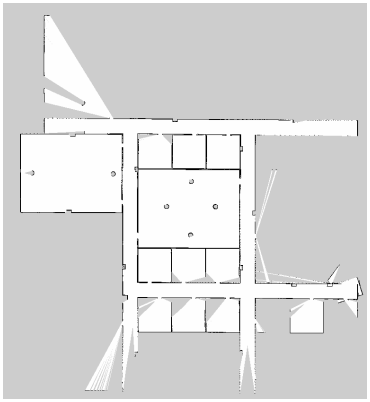

Figure 6.1: Map produced with gmapping and frontier exploration.

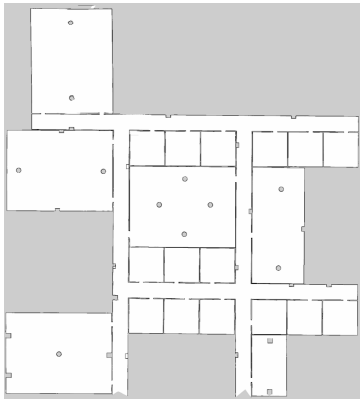

Figure 6.2: Map produced with google cartographer and teleoperation.

8. References

Abdulhai, R. (2019). rumaisaabdulhai/PathPlanning. Retrieved from https://github.com/rumaisaabdulhai/PathPlanning.

Abdulhai, R. (2020). rumaisaabdulhai/quadcopter_sim. Retrieved from https://github.com/rumaisaabdulhai/quad_sim.

Ackerman, E., & Strickland, E. (2018, Jan). Medical delivery drones take flight in east africa. *IEEE Spectrum, 55*, 34-35. doi:10. 1109/MSPEC. 2018. 8241731 Retrieved from https://ieeexplore.ieee.org/document/8241731

Chin, T. (2019, February 26). Robotic Path Planning: RRT and RRT*. Retrieved from https://medium. com/@theclassytim/robotic-path-planning-rrt-and-rrt-212319121378.

Correll, N. (2014, October 14). Introduction to Robotics #4: Path-Planning. Retrieved from http://correll. cs. colorado.edu/?p=965.

Guermonprez, P. (2017, November 13). Autonomous Drone Engineer - A1 - Intel Aero in 5mn. Retrieved from https://www. youtube. com/watch?v=7t7l885g8dI.

Khosiawan, Y. , & Nielsen, I. (2016). A system of UAV application in indoor environment. *Production & Manufacturing Research, 4*(1), 2-22. doi:10. 1080/21693277. 2016. 1195304

Kudan. (2016, May 23). An Introduction to Simultaneous Localisation and Mapping. Retrieved from https://www.kudan.io/post/an-introduction-to-simultaneous-localisation-and-mapping.

Matarić, Maja J. (2008). *The Robotics Primer*. Cambridge, Mass: The MIT Press.

Yan, M. (2014). Dijkstra's Algorithm [PDF file]. Retrieved from https://math. mit. edu/~rothvoss/18. 304. 3PM/Presentations/1-Melissa. Pdf

Li, J., Bi, Y. , Lan, M., Qin, H., Shan, M., Lin, F., & Chen, B. M. (2016). *Real-time Simultaneous Localization and Mapping for Uav: A Survey*.

OSRF. (n. d.). Beginner: Overview. Retrieved from http://gazebosim. org/tutorials?tut=guided_b1&cat=.

Puri, V., Nayyar, A., & Raja, L. (2017). Agriculture drones: A modern breakthrough in precision agriculture. *Journal of Statistics and Management Systems, 20*(4), 507-518. doi:10. 1080/09720510. 2017. 1395171

ROS Wiki. (n. d.). Retrieved from http://wiki.ros.org/ROS/Introduction.

Skydio Inc. (2019). How does Skydio 2 work? (n. d.). Retrieved from https://support.skydio.com/hc/en-us/articles/360036116834-How-does-Skydio-2-work-.

Swift, N. (2017, March 1). Easy A* (star) Pathfinding. Retrieved from https://medium.com/@nicholas.w.swift/easy-a-star-pathfinding-7e6689c7f7b2.

Thrun, S., Burgard, W., & Fox, D. (2010). *Probabilistic Robotics*. Retrieved from https://docs.ufpr.br/~danielsantos/ProbabilisticRobotics. pdf

Wenderlich, R. (2011, September 29). Introduction to A* Pathfinding. Retrieved from https://www.raywenderlich.com/3016-introduction-to-a-pathfinding.

Williams, B., & Frazzoli, E. (2010). *16. 410 Principles of Autonomy and Decision Making.* Massachusetts Institute of Technology: MIT OpenCourseWare. Retrieved from https://ocw.mit.edu.

