UAV 3D Path Planning Simulation for Obstacle Avoidance

Auburn REU on SMART UAVs 2019

Coleman, Valencia
Columbus State University
coleman_valencia@columbusstate.edu

Shatney, Christina
Plymouth State University
cds1027@plymouth.edu

Biaz, Saad
Auburn University
biazsaa@auburn.edu

Chapman, Richard
Auburn University
chapmro@auburn.edu

September 23, 2019

Technical Report number: CSSE19-03

Abstract

Drone navigation is one of the hottest topics in Unmanned Aerial Vehicles. Simulating UAVs is a way to visualize how a drone can move without the risk of crashing a real drone. One way to navigate a drone in either a simulated or real-life environment is for the drone to have a path planning algorithm built into its programming. A path planning algorithm is a path detection system that can map out the most efficient way to get to a given target, before the drone actually begins moving. Up to this point, these path planning algorithms have only been created for a 2D environment. This report is attempting to create a 3D path planning algorithm using a drone in a pre-made obstacle course simulation. The main algorithms to be implemented in the simulation are A* and Dijkstra. Using a robotics framework, ROS [1], a simulation environment, Gazebo [2], and a sensor data visualizer, RViz [3], a simulated environment will be created which can run a 3D path planning script. The goal is to use the simulation to find the most efficient algorithm for the drone to find its way to its set destination in a 3D environment.

1 Introduction

In today's world, nearly everything is about improving technology and finding ways to make existing software better. One piece of technology that is becoming more widely used is the unmanned aerial vehicle - or UAV. These UAVs have a multitude of uses from recreational ones, to inspecting buildings, to delivering your package. There are many ways to use UAVs, and the computer science community is constantly working to try to improve how they are run and what they can do. One major issue in UAV simulation is that it is always assumed that the simulation is done in 2D. There is a lot of information about simulation in a 2D environment, but the biggest concept that separates UAVs from other robots is that it can move up and down. Why dumb down the uses of a UAV rather than use it to its full potential?

The goal of this report is to change the way that UAVs are viewed in simulation and in reality. One concept that can be done using drones is path planning. Path planning is used in many occasions, maybe most notable flight paths. Before an airplane takes off, there is a general flight plan that the pilot has. Although in an airplane, the pilot is high enough in altitude that obstacles are nearly non-existent, the paths of other airplanes must be taken into consideration. For a UAV, they are often flown at low enough altitudes

that other obstacles may very well be in the way. The concept of path planning allows the user to know the shortest way to get from point a to point b avoiding all known obstacles in its path. There are resources which discuss UAV path planning in 2D [4] [5] [6] [7] [8] [9], or the UAV is in a 3D environment but projected onto a 2D plane. However, the goal of this report is to change that and use UAVs to their full potential, allowing them to also move up and down.

Using the Robot Operating System (ROS), Gazebo, and Rviz, a drone simulation is created which can do the very path planning described above. The drone will be manually flown throughout an obstacle course - octomapping as it goes. Octomapping will be explained in more detail later, but in short, it is way to create a 3D map which contains the occupancy of each node. Once a full octomap has been created, a path planning algorithm (either Dijkstra's algorithm or the A* algorithm) will be run. Finally, a waypoint script will be run which will navigate the UAV throughout the obstacle course using the waypoints provided by the path planning algorithm.

2 The Problem

Drones are versatile pieces of hardware which can do much more than anyone could have ever imagined when they were first thought up. These UAVs have the ability to fly in so many different conditions and there are many different ways to make them work. The goal of this report is to give the drone the ability to find its way to a set destination using 3D navigation through path planning. The problem that this report would like to address, is that path planning has been used in the past to navigate drone simulations, but has always been done in a 2D environment. 2D navigation is likely used in simulations because there is much more documentation and software for it. The reason for this report is the fact that 2D navigation ignores the drone's ability to fly on a z-axis. Path planning in a 3D environment has either never been done or has very poor documentation. Figuring out how to path plan with a UAV in a 3D environment is a whole new problem in itself. It is important to attempt to path plan in 3D because that is why UAVs are so useful. If UAVs have the ability to fly up and down, that should be taken advantage of and it has never been taken advantage of in the past in path planning.

3 Related Work

Other work has been done before relating to path planning, but it has always been done in 2D. There are also papers that work with slightly altering the terrain of the environment, but it is still 2D as shown in a paper by He and Zhao [4]. He and Zhao's [4] paper is very useful for the purposes of this report because it discusses the exact algorithms that are intended to be used and it discusses movement along rough terrain. For the intentions of this report, it would be great if He and Zhao [4] considered moving along rough terrain like potentially moving over obstacles, but this is not the case. This report would like the UAV to be able to move straight up in the air and not entirely against terrain if that is the best possible path. He and Zhao [4] assume that the UAV is moving along the terrain the whole time. Essentially, they are considering the third dimension but they are doing so in a 2D way and the goal of this report is to consider the 3D whole environment. He and Zhao [4] claim that Dijkstra's Algorithm is the best algorithm to use, both in computation time and in shortest distance to goal, when compared to the A* algorithm, Floyd Algorithm, and Ant Colony Algorithm [4]. This statement is interesting because typically A* is considered to be the most efficient algorithm to use. Although He and Zhao's [4] paper contains information about the algorithms intended to be used in this report, it does not provide any information about path planning in 3D which is the main issue at the core of this report.

Another objective this report would like to achieve is having a UAV be able to identify random obstacles as well as obstacles in the preexisting obstacle course that has already been mapped. Thus, it is important to analyze articles that discuss how to identify random objects as well. In a paper by McGuire, Croon, Wagter, Tuyls, and Kappen [10], they discuss obstacle avoidance using a 4 G stereo camera. A 4 G stereo camera is a camera that uses binocular vision (as the human eye sees) and it connects to its home base

through 4 G internet. This report intends to use a VI-sensor in order to produce an octomap. The VI-sensor this report uses, contains a stereo camera so there is a chance that it may be very easy to integrate random obstacle avoidance into Dijkstra's algorithm and A* algorithm using the already existing sensors. This paper discusses the computational restrictions of using a stereo camera so they created their own extension called Edge-FS - a mixture of EdgeFlow and EdgeStereo [10]. McGuire, Croon, Wagter, Tuyls, and Kappen's [10] paper is very useful, but they use a pocket drone which does not have the computing power to handle the stereo camera. For the drones and simulations that this report intends to use, the stereo camera works just fine. This report uses drones from the RotorS package. This package contains many different copters such as the Iris, the Firefly, a fixed wing plane, and the Hummingbird. All of the UAVs in the RotorS package are full-size drones that have the computational ability to handle typical stereo camera algorithms. There is really no use in considering their algorithm in the situation given in this report, although learning about obstacle avoidance in general is very useful. Also, McGuire, Croon, Wagter, Tuyls, and Kappen [10] showed how to break down and read in data from the camera as well parse data for the information needed to actually avoid obstacles [10], but that is what ROS does for this report. The Robot Operating System provides resources that allow its users to just attach sensors and write the code to avoid obstacles. Users do not need to understand how the information is parsed because that is just done for them. Overall, the paper by McGuire, Croon, Wagter, Tuyls, and Kappen [10] is a useful source for using micro aerial vehicles or learning about parsing sensor information. Neither of these are the case for this report so this article was not a major resource; however, this paper does provide a better understanding of obstacle avoidance in general.

Another very useful resource is a paper by Rathburn, Kragelund, and Pongpunwattana [5]. They are also using path planning in UAVs so it is very useful to get an idea of how to path plan on a UAV. The idea of the paper by Rathburn, Kragelund, and Pongpunwattana [5] is to path plan using other drones as the obstacles. Thus, the UAVs must be able to avoid stationary obstacles as well as non-stationary obstacles. They discuss how this idea of avoiding other drones turns this into a stochastic problem. Now the drone must have an idea of a planned out path that will avoid an object at some *approximate* location at an *approximate* time [5]. It is near impossible to exactly predict when a drone will be at some very exact position. Thus that adds much more difficulty to the problem of path planning in a UAV. Rathburn, Kragelund, and Pongpunwattana's [5] paper is extremely useful because it presents path planning in general as well as path planning using other drone paths as obstacles. This is something that this report would like to be able to do in the future if lower level goals are completed first. This is one of the articles that really gave a basis for the ideas in this report.

