Abstract

Area inspection and analysis can be a time-consuming and costly operation. Inspection of high-rise buildings poses security challenges while rail inspections, in remote areas, entail long-distance travel and challenging terrain. Using technologically advanced drones, algorithms and manually operated analytical tools, this operation can be automated to provide an economically competitive advantage. We present an early stage of such a program with the use of modern simulation technologies and a user-oriented web interface. The Autonomous Obstacle Avoidance algorithm can be effectively used as part of a broader framework, while the online platform offers market-rivaled analysis tools.

By utilizing stereo cameras we can interpret a 3D environment and navigate it safely. During this flight we collect data and communicate with our web-platform. Our web-platform presents captured data and offers a variety of research tools, such as displaying flight images, trajectory animation and comparison between scheduled and simulated flight. This results in a completely operational product capable of reviewing, evaluating and preparing a new aerial operation.

Acknowledgements

When stretching as far out of our comfort zone as we did in this thesis it is obvious that we would not have come as far as we have without some serious help and support. We would like to thank the entire team at Nordic Unamanned and Staaker for helping us define, plan and develop a bachelor thesis as complex as this one. Nordic Unmanned gave us the use-case and motivation we needed to work through. Staaker has helped us with the software-related challenges that we were bound to meet during this project.

We seldom write large reports like this so we needed support from the school in organizing this thesis. We would like to extend our gratitude to Erlend Tøssebro for helping us stay on track and write a report for this complex product.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Our business is not about drones. The drones we use are simply a tool to convey a sensor into a desired position, enabling us to collect data from locations and positions that would not be feasible by other means.[20] Nordic Unmanned

Nordic Unmanned (NU) is one of Europe's leading suppliers of Unmanned Aerial Vehicle (UAV) services, progressing towards a goal of becoming the preferred provider of unmanned systems and services within Energy, Maritime and Governmental segments.[21] They deliver several types of industrial RPAS (Remotely Piloted Aircraft Services), such as inspections, photogrammetry, thermographic imaging and 3D-modeling of terrain and constructions.

This thesis is also written on the account of Avium, a student organization newly founded by the four authors of this bachelor's project. Avium focuses on the development of software aimed at solving different tasks with the end goal of competing in international drone competitions. This project has an important role in the early experience of the four founders of Avium and our journey diving into the technological landscape of drones. NU became a supporting advocate in Avium's early stages, eventually evolving to the collaboration that produced this bachelor's project.

1.1 Initial project description

The agreement with Nordic Unmanned states: "The intent of this project is to combine four students and two bachelor's theses' to be able to deliver a larger, more complete solution. We will be trying to solve a specific problem that can be solved more efficiently by using autonomous drones and intelligent image interpretation." So this is thus a two-part bachelor project where this group will handle the first part involving drone navigation, obstacle avoidance and data collection.

We will focus on achieving custom drone navigating in aerial space autonomously by employing a simulator. Using a simulator is a good method of ensuring that a drone is controlled, limiting its potential to crash and be destroyed. We will able to capture data as a drone would in real life, and send that information to Group 2 to present. This will be the first stage of connection between the two parts.

The second part of this two-part project will seek to create a complete UI that will be able to handle the presentation of various point clouds. This part is connected to the first part by planning a route, and presenting the actual data simulated by us.

This thesis is also written on the account of our newly started student organization Avium. Avium focuses on development of software for to solve different tasks and compete in international

drone competitions. This project has an important role in the early experience of the four founders of Avium and our journey diving into the technological landscape of drones.

1.2 Redefining the thesis for simulation

Originally the plan for this thesis was to create a product that we could use for a physical demo. We had a self-built drone at hand that we could fix and use. If that plan did not seem to work out, our backup path was to get support from Nordic Unmanned with hardware challenges and use one of their drones. However time and money became obstacles rather than advantages. As students we were severely limited by our budget, and simply did not have the funds to get the compatible drone or sensors, as LiDAR sensors are expensive. Nordic Unmanned had an untimely surge of activity at this time. This restricted their time capacity so that we simply did not have time to develop a compatible drone in addition to the software we are creating.

As we delved deeper into the time and capital problems we learned a lot about the landscape of drone technology development. We realized that even if we had the resources, using a physical drone for development is nearly futile. Virtually all development regarding autonomous vehicles is simulated. Upon recognizing the limits we contacted Nordic Unmanned and redefined some of our initial plans regarding physical flight. We came to an understanding that this thesis will solely focus on developing an obstacle avoidance algorithm in the AirSim simulator. This will be further discussed in the discussion chapter at the end of this thesis.

1.3 Outline

The outline of this thesis contains chapter one that contains an introduction to the project in general. Thereafter a chapter that lists and describes the technologies used in the thesis. In chapter three we explain the advantages of simulation, how and why we used our designated simulation, AirSim. Chapter four is a explanation on how the main project was completed. The extra ease-of-use modules were clarified in chapter five. Next follows chapter six in were we delve into some ideas for further development. Finally in chapter seven the finished product and development experience is discussed and concluded.

Chapter 2

Technological Background

Most of the technologies used in this study are familiar to us already from our studies at Stavanger University. These include Python, Git and GitHub. We have had to learn additional technologies and theory during this project to achieve a satisfactory result. This chapter sums up the theory and technologies we used to build our study.

2.1 Light Detection and Ranging (LiDAR)

This projects calls upon our group to collect data that describes the drones surroundings[4]. This data is going to be used for visualization in Group 2s platform. Our data structure of choice will be LiDAR point clouds. A physical LiDAR scanner uses pulses of light to determine the distance to any given point by measuring the time from sending the light pulse to receiving its reflection. Each point is calculated from the drones position, the sensors angle, the time of the pulse and the speed of light. A collection of such points is called a point cloud and is useful for analysis and image localization.

2.2 Python

Python is an interpreted, high-level, object-oriented programming language [1]. Pythons user friendliness is what has made it the second most used programming language of 2019 [5]. Python has recently been at the forefront in AI and Machine Learning work. In this thesis we will use Python for decision making with our obstacle avoidance algorithm.

Our preference derives from a variety of factors. We have studied Python in school and are looking to expand on our knowledge while also using it to control our drone. It is critical to have clear and readable code when embarking on a large project. Python provides the developer with an easy-to-understand syntax without unnecessary clutter. It will benefit both us during production and potential readers looking to expand the project further. Python connects seamlessly with the AirSim API[12] - which we will dive deeper into later. For a straight forward script like this algorithm, Python was the obvious choice.

Numpy

In this study, we are going to want to process a lot of image and LiDAR data, so we need a math math tool. Numpy is a library that implements functions which are easy to use in Python[6].

Numpy is capable of separating, reshaping, or concatenating matrices. These matrices will in our case be primarily image data. Numpy employs specialized computer science to produce the most efficient matrix transformations possible. We chose Numpy because it is the single most used library in Python[7] for heavy matrix and array operations.

ArgParse

There was a need to quickly adjust the values or to have the data collected during testing to be stored.To address this, we utilized the ArgParse library to quickly read and interpret data from the command line interface.

2.3 Git & GitHub

Git is the leading technology used for version control in programming projects[8]. It gives you all the resources you need to track changes in source code and work seamlessly with other developers. Both version tracking and collaboration tools are important to us when working together on a project of this scale. We get the help we need to restore earlier written code in case we make an mistake, and we get more freedom to work independently through the whole thesis.

GitHub is the "world's leading software development platform"[9] and our git platform of choice. Hours and hours spent on GitHub in earlier projects give us all the familiarity we need when developing. We use some of the GitHub tools to track changes and workflow. Issues help us track bugs and assign tasks. We transferred much of our issue monitoring and workflow control to *Trello* on this project.

2.4 Trello

Trello is a web-based list program to coordinate any kind of operation[10]. The key feature is a powerful Kanban board. The combination of an easy-to-use framework with all the functionality you may like made this an essential process planning tool for us. We have used the conventional TODO, IN Development, Finished workflow. We had two different backlogs for the project and the report by being able to break each level into two tabs in Trello. We have learned from this by still being able to pick the next goal without having to address it in any way. We could then tailor the workload more easily to our individual schedules.

2.5 Unreal Engine

Unreal Engine is a platform for developing 3D games developed by Epic Games[11]. Originally developed to create first-person shooters, it has grown into a medium that incorporates realistic physics, lighting and graphics. Because of this, there are now hundreds of uses-cases that are not explicitly related to game development[19]. One of these applications, and the most interesting to us, is simulation. Unreal Engine gives you the ability to create your own maps and use their physical engine to simulate real-life situations.

2.6 AirSim

AirSim is an open source project[13] funded and maintained by Microsoft. The AirSim team has built a realistic framework on top of Unreal Engine and has revealed all the APIs that we

need to program a UAV or a car in their simulation. Additional APIs allow us to collect data from the drone and its sensors. We can also describe our own drone by changing what sensors we want to use and what kind of drone we want to use. In the dedicated Simulation chapter, we will further explore AirSim.

Chapter 3

Simulation

There are plenty of reasons why using simulation software is helpful, if not essential, whenever we are working with an expensive physical product like a drone. In this chapter we will explore why and how we used simulation to our advantage. We will contemplate what advantages and challenges we faced.

3.1 What is AirSim?

As we mentioned in the earlier AirSim explanation, AirSim is a simulation project provided by Microsoft and based on Unreal Engine and C++. AirSim is an open source [13] project. This gives us the accessibility to track down the wanted functionality whenever it is not apparently available in the documentation. Additionally this is helpful for Microsoft, as users all over the world can contribute to the AirSim source code whenever they create a new piece of software that might be helpful for other users. Unreal Engine is also a project with open source code, even though, strictly, it is not open sourced.Nevertheless, the open source code is appropriate for this purpose. We can monitor how different features are implemented in the AirSim code and reproduce new features that we need by writing similar code and connecting it to Unreal Engine as AirSim developers have done. The combination of both the platform(AirSim) and the engine(Unreal Engine) being available to the public is the most prominent explanation why we have chosen to use AirSim for this study.