9. Acknowledgments

I would like to thank my Beaver Works Summer Institute (BWSI) Instructors Dr. Mark Mazumder and Dr. Ross Allen for motivating me to embark on this project. I would also like to thank my STEM advisors Dr. Kevin Crowthers and Dr. Raghavendra Cowlagi for their persistent guidance and holding me accountable to the phases and deadlines throughout my project. I would also like to thank WPI graduate student Hanqing Zhang for introducing me to Gazebo and answering difficult technical questions that arose from time to time. Finally, I would like to thank my parents for mental support and staying up late alongside me as I worked long hours to develop this autonomous drone application.

10. Appendices

This section contains additional information on the project, including the limitations and assumptions, challenges, extensions, installation instructions, source code, and project notes file.

10.1. Appendix A: Limitations

At the beginning of this project, I wanted to work with a real research drone such as the Intel Ready to Fly (RTF) quadcopter and develop the autonomous drone application to fight mass shootings. I had previously built a quadcopter two years ago, but I could not use that as the quadcopter did not have an onboard computer such as an Nvidia GPU and a development platform. I was not looking for a commercial quadcopter as I needed the freedom to develop my application and add sensors to the drone as needed. I needed an onboard computer on the drone so that the quadcopter could perform mapping and path planning and make decisions without connecting to an external source such as WiFi or Bluetooth.

I tried to acquire an Intel RTF drone and contacted the official sellers, however, I was unable to acquire one as they were no longer selling the drones. I also tried to reach out to my professors from Lincoln Laboratories who taught the Beaver Works UAV racing program, a summer program I had attended last year in 2019, to see if I could use one of their Intel research drones. However, I was unable to use the drones as they were being used for other classes. I searched for research drones online and came across a company called Uvify that makes quadcopters with

onboard computers and stereo cameras. I reached out to them and had asked my mentor Professor Cowlagi to see if WPI could supply them, but I needed a sufficient reason to order them. WPI does not have research drones available for use. They have small drones with a Raspberry Pi compute board, but that will not be enough to run the mapping and navigation phases for my project. The drones also did not have the necessary sensors such as the LiDAR or depth camera I was looking for. I even tried ordering a commercial AR Parrot drone as I had heard good things about it from researchers online, but the drone's battery was faulty and I could not use the quadcopter. Even if I acquired a drone, I would not have been able to use it. WPI does not have facilities for testing a drone in the hallways of its indoor buildings. There are also very limited areas outdoors to test my code. I had to find another option instead of using a real-life drone.

My mentors Dr. Mark Mazumder and Ross Allen advised me to start with a development and simulation platform such as Gazebo as it was a huge undertaking to develop an autonomous drone application using a real research drone right in the first step. Several modules for mapping and navigation have to be developed and tested and lots of work could be done initially without the actual drone on hand. In addition during the initial testing the drone would crash often and it would not be advisable to damage expensive drone equipment. With all these considerations in mind. I decided to use the Gazebo platform for my project. The great aspect of using Gazebo is that I am able to test all the software modules that will exactly be used on a real research quadcopter. I will still be able to use the packages and code that I developed with ROS on the Linux Machine as the research drone I may be using in the future will have an onboard Linux Machine that will run ROS as well. The only component that will be in simulation is the drone hardware itself. It is also beneficial to use a simulator as I am able to make sure everything is safe and working before I deploy it on a real drone. Furthermore, it is an industry-standard to use a development and simulation platform such as Gazebo as the application can easily be modified to suit other applications as well. Using a simulator also gives the added benefit of working from home. I do not have to be constrained to the working hours of a research lab and can work on developing the code for the application in the meantime. In the future, I plan on testing my code on a real quadcopter. I have reached out to a professor from Lincoln Labs and I may be able to work in his lab. However, I want to test as much as I can on the Gazebo Simulator so that I do not have to debug many code problems during the lab and real-world testing.

10.2. Appendix B: Assumptions

The following are the assumptions made during the project:

1. The exact location of the shooter in the indoor environment is known using a sensor.
2. There is a height above the ground where there is no disturbance or moving obstacles so the drone is able to navigate using solely a static map.
3. A drone with a 2D LiDAR sensor is available and can be tested with the code developed during this project.

10.3. Appendix C: Challenges

This section will detail the struggles encountered during the project and the lessons learned through each one.

10.3.1. ROS Melodic vs. Kinetic

At the beginning of this project, I had to configure a laptop for running the simulator. I had previously installed Ubuntu 18.04 with the latest ROS Melodic and Gazebo 9 on my Lenovo Laptop. However, many packages and resources are available for ROS Kinetic but not ROS Melodic as it was just released in 2019, so I decided to get Ubuntu 16.04. In addition, more tutorials and guidance were more available for ROS Kinetic, which is currently the most common ROS distro used by researchers. I installed Ubuntu 16.04 on my laptop using a USB stick. I changed the priority of booting the laptop to the USB stick and then when I restarted the laptop again I was able to install Ubuntu 16.04 by erasing Ubuntu 18.04.