Another useful article to analyze is a paper by Radmanesh, Kumar, Guentert, and Sarim [6]. They specifically presents information about UAVs using path planning in a 3D environment. However, the only mention about path planning in 3D is that the Genetic Algorithm performs well in 3D environments and the A* algorithm performs poorly. They don't even explain why or how it just states that they do. Other than this, there is very little mention about path planning in 3D at all. Radmanesh, Kumar, Guentert, and Sarim [6] begin by explaining that the first step in using many path planning approaches is to break down the environment into different navigable areas. They recommend breaking the area into tessellations or to break up the area based upon matrix decomposition [6]. Any idea about how to separate an environment into navigable nodes is highly appreciated. However, the actual detail about breaking up the environment into nodes is almost not spoken about at all. This is the type of information that can help this report progress, however, all that this article says is to use tessellations or matrix decomposition, but it does not explain how to make that possible. Radmanesh, Kumar, Guentert, and Sarim [6] then go on to discuss many different types of algorithms and how they can all be used in UAV path planning. This is *very* helpful for this report. When deciding what algorithms to pursue using, it is much easier to see what other people have done and what they have looked into before. Radmanesh, Kumar, Guentert, and Sarim [6] even provide the pseudo code so that it is easier to see what each of these algorithms really entails [6]. As helpful as the work of Radmanesh, Kumar, Guentert, and Sarim [6] is when analyzing algorithms, it is far too focused on the algorithms themselves. The focus of this report in the beginning stages need not be on the algorithms themselves, but on how to break up the environment into the nodes necessary to actually be able to utilize these algorithms at all. The information that this report really needs just is not as well

displayed in this article as would be hoped for; however, it triggers interest in looking into tessellations and matrix decomposition to potentially break up the UAV environment into nodes.

After learning about the idea of tessellations, it would be a good idea to look into how to break an environment into tessellations to make it navigable. The thesis by Pfaff [11] is just what is needed to look into tessellations. His thesis presents more about tessellating an environment which is what really needs to be learned about to begin the research done in this report. Pfaff [11] explains that tessellating an environment is for a 2D environment or for an environment with very little 3D movement. It does break the environment into different navigable shapes as this report is intending to do, however, it does not work in the way that this report needs it to. Pfaff's [11] thesis is intending on flying the drone far in the air where there will be virtually no objects to avoid by moving up and down. His thesis even mentions a third dimension can be added to the tessellations but it does not discuss how to do so [11]. This is likely due to the fact that Pfaff [11] is not intending in flying up and over obstacles. Pfaff's [11] thesis was extremely informative on many different topics, but the information that this report is focused on is the information about tessellations. This source, although it was not able to show the exact information that is needed for this report, helped to understand that tessellations likely are not the solution to the problem that this report is pursuing.

Another informative resource to analyze is a paper by Zhao, Zheng, and Liu [7]. Zhao, Zheng, and Liu [7] compare 231 other papers and discuss path planning in UAVs. This is a useful resource because it allows the reader to see other papers and how they all compare to each other. If there is a certain paper that seems informative, all of its information can be found in the resources area of their paper. Zhao, Zheng, and Liu [7] combined many different papers into one accessible place. They focus on trying to steer away from A* which is one of the main algorithms that this report is interested in using, so it would be more helpful if they were slightly less bias against the A* algorithm. Nonetheless it provides information about other algorithms to consider. Zhao, Zheng, and Liu [7] also present some of the drawbacks of using the A* algorithm. They are very informative when presenting the constraints to consider when path planning with a UAV [7]. This is something that this report really needs to consider. Since this report is intending to use the A* algorithm, it is also important to consider what heuristic should be used. Originally, it may be the most straightforward to just use a distance from the goal heuristic, but this article presents that it is also important to consider other heuristics such as stealth, feasibility, and realistic implementation [7]. Overall, Zhao, Zheng, and Liu [7] show all of the different algorithms to consider, as well as show many different resources that can be used. Although they do not show exactly the information that could be useful for this report, it provides a stepping stone to launch off of.

The last really useful related paper is a paper by Lin, Xiao, and Michalewicz [8]. Lin, Xiao, and Michalewicz [8] discuss path planning in open space rather than through the use of nodes as A* and Dijkstra's algorithms do. They explain that, in the way that they chooses to do path planning though their new Evolutionary Navigator algorithm, it is ensured that the robot is keeping a sufficiently safe distance from obstacles whereas some other algorithms that follow a graph may navigate too closely to obstacles. Their new algorithm also learns as it continues to find paths so it will find better paths as it continues to be used [8]. Their algorithm provides information for this report and using free space is something that may be easier since there would be no need to figure out how to divide the UAV environment into a grid system... Unfortunately, their algorithm currently only works in 2D and the whole point of this research is to do the research in 3D. Rather than adapting this new algorithm to be 3D, it appears that it may be more reasonable to find resources which can help turn A* and Dijkstra's algorithm into a 3D algorithm. Although the algorithm created by Lin, Xiao, and Michalewicz [8] likely works better than Dijkstra's and A* algorithms, there is definitely going to be less resources to help adapt it into 3D.

Overall, the other works related to the research to be done in this report, provide information to go off of. All of the papers help to decide what to research and they provide resources to analyze. However, for the most part, these papers are fairly vague and do not outline the true course of action to complete their goals. It appears that most of what is done in this report will be learned through tutorials and trial and error since there is very little documentation based around the goals of this report. There is an abundant amount of information about path planning in a 2D environment [4] [6] but *no* information about path planning in a 3D environment at all. The goal of this report is to improve upon these related works, do research that

has never been done before by doing path planning in a 3D environment, and most importantly to discuss the approach in a detailed enough fashion that others are capable of starting where this report leaves off. These other related papers provide important information, but none of the papers really outline their exact process, and that is why it is so important for this report to do just that. This report aims to guide others in the use of ROS, the use of UAV simulation, and the use of 3D path planning.

4 The Approach

When approaching how to path plan in 3D, there are many details to take into consideration. Many of the related works have very little to do with 3D path planning so it appears that everything that needs to be done to achieve the goals of this report will be discovered through trial and error. This report intends to achieve 3D path planning through a simulation approach since there is so much trial and error going into it. The simulation will be done using ROS, Gazebo, and the RotorS [12] package along with many packages that integrate with ROS which will be discussed later. The UAV used in this report is the Iris, however, any UAV in the RotorS package has the same abilities as the Iris and could be used to carry out the work done in this report. This report will use octomapping in order to create a 3D map of the environment which shows the occupancy of nodes in the environment. Using the occupancy information from the octomap, an algorithm written in c++ will be made for both Dijkstra's path planning algorithm and the A* path planning algorithm. These algorithms will output a vector of the waypoints needed to reach the goal position in the most efficient way possible based on what is considered efficient by that algorithm. This report chose to use Dijkstra's algorithm and the A* algorithm because, although there is no documentation of them being used in 3D, these algorithms are used frequently throughout the related works [4] [6] [7]. There are other algorithms that are said to work faster and find paths with shorter distances to the goal, however the documentation for Dijkstra's algorithm and the A* algorithm is supple. There will likely be more resources to adapt these algorithms for a 3D approach rather than some of the other algorithms such as Ant colony [4] and the Evolutionary Navigator [8] that have less documentation and support. After writing the algorithms which produce these waypoints, a waypoint script must be created which can actually moves the UAV waypoint to waypoint until the UAV reaches the goal position. This waypoint script - discussed in further detail later - can be adapted from the existing waypoint script provided in the rotorS package. After all of the pieces are there, everything can be put together to create an obstacle course, octomap it, path plan it, and then navigate it.