We chose AirSim before we realized, like mentioned in the previous section, that we had to abandon our plans of using a physical drone. AirSim provides APIs for direct two-way communication with all the largest providers of consumer flight controllers. For example our self-built drone was controlled from a PixHawk PX4. AirSim included functions for sending commands to this flight controller as well as receiving flight data from the PixHawk. By using these developers might be able to use the same code for flying in AirSim and flying the physical drone. Additionally the data from the physical drone could have been visualized, or otherwise interpreted in AirSim.

3.1.1 Why AirSim?

There were several reasons why we chose to use AirSim over other simulation technologies, but it was crucial for us take a look at the different forms of simulations out there. The table 3.13.1 offers a brief overview of a variety simulators for drone flight.

Simulators	**Availability**	**Company**
Dji Simulator	Free trial/Enterprise	Dji
Airsim	Open Source	Microsoft
Matlab and Simulink	License needed	Mathworks

Table 3.1: Overview for simulations

Dji Phantom Simulator

Let us first take a look at the Dji Phantom simulator. The simulator is from a reputable brand that is well known in the drone community. The Dji simulator supports several different point of views, three flight modes and "Auhtentic Flying Experience"[16]. The models supported are specific models from Djis line up, including mavic series and phantom, but also a couple more. However this simulator is mainly targeted at training pilots in different modules and would not suit our demand since this means that it only has a couple "modules" and you need to have a controller.

MATLAB

MATLAB is a research tool that we have come across earlier in the studies of computer engineering on this university and it is a versatile tool, particularly with the use of simulink. This will enable you to build drones, with different setups like octocopter anything to your liking. Everything has to be built almost from scratch, which means there would be a ton of work setting it up. This would mean we had to use a lot of time to design the model, motors and controller for our drone and program most of it our self.

Summary

The reason we chose exactly *AirSim* was because it suited our goal with this project. We did not want to use our time and effort of designing the drone and everything surrounding it. Therefore the MATLAB approach would take a lot of time and the whole project would have had another focus. Ease of use versus MATLAB and the opportunities we had with AirSim and the Unreal Editor made us certain that AirSim was a good pick.

3.2 The AirSim coordinate system

Before moving the drone we need to understand the AirSim coordinate system. AirSim uses a static NED (North-East-Down)[22] world reference system. We call this a static system because once the simulator is started the coordinate system will never rotate or transform. The coordinate system begins with zero-values at the drones spawning point. This means that when the drone rotates or shifts its position the drones position is changing, whilst the system stays still. In some situations, like LiDAR scans and and saving the drones position, this type of static system is easily use-able. For other calculations, such as finding the angle between the drone and a goal point, a coordinate system where the drone is always in the origin would be easier to use.

A NED system defines which direction is positive for each axis. In the North-East-Down system positive X-axis values represent north, whilst the Y axis represents east and positive Z-axis values are pointing down. The most impactful change when using this coordinate system is that we need to remember that the positive Z-axis is downwards. Therefore we need to use negative Z values for every move we want to make over the ground. We will also see later that we

need to invert the Z-axis when exporting data so that the data is compatible with other systems that use positive Z-values as up.

NED systems are conventional specifically for flying objects like missiles, rockets and in this case drones. For geography a ENU (East-North-Up) system is used.

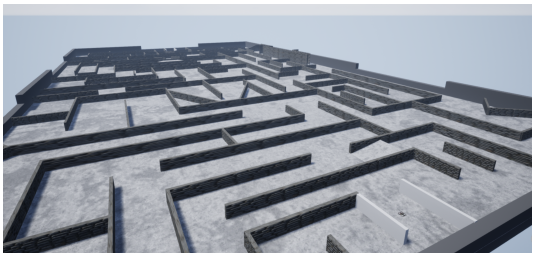
3.3 AirSim maps

A significant part of what we want to simulate is that we can test our algorithm in a variety of different environments. Since AirSim is built on Unreal Engine we are able to use any Unreal Engine map for flying. Within this section, we will take a look at some of the binary maps that AirSim has already compiled and see how we we've created our own map to suit our exact needs.

3.3.1 Binaries

Binary files are files that are stored in the binary format; zeroes and ones. Files like these are readable by a computed but not readable by humans. These files are created by the compiler when you compile a program in many different programming languages. Unreal Engine is able to create binary files that can be directly run by Windows because they are compiled to an *.exe* file. These *.exe* files already include all the AirSim logic like the drone physics and the API calls. When running the application from an *.exe* file it opens a small window ready for API commands and loads the compiled map.

By using binaries we bypass a lot of heavy compile operations every time we need to test the algorithm. If we had not precompiled the binaries we would need to compile the entire AirSim C++ code and open the Unreal Engine editor every time we wanted to use AirSim. AirSim provides some different precompiled maps[15] that we used heavily during the development process. The ones we used the most are pictured here:



(a) "Simple Maze" map. Drone starting point in lower right corner

(b) "Neighborhood" map

Figure 3.2: City map with moving cars and humans

From these three images we can imagine how many different use cases we can test from a single simulator. We tested early versions of the algorithm on the simple maze map against flat walls, a flat ground and sharp, distinct edges and turns. When the algorithm worked in this simplified environment we moved on to the neighborhood to test against trees, signposts, cars, hedges and houses. This was a larger challenge and needed some fine-tuning of the algorithm.

Finally we tested our algorithm on the City map. This is an even more realistic environment for using this thesis. If this algorithm was used for inspection in a city it would be helpful to already have tested the algorithm in a similar territory. This map includes several moving cars and humans at the ground level. The ability to add these moving objects is a strength of using a simulator based on Unreal Engine.

3.3.2 Building our own map

To begin making our own map we had to make sure AirSim is installed and the plugins from this are present when making a project. We however started with the basic project provided from installing and generating a block environment from the AirSim documents. This file should be present in AirSim_Unreal_Environments_Blocks and named Blocks.uproject and will open the Unreal Editor. The base map or "testcourse" had a couple of blocks of the same size all around the drones starting point in a circle. It is also possible to import high definition assets or maps from the Unreal marketplace. The figure 3.3 down below shows the drone and how the map looked in this original form.

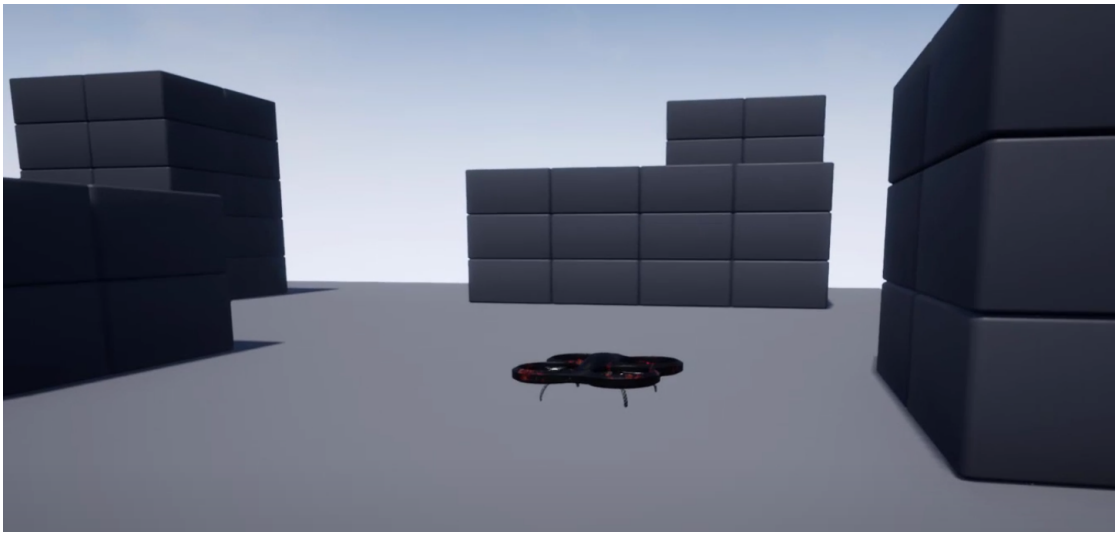


Figure 3.3: Original blocks

It is also possible to open the blocks file in visual studio 2017 or newer to get a set of debugging features. The Unreal Editor was how we were able to customize the existing map to what we needed. The figure 3.4 displays the GUI that is present with Unreal Editor. The editor had tons of features for different kinds of tasks, but in our case it was used to make an obstacle course for the simulated drone. In the beginning the base map was used so that we could try to avoid objects in general. From there the next step was to build something more complicated with objects that can be seen on the left side of figure 3.4 were used as the base figures. These create basic a "StaticMeshActor" of the geometric form, and are instances of the "UStaticMesh" and the information that is needed in this context is that it is a static mesh and will not deform and alter its appearance if something interacts with it, however this could be changed if needed be.