10.3.2. Incorrect Nvidia Drivers

When I was trying to make the laser sensor detect objects in Gazebo, it was not working initially. I placed an object in front of the simple chassis robot I was testing with and the laser signals did not seem to be blocked at all as they should have been. I searched for this error online and someone suggested installing the latest Nvidia drivers as it worked for their laptop. I installed the Nvidia driver 430 on my Lenovo and rebooted my laptop. I got an error saying that “the system is running in low graphics mode” and that I had to configure the graphics settings myself. Whatever option I selected did not help and led to the laptop restarting to the same error. I realized I had installed the wrong driver as it was known to cause problems. So, I attempted to install Nvidia driver 384 by logging onto the

laptop and purging all the drivers. After the correct driver was installed, I was able to view the laser output in the Gazebo Simulator.

10.3.3. Hovering the Quadcopter

Another issue I encountered was getting the drone to hover at a fixed height. In the beginning, I manually took off the drone by publishing a positive linear z command velocity of 0.3 m/s until it was at a reasonable hover height by glance (below the wall height of 2.8 meters). I would then stop publishing the z velocity thinking that the drone would remain at that height. However, the drone was plunging back to the ground. I was not aware that in order for the drone to hover at a fixed height, a linear z command velocity of 0 m/s had to be published constantly. Once I learned this the hard way, I took off the drone publishing the z velocity for 0.3 m/s, just like before and once the drone reached the acceptable hover height, I would quickly execute the teleoperation node from another terminal tab which published a linear z command velocity of 0 m/s. The drone stayed in position and did not fall to the ground. I had solved the original problem even though there was a manual step. Later on, I programmed the drone to take off and hover in Python so that the process was more automated. In code, I constantly checked for the height of the drone above the ground, and once the drone had reached its hover height which I specified to be about 1.3 meters above the ground, the drone would hover in position.

10.3.4. Ghost Wall

Another issue I encountered was getting an accurate occupancy grid map of the simple indoor environment. After constructing the simple indoor environment, I was ready to perform mapping using the gmapping slam algorithm and the laser scans from the quadcopter. I took off and hovered the drone using the takeoff code and navigated the drone using the turtlebot teleoperation code. I was able to control the drone using keys on my keyboard. While I was mapping the indoor environment, I realized the quadcopter was unable to see the full size of the back walls (larger than 7.5 m) as the laser scans were not registered in that area. At first, it did not make sense to me as the walls appeared visually long enough and it was strange the drone was unable to detect them. In fact, when I tried navigating the drone close to the back wall which was larger than 7.5 m to see if it would detect it, the drone simply passed through the wall. Even though for me visually there was a wall, for the drone the wall did not exist. This was the problem. I realized that it had something to do with extending the walls to more than 7.5 m when I was

first creating the indoor environment. I remember changing the size of the back walls to 20 m but it seems I did not change the collision size which remained 7.5 m. I searched this up and came to the conclusion that you can change the visual appearance of an object and make it look bigger, but if you do not change the collision size, some parts of the object will not be detected because it is really not existent for the drone. This explains why my drone was able to fly through the wall. After changing the collision aspect of the wall 20 m in line with its visual size, I was able to get the drone to recognize the entire back walls and achieve my first acceptable map of an indoor environment. Figure 9.1 shows a comparison of the mapping done prior to changing the collision size versus after.

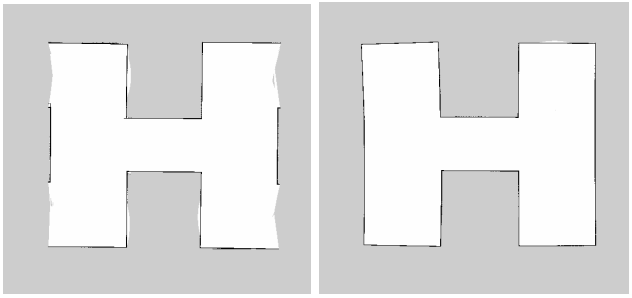

Figure 9.1: The occupancy grid map prior to changing the collision size vs. after.

10.3.5. Conflicting Command Velocity Messages

When I was trying to specify a 2D navigation goal during the navigation process, I realized that my drone was moving very slowly and often tried to reset itself by going into recovery mode. I spent many nights and meetings with my mentor trying to figure out what the issue was. I finally understood why the drone was moving unusually slowly: the drone was receiving cmd vel messages from two separate places; the quad_takeoff_land node and the move_base node and therefore the drone was conflicted in its movements. When the drone took off using the quad_takeoff_land node, the code published a positive z velocity until the drone reached its hover height. When the drone was hovering, the quad_takeoff_land node published a zero z velocity and everything else with zero velocities. When I then ran the move_base node to specify a 2D navigation goal, the move_base node would publish some non-zero linear and rotational velocities. While the move_base node was publishing non-zero velocities in some directions, the takeoff_land node was still publishing zero velocities in all directions for hovering purposes. These conflicting command velocities caused the drone to move extremely slowly.