4.1 Installing Software

The first step to this whole process is to download all of the necessary software. This report uses a Windows 10 operating system. ROS was created to be used on a Linux system therefore to adapt to the use of a Windows 10 operating system, a WSL can be used since ROS only uses a Linux terminal. This report uses Ubuntu 16.04 LTS. There is a newer version of Ubuntu available (Ubuntu 18.04), however, most documentation and tutorials written at the time of this report are still done using ROS Kinetic which must be downloaded on Ubuntu 16.04. At this time, there is a newer version of ROS available as well called ROS Melodic. However, again, most of the resources at this time use ROS Kinetic and it is easier for users new to ROS to have more documentation available. The steps to download all of the necessary software for ROS are included in the following.

4.1.1 Installing a WSL

Jan Bernlöhr [öhr2017] created a page on his blog which explains how to install ROS on a Windows 10 operating system very nicely. He uses Ubuntu 18.04 and ROS Melodic, however, the same steps can be taken just adjusting ROS Melodic to ROS Kinetic and adjusting Ubuntu 18.04 to Ubuntu 16.04. Since this report uses Windows 10, this is an useful resource to get everything installed. First, update Windows 10 to be at least version 1703. This can be checked by going to settings, the systems tab, the about tab, and then

looking at the version number. If it is not version 1703 or higher, update it now. This can be done by going to settings, update and security tab, the windows update tab, and then clicking "check for updates" to see if there are any available. A version of at least 1703 must be installed for this ROS installation to work. Once the operating system is updated, Bernlöhr [öhr2017] points to another website which explains how to install the Ubuntu 16.04 WSL [öhr2017].

To install the WSL, use the steps shown in the provided website [13]. This website explains that before a WSL gets installed, first the WSL feature must be enabled. Do this by running

```
Enable-WindowsOptionalFeature -Online -FeatureName Microsoft-Windows-Subsystem-Linux
```

[14] as an administrator on the powershell. After this has been installed, restart the computer. Once the computer has restarted, now the actual WSL must be installed. This can be installed off of the Microsoft Store by typing in Ubuntu 16.04 in the Microsoft store [15]. This is all that is required to install the WSL. [13] After this WSL is installed, ROS must be installed.

4.1.2 Installing ROS and Gazebo

The rest of the information to install ROS can be found on Bernlöhr's [öhr2017] blog. To install ROS follow everything that Bernlöhr [öhr2017] does in his blog, however, do not install the full desktop version because the full desktop version installs the Gazebo version associated with the ROS version. Gazebo 7 is the Gazebo version associated with ROS Kinetic and Gazebo 9 is much smoother. Gazebo 9 is the Gazebo version used in this report because it runs with less lag, however, Gazebo 7 can be used with this report to achieve the same results. In order to download ROS Kinetic with Gazebo 9 follow the steps that Bernlöhr [öhr2017] provides just adjusting the "sudo apt install -y ros-melodic-desktop-full" line. As Bernlöhr [öhr2017] states, do the first three bash commands in the Ubuntu WSL terminal.

```
$ sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main" >
    /etc/apt/sources.list.d/ros-latest.list'
$ curl -sL
    "http://keyserver.ubuntu.com/pks/lookup?op=get&search=0x421C365BD9FF1F717815A3895523BAEEB01FA116"
    | sudo apt-key add
$ sudo apt update
```

[öhr2017] However, for the next command there are different lines to implement that come from the wiki.ros.org website [16].

```
$ sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main" >
    /etc/apt/sources.list.d/ros-latest.list'
$ sudo apt-key adv --keyserver 'hkp://keyserver.ubuntu.com:80' --recv-key
    C1CF6E31E6BADE8868B172B4F42ED6FBAB17C654
$ sudo apt-get install ros-kinetic-desktop
```

The first line is run to allow the computer to accept software from ROS. The second line is to set up the keys. The last line will install ROS and Rviz, but not Gazebo. Now to continue installing ROS, continue to use the wiki.ros.org website [16].

```
$ sudo rosdep init
$ rosdep update
$ echo "source /opt/ros/kinetic/setup.bash" >> ~/.bashrc
$ source ~/.bashrc
```

The first two lines of code allows installing dependencies through ROS easily. The second two lines initialize environment variables. The base of ROS is now installed, however there are many dependencies

and packages to still be installed. By running

```
$ sudo apt install python-rosinstall python-rosinstall-generator python-wstool build-essential
```

Rosinstall gets installed which is used to install many ROS packages that will be needed in the future [16]. Now that ROS is installed without Gazebo, Gazebo 9 must be installed separately. To install Gazebo 9, use the information provided by the Gazebo website [17]. The step-by-step install is best because this tends to be better at catching errors. To install Gazebo run the following lines of code.

```
$ sudo sh -c 'echo "deb http://packages.osrfoundation.org/gazebo/ubuntu-stable `lsb_release -cs`
    main" > /etc/apt/sources.list.d/gazebo-stable.list'
$ wget http://packages.osrfoundation.org/gazebo.key -O - | sudo apt-key add -
$ sudo apt-get update
$ sudo apt-get install gazebo9
$ sudo apt-get install libgazebo9-dev
```

The first line ensures that the computer is going to accept the software provided through packages.osrfoundation.org. The second line sets up the keys. The third line updates the Ubuntu database. The last two lines install Gazebo 9 and an extra package [17]. Now that Gazebo is installed, Gazebo packages must be installed. Again, this information comes from the Gazebo website but this time it comes from a different site [18]. This page shows how to create a catkin workspace and then begin installing the extra packages. A catkin workspace is needed in order to install and run catkin packages. In order to run just about any package, a catkin workspace is needed so creating that now is a good idea. To create a catkin workspace, run the following lines in the Ubuntu terminal.

```
$ mkdir -p ~/catkin_ws/src
$ cd ~/catkin_ws/src
$ catkin_init_workspace
$ cd ~/catkin_ws
```

The first line creates the catkin workspace along with creating a source folder within the catkin workspace. The second line moves into the source folder within the catkin workspace. The third line builds the catkin directory and adds a CMakeList.txt file and a package.xml file. Both of these files are needed in order to run code within the catkin workspace. The fourth line moves the terminal back into the catkin workspace folder rather than the source folder [18]. From here, a different command must be used from the last one given in the website. The website uses "catkin_make" which typically works fine. Catkin_make is used to build everything within the catkin workspace. It is required to compile all of the files. In the case of this report, "catkin build" instead of "catkin_make" is used. The idea of each command is the same but "catkin build" creates log files while "catkin_make" does not. It is important to use "catkin build" in this case because in the future the RotorS package will need to be compiled and this package requires "catkin build" because it needs the log files to compile correctly. Thus, the next line that must be run is

```
$ catkin build
```

Again, this builds and compiles the whole workspace. Now the website [18] can be followed again. The next bash command is the following commands.

```
$ echo "source ~/catkin_ws/devel/setup.bash" >> ~/.bashrc
```

This command sources the environment. Next git must be installed onto the terminal. Git is used to install many packages and dependencies. To install git run the following command in your terminal.

```
$ sudo apt-get install git
```

Now the actual code must get downloaded. This is achieved by running the following lines of code.

```
$ cd ~/catkin_ws/src
$ git clone https://github.com/ros-simulation/gazebo_ros_pkgs.git -b kinetic-devel
```

Often when installing packages, there are random dependencies that are missing. To find these missing dependencies, run the following bash commands.

```
$ rosdep update
$ rosdep check --from-paths . --ignore-src --rosdistro kinetic
$ rosdep install --from-paths . --ignore-src --rosdistro kinetic -y
```

The last line will actually install those dependencies if there are any missing [18]. Again, to build these packages, the website [18] says to use "catkin_make" however "catkin build" will be used in place of "catkin_make" throughout this report. Thus, to build the packages follow the commands below.

```
$ cd ~/catkin_ws/
$ catkin build
```

Everything is now installed in order to run Gazebo if the project was on Ubuntu OS. However, since this report uses Windows 10 and a WSL, an X Server must get installed. An X Server allows GUIs within WSL since the WSL only takes care of the Ubuntu terminal. To install the X Server, Jan Bernlöhr's [öhr2017] blog will be used again. In his blog, he provides a link to the the installation of VcXsrv [19]. This is the X Server used in this report. Simply go to this link, click download, and follow the directions for the installation. After the X Server is installed, more bash commands must be run in order for the WSL to use the X Server. These commands are the following.

```
$ echo "export DISPLAY=:0" >> ~/.bashrc
$ source ~/.bashrc
```

After the X Server has been sourced for the WSL to use it, the X Server must be launched from the start menu of the computer. *Each time after start up, the X Server must be launched from the start menu of the computer in order to use a GUI within the WSL.* When launching the X Server, all of the default settings can be used except on the third page, Native OpenGL must be unchecked [öhr2017]. Now everything should be installed to run Gazebo on a windows machine. To test if it works, go to the WSL terminal and type in "gazebo". This may take a few minutes on start up, but eventually an empty gazebo world should appear.