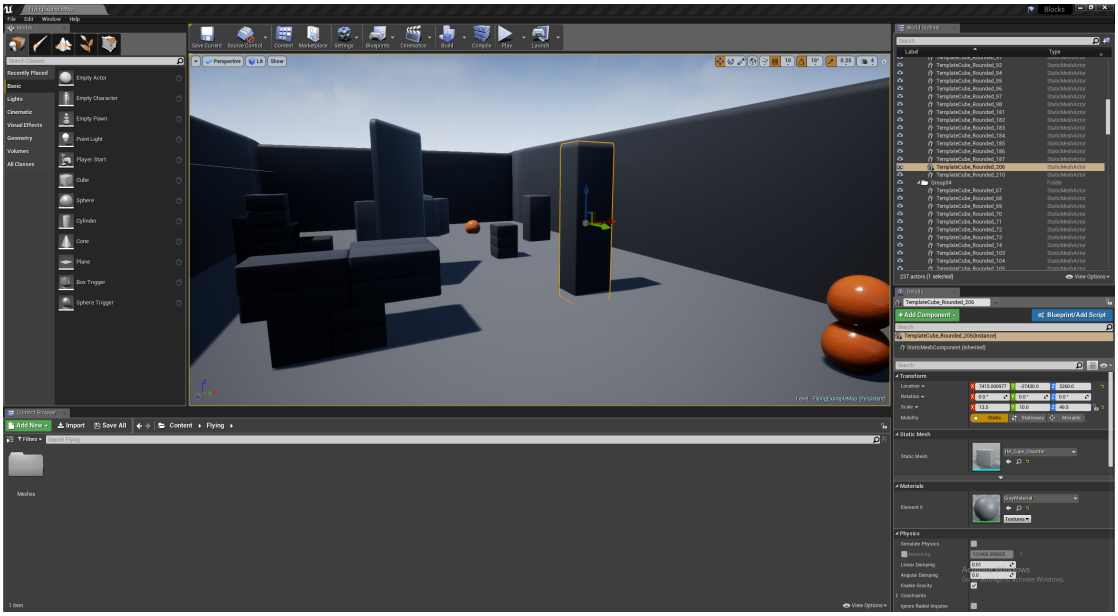


Figure 3.4: Unreal Editor view

Once a actor has been placed it is possible modify it, and this could be done by pressing the different icons on the top of the work area or the respective keyboard shortcuts: "**W**", "**E**" and "**R**". This made it possible to make other types of structures and scale, rotate or move them to different positions. That is how we made the more complex one like towers or hallways.

Since the drone should be able to somewhat follow a path or find its way to a point the map is designed as a long tunnel of sorts. This means that the drone cannot just take off in a direction with no obstacles and just continue to fly. The tunnel consist of many different kinds of objects that was made with the Unreal Editor. The player actor or drone starts in the far left of the map by the wall, as seen on the 3.5 below.

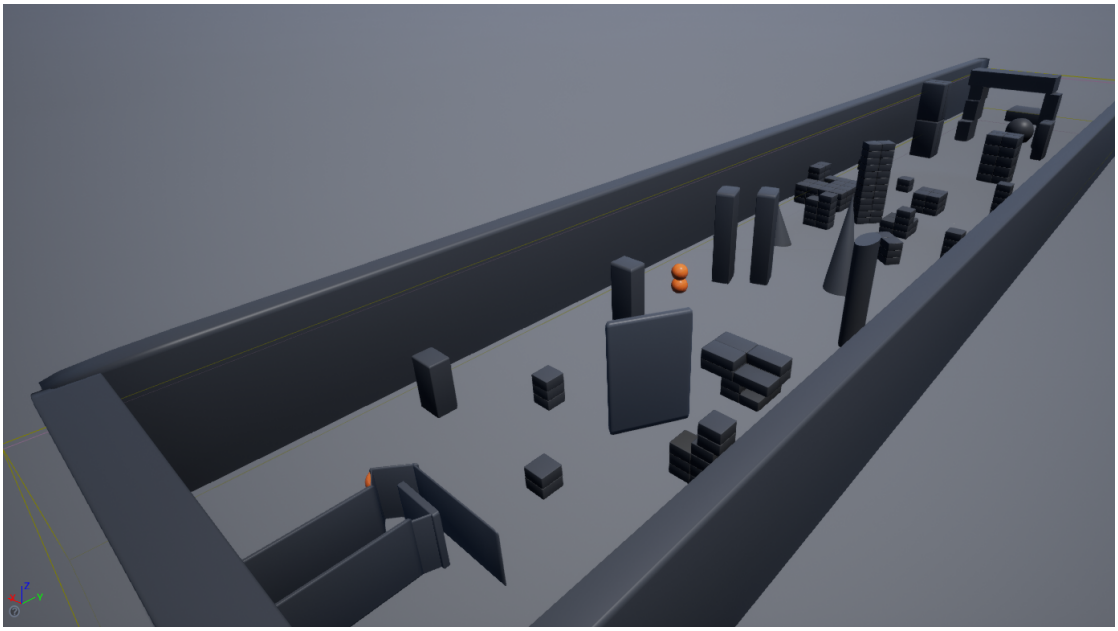


Figure 3.5: Tunnel map

The drone would hopefully get to the end of the obstacle course, but the map should also give a challenge so the algorithm has to put in some work. It was not supposed to look photo realistic, however it is possible tom implement already mapped real life areas or make some with other assets and more complicated stuff like foliage. The map was a good challenge because there were many different kinds of obstacles for the drone to overcome, and somewhat long, but it is the same layout every time. For further challenging the algorithm, a randomized map layout could be the next step. This could be done by making a script in C++ and implement it into the .uproject-file.

3.4 API calls

Our only connection with the AirSim simulator is through the provided Python APIs. An API (Application Programming Interface) can, like in our case, be an interface that provides access points to a larger program through documented function calls. For AirSim that means functions for getting information from the simulator or sending commands to the vehicle. In this section we will look at the setup needed for taking off, some API calls to statically move the drone to any position and a list of API calls that are used in this thesis.

3.4.1 Taking off

First of all we needed to learn how to use the basic drone APIs provided by AirSim. This was pretty well documented and used, therefore we could use both the documentation and the examples to create some working code for this.

To get AirSim to work with our Python files we need to setup the correct folder structure. In the AirSim git[13] there is a folder called PythonClient. This folder contains all the APIs that are available for use directly in Python, as long as these are in the same workspace. The

folder contains files with utils, types and the client itself. All of these are originally written in C++, but the Python client has been created as a link between the C++ source code and the easy-to-use Python language. In the *client.py* file the Python code is able to send messages to the compiled C++ program which executes the functions in the simulator.

Luckily for us, all of this happens under the hood of the AirSim API. Thus we can do the simple following setup to get the drone to take off:

```
1  import setup_path
2  import airsim
3
4  client = airsim.MultirotorClient()
5  client.confirmConnection()
6  client.enableApiControl(True)
7  client.armDisarm(True)
8
9  print("Taking off")
10 client.takeoffAsync().join()
```

In the above code we are doing the general setup that has to be done in every Python file that is supposed to be the connection with AirSim.

Line 1 & 2: These lines simply import all the previously explained AirSim functionality. To use the APIs that AirSim provides the current Python file needs to be able to access the Python files in the AirSim directory. To make the import a bit easier we are firstly importing the *setup_path.py* file. This file is provided by the AirSim team and the only job this file does is to locate the AirSim installation and use the correct version. *setup_path.py* has to be present in the same directory as the file we are importing it in. This saves us the hassle of having to copy the entire AirSim folder every time we want to work in a new folder. After the path is set up we import the *airsim* folder itself. This is the aforementioned folder that contains the Python files similar to those in the *Python Client* folder in the root AirSim git[13]. Later in this code snippet we use this import to setup the connection with AirSim.

Line 4 - 7: AirSim has the functionality for both cars and multirotor drones. On line 4 we use the imported airsim object to declare our drone with the name *client*. This instantiates an object with all the functionality of the multirotor API from AirSim. We will use this client object whenever referring to the drone for the rest of the file. The rest of the lines 5-7 are some statuses that need to be set for the drone to be usable. On line 5 we confirm the connection, this line would have thrown an error if the connection was not setup. On line 6 we tell AirSim that we want to be able to control our multirotor through API calls to the client object. Real life, physical drones have two safety states; armed and disarmed. The *.armDisarm()* function takes in a boolean and sets the arm/disarm accordingly. (True = armed, False = disarmed). This functionality will not be very useful for us, but to replicate a real life drone as close as possible this is a nice addition in funcionality.

Line 9 & 10: On line 9 we just print a message for the user telling them that the drone is now taking off. On line 10 we use another of the AirSim provided API calls. This is the first point at which we saw some drone movement. Most of the API calls in AirSim have recently been moved from synchronous function calls to being able to use the functions asynchronously[1]. This means that there was earlier a *.takeoff()* function that is now deprecated. Therefore we have to use the *.takeoffAsync()* function. However in most of our programs we want the take-off to be synchronous. This can then be done by using the *.join()* function. This works with all

[1]Synchrounous functions will stop the rest of the program from running. This means that all AirSim API calls that are synchronous pauses the rest of the code until the command is finished in the simulator. The power of asynchronous functions becomes apparent when we want to do several things at once. For example we want to move the drone and calculate our next move at the same time.

API call	Example file	Description
.moveByVelocityZAsync()	multirotor/box.py	Moves by x and y velocities at a fixed height z.
.simGetImage()	multirotor/navigate.py	Gets a single image from the simulator.
.simGetImages()	multirotor/multi_agent_drone.py	Gets multiple images from the simulator.
.simGetVehiclePose()	multirotor/navigate.py	Gets the orientation and position of the drone.

Table 3.2: API calls with example files

of the AirSim API calls. *.join()* creates a queue in which it puts all functions that it is called on. When a function that is queued with *.join()* is finished, the queue is checked and the next function is ran. With this functionality we can easily define when we want to use asynchronous or synchronous functions. Read more about the *.join()* function in Appendix A.2.