In order to fix this issue, I had to stop running the hover code within the quad_takeoff_land node temporarily while the drone was navigating using the move_base node. Luckily, stopping the hover code temporarily would not make the drone fall to the ground as the move_base node specified a linear z velocity in its command velocity messages. I made the quad_takeoff_land node a subscriber of the /move_base/feedback topic from the 2D navigation goal to see if a goal was currently set for the drone. If that was true, a boolean variable turned true and that in turn stopped the hover code. After the goal was finished, I needed the boolean variable to revert to false in order for the hovering from the quad_takeoff_land node to resume. If the command velocities were all set to zero, which meant the drone was temporarily not moving, I switched to the hover code again. After adding this functionality, my drone would not crash to the ground after reaching a specified goal position. I also added the landing functionality to the takeoff_land node so the drone could land gracefully when it was done reaching its final goal. Now that I have perfected the taking off, hovering, and landing safely, everything is more automated, which will be better for my project moving forward.

10.3.6. Increasing Obstacles for Mapping Accuracy

Another issue I encountered was achieving a proper occupancy grid map of the complex indoor environment. As mentioned before, I made sure all the collision sizes matched with the visual sizes of the walls. However, the drone was still unable to create an accurate occupancy grid map. The problem was due to the fact that the rooms were too large and therefore the quadcopter could not properly localize itself within the indoor environment, resulting in the distortion of the grid map. I had created large empty halls for the cafeteria, auditorium and the courtyard, with not enough features or obstacles in these empty spaces for the drone to localize itself. After adding additional pillars in the cafeteria, auditorium, and courtyard, the localization improved, and I was able to generate a more accurate occupancy grid map of the school environment.

10.4. Appendix D: Extensions and Improvements

So far in this project, I have developed an autonomous drone application that clearly demonstrates how a drone could be used to assist law enforcement during mass shootings. Except for the drone hardware which is simulated, all the required modules such as the mapping, navigation, and live stream recording of the scene can be

used on a real-world drone application. However, there are still many possible extensions and improvements that can be made to this application. There were many features that were initially planned to be incorporated, but due to time constraints and the need to limit the scope of this project, they were not able to be incorporated into this paper. I have listed these improvements and extensions below.

10.4.1. Faster Drone Navigation

The first extension that can be made is to have the drone navigate the school environment faster using the occupancy grid map that has already been built. The average speed that has been achieved so far is 0.2 m/s. I have tried tuning the parameters of the drone's speed in the node but have not seen a noticeable improvement in the average speed of the drone. The average school shooting lasts for about 5-10 minutes. If the drone from its initial station to the point of emergency in 1-2 minutes many lives can be saved. So it is critical for the drone to not only be able to avoid obstacles but navigate with a faster average speed of 1 m/s.

10.4.2. Depth Camera Mapping & Navigation

Another extension to this project would be the use of depth cameras instead of laser sensors to build an occupancy grid map and navigate the environment. Depth cameras are more desirable as they are more available on the market and are less expensive. They could be used to construct an occupancy grid map of the environment and localize the drone instead of using 2D LiDAR laser scans.

There are three possible ways to perform mapping using the depth camera. The first method involves using a ROS package called depthimage_to_laserscan. This package subscribes to the raw depth image from a depth camera and publishes a 2D laser scan to the /scan topic. In essence, this method just converts the depth image information into the format of the laser scans which can then be used with the same gmapping SLAM algorithm. I have made some progress in this direction. Using a different hector quadrotor drone model with a Kinect depth camera, I have successfully converted the depth camera information into laser scans, but I still have to make the mapping part work. The second method uses a similar ROS package called pointcloud_to_laserscan. This package converts the point cloud data from a depth camera such as a Kinect sensor or an Intel real sense camera into 2D laser scans that can be used in the gmapping algorithm. Some progress has been achieved in this direction, with the point cloud data converting into 2D laser scans on a PX4 quadcopter. However, the localization of the drone is poor and

the drone is unable to generate accurate laser scans. The third method involves using the rtabmap_ros package which is a Real-Time Appearance-Based Mapping RGBD SLAM implementation. It subscribes to the depth image and publishes a 2D occupancy grid map or a 3D point cloud. If I work on this extension, I would have to try all three approaches and determine which approach is more optimal for depth camera mapping. Navigation with depth cameras is expected to work the same way as 2D LiDAR navigation as long as a 2D occupancy grid map is used and laser scans can be used to localize the quadcopter.