4.1.3 Installing RotorS package

Now that ROS and Gazebo are installed, the extra packages for to run the simulator must be installed. This report uses UAVs from the RotorS package. The RotorS package is a package that provides multiple UAVs along with many different launch files and world files that can be used to launch the mavs in different worlds within Gazebo. In order to use the RotorS package, MAVROS must be installed [20]. The dev.px4.io website shows how to do this MAVROS installation in a very detailed way. Again, the source installation helps to pinpoint errors if any occur so the source installation is what this report uses. In order to install MAVROS, wstool and catkin_tools needs to be installed. Catkin_tools and wstool should be installed when Ros is on the computer, however, if any errors occurred, it can be installed manually through the following bash command.

```
$ sudo apt-get install python-catkin-tools python-rosinstall-generator -y
```

Now MAVLink must be installed. First move into the catkin_ws, then initialize the workspace and add the .rosinstall file by doing the following commands.

```
$ cd ~/catkin_ws
$ catkin init
$ wstool init src
$ wstool init ~/catkin_ws/src
```

The first line moves into the catkin_ws, the second two lines initialize the workspace and add the .rosinstall file, and the last line sources the space. Now, Mavlink needs to be installed using the following bash command.

```
$ rosinstall_generator --rosdistro kinetic mavlink | tee /tmp/mavros.rosinstall
```

After Mavlink is installed, MAVROS can finally be installed. This report chooses to install from the released/stable version because the stable version will likely work with less issues. Thus, to install from the stable version enter the following bash command into the terminal.

```
$ rosinstall_generator --upstream mavros | tee -a /tmp/mavros.rosinstall
```

Now, the workspace needs to be created and all of the dependencies must be downloaded. This can be created using the following bash commands.

```
$ wstool merge -t src /tmp/mavros.rosinstall
$ wstool update -t src -j4
$ rosdep install --from-paths src --ignore-src -y
$ ./src/mavros/mavros/scripts/install_geographiclib_datasets.sh
```

[20] The last line in this code may need to be run with "sudo" in the beginning depending on the privileges of your account. Now MAVROS is installed. In order to compile everything run the following bash commands in your terminal.

```
$ catkin build
$ source devel/setup.bash
```

This "source devel/setup.bash" command must be run anytime after a new package is added in order to access the nodes within the workspace. If the terminal says that some file isn't within the folder, but it is definitely there, the issue is likely that this source command needs to run to re-source the environment [20]. Now that MAVROS has been installed, the RotorS package can get installed using the RotorS simulator github [12]. Notice that some of the commands within the github have already been run so some edits need to be made in order to install the RotorS package now. First, run the following bash commands in the terminal.

```
$ sudo apt-get update
$ sudo apt-get install ros-kinetic-joy ros-kinetic-octomap-ros ros-kinetic-mavlink python-wstool
    protobuf-compiler libgoogle-glog-dev ros-kinetic-control-toolbox
$ sudo rosdep init
$ rosdep update
$ source /opt/ros/kinetic/setup.bash
```

The first line updates all packages. The second line installs extra packages (note that even if these packages are already installed on the computer, it will do no harm to run the installation again). The third and fourth line initialize and update dependencies. The last line sources the environment. Now some more commands can be run to ensure that the workspace is fully prepared for the installation of the RotorS package.

```
$ cd ~/catkin_ws/src
$ catkin_init_workspace # initialize your catkin workspace
$ wstool init
$ wget https://raw.githubusercontent.com/ethz-asl/rotors_simulator/master/rotors_hil.rosinstall
$ wstool merge rotors_hil.rosinstall
$ wstool update
```

The first line puts in the terminal in the source folder within the catkin workspace. The second and third lines initialize the workspace. The last three lines install the RotorS package and update all packages. Now the workspace must be built and sourced.

```
$ cd ~/catkin_ws/
$ catkin build
$ echo "source ~/catkin_ws/devel/setup.bash" >> ~/.bashrc
$ source ~/.bashrc
```

Now the RotorS package is built and ready for use [12]. To view the launch files that come with the package, enter the following bash commands.

```
$ cd ~/catkin_ws/src/rotors_simulator/rotors_gazebo/launch
$ ls
```

To view the worlds that come with the package, enter the following bash commands.

```
$ cd ~/catkin_ws/src/rotors_simulator/rotors_gazebo/worlds
$ ls
```

To experiment with the launch files, world files, and mav names, use the following bash command.

```
$ roslaunch rotors_gazebo mav_hovering_example.launch mav_name:=firefly world_name:=basic
```

Changing the "mav_hovering_example.launch" to the name of a different launch file will launch the mav differently in Gazebo. Changing "firefly" to the name of a different mav provided in the package such as "iris", or "hummingbird" will change the mav that spawns in the Gazebo environment. Changing "basic" to the name of any of the other ".world" files found within the worlds folder will change the world that the mav spawns in in Gazebo.

If at any point in this installation process, there appears to be an error with dependencies, the following bash command can be run.

```
$ rosdep install --from-paths src --ignore-src --rosdistro kinetic -y
```

This command finds any missing dependencies and downloads them. [21] Also notice that as ROS grows and updates, some of these commands may no longer work. If any errors occur during installation, often the error message can be looked up and usually the case is just that a package or dependency is missing.

4.2 Creating a World

This report creates its own .world files within Gazebo because the goal of this report is to octomap and 3D path plan around an obstacle course. There are no obstacle course worlds prebuilt into the RotorS package, thus in this report, obstacle course worlds are created. To create a world, first launch gazebo by entering the following command into a WSL terminal.

```
$ gazebo
```

This launches an empty gazebo world. From here, make the obstacle course however is desired using the objects within Gazebo. There are house objects available, houses available, or the cube and sphere blocks work just fine. An example obstacle course is shown in figure 1. This obstacle course would be best used to start the UAV on one side of the large center block and try to make it navigate to the other side of the large block. Once the obstacle course is complete, the world must be saved. First, go to "Save world as" under the file tab in the top left. Navigate into the "catkin_ws/src/rotors_simulator/rotors_gazebo/worlds" folder. Name the world and make sure it ends in .world.

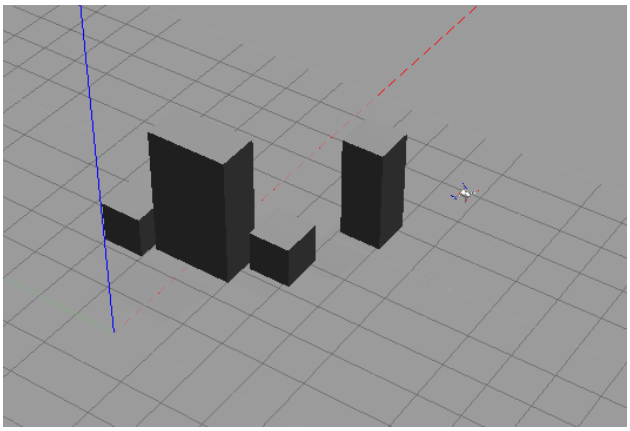

Figure 1: Example of World Created in Gazebo

4.3 Octomapping

Now that the RotorS simulator has been installed, the software needed in order to octomap can be installed. The octomap software directly coincides with the RotorS package so that needed to be installed first. Octomaps show the environment in very great detail far beyond what many other 3D maps are able to show. Octomaps are also compressed into very reasonably sized files. They contain massive amounts of information but hold onto that information in a very efficient manner. Along with the fact that octomaps specifically supply occupancy information directly, these are the reasons this report uses octomapping. The following bash commands come from the PX4 website [22], however the ROS distro has been changed to Kinetic rather than Melodic to suit the needs of this report.

```
$ sudo apt-get install ros-kinetic-octomap ros-kinetic-octomap-mapping
$ rosdep install octomap_mapping
$ rosmake octomap_mapping
```

These commands combine together to install the octomapping library - the packages and dependencies. Now some adjustments must be made to existing files in the catkin workspace. In the WSL terminal, run the following commands.

```
$ cd ~/catkin_ws/src/rotors_simulator/rotors_gazebo
$ vim CMakeLists.txt
```

This brings up the text editor to adjust the CMakeLists.txt file. Most website will say not to touch this file because it adjusts as packages and files are added, however, the file must be edited to work for octomapping. Arrow down to the bottom of the file using the arrow keys. To begin typing type "i". This puts vim into insert mode. Begin typing and add the following three lines to the very bottom of the file.