3.4.2 Statically moving the drone

After we have successfully taken off we can use one of the many AirSim APIs made for moving a multirotor around. This section could have been several pages long if we were to discuss all the available movement APIs. Therefore we are only going to go through the ones that we will not explain later, and refer to the AirSim movement documentation [14] for the other APIs.

```
1 client.moveToPositionAsync(-5, -5, -5, 10).join();
2 client.moveToPositionAsync(-10, -5, -10, 10).join();
3 client.moveToPositionAsync(-10, -10, -10, 10).join();
4 client.moveToPositionAsync(-10, -10, -1, 10).join();
```

A good start is to use one of the highest level APIs that AirSim provides. *moveToPositionAsync()* does exactly what the name implies, it moves the drone in the shortest line possible to the specified point. The point in question is the first three parameters of the function. The last of the four parameters is the speed we want the drone to fly at, defined in m/s. On line 1 we tell the drone to move from its current position to the point (x: -5, y: -5, z:-5) at a speed of 10 m/s. Line two moves to a new point, (x: -10, y: -5, z: -10) at the same speed. The last two lines are similar, but to different points in the 3D environment. Remember that we do not want the drone to get new commands while it is moving, which is why we are adding the *.join()* function call to every command. Also remember that the Z-value has to be negative to represent values that are over the ground.

3.4.3 API calls used in this thesis

All these API calls will be further described when we use them in the next two chapters. However it can be helpful to read through the AirSim documentation for each of these and look at the Python AirSim example files in their GitHub repository [12]. The table contains the filename of a file in the AirSim Python examples that utilizes the API call.

Chapter 4

Collision Avoidance

Collision avoidance for drones is a broad and advanced field. For the purpose of this thesis we have defined our task as creating an algorithm that navigates the drone between two given points while flying clear of any obstacles on the way. This algorithm should also be able to follow a path of several points. To achieve this the drone has to go through several steps before finally moving. We have to somehow interpret our surroundings, then calculate the optimal move to make, before finally moving the drone in the desired direction. All these steps have to be done every single tick of the algorithm.

4.1 Introduction

Before diving into creating a complex algorithm it is helpful to have an overview of the file setup and the flow of the algorithm.

4.1.1 Flow Chart

The following flow chart summarizes the flow of the algorithm. When explaining the algorithm in the rest of this chapter it can be helpful to look back at this flowchart and realize how each module is used.

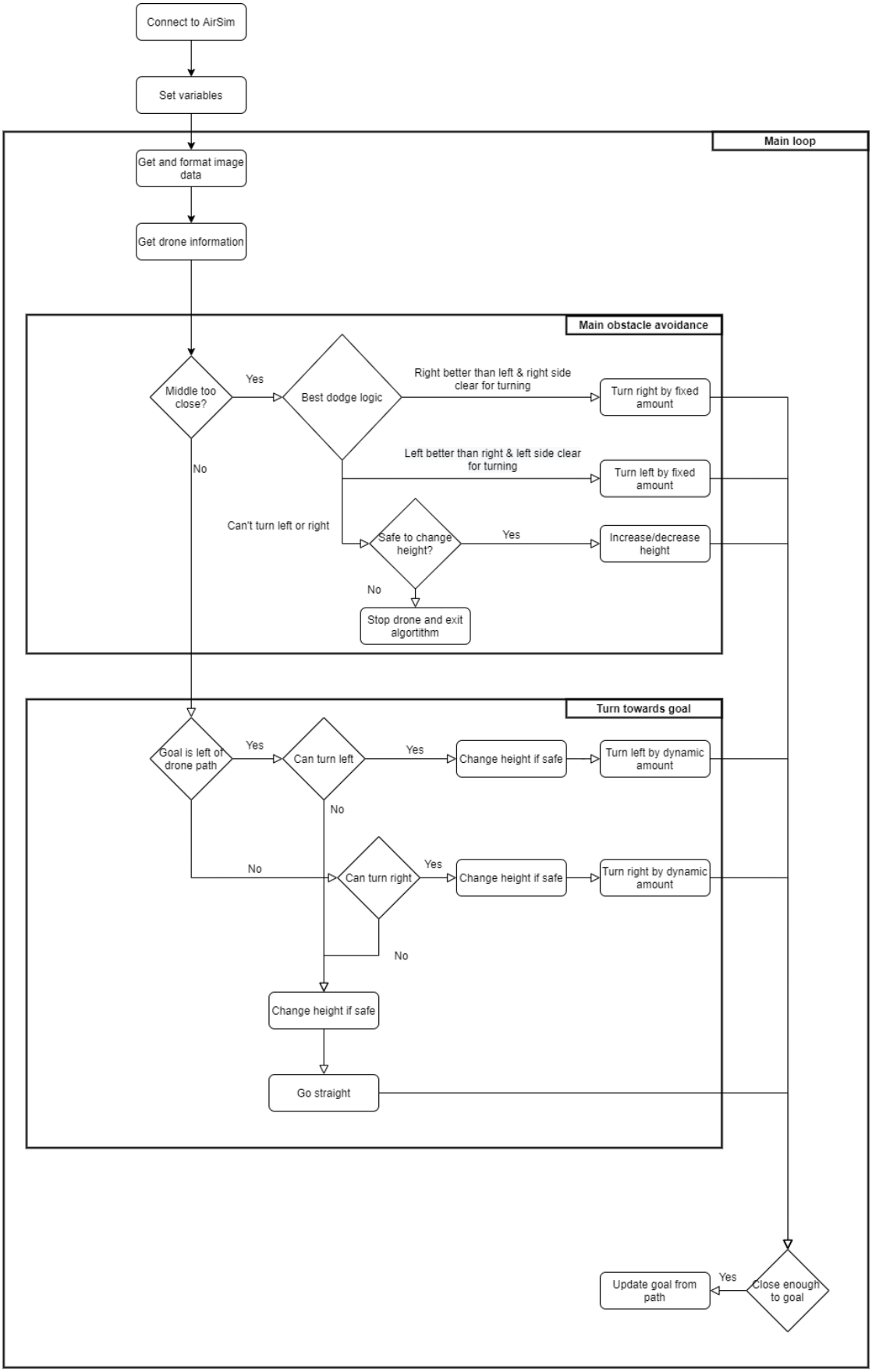


Figure 4.1: Flow chart

From the flow chart we can discern three main modules. The largest module is called *Main loop* this is the loop that runs most of the algorithm every tick. Everything inside this module will be looped. In the bottom right is the last step for each loop of the algorithm, checking if we should update our goal. After this step the algorithm is looped back to the top of the *Main loop*.

In this section we will systematically explain each step of the algorithm in the same flow as in the flow chart. Firstly how to dodge obstacles, secondly how to turn towards our goal and lastly how these work together to achieve safe navigation.

4.1.2 The File Setup

Official AirSim files use a particular setup for Python files that we also used in our algorithm. This setup provides several benefits that we shall discuss in this section. In AirSim there are some operations that should be done every time the algorithm is run. For example the script has to safely connect to and disconnect from AirSim every time. To further explain this we can refer to this shell code:

```
class AirSimClass:
    def __init__(self):
        """ Setup parameters and take off """
        self.client = airsim.MultirotorClient()
        self.client.confirmConnection()
        self.client.enableApiControl(True)
        self.client.takeoffAsync().join()

    def execute(self):
        """ Main code """
        # Algorithm logic

    def stop(self):
        """ Safely reset and release AirSim control """
        airsim.wait_key('Press any key to reset to original state')
        self.client.armDisarm(False)
        self.client.reset()
        self.client.enableApiControl(False)
        print("Done!\n")

# Main
if __name__ == "__main__":
    obj = AirSimClass()
    try:
        obj.execute()
    finally:
        obj.stop()
```

This shell code will be the same for all AirSim programs we create. It provides a standard setup and connection for the AirSim connection in the *__init__(self)* method. This has been explained in 3.4. We save this to the *self* object so that we can access it later, when we want to command the drone. After we have created the *self.client* object, which represents the drone connection, we can start using it. The following three lines confirm the connection, and throw an error if the connection is not correctly set, before making the drone take off. The *__init__(self)* function is automatically called when AirSimClass is instantiated.[17] This is helpful as we always want this code to run when starting the program.

At the bottom of the code snippet the code is run. First we instantiate the entire *AirSimClass* and therefore the *__init__(self)* method is run. When the object is ready we execute the rest of the code, firstly we run the main method of this program called *execute(self)* before finally cleanly

stopping the code. By using a *try* block like this we ensure that the *stop(self)* method is run. Even though the program crashes in *execute(self)* the *stop(self)* will be fully executed and control will be released from AirSim. From the code snippet we can see that in the *stop(self)* method we do the opposite of what is done in the *__init__(self)* method. This tells AirSim that it should no more be waiting for input from this script, as it is stopped or has crashed. By doing this we ensure that AirSim does not get overloaded and crashes after several scripts have been run.

The *execute(self)* method contains the actual code we want to run in the script, and will be further used and made example of when we walk through our own algorithm. When we are discussing a single main file, like our algorithm, all the benefits of a setup like this might not be apparent. However one will quickly learn to use this tool when developing or further evolving an AirSim program. To us this setup mainly provided two things; the ability to run programs without having AirSim crash because of too many unclosed connections, like mentioned above, and the ability to rapidly create new, short scripts to test code. We would be able to create a new file and paste this setup before simply adding our new code snippet in the *execute(self)* method. This is a powerful tool and work method that anyone that works with projects similar to this should be using.

4.2 Recording the surroundings

The first step the drone needs to go through to be able to move is to collect and format some data that gives us some crucial information about the surrounding environment. There are several different types of data, and different ways to capture and programmatically interpret the surroundings of a drone. In this section we will look at the advantages and disadvantages of using LiDAR scans and front-facing stereo cameras for interpretation. This section explains some of the first steps in figure 4.1, *Get and format image data* and *Get drone information*.