10.4.3. PX4 Flight Controller Integration

In order for the drone to be tested in a real-life environment, the drone needs a flight controller. A common open-source flight controller software PX4 could be integrated with the Gazebo ROS Simulation so that when the time comes the code can be tested on a research drone such as the Intel RTF drone. Some progress has been made with a simulated PX4 Iris quadcopter with an Intel Realsense r200 depth camera drone. The drone is able to take off, hover, and navigate using teleoperation. The Iris quadcopter with the PX4 firmware does not have a 2D LiDAR so the drone cannot map the environment using Laser Scans and must use a depth camera to construct a 2D occupancy grid map. However, it is possible to add a 2D LiDAR using Gazebo Plugins so that the code developed in this project can be used on the PX4 quadcopter.

10.4.4. Switching the Global Path Planning Algorithm

In this project, Dijkstra's algorithm was used for global path planning. Another addition to this project would be considering other path planning algorithms to switch out with the Dijkstra's algorithm. A path planning algorithm such as A Star or RRT Star could be integrated into the code to test and evaluate the best path planning algorithm for the drone that would minimize the time and the path is taken. It is possible that A Star and RRT Star may prove to be more efficient than Dijkstra's Algorithm.

10.4.5. AR Marker Localization

Other additions for the project would be entirely new localization, mapping, and navigation methods. For example, AR Marker Localization could be used along with computer vision and obstacle avoidance to navigate the

drone. AR Markers could be placed near entrances of rooms to indicate to the drone the location of important places. The drone would use cameras to avoid obstacles and identify the AR Markers.

10.4.6. 3D Mapping and Path Planning

Another method would be 3D mapping. A package for 3D mapping that would be considered is Octomap. This map would be more beneficial as it allows the drone to move up and down in case it is unable to move forward at a constant height. This would be more realistic because, in reality, the height of the classrooms and hallways are not the same, and the height of the door must be considered and compared to the height of the school hallway.

10.4.7. Computer Vision-Based Obstacle Avoidance

Depth cameras would also be desirable as they could be used for dynamic obstacle avoidance. Using depth cameras, the quadcopter can detect how close it is to other obstacles and identify what type of obstacles are in its field of view.

10.4.8. More Realistic Environments

The indoor environment could be modified to be more realistic so that the drone can keep track of more features in the environment. Models of real objects including desks, chairs, lockers, and closets could be placed in the indoor environment.

10.4.9. Sensor-Based Initiation

One important addition to this project that would complete the application would be sensor-based initiation of the drone. First, an optimal sensor must be found. Next, the quadcopter must know the location of the sensors that will be located at multiple places throughout the building. As soon as a sensor activates, the drone must know which sensor activated and navigate to the approximate location of the sensor. A more obvious sensor would be a gunshot detection device that would know when and where a gunshot has been fired and the drone could navigate to that location. This would require a lot of investigation and design.

10.5. Appendix E: ROS Architecture

The Robot Operating System (ROS) is a powerful tool for autonomous robotics programming as it is open source and is available in many programming languages such as Python, Java, or C++. It provides various libraries for building and writing code and runs of Unix-based platforms including Ubuntu. A ROS package is an organized collection of nodes, libraries, or configuration files. In this project, 5 ROS Packages are created and used throughout the mapping and navigation processes. ROS must be actively running in Terminal in order for it to be used. The ROS master allows nodes to find each other and exchange messages. A ROS node is a program that runs inside a user's computer under the ROS master. Each node has a single purpose and is executable by the user. Nodes in this project would be considered Python or C++ files that are executed by launch files. A ROS topic is a stream of messages through which nodes can communicate over. Programs use this by having a node publish its information to a topic, where subscribing nodes can read the information on the topic. A ROS message defines what happens in a topic. A node sends a message to a topic. Other notes may listen to this topic to receive the message. The action of sending messages is called publishing and receiving messages is called subscribing. A node that publishes to a topic is known as a publisher of that topic, and a node that subscribes to a topic is known as a subscriber of that topic.

These concepts are illustrated through an example of the quadcopter lifting off the ground. The node responsible for taking off the quadcopter is called /quad_takeoff_land. When the launch file is run, the node is initialized and the node publishes command velocities in the form of a Twist message to the /cmd_vel topic. The /gazebo node, which represents the drone, subscribes to the /cmd_vel topic to get the Twist messages. Through these publishing and subscribing actions, the drone starts to take off.

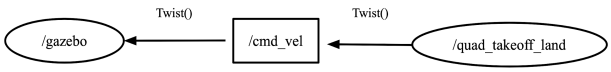

Figure 9.2: ROS Publisher/Subscriber Architecture Example in quad_takeoff_land.