When the lines are typed, enter the "esc" key and type ":x". The "esc" key brings vim out of the insert mode and the ":x" closes the vim editor. Now the package.xml code must be edited. This is another file

that many website say not to adjust, however, the file must be edited in order to do octomapping. In the WSL terminal type the following commands.

```
$ cd ~/catkin_ws/src/rotors_simulator/rotors_gazebo
$ vim package.xml
```

[22] This brings up the vim text editor to adjust the package.xml file. At the very bottom of the file but before "</package>" line, add the following two lines.

Notice that the PX4 website [22] says to use "<run_depend>" instead of "<exec_depend>", however <run_depend> gives errors. Now the octomap_tracking_server.launch file needs to be edited within the octomap_server. In order to reach the octomap_server there has to be access to root files from within the WSL. To do this type the following bash commands into the WSL terminal.

```
$ cd
$ cd ../..
$ cd opt/ros/kinetic/share/octomap_server/launch
$ sudo chmod +w octomap_tracking_server.launch
$ sudo vim octomap_tracking_server.launch
```

The first three lines move the terminal into the octomap_server. The fourth line allows write access to the octomap_tracking_server.launch file and the last line opens the vim text editor on the octomap_tracking_server.launch file. Within this file there are a few lines that need to be edited. First, change the line

to

This changes the frame_id to be the world rather than the map. This is needed for Rviz when octomapping. Next change the line

to

Those are the changes that need to be made to the octomap_server. Now when actually running the simulation for octomapping, three different terminals are needed. In the first terminal enter the following command.

```
$ roslaunch rotors_gazebo mav_hovering_example_with_vi_sensor.launch mav_name:=iris
    world_name:=obstaclecourse1
```

The world name should be the name of the obstacle course that is being octomapped is called. This command launches the mav_hovering_example_with_vi_sensor.launch file. This file launches the specified mav into the Gazebo world with a vi-sensor attached. This is important for octomapping because the vi-sensor is required in order to have the proper equipment to octomap. This file will also launch the UAV one meter into the air in the z-axis direction. The mav_name is iris because that is the UAV that this report uses. The world_name is obstaclecourse1 because that is the world that was saved that is now going to be octomapped. In the second terminal run the following command.

```
$ rviz
```

This command opens Rviz. Rviz has been mentioned before but never discussed in detail. Rviz is a GUI that allows the user to see what the sensors in Gazebo are seeing. It is what allows the user to view what the vi-sensor is seeing thus allowing the user to view the 3D octomap. In the last terminal run the following command.

```
$ roslaunch octomap_server octomap_tracking_server.launch
```

[22] This command launches the octomap_tracking_server which is essentially what allows the UAV to octomap at all. The UAV may have the vi-sensor attached to its body but the server isn't actually reading

in information until this command is run. After all of the terminals are running, some adjustments need to be made within Rviz. Change the "fixed frame" from "map" to "world". Then, click on "add" and scroll to "MarkerArray". Add this marker array. Then within the marker array on the left side tool bar, change the topic to "occupied_cells_via_array" [22]. Now an octomap should be beginning to form on Rviz. In order to octomap the whole area, the UAV must be moved using waypoints. To move using waypoints open yet another WSL terminal. In this terminal type the following command.

```
rosrun rotors_gazebo waypoint_publisher x y z yaw delay __ns:=iris
```

In place of x, y, z, yaw, and delay, enter the actual waypoint values. To move one grid space in the positive x direction, enter "1 0 1 0 0" in place of x y z yaw delay. The delay should always be 0 and the yaw is in degrees based on the number of degrees away from facing the positive x direction that the UAV would be facing. When entering waypoints, try to only move one meter at a time. The simulator is slow and the UAV moves at a realistically unstable pace. Thus, moving more than one meter at a time is likely to cause issues.

After the whole map is octomapped as shown in figure 2, now the octomap must be saved. In the same terminal where the waypoints were being entered, enter the following command.

```
rosrun octomap_server octomap_saver -f filename.ot
```

This may take a few moments but it will save the map as a .ot (OcTree file) as shown in figure 3. Then once that has been saved, run the following command.

```
rosrun octomap_server octomap_saver -f filename.bt
```

This will likely also take a few minutes but this will save the map as a .bt (Bonsai Tree file) as shown in figure 4. Both the .ot and the .bt file can be used for the purposes of 3D path planning. The .ot file is just a slightly more compressed .bt file but they will both work well for the intended uses in this report. To visualize the saved octomap files - either .ot or .bt - there is a program called Octovis [23]. This program can be quickly installed by enter the following bash command into the WSL terminal.

```
$ sudo apt-get install ros-kinetic-octovis
```

[23] To then run Octovis simply type "octovis" into the WSL terminal. This will open an empty Octovis world. To visualize the saved octomaps, go to the file tab and select the octomap to open. Now the octomap file can been analyzed more closely.

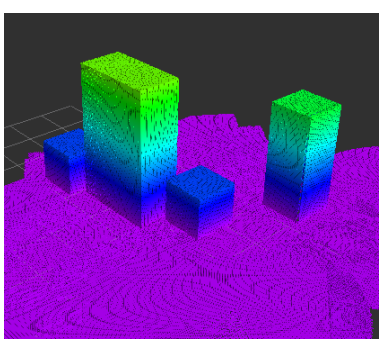

Figure 2: Complete Octomap

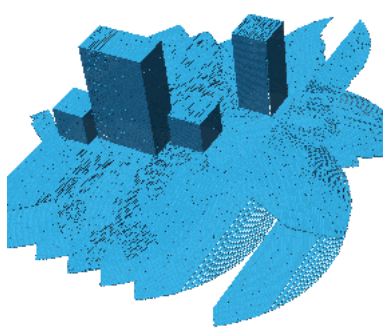

Figure 3: Example of an .ot file

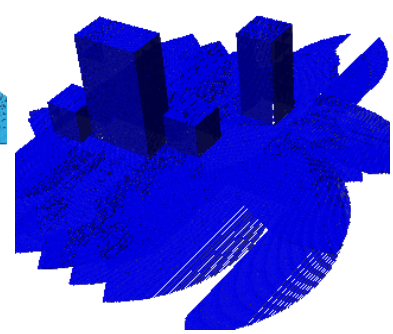

Figure 4: Example of an .bt file

4.4 Creating Algorithms

The main focus of this project is based around path planning in 3D. This means that there needs to be algorithms which are used to path plan. For the purposes of this report, Dijkstra's Algorithm and the A* algorithm will be used for path planning. As stated before, these algorithms are very frequently used and compared in 2D path planning so the idea to convert both of them into 3D only makes sense. Path planning algorithms are just what they sound like; they are algorithms used to plan a path. These algorithms are typically run with a whole map provided so that all obstacles can be viewed before the path is planned out. There are also ways to make path planning algorithms that regenerate the path as obstacles appear, however, for this report, just a general pre-decided path plan is the goal.