4.2.1 LiDAR Sensors

LiDAR (Light Detection and Ranging) sensors are used to create 3D visualizations of environments. The LiDAR sensor checks the range from the sensor to a given point by using a timed laser. This technology is often used in self-driving cars and to inspect and review 3D data of a specified area. When using LiDAR data for navigation we are able to look at the 3D data and determine if a given path has enough space for the drone. In this case we can utilize established path finding algorithms to find possible paths that lead us toward the goal. When we have this path we can calculate the range from the drone to the LiDAR scanned environment and adjust the path so there are safe distances on every side of the drone. With this approach we can calculate a route and check if that route is still the optimal route every tick.

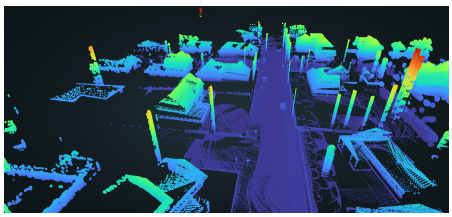


Figure 4.2: Visualization of a LiDAR point cloud of a neighborhood.

4.2.2 Stereo Cameras

Stereo cameras can be used to perceive depth in an image by using two separate images and using some math. This is how our eyes are able to perceive depth when we look around. By using two separate cameras, like our eyes, and knowing the distance between them, we can calculate the range from the middle of the two cameras to every single point in the image. This technology

is easy and cheap to use on physical drones, and often used in addition to LiDAR sensors for autonomous navigation. AirSim uses Unreal Engine to simulate a gray-scale depth perception which we can translate to depth measurements in meters.

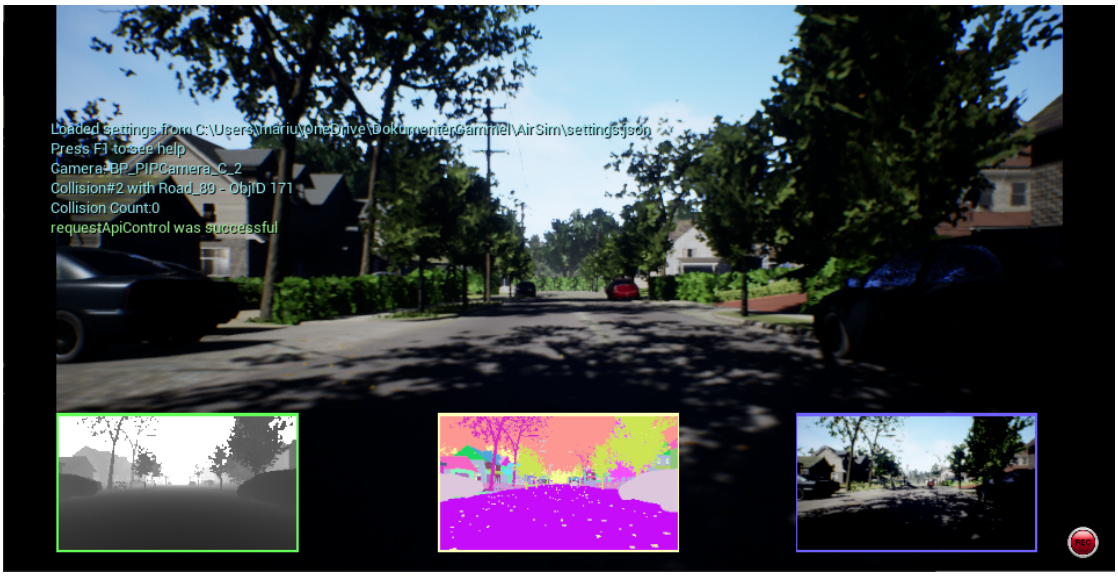


Figure 4.3: Image showing different AirSim image modes at the bottom. From left: Depth perception, object detection and regular camera.

The image above is a screenshot taken from running AirSim on the "Neighborhood" map which is available from the AirSim binaries.[15] The full screen image is a direct video stream from the drone and is the same as the rightmost of the three smaller photos. The center of the three smaller photos is a camera that uses object detection. This shows a different color for each object. For example the ground is a dark pink whilst the yellow dots on the ground are leaves. The trees further in the background are yellow and the cars on the right and left sides of the image are gray. Extracting this data is easy when in a simulation as we can ask the engine, in this case Unreal Engine, for each object and plot it in a different color. In real life this is a very complex computation which is mostly handled by intricate machine learning algorithms. Because of the difficulty of extracting this data on a physical drone we have decided not to use this data and rather focus on the leftmost approach.

The gray-scale image to the left simulates the stereo camera approach. This image projects the range from the drone in different gray scale values for reach pixel in the image. Entirely white pixels represent distances of 100 meters or more, while entirely black pixels represent a distance of 0 meters. By using this data we can create matrices that contain the range to every single pixel in the image. This data will be very useful when writing an algorithm that will make a drone choose the best path. For example one would rather turn left than going forward if the center of the image is very close to the drone when there is free space to the left.

4.2.3 LiDAR vs. Stereo Imaging

To choose sensors there are many factors to consider. Mainly we need to consider the trade-off between more accurate obstacle avoidance and the algorithms running time. Both LiDAR sensors and stereo cameras can be tailored to provide the same data to the drone. We can

limit the LiDAR sensors field of view and lower its polling rate to combat huge data sets and slow computation times. Correspondingly, we can mount multiple stereo cameras on the same drone to increase the seemingly limited field of view of a single camera. This brings us to the conclusion that the choice of sensors is based on our personal preference. We have learned some image handling through courses at the University of Stavanger which lead us to choose the stereo camera for main obstacle avoidance. For us to be able explore the LiDAR technology we also added LiDAR sensors as safety tests when adjusting the altitude of the drone. This approach makes it possible for us to explore both types of technology, yielding a greater learning experience.

4.2.4 Implementation

To get and format the image data we have created a method called *getImageData*. This method is run once per iteration to make sure that the drone has access to the latest image data as it decides where to navigate. *getImageData* asks the AirSim API for the image, and receives a one-dimensional list in return. As we have experienced some problems with the image data AirSim provides, we loop over the values in the list and sanitize them. In the same loop we also convert the data to a more humanly readable format, but multiplying by 100. This way whenever we need data from the image it is instantly presented in meters rather than a float between 0 and 1. Finally, we reshape this list into a matrix of pixels and save this value so it can be used in the next rest of the algorithm.

```
1  def getImageData(self):
2      """
3          Sets the depth data to the formated matrix
4          Values in the matrix are meters from the camera
5      """
6      responses = self.client.simGetImages([airsim.ImageRequest(0, airsim.ImageType.
       DepthVis, True)])
7      result = responses[0]
8      depth = np.array(result.image_data_float, dtype=np.float32)
9      for i in range(len(depth)):
10         if depth[i] > 1: depth[i] = 1
11
12         depth[i] *= 100
13     depth = depth.reshape(result.height, result.width)
14     self.depth = depth
```

On line 6 we use the AirSim API to get the images we need. In our case, this is a *DepthVis* image. The *DepthVis* image is similar to the gray-scale preview from the bottom left of figure 4.3. Because the *.simGetImages()* function returns a list of images, we extract the first image from the list on line 7. The result object includes valuable data about the image that is returned, as well as the actual image data for each pixel. When we pass *ImageType.DepthVis* as a parameter it will return the dimensions of the image, as well as an array of float values that represent the length from the camera to each pixel. On line 8 we extract the float for each pixel and save it in a *numpy* array. This data is simulated and, for an unknown reason, some of the values might be beyond of the intended range of 0 to 1. Therefore, we test if any of the values are above 1 and set these to our maximum; 1. We know from the AirSim documentation that a value of 1 means 100, or more, meters away, whilst 0 means 0 meters away. For human readability, we multiply each value by 100. This helps us as developers in the rest of the program by allowing us to use a an integer that represents meters rather than gray-scale float values. Finally, we use the *numpy* reshape function to reshape our linear array into a matrix of values representing meters from the camera for each pixel and save this to *self*.

This method makes it easy for us to use the depth matrix for image operations later. In this matrix each value (self.depth[x][y]) represents the number of meters from the camera to the

surface that is pictured in pixel (x, y) in the grey scale image. Because this matrix is in the same format as an image we can easily segment, draw or interpret it as will be further discussed and made example of in the rest of this chapter.

4.2.5 Drone pose

In order for the drone to act on information about its environment, it must also have some knowledge about its own location and orientation. This will be achieved on actual drones with a multitude of different sensors, such as GPS, barometers and gyroscopes. This is available in AirSim via the simulation API. Therefore, the pitch, roll and yaw are simply extracted from the API and stored in the *self* object.

4.3 Interpreting surroundings

Now that the data from the image of the environment in front of the drone is saved and formatted, we can start working on the next stage of the algorithm; extracting some useful information from the image. In our case, we want to use the gray scale image to determine the best next step for our drone. We split the image into several sections that will be needed for the drone to later compare possible moves.

Figure 4.4: Segmented depth perception

Figure 4.4 shows depth perception image similar to a gray scale image the drone uses, but with different colors. In this image, bright yellow represents ranges of 100 meters or more from the camera, while the darker purple represents objects that are closer to the drone. This image also shows the segmentation the drone uses to calculate where to move. There are three boxes, each of these representing the possibility of turning left, staying on course or turning right. Additionally we use the upper and lower middle as a fail-safe when changing the altitude of the drone. Lets look at the code where this segmentation is done, using the powerful *numpy* library.

```
1 def splitImageData(self):
2     """ Splits the image data in 5 matrices
      which indicate the possible moves (left,
      middle, right, up and down) """
3     vsplits = np.array_split(self.depth, 3)
4     self.top = np.array_split(vsplits[0], 3, 1)
[1]
5     self.bottom = vsplits[2]
6     splits = np.array_split(vsplits[1], 3, 1)
7     self.left = splits[0]
8     self.middle = splits[1]
9     self.right = splits[2]
```

The purpose of this method is to split the image into the segments shown in figure 4.4. Having a method that does this at the beginning of every iteration makes it easier to use this data in the rest of the calculations and helps ensure that we do not perform heavy matrix operations more than once. The code on line 4 divides the entire image into 3 equally-sized slices; top, middle and bottom. We split the top slice of the image and top middle part for later use. It will be used to test whether flying above an obstacle is safe. We save the whole bottom slice so that we can determine whether decreasing the altitude is safe. Finally, on line 6, we split the middle slice of

the image into left, right and middle parts. These are saved and will later be used for horizontal movement.