Figure 9.2 above demonstrates one action of the /quad_takeoff_land node. The oval shapes represent nodes, the rectangles represent topics, and the arrows carry messages between the nodes. The arrow coming out of /quad_takeoff_land signifies that the node is publishing to the /cmd_vel topic, and the arrow going into the /gazebo node signifies that the drone is subscribing to the /cmd_vel topic which contains the Twist message. It can also be

said that /quad_takeoff_land is a publisher of the /cmd_vel topic and /gazebo is a subscriber of the /gazebo topic. The Twist message has two parts: linear and angular, and within each part, there is an x, y, and z velocity. Therefore, there are 6 different command velocities that can be specified in the Twist message. For taking off the drone, the only velocity that is specified is the linear z velocity, and all other velocities are published as zero. Note that the arrows are not touching the /cmd_vel topic because topics do not perform actions. They are just a place for the messages to be sent and received from. In this way, nodes are not aware that they are receiving information from other nodes. Nodes are completely independent of each other. This characteristic of ROS allows nodes to collect and publish information to multiple nodes. All the nodes, topics, and messages for the mapping and navigation phases will be described in detail in the next two sections.

10.5.1. Mapping Architecture

The first step in the mapping phase was to launch the gazebo environment and instantiate the drone at the center of the indoor environment. 6 Terminal Tabs were created for launching the various nodes that will be detailed below. The following line was executed in the first Terminal Tab:

```
roslaunch quadcopter_gazebo quadcopter.launch
```

The line above executed the launch file located in the quadcopter_gazebo package. At this point, the main node that has been initialized is the /gazebo node for publishing and subscribing to important information for the quadcopter. It is currently listening to the /cmd_vel topic, however, no Twist() messages are being published to the /cmd_vel topic yet, so the quadcopter remains stationary. Figure 9.3 below shows /gazebo is a subscriber of the /cmd_vel topic. If there is unfamiliarity with the publishing and subscribing mechanism, more details about the ROS Architecture can be found in Appendix E.

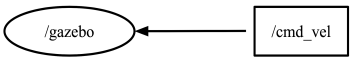

Figure 9.3: The /gazebo node is a subscriber of the /cmd_vel topic.

The second step in the mapping phase was to launch the gmapping node for mapping the indoor environment. The following line was executed in the second Terminal Tab:

```
roslaunch quadcopter_mapping quadcopter_mapping.launch
```

The line above executed the launch file located in the quadcopter_mapping package. It initializes the /slam_gmapping node and launches the mapping RViz file. RViz is a visualization tool used in ROS to view the information being broadcasted by nodes. RViz is used to view the laser scans and the map being updated in real-time. As illustrated in Figure 9.4, the /gazebo node continues to subscribe to the /cmd_vel topic, but now /gazebo is publishing the laser scans from the drone to the /scan topic. The /slam_gmapping node subscribes to the /scan topic which contains messages of type LaserScan. The node turns the LaserScans into a 2D probabilistic grid map of the environment and publishes it to the /map topic using the gmapping algorithm, which is an implementation of the Simultaneous Localization & Mapping (SLAM) problem. In this stage, the quadcopter begins to collect information about the indoor environment whilst being on the ground. Figure 9.4 below shows the updated map of nodes and topics:

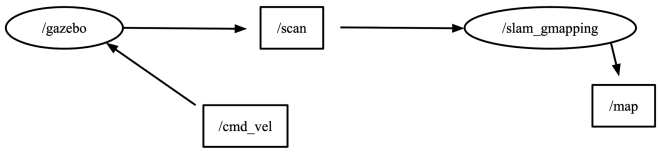

Figure 9.4: After launching the /slam_gmapping node.

The third step in the mapping phase allowed the drone to take off from rest to a stable hover height between 1.3 and 1.5 meters above the ground. The height of the drone can be changed to be higher, but it does not matter as the laser scans from the drone are the same at every height between the ground and ceiling of the school. The following line was executed in the third Terminal Tab:

```
roslaunch quadcopter_takeoff_land quadcopter_takeoff_land.launch
```

The line above executed the launch file located in the quadcopter_takeoff_land package. It initializes the /quad_takeoff_land node. As mentioned previously in the ROS Architecture Example, the /quad_takeoff_land node publishes a linear z velocity of 0.3 m/s to the /cmd_vel topic which the /gazebo node subscribes to for the drone to start taking off. The /quad_takeoff_land node also constantly listens to the /ground_truth_to_tf/pose topic containing the pose information of the drone in the simulator as the quadcopter takes off to see whether the drone has

approached the hovering height. The /gazebo node initially publishes this information to the /ground_truth/state which contains information about the complete state of the quadcopter, which is subscribed to by the /ground_truth_to_tf node. The /ground_truth_to_tf node publishes the pose information of type PoseStamped which contains the pose and orientation of the drone in the x, y, and z dimensions to the /ground_truth_to_tf/pose topic. As the drone approaches the hovering height, the node publishes a lower linear velocity of 0.1 m/s as to coast the drone to a stable hover. When the drone is hovering, a z velocity of 0 m/s is constantly being published. Figure 9.5 shows the map of nodes and topics with the /quad_takeoff_node:

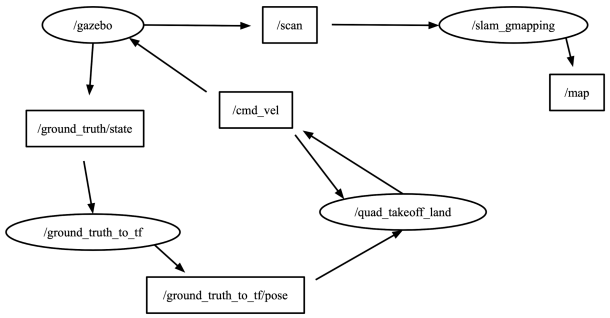

Figure 9.5: After adding the /quad_takeoff_land node.

The fourth step in the mapping phase was to launch the /quad_teleop node to allow keyboard control of the quadcopter in the x-y coordinate plane. The following line was executed in the fourth Terminal Tab:

```
roslaunch quadcopter_teleop quadcopter_teleop.launch
```

The line above executes the launch file located in the quadcopter_teleop package. It initializes the /quad_teleop node. Upon the input of the user, the /quad_teleop node publishes the input velocities to the /cmd_vel topic, as illustrated in Figure 9.6. The code allows the drone with 2 controllable degrees of freedom out of the 6 degrees of freedom: y-axis motion and the yaw motion. There are also keys for increasing and/or decreasing the linear and/or angular speeds of the drone. The map continuously updates as the drone explores the environment. The map of nodes and topics with the /quad_teleop node is shown in Figure 9.6.

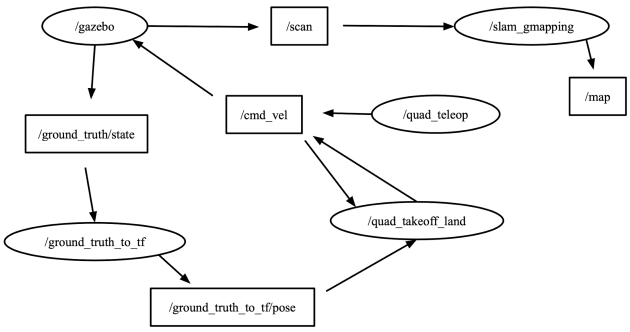

Figure 9.6: After adding the /quad_teleop node.

The fifth step in the mapping phase was to move the drone around to complete the map of the indoor environment. Once the indoor environment had been mapped, the map was saved to the map_server node. The following command was executed in the fifth Terminal Tab to save the map:

```
rosrun map_server mapsaver -f
/home/<username>/catkin_ws/src/quadcopter_navigation/maps/new_map
```

This command saves the map to the following directory. In the directory, a probabilistic grid map file is created of the map and a YAML file is also created that points to the directory of the PGM file so that it may be used later in the navigation phase. Now, the drone will be landed safely. Execute the following command in the last Terminal Tab:

```
rostopic pub /quadcopter_land -r 5 std_msgs/Empty "{}"
```

The statement above publishes an Empty() message to the /quadcopter_land topic from the Terminal at a certain rate. The /quad_takeoff_land node is a subscriber of the /quadcopter_land topic and while the drone is above the ground, the /quad_takeoff_land node publishes a negative command velocity of -0.3 m/s. When the drone has reached the ground, the node is killed. Now, all the processes can be terminated with CTRL-C. The Gazebo environment and RViz will close, and the steps for navigation will now be carried through. The resulting ROS Mapping Architecture is shown below in Figure 9.7.

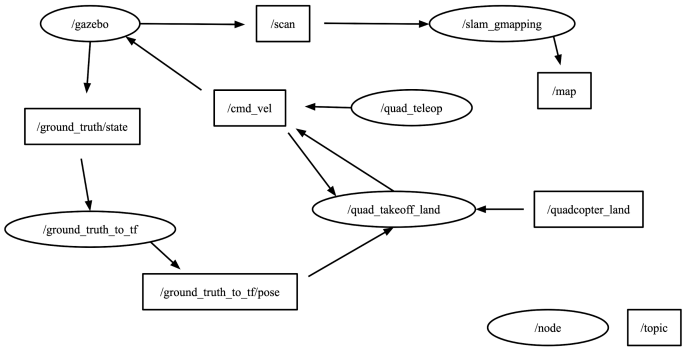

Figure 9.7: The ROS Mapping Architecture.