In summary, Dijkstra's algorithm is a greedy path planning algorithm. This means in all of the choices that are made, the "best" choice is considered the choice that will gives the shortest distance immediately whether or not this is the "best" choice overall. To start, there is a graph of nodes and there is a "weight" between each node which is determined based on what the goal of the algorithm is. In the case of this report, the goal is to move from a start position to a goal position in the shortest distance possible, thus the weights between the nodes are the distances between the nodes. An example of a weighted graph is shown in figure 5.

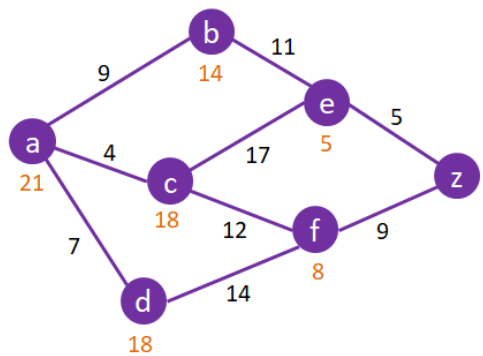

Figure 5: Graph [24] to demonstrate A* and Dijkstra's algorithms

This graph is not to size but it can be assumed that the black numbers between nodes is the distance between the two nodes. Also notice that there is an orange number associated with each node itself. The goal is make these numbers as small as possible because these numbers represent the distance from the start node. Thus, making the orange number on the goal node as small as possible means that the distance from start to finish is as small as possible. In the beginning, assume that these orange numbers start at infinity. To start the algorithm, there is a list of the visited nodes which begins as empty. Add the start node to the list of visited nodes. Look at all of the nodes neighboring the start node (attached to the start node by a weighted line). In the case of figure 5, assume that the start goal is node `a`. The neighboring nodes to node `a` are nodes `b`, `c`, and `d`. The distance from `a` to `a` is 0, thus the orange number on node `a` would become 0. The orange number on `b` becomes 9 because that is the distance from `a` to `b` and 9 is smaller than infinity. The orange number on `c` becomes 4 because the distance between nodes `a` and `c` is 4 and 4 is smaller than infinity. Then the orange number for node `d` becomes 7 because the distance between nodes `a` and `d` is 7 and 7 is smaller than infinity. Now that all of the neighboring nodes have been checked, the next step is to move the next smallest node into "front". This "front" is a list of all of the nodes that have been viewed in the past and since nodes `b`, `c`, and `d` are neighbors of node `a` they have been viewed. Thus, add node `c` to the visited list because it has the smallest associated orange number. Continue with this pattern, now looking at node `c`'s neighbors. Notice if there is a way to get to a neighboring node with a smaller associated cost even if it takes more nodes to get there, adjust the orange number to represent this new shorter path. Once all neighbors have been analyzed, move to the next smallest node in the "front" and so on until the goal node

is reached. In the end, there will be a chain of nodes which leads to the goal node in the shortest possible path. Dijkstra's algorithm can definitely be hard to comprehend in words but trying it out helps it make a lot more sense.

To make the transition between Dijkstra's algorithm and the A* algorithm, simply add a heuristic. The heuristic changes based on what the goal of the algorithm is so, again, since in this case the goal is to get to the goal node in the shortest path possible, the heuristic used is the straight line distance between the start node and the goal node. Now, when considering the orange number of each node, it will not only consist of the weight in between nodes, but also the distance to the goal node. To calculate this new orange number, do it just as it was done before but simply add the distance from the current node being analyzed to the goal to the orange number as well. This will drag the path towards the goal much faster because nodes that are extremely far from the goal but have a short cost likely won't be considered.

When it comes to programming these algorithms, there is a github provided by Juan Crespo [9] which executes Dijkstra's algorithm and the A* algorithm in a way which could be very useful to what this report needs to do. Crespo's [9] code is in python and this report uses C++. His code is also assuming that a provided GUI has been downloaded and this report is using Gazebo simulator. Crespo's [9] code is also in 2D and this report is trying to execute the algorithm in 3D. There are many things that need to be adapted, but Crespo's [9] code is attractive because it makes its own nodes. One major issue with doing path planning in 3D is that path planning algorithms are all based off of graphs with nodes and there are no nodes within Gazebo. That is why Crespo's [9] code stands out so much. In the following, a general summary of how both Dijkstra's algorithm and the A* algorithms are coded is provided.

In this report, the general form of the scripts are very similar to Crespo's [9] although somethings have been edited for the sake of 3D versus 2D, for the sake of C++ versus Python, and for the sake of Gazebo versus an obstacle GUI. Notice that either a .bt or .ot octomap file can be run, however the code to be used changes slightly depending on what octomap version is used. Both algorithms adapted to this report can be seen in the appendix Dijkstra's algorithm running a .ot file being located at A.3, Dijkstra's algorithm running a .bt file being located at A.4, A* algorithm running a .ot file being located at A.5, and A* algorithm running a .bt file being located at A.6. An explanation of Dijkstra's algorithm code is in the following.

```
begin
  comment: This struct will represent waypoints and the "nodes" being used within the algorithm
  Create a Point struct containing three doubles x, y, and z
  comment: This struct represents the same points as the Point struct but now the cost of the points
  comment: are also being taken into consideration
  Create a PointCost struct containing three doubles x, y, and z and a float cost
  comment: This struct represents nodes when they are in the "front"
  comment:  The cost represents the most up-to-date cost of the node, the pos is the postion of the node,
  comment: and previous is the node used to reach that point
  Create a struct consisting of a float cost, a Point pos, and a Point previous
  comment: Later Fronts must be compared, but there is no comparator for them
  Create a comparator class which compares elements in the Front based on cost [25]
  begin
    comment: Now the actual coding for Dijkstra's algorithm may begin.
    comment: The Dijkstra's function takes the start node, the goal node, and the octomap as parameters
    Create variables for the minimum and maximum values of the environment are created
    comment:  This will represent the "front" and contain all Front objects
    Create a priority queue called "front"
    Add an initial Front object containing a cost of nearly 0, a current position of
    the start node, and a previous NULL node
    Create a 3D array of doubles - initialized to all zeroes - called "visited" where
    the size of each dimension of the array is represented by the lengths of the x, y
    and z axes
```

```
    Create a 3D array of Points called came_from where the size of each dimension is
    dependent on the size of the axes in the environment
    comment: Since this report is moving in 3D, there are 6 possible movements to make. Up or down
    comment: on the x axis, up or down on the y axis, or up or down on the z axis
    Create an array of the possible movements that can be made
    while the size of the front is non empty
        Pop the node with the smallest cost off of the "front"
        if This node has been visited
            continue
        Mark the nodes position in the visited array to be the cost of the node
        Mark the nodes position in the came_from array with the value of the previous
        node
        if this node is the goal
            break
        For i = 0 to 6 step 1
            Create a new node from the possible movements in the movements array
            if node is outside of environment constraints
                continue
            search for the new node's position in the octomap
            if node is null and unvisited
                Create a bounding box iterator around the node in the octomap and ensure
                that there are no objects within .8 meters of the bounding box
                if The bounding box contains no obstacles
                    Push the new node onto the "front"
            if node is unvisited and node is unoccupied
                Create a bounding box iterator around the node in the octomap
                and ensure that there are no objects within .8 meters of the
                bounding box
                if The bounding box contains no obstacles
                    Push the new node onto the "front"
    Create a vector for the path
    if the position is the goal
        while The position of the node isn't the start node
            Add the position of the node onto the vector
            Change the position of the node to be the value of the previous node from the
            came_from array.
        Add the start node to the path
        Reverse the path
    return the path
  end
end
```

[9]

Notice this algorithm acts just as a normal Dijkstra's algorithm would run on a graph with nodes. The only thing that is changed is that there is a bounding box around each node to check if surrounding octomap nodes are also free. This is necessary because, even in simulation, the UAVs move very unstably so it is important that there is sufficient room for them to move. In the main method of the code, there are two points, the start and the goal point. In the case of this report, the start node will always been $\{0, 0, 1\}$ because that is the position of the UAV when it takes off using the "mav_hovering_example_with_vi_sensor.launch" file. The goal can be adjusted to whatever goal position is desired. The octomap gets read in the main method. Then, Dijkstra's algorithm is called and the results of the path are printed to an out file which is the text file that will contain all of the waypoints and be fed into the waypoint file. That is it for Dijkstra's

algorithm. A.3

The A* algorithm is designed in the exact same way as Dijkstra's. The only major changes are made when pushing a Front object onto the "front". This time the cost of the node being added to the front has the distance from the current node to the goal node added to the cost. Other than minor changes to adapt to this change, everything is the same. A.5