Once again every variable is saved to the global *self* object so that the entirety of our program can access these when needed.

4.4 Deciding where to move

Now that we have gathered some information about the surroundings and the drones own pose, we can create an algorithm that uses this information to act. Our algorithm is always going to try to turn towards the given goal. If that is not possible, it will try to go straight forward. However, if the drone is too close to an obstacle, a decision will be made whether it is safer to turn left, right or adjust the drones altitude to avoid the obstacle. This is shown in the flowchart in figure 4.1. In this section, we will explain the rest of the logic in the *Main Loop* in the flow chart.

4.4.1 Understanding the execute loop

In order for our drone to be able to navigate for an unknown amount of time we need a while loop. The loop is running off a boolean called *running*. By saving the boolean in the *self* object and we can change it from anywhere in the algorithm when the while loop needs to stop. An example can be seen when we reach our goal. The *running* is set to false and the drone never executes the next iteration of the loop.

We originally had a wait time when we created this loop so that every single iteration of the loop would use the same amount of time. This was done by recording the time it took for the current iteration to be done and then waiting the remainder of the preset time for a single tick. However, we realized that there are no major benefits to this strategy. We removed the wait time so that each iteration of the loop finishes as fast as possible. This approach allows the algorithm to run even faster on more powerful computers and does not present any disadvantages compared to having a constant iteration time.

Finally, for debugging and development reasons, we count the number of iterations. This is a powerful tool for understanding when a bug occurs, but it is not significant when the program is not being developed.

4.4.2 Calculating possible moves

For our drone there are five possible moves; *turn left*, *go straight*, *turn right*, *turn left towards goal* and *turn right towards goal*. In this section, we will explain the logic in the *Main obstacle avoidance* module of the flow chart in figure 4.1. We are going to look at how the drone determines whether a section of the image is too near to the drone and how it uses the image to determine which way to move. Let us first look at the related part of the flow chart:

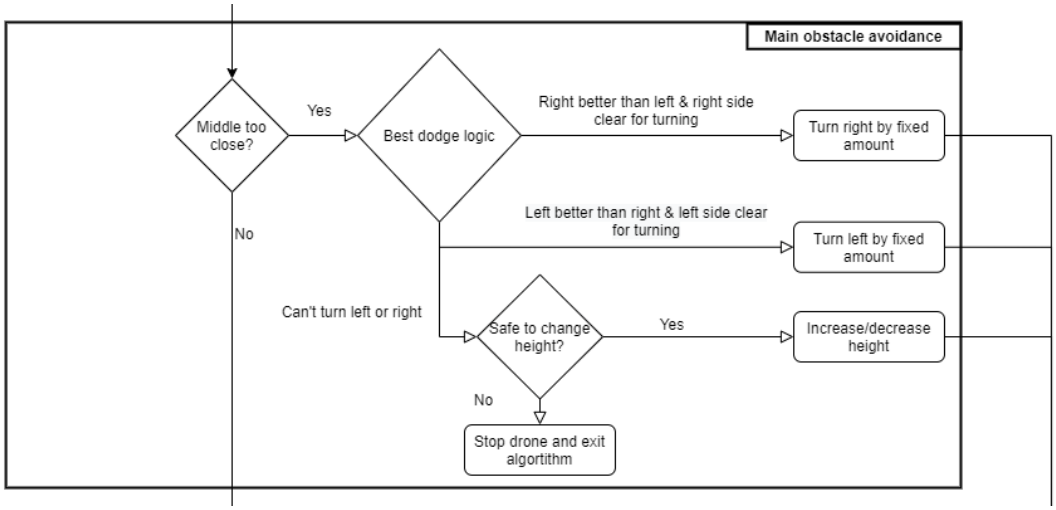


Figure 4.5: Flow chart of logic for dodging an obstacle

Checking if a segment is too close

As stated when we formatted the image in the previous chapter, the image is now divided into three sections; left, middle and right. There are many different ways to define when you want each of these segments to count as "too close" to the drone. The data we use in this algorithm is an image taken by a stereo camera. Programmatically, we use this image as a matrix of values that represent how far each pixel is from the camera. It is, therefore, natural to use "number of pixels too close" as an indicator for how near the objects in the segment are. To use this approach we need to define what a pixel that is "too close" means. Luckily for us, every single pixel is already stored in a matrix where the values represent meters away from the camera. Therefore, we can choose an arbitrary number of meters that we define to be "too close" to the drone. In our algorithm this can be changed in the *set variables* part of our flowchart. By default, this value is set to 5 meters, meaning that every pixel that is less than 5 meters away should be interpreted by the drone as "too close". Now that we know when a pixel is too close, we need to define when we think that the entire segment of the image is unsuitable for navigation. Here is an outline of some different approaches with their key advantages and disadvantages:

Approach	**Advantage**	**Disadvantage**
Average of all pixels	Simple calculation	Easily skewed by pixels that are very close or very far
Median of all pixels	Simple calculation	Polarized if there are zones of pixels with very different ranges
Percentage of all pixels	Does not consider pixels that are not "too close"	Might overlook objects smaller than threshold

All three of these approaches make sense in their own way. For example, a low average pixel value from a segment would mean that the entire segment is most likely close to the drone. However, the issue occurs when part of the segment may be near but the remainder of the segment is very far away. Consider, for example, an image segment where there is an object in the middle that occupies two thirds of the segment. The object is one meter away from the drone. However, the remainder of the segment can only see the sky, which is more than 100 meters away from the

drone. In this case, the average length in the segment is around 34 meters, and the drone will crash into the object that is 1 meter away.

This problem can be partially solved by using the median. In the same scenario, the median range in the segment would be one meter, and the drone would turn away from the object. However consider a scenario where the object that is 1 meter from the drone only covers 49% of the image, and the sky covers the remaining 51%. In this case the median calculation would tell us that the range value in the segment is 100 meters, and once again the drone would crash into the object.

Since these two options have clear cases where they would falter, we needed to create a new approach for the drone to assess whether or not a segment is safe for flight. In this algorithm, we test whether each pixel is "too close" to the drone and calculate the percentage of pixels that are too close to the drone. Through doing so, we can check if there is enough space to fly through the tested section. In our algorithm the default value for, "enough free space" is 20%. This means that if 20, or more, percent of the pixels are less than 5 meters away our drone will consider the segment unsuitable for flight. The percentage is and arbitrary factor that can be adjusted before flying. This allows each user the flexibility to tailor the algorithm to their needs. A higher percentage would result in more reckless flight, but might for example allow for higher flying speeds. On the contrary, a lower percentage would create a more cautious flight, useful for flying in spaces where small objects could be found in the drones path. In some cases one would not have to care if only the leftmost 5% of the middle segment is covered, whilst in other cases a small object in the middle of the segment would disrupt flight.[1] By adjusting the values of the threshold and the percentage of "too close" pixels one would be able to achieve safe, but effective, flight in any environment.

The code used for this calculation is a simple matrix operation. We loop over every single pixel in the matrix that represents the image, and count each pixel that has a value lower than our threshold. Then we divide the number of pixels under the threshold by the total number of pixels and multiply it by 100 to return a clean percentage.

$$\frac{\text{"too close" pixels}}{\text{total pixels}} * 100$$

Since our function returns this percentage we can easily compare each segment with our predefined threshold or with other segments to select the most optimal path.

Checking if vertical movement is safe

Drones have the ability to change their altitude without having to change either latitude or longitude. When a drone moves directly upwards or downwards the camera has no information about that area. Our solution to this is to add two more sensors to the drone. This is possible on a physical drone by adding a cheap, single-point stereo sensor. Simple distance sensors in AirSim are only capable of pointing forwards, as rotating them has not been implemented yet.

The second best sensor would be a low polling-rate LiDAR sensor which scans only a small area. We are attaching two additional LiDAR sensors, one on the top of the drone and one on the bottom. These sensors provide information about the area immediately above or underneath the drone. We check the LiDAR scanning in a small square directly above and below the middle of the drone and have defined it as "safe to change height" if the corresponding LiDAR scanning returns ranges of more than 1 meter.

[1] An idea we debated for this part of the algorithm would be to weigh pixels closer to the center more than peripheral pixels. This would make it possible for the drone to ignore a small object on the edge of the middle image segment, but dodge it if it is directly in front of the drone.

Going left, forward or right

For simplicity, in this part, we are going to assume that the drone is trying to reach a point that is right in front of the drone. Between the drone and the target are some obstacles that need to be circumvented. In this scenario, the drone will always try to fly straight towards the target, and only divert if there is anything too close in the front of the drone. Because of the functions explained above, we can calculate the percentages for each segment and compare them:

```
1     def avoid(self):
2         """ Main collision avoidance logic """
3         m = self.calculateTooClosePercentage(self.middle)
4         if  m > self.percentage:
5             l = self.calculateTooClosePercentage(self.left)
6             r = self.calculateTooClosePercentage(self.right)
7             if l > r and r < self.percentage * 3:
8                 self.turnRight()
9             elif r > l and l < self.percentage * 3:
10                self.turnLeft()
11            else:
12                # Adjust altitude logic
13        else:
14            self.goStraight()
```

This code represents the flow chart shown in Figure 4.5. Referring to the flow chart might be helpful in interpreting the corresponding code. In this code, we calculate the "too close" percentage once for every segment. Firstly, on line 4, we test if the middle segment is blocked enough to cause a turn.