10.5.2. Navigation Architecture

The first step in the navigation phase is to launch the gazebo environment and instantiate the drone at the center of the environment. Four Terminal Tabs were created for launching the various nodes that will be detailed below. The following line is executed in the first Terminal Tab:

```
roslaunch quadcopter_gazebo quadcopter.launch
```

Refer to Figure 4.2 above in the Mapping Phase that shows /gazebo is a subscriber of the /cmd_vel topic. The second step in the navigation phase launches the /move_base node for mapping the indoor environment. In the second Terminal Tab, the following line was executed:

```
roslaunch quadcopter_navigation quadcopter_move_base.launch
```

The line above executes the launch file located in the quadcopter_navigation package. It initializes the /move_base node and launches the navigation Rviz is used to view the laser scans, the overlaid map, and the trajectory of the drone as it navigates to its destination point. It also launches the /amcl node for localization as the launch file for this node is included in the above launch file.

For the navigation phase, the amcl and move_base nodes are mainly used. AMCL stands for Adaptive Monte-Carlo Localization. It is a localization method that uses a particle filter to estimate the pose of the drone in the environment. The node subscribes to the current laser scans and the occupancy grid map which are compared against each other to localize the drone. The node outputs or publishes the particle filter which contains all the estimates of the robot's pose. The /move_base node allows for the 2D navigation goal to be used in rviz. It listens for

the goal to be set in rviz and outputs the necessary velocities needed for the drone to navigate from its current pose to the goal pose. The node contains global and local path planners. The global planner makes a path from the start point to the goal point. The global planner may be changed through the YAML parameter files. The local path planning method used is called the Dynamic-Window Approach (DWA) planner. For each time step, the local planner finds the optimal trajectory of the quadcopter based on the distance to the goal, the speed of the robot, distance to obstacles, and the robot's distance to the global path. Rviz was used to view the local path planning trajectory in real-time. The third step in the navigation phase was to initialize the /quad_takeoff_land node to start the take-off process. The following line was executed in the third Terminal Tab:

```
roslaunch quadcopter_takeoff_land quadcopter_takeoff_land.launch
```

When a 2D navigation goal is set, the /quad_takeoff_land node temporarily stops publishing the command velocities. The /quad_takeoff_land listens to the /move_base_feedback topic which information is published by the /move_base node about the state of the goal. When the navigation goal is reached, the /quad_takeoff_land node listens to the /cmd_vel topic to make sure no velocities are being published and then resumes the hovering code. The fourth step in the navigation phase was to land the drone safely:

```
rostopic pub /quadcopter_land -r 5 std_msgs/Empty "{}"
```

The resulting ROS Navigation Architecture is shown in Figure 9.8.

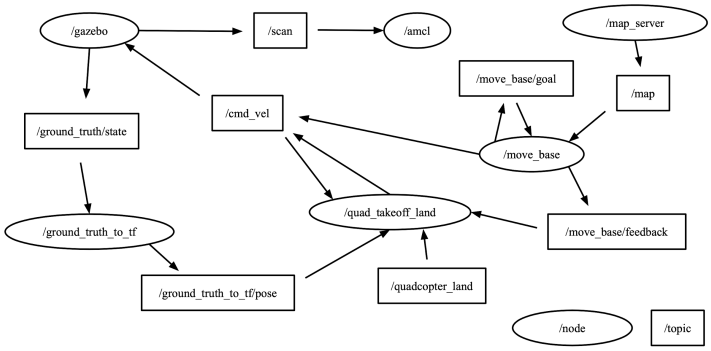

Figure 9.8: The ROS Navigation Architecture.

10.6. Appendix F: Installation Procedure

This Appendix section will cover installing ROS as well as the required packages for the project.

10.6.1. Installing ROS

Follow the instructions in the link below to install the full version of ROS Kinetic and make a catkin_ws folder: http://wiki.ros.org/kinetic/Installation/Ubuntu

10.6.2. Installing the Packages

Clone the hector quadrotor stack into your ~/catkin_ws/src folder. This package is required for the drone model.

```
git clone https://github.com/tu-darmstadt-ros-pkg/hector_quadrotor.git
```

Install the gmapping package, amcl package, and move_base package required for this project. These packages are required for mapping, localization, and navigation, respectively.

```
sudo apt-get install ros-kinetic-gmapping ros-kinetic-amcl
ros-kinetic-move-base
```

You can view the front camera through Rviz, but you can also use video streaming. Install the web_video_server package on your laptop:

```
sudo apt-get install ros-kinetic-web-video-server
```

10.6.3. GitHub Code

If you would like to work with the code and try it yourself, you can clone or download the code from the project's GitHub page. All the instructions for running the mapping, navigation, and video streaming are also provided on the page. Note that a Linux machine running ROS Kinetic and Ubuntu 16.04 were used to make the code for this project. This code is not guaranteed to work for other ROS versions, but many packages used in this project are available in other ROS distros. Clone this quadcopter_sim package into your ~/catkin_ws/src folder.

```
git clone https://github.com/rumaisaabdulhai/quad_sim.git
```