4.5 Running the code

Once the script is created then it must be put in the right package. It is the package that will run the code. For this script, it is placed in the rotors_gazebo package within the rotors_simulator. The following command allows access to the package.

```
$ cd ~/catkin_ws/src/rotors_simulator/rotors_gazebo
```

In this package the .ot and .bt files of the obstacle course should be added to this package. If they are not already, navigate to the directory where the .ot and .bt files are saved and follow the command below to move them to the rotors_gazebo package.

```
$  mv  obstaclecourse1.ot ~/catkin_ws/src/rotors_simulator/rotors_gazebo
```

OR

```
$  mv obstaclecourse1.bt ~/catkin_ws/src/rotors_simulator/rotors_gazebo
```

Since the script is created, the next step is to add an executable and target link to the script in the CMakeLists text file. CMakeLists.txt is a text file used to compile scripts in ROS. Executables are what the user would call when running the script. For example,the A* script is called AStar_OT when running the script with .ot octomap files or AStar_BT when running the script with .bt octomap files. Target links connect the scripts to the ROS libraries. The edits to the CMakeLists.txt file are attached at A.2. Notice that the CMakeLists.txt file currently assumes the scripts using .ot octomap files are being used. The executables/targets are commented out for the scripts that uses .bt octomap files. If the scripts using .bt octomap files are being used, these lines may be uncommented. Also, include the commented lines related to adding a target for "wp_pub_file.cpp". Those three lines will be needed later. After the CMakeLists.txt file is complete, catkin build the package using the following command.

```
$cd ~/catkin_ws
$ catkin build rotors_gazebo
$ source devel/setup.bash
```

Now, that the code is compiled and the .ot or .bt are in the rotors_gazebo package, the code is ready to execute. The following codes go over how to run the script in a package.
Running A* script with .ot octomap file

```
$cd ~/catkin_ws/src/rotors_simulator/rotors_gazebo
$rosrun rotors_gazebo AStar_OT
```

Running A* script with .bt octomap file

```
$cd ~/catkin_ws/src/rotors_simulator/rotors_gazebo
$rosrun rotors_gazebo AStar_BT
```

Running Dijkstra script with .ot octomap file

```
$cd ~/catkin_ws/src/rotors_simulator/rotors_gazebo
$rosrun rotors_gazebo DJ_OT
```

Running Dijkstra script with .bt octomap file

```
$cd ~/catkin_ws/src/rotors_simulator/rotors_gazebo
$rosrun rotors_gazebo DJ_BT
```

After a few seconds,the script should run and the output will display on the command terminal as shown in figure 6 and figure 7. The values will likely be different.

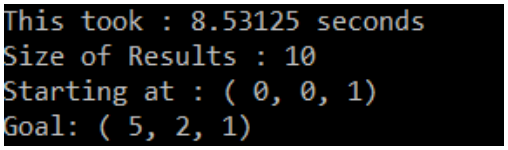

Figure 6: Output from AStar script

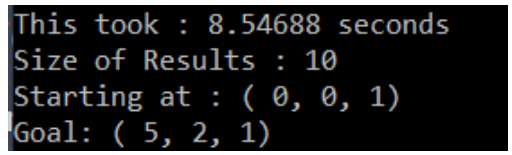

Figure 7: Output from Dijkstra script

4.6 Creating Waypoint Script

The waypoint script is a modified version of the waypoint_publisher and the waypoint_publisher_file.cpp which came with the RotorS package [12]. In this project, the modified script is called wp_pub_file.cpp and can be found at A.7. This was modified from the original scripts because they did not implement the waypoints from the path planning algorithm as intended. The waypoint_publisher only published one waypoint. The waypoint_publisher_file published multiple but it did not run them one by one. It created the trajectory of all the waypoints which caused the drone to crash in the path planning simulation. [12] Thus a mixture of the two waypoint is created through wp_pub_file.cpp A.7. In this file, the waypoints are read from a text file full of waypoints. Each waypoint contains a waiting time or delay, position, and yaw. An example of a waypoint would be shown as below.
0 1 0 1 0 (delay, x, y, z, yaw)
This waypoint has a delay of 0 seconds, x position of 1, y position of 0, z position of 1, and yaw of 0 degrees.

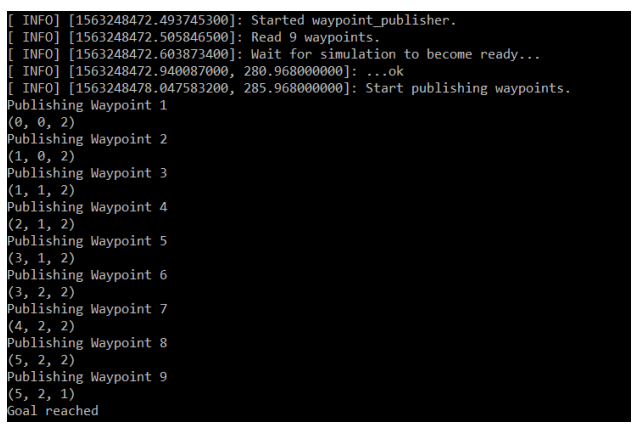

Figure 8: Example of Ouput from Waypoint Script

Once the waypoint vector has all the waypoints from the text file, the file will be closed. Next, the subscriber and publisher ros node are to be created. The subscriber topic would be the IMU topic from the simulation and the publisher topic would be the trajectory messages from the drone in gazebo. Finally, it

goes through every waypoint in the waypoint vector and publishes them one by one. The figure 8 shows an example output of the script.

4.7 Run It All Together

After all the packages are installed and scripts are created, the simulation is ready to be implemented. The following are how to run the path planning algorithm and simulation. First, start a roscore in a new terminal.

```
$ roscore
```

A roscore starts a master node. This is important when running any code that doesn't automatically start a master node itself. Next,the catkin work space must be built again to make sure everything complies correctly. Again to build the catkin work space, follow the commands below.

```
$ cd ~/catkin_ws
$ catkin build rotors_gazebo
$ source devel/setup.bash
```

After everything complies, follow one the commands below to run the script. Notice that whether a .bt or .ot file is being used for the octomap, affects the way that the script must be run. The output should be similar to the figures 6 and 7 however the values will likely be different.
Running A* script with a .ot octomap file

```
$cd ~/catkin_ws/src/rotors_simulator/rotors_gazebo
$rosrun rotors_gazebo AStar_OT
```

Running A* script with a .bt octomap file

```
$cd ~/catkin_ws/src/rotors_simulator/rotors_gazebo
$rosrun rotors_gazebo AStar_BT
```

Running Dijkstra script with a .ot

```
$cd ~/catkin_ws/src/rotors_simulator/rotors_gazebo
$rosrun rotors_gazebo DJ_OT
```

Running Dijkstra script with a .bt

```
$cd ~/catkin_ws/src/rotors_simulator/rotors_gazebo
$rosrun rotors_gazebo DJ_BT
```

After running the algorithm scripts, text files should be created in the current directory. The text files in this report are called Waypoints_AStarOT.txt when running AStarwithOT.cpp, Waypoints_AStarBT.txt when running AStarwithBT.cpp, Waypoints_DJOT.txt when running DijkstrasWithOT.cpp, and Waypoints_DJBT.txt when running DijkstrasWithBT.cpp. After the files are created, run the simulation using roslaunch as shown below.

```
$cd ~/catkin_ws/
$rolaunch rotors_gazebo mav_hovering_example.launch mav_name:=iris world_name:=obstaclecourse1
```

After everything loads and iris is hovering in the environment, run the waypoint script in a new terminal. The following commands show how to run the waypoints depending on which algorithm was run. The output

should be similar to figure 8, however the values will likely be different.
Waypoint Script for A* with .ot octomap files

```
$cd ~/catkin_ws/src/rotors_simulator/rotors_gazebo
$rosrun wp_pub_file Waypoint_AStarOT.txt __ns:=iris
```

Waypoint Script for A* with .bt octomap files

```
$cd ~/catkin_ws/src/rotors_simulator/rotors_gazebo
$rosrun wp_pub_file Waypoint_AStarBT.txt __ns:=iris
```

Waypoint Script for Dijkstra with .ot octomap files

```
$cd ~/catkin_ws/src/rotors_simulator/rotors_gazebo
$rosrun wp_pub_file Waypoint_DJOT.txt __ns:=iris
```

Waypoint Script for Dijkstra with .bt octomap files

```
$cd ~/catkin_ws/src/rotors_simulator/rotors_gazebo
$rosrun wp_pub_file Waypoint_DJBT.txt __ns:=iris
```

The drone should move according to the waypoints in the text file to its destination.