When there is an obstacle directly in front of the drone, we have got to go through the steps to find the best dodge. We would rather turn left or right than adjust the altitude. Changing altitude usually implies a greater deviation from our path than dodging left or right. The first possibility we test is turning right. If the right side is better suited than the left side and is "clear enough"[2], the drone turns right. The same logic applies to a potential left turn. As a last resort, the drone will be able to adjust its altitude if all other options are blocked. This logic will be further clarified at the end of this chapter.

Turning the drone simply means we have to change the drones yaw. A negative yaw represents counter-clockwise motion in the AirSims coordinate system, and a positive yaw represents clockwise motion. Therefore we can get the drones yaw and subtract from it if we want to turn left, or add to it if we want to turn right. The following code turns the drone a constant value to the left. Note that the only difference in the *turnRight()* function is that the value would be added to the current yaw, not subtracted.

```
1 def turnLeft(self):
2     """ Turns the drone to the left by PI/32 radians """
3     yaw = self.yaw - math.pi / 32
4     self.fly(yaw)
```

4.4.3 Moving towards a goal

With the avoid logic in place, the drone is now able to dodge any obstacles that may occur in its way. However, there are limited use cases for drones that can only go straight and avoid obstacles in their paths. In this section we will look at the additional logic needed to make the

[2]We define "clear enough" as three times the threshold for going forwards. Turning towards a possible obstacle is preferred to flying straight towards a confirmed obstacle in front. Scaling this arbitrary value down results in more frequent changes in altitude.

drone follow a path of points. Our algorithm treats a path as a list of points to be visited before arriving at the final target. Therefore, all we need is the code for moving to a single point, and loop it for each point in the list.

Moving towards a yaw

Our algorithm should be able to operate a real drone in the same way it does in our simulator. We decided to make the algorithm high-level by using an API call in AirSim that is not specific to each different drone[3]. This is useful as further work can simply create a similar API for their own drone and instantly use our algorithm on any physical drone. The AirSim API we used is called *moveByVelocityZAsync()* and our fly method looks like this:

```
1  def fly(self, yaw):
2      """ Flies toward the given yaw """
3      vx = math.cos(yaw)
4      vy = math.sin(yaw)
5      self.client.moveByVelocityZAsync(
6          vx,
7          vy,
8          self.height,
9          2,
10         airsim.DrivetrainType.ForwardOnly,
11         airsim.YawMode(False, 0)
12     )
```

This is the most advanced, most used and most important API call in this algorithm. Read more about the AirSim movement APIs in the AirSim documentation [14]. To understand this method we need to understand the *YawMode* and the *DrivetrainType* in the two last parameters of the line 5 method call.

There are two *DrivetrainTypes* in AirSim; *MaxDegreesOfFreedom* and *ForwardOnly*. The first one lets us turn the drone in any direction we like by giving a particular angle to the *YawMode*. The second, which is the one we use, automatically turns the drone in the direction the movement is set. For example if we tell the drone to move east as it points north, the drone will automatically turn east as it moves. This is beneficial because our algorithm is based entirely on a forward-facing camera, and this approach keeps that camera in the direction of travel, at all times.

YawMode handles turning the drone in AirSim. Since we have already established that the drone should always turn with its motion, we do not need to specify a yaw shift in the *YawMode* parameter. We set the *YawMode* parameters to what is needed for the *ForwardOnly Drivetrain* to function properly. This is also further elaborated in the AirSim documentation.

The parameter on line 9 is simply the time that this command should be run. This is set to an arbitrary value of 2 seconds. This entire function will run every tick of the algorithm, and it should be cancelled and re-run with new values before the two seconds have passed. As explained in Appendix A.2 we choose to omit the *.join()* function on line 12, so that this function will block the next tick of the algorithm. This 2 second value should always be held higher than the slowest possible iteration of the algorithm for smooth flight.

The third parameter in this function, is the height in the coordinate system. As previously mentioned this value has to be negative to represent a height above the ground. This value is changed by the obstacle avoidance and goal targeting modules.

[3]To achieve a lower-level approach we would have to handle different RPM adjustments or how the drones stabilizes. If we chose this approach our algorithm would need modifications to work on any other drone, something we do not want.

The first two, and most important, parameters in this method call are the velocity in the X and Y directions. Since we have used the *ForwardOnly DrivetrainType* all we need to do is set the speed in the direction we need to move. The direction we need to travel is passed into our *fly* function through the yaw parameter.

The figure to the right is a view of the drone from above. This is what it would look like when the drone faces east and flies along the x-axis. The thicker blue line to the top left is the direction the drone needs to fly. By using the unit circle, we realize that the cosine would yield the value on the x-axis while the sine will give us the value on the y-axis. In the code above, we use this math to get the correct velocity value for each axis. Note that this velocity can be multiplied to attain higher speeds. Without multiplication, the drone will always travel at 1 m/s in the direction it is heading.

We achieve smooth flight toward the specified yaw by using these parameters. The camera will always be faced in the direction of movement and ready to avoid objects. This *fly* function is used for any move the drone makes our algorithm, whether it is dodging obstacles, going straight or heading towards a target.

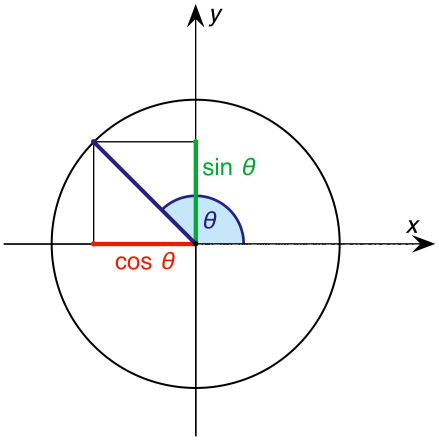


Figure 4.6: Calculation of X and Y velocities

Choosing a yaw to fly towards

This section explores how we can figure out which yaw to input into the fly method for each iteration. We have already addressed how to move when dodging obstacles, so we are only going to look at moving towards targets in this segment. Lets take a look at the relevant part of the flow chart.

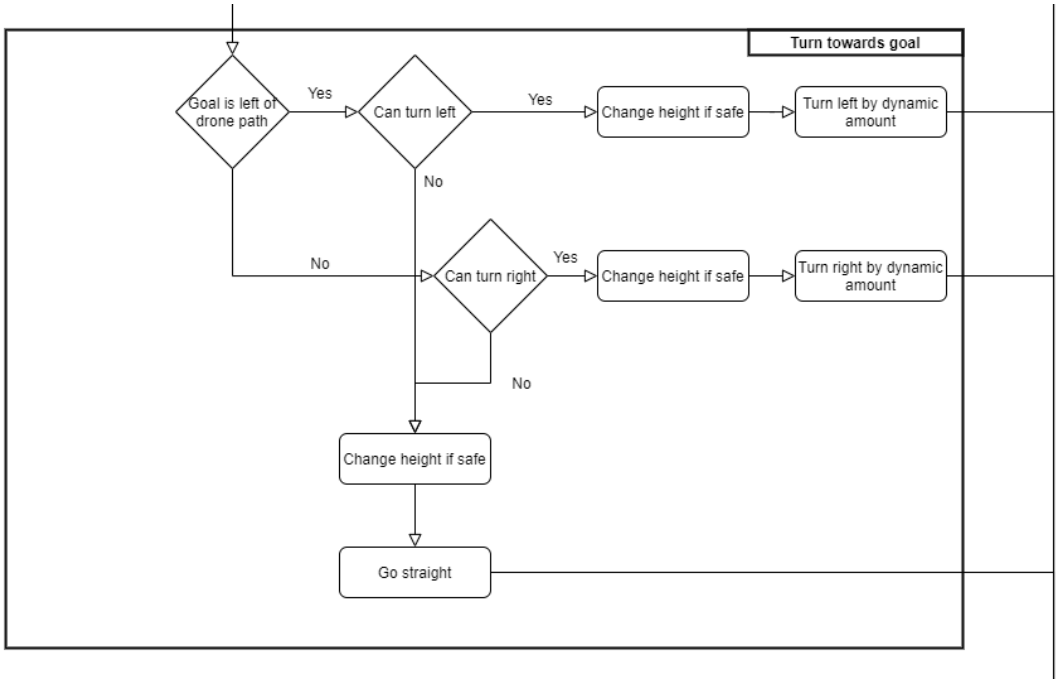


Figure 4.7: Flow chart of logic for turning towards a goal

The first thing we need to do is find the difference in the yaw between the drones current path and the target point. AirSim uses a static coordinate system, so when the drone turns the yaw of the drone changes while the coordinate system stays in place. The drones yaw is set to 0 at the forward position when AirSim starts. Therefore we can once again interpret the drone to be in a 2D coordinate system like in Figure 4.6, and do calculations without having to worry about rotating the entire system. We will consider two code snippets in this section, both are taken from our method *turnTowardsGoal()*. This first one handles calculating the angle between the drones direction and the goal point.

```
pos = self.client.simGetVehiclePose().position
dX = self.goal.x_val - pos.x_val
dY = self.goal.y_val - pos.y_val
dest = math.atan2(dY, dX)
yaw = self.yaw

a = dest - yaw
if a > math.pi:
    a -= math.pi * 2
elif a < -math.pi:
    a += math.pi * 2
```

We need the position of the drone and the location of the target point. The *atan2()* function calculates the angle from the x-axis to a vector. The vector we use will be a vector from the origin to the point in the methods parameters. The origin point is the (0, 0, 0) point in the coordinate system, which in AirSim is defined by where the drone starts.