5 Future Work and Conclusion

The focus of this research is to create a 3D path planning script in a 3D environment. This is because there is a lack of documentation for 3D path planning. Most other works use a 2D path planning algorithm. One way this simulation can be used in the future is to add multiple drones to the environment and have the main drone treat the other drones' paths as the obstacles. Another possibility for the future for this simulation is to use a real life drone with the path planning scripts. The drone would need a vi sensor to octomap the environment. Then the script would find the best path using the octomap. One more future idea for this simulation is upgrading it. The simulation is made in Ubuntu 16.04 LTS and ROS Kinetic, which is not the latest version of Ubuntu nor ROS. The project should be upgraded to Ubuntu 18.04 LTS and ROS Melodic. There are many ways that this project could be expanded upon to create more ideas within the UAV community.

Through this research, it is discovered that A* is slightly faster than Dijkstra's algorithm. This isn't the case for all obstacle course configurations, however, in this project there are clear results that show A* typically is faster than Dijkstra's. Another result that was discovered through this research, is that octomap nodes cannot be used solely for navigation. Octomapping an obstacle course only shows the occupied nodes and some of the unoccupied nodes that are within a very close vacinity to those occupied nodes. In order to navigate with a UAV, there must be sufficient space to move and the octomap nodes do not provide enough unoccupied nodes to allow for navigation using solely the map. That is why, in this research, it is assumed that if a node is null, that it is unoccupied. This could be dangerous if a random obstacle occurs in the environment, however, it is assumed in this research that no random obstacles occur. In order to combat this issue, a laser sensor could be attached to the drone. This laser sensor will identify random obstacles if random obstacle avoidance is the goal.

This research is important because 3D path planning does not have a lot of research in the UAV community. It could spark more interest in 3D path planning algorithms in 3D environments in the future. The obstacle course created in this research can be replaced with an actual 3D environment such as a university, hallway or a house in Gazebo. Overall, this project would help future drone research be able to path planning and control drone in different octomapped environments, especially when flying a drone indoors.

References Cited

[1] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng, "Ros: an open-source robot operating system," in *Proc. of the IEEE Intl. Conf. on Robotics and Automation (ICRA) Workshop on Open Source Robotics*, (Kobe, Japan), May 2009.

[2] N. Koenig and A. Howard, "Design and use paradigms for gazebo, an open-source multi-robot simulator," in *In IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 2149–2154, 2004.

[3] "ros-visualization/rviz, howpublished= https://github.com/ros-visualization/rviz."

[4] Z. He and L. Zhao, "The comparison of four uav path planning algorithms based on geometry search algorithm," in *2017 9th International Conference on Intelligent Human-Machine Systems and Cybernetics (IHMSC)*, vol. 2, pp. 33–36, Aug 2017.

[5] D. Rathbun, S. Kragelund, A. Pongpunwattana, and B. Capozzi, "An evolution based path planning algorithm for autonomous motion of a uav through uncertain environments," in *Proceedings. The 21st Digital Avionics Systems Conference*, vol. 2, pp. 8D2–8D2, Oct 2002.

[6] M. Radmanesh, M. Kumar, P. H. Guentert, and M. Sarim, "Overview of path planning and obstacle avoidance algorithms for uavs: A comparative study," *Unmanned Systems*, vol. 6, pp. 1–24, 04 2018.

[7] Y. Zhao, Z. Zheng, and Y. Liu, "Survey on computational-intelligence-based uav path planning," *Knowledge-Based Systems*, vol. 158, pp. 54 – 64, 2018.

[8] Hoi-Shan Lin, Jing Xiao, and Z. Michalewicz, "Evolutionary navigator for a mobile robot," in *Proceedings of the 1994 IEEE International Conference on Robotics and Automation*, pp. 2199–2204 vol.3, May 1994.

[9] J. F. R. Crespo, "SLAM_AND_PATH_PLANNING_ALGORITHMS, howpublished = https://github.com/jfrascon/slam_and_path_planning_algorithms/tree/master/08-path_planning/code," Oct 2017.

[10] K. McGuire, G. de Croon, C. De Wagter, K. Tuyls, and H. Kappen, "Efficient optical flow and stereo vision for velocity estimation and obstacle avoidance on an autonomous pocket drone," *IEEE Robotics and Automation Letters*, vol. 2, pp. 1070–1076, April 2017.

[11] B. Pfaff, "Overwhelming the saa system of delivery uavs by drone swarming," Master's thesis, Wright State University, 2016.

[12] F. Furrer, M. Burri, M. Achtelik, and R. Siegwart, *Robot Operating System (ROS): The Complete Reference (Volume 1)*, ch. RotorS—A Modular Gazebo MAV Simulator Framework, pp. 595–625. Cham: Springer International Publishing, 2016.

[13] S. Cooley, M. Wojciakowski, C. Loewen, R. Turner, V. Silva, L. QIN, N. Srinivas, J. Sejkora, S. Hanselman, M. Satran, and et al., "Install windows subsystem for linux (wsl) on windows 10," Jul 2018.

[14] R. Mainer, "rmainer/latex-listings-powershell, howpublished= https://github.com/rmainer/latex-listings-powershell," Aug 2018.

[15] "Get Ubuntu 16.04 LTS, howpublished = https://www.microsoft.com/en-us/p/ubuntu-1604-lts/9pjn388hp8c9?activetab=pivot:overviewtab," May 2018.

[16] "Ubuntu install of ROS Kinetic, howpublished= http://wiki.ros.org/kinetic/Installation/Ubuntu," Jul 2017.

[17] "Install Gazebo using Ubuntu packages, howpublished = http://gazebosim.org/tutorials?tut=install_ubuntu&cat=install," 2014.

[18] "Installing gazebo_ros_pkgs (ROS 1), howpublished = http://gazebosim.org/tutorials?tut=ros_installing," 2014.

[19] "VcXsrv Windows X Server, howpublished = https://sourceforge.net/projects/vcxsrv/," Jun 2019.

[20] "MAVROS (MAVLink on ROS), howpublished = https://dev.px4.io/v1.9.0/en/ros/mavros_installation.html."

[21] Marguedas, "Installation/Source, howpublished = http://wiki.ros.org/installation/source," Feb 2018.

[22] "OctoMap Models with ROS, howpublished = https://dev.px4.io/v1.9.0/en/simulation/gazebo_octomap.html#octomap-3d-models-with-rosgazebo."

[23] "Octovis, howpublished = http://wiki.ros.org/octovis."

[24] "A* Search Algorithm, howpublished = https://www.101computing.net/a-star-search-algorithm/," Feb 2018.

[25] "How to implement Min Heap using STL?, howpublished = https://www.geeksforgeeks.org/implement-min-heap-using-stl/," Oct 2018.

A Appendices

A.1 Rotors Gazebo Package.xml

A.2 Rotors Gazebo CMakeLists.txt

A.3 Dijkstra Script with OT file

A.4 Dijkstra Script with BT file

A.5 A* Script with OT file

A.6 A* Script with BT file

A.7 Waypoint Script