The angle we pass to the fly function is defined so that 0 degrees means no change of direction. We need to find the vector between our drone and the target to get this angles. We find vector coordinates by subtracting the current drone position from the

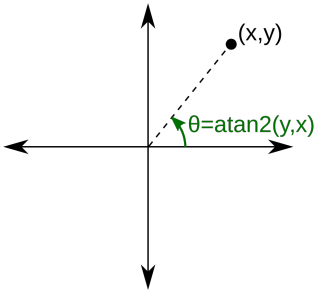


Figure 4.8: Getting the angle between the x-axis and a point

target. We can picture this as shifting the drone to the origin of the system before calculating the angle. The *atan2()* now returns the angle between the x-axis and the target. In the same way that the drone can have different position than the starting point, it may also not aligned with the x-axis, as it is when AirSim starts. Therefore we need to subtract the drones current angle from the target angle to get the difference between the two. Because both of these values are already in radians we are now left with the value a. This value is a signed number of radians that tells us how far the drone will turn to the left(negative values) or to the right(positive values).

In most cases, simply adding the a value to the drones current yaw yields the result we need. There are several cases, however, where this implementation does not function correctly due to the circular nature of the coordinate system. In such cases we need to shift the yaw value so that we get the shortest possible value. Let us take a look at another figure and the math behind the shift.

Line 8-11 of our code handles angles that are outside AirSims yaw scope, which ranges from -π to +π. If the angle is outside of this range it will be pushed back in by adding or subtracting the entire range of 2π. The combination of this shift and the way we calculate the angle effectively solves the complicated problem of finding the shortest angle in every scenario. In Figure 4.9 the red arrow is the direction of the goal and the green arrow is the current direction of our drone. By using simple subtraction we would always get an angle that is inside the scope of -π to +π. In the figure, that would represent the blue angle. However we would always like to get the shortest possible angle between the yaw of our drone and direction of the target, which is represented by the yellow angle in the figure. By adding or subtracting and entire range of the circle if the yaw is outside of our circular range, we will always end up with the shortest possible angle. We will use the figure as an example. The yaw of the drone is around 3/4 π, and the yaw of the target is approximately -3/4 π. This gives us [2]:

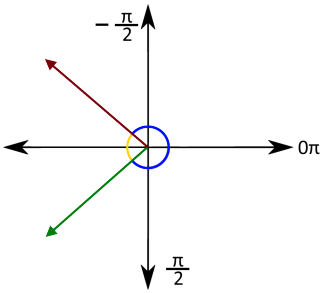


Figure 4.9: Difference between longest and shortest angle

$$a = \text{destination} - \text{current drone path}$$

$$a = -\frac{3}{4}\pi - \frac{3}{4}\pi = -\frac{3}{2}pi$$

-3/2 π is outside our range and therefore we shift it back in:

$$a = a + 2\pi$$

$$a = -\frac{3}{2}\pi + 2\pi = \frac{1}{2}\pi$$

From the figure, we can infer that the target is about 1/2 π to the right of the drones current direction, which matches the calculation even in the out-of-range scenario. It is important to understand this math to have insight into how we use modulus mathematics with a range that does not start at 0.

The rest of the calculations for finding a yaw to fly against utilize functions we have previously described:

```
if(a > 0):
    if(self.calculateTooClosePercentage(self.right) < self.percentage):
        yaw = (self.yaw + a / 5)
else:
    if(self.calculateTooClosePercentage(self.left) < self.percentage):
        yaw = (self.yaw + a / 5)
self.fly(yaw)
```

Before turning to either the sides we need to test whether it is possible turn. First we test if the newly calculated a value is greater than 0. As stated, positive values represent right turns and negative values, left turns. We calculate "too close" percentage of the corresponding side and compare it to the predefined percentage. If it is clear we can begin to turn towards the target. For smooth turns the angle between the drones direction and the target is divided by 5. The new angle overrides the drones current yaw and is used in the fly method. If the side the drone wants to turn against is not clear, the yaw will remain unchanged and the drone will fly straight. It is important to remember that this logic is applied after the drone has already checked if any obstacles need to be avoided. This means that if this code is run, it is always possible to fly straight forwards safely.

Changing the height

To support a path that has points at various heights, we need to adjust the height according to the height of the target and the movements that are currently safe. If the corresponding LiDAR sensor confirms that changing the height is safe, the difference in altitude as calculated. Finally, we apply the same smoothing procedure as when adjusting the yaw. The drone considers the difference between the drones current height and the goal height, before raising or lowering the drone 1/5th of the difference each tick.

Updating the goal

When the drone follows a path, the user has provided algorithm with a list of points to be visited. When the drone is within a certain range of the current target, the target is overwritten and set to the next of the points in the specified list. If the algorithm is run with only a single point, or the current goal is the last one in the list the program will exit safely with a message for the user.

4.5 Summary and results

4.5.1 What objects will the drone be able to dodge?

The drone will view its surrounds as pixels that are too near or at a safe distance. Once this is done, the drone must test whether or not the number of pixels that are too near are over a threshold. This means that if an object is too small (less than 20% of the middle image segment when 0 meters from the camera) the drone can not perceive this as an obstacle. However this can be modified by adjusting the threshold before running the algorithm. The algorithm has been tested in the Neighborhood map, which contains small signposts and small trees. The drone is able to steer clear of objects as small as a sign post. If this algorithm were to be used in an environment that demanded even more careful navigation, the threshold would have to be lowered. The drawback is the possibility of the drone interpreting any unintended noise as an obstacle.

Avoiding moving objects perfectly requires an extra level of logic. Because the drone only monitors the middle section of the camera, it will not be searching for objects traveling from the edges of the camera to the centre. Therefore there is no way to dodge an obstacle until it is directly in front of the drone. Whereas if two drones fly together and the first flies too close to, but still directly in front of, the second drone, the second drone will prioritize dodging and move to the side. The algorithm has been tested on the *City* map which contains moving cars and humans. If there is an obstruction in the front of the drone, it is able to avoid them by using the same logic as for stationary objects. However the drone has no way to interpret if an object is coming from the side or the back and can not dodge these. This is further discussed in Section 6.1.3

4.5.2 Results

This chapter describes the main algorithm that this thesis is built around. The drone is now able to complete a variety of tasks by using this code. We can achieve safe flight between any two points by dodging objects on the way to our target. If we want more complex flight routes we can use the same logic to move from target to target until we reach our destination. This algorithm excels when combined with the extra modules that we have created, which are described in the next chapter.

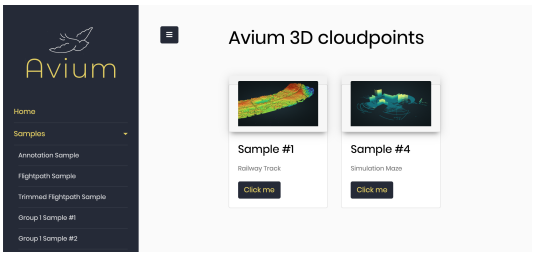
Chapter 5

Extra modules

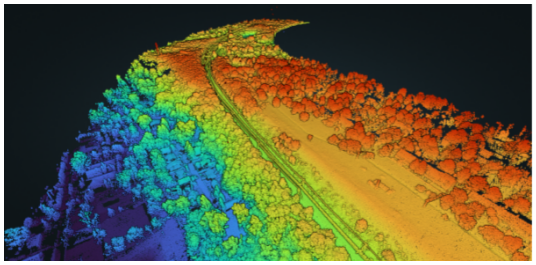
As stated in the original project description this thesis focuses on creating an algorithm for safe navigation of drones in any environment. This algorithm has been explained in Chapter 4. For this algorithm to be useful in we have created several modules that alters the in- and output of our program. In this chapter we will discuss how these are made and how they are used to be able to provide a powerful simulation tool for a larger platform. We will also look at how these modules create a link between this algorithm and the platform created by Group 2.

5.1 Merging with Group 2

This thesis is written as a part of a two-part task form Nordic Unmanned. While our task is to navigate in collect data the other groups task is to visualize and analyze this data to provide useful insights for users.



(a) Group 2 landing page



(b) Group 2 visualization of point cloud

Group 2 has created a platform the visualizes point clouds with different information. A user would have the ability to plot the drones path in the point cloud, look at images taken during flight, and send a new path back to our algorithm for further flights. For our product to be usable with their platform we needed functions that could output all this data and take a path as input for new flights.

5.1.1 LiDAR data

A point cloud is a structure that contains a multitude of points to visualize an area in 3D. To create a point cloud we had to use the inbuilt AirSim LiDAR sensors. These replicate real life

LiDAR sensors by scanning the surroundings in AirSim and returning a list of points. This list is a continuous stream of alternating X, Y and Z coordinates for each point in the point cloud. To clean this up we use numpy to reshape the point could to a list containing nested lists that each represent a single point.

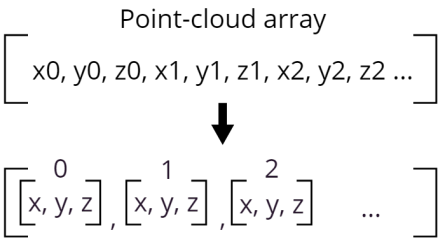


Figure 5.2: Organizing the point cloud data

This data can be written to a *.txt* file that can be used by Group 2 as visualization data. Group 2 needs this file to be formatted in a specific way for it to be usable. Each point has to be on a single line, and each of the X, Y and Z values have to be separated by a space. In addition to this we have to remember that AirSim is using a NED coordinate system. For the point cloud to not be mirrored around the ground plane we need to reverse every single Z value. This is done by the following code.

Additionally we have the ability to use customized file- and foldernames. This also means we need error handling for opening and writing to the file.

```
def writeLidarDataToDisk(data, foldername, filename):
    """Writes the lidar data to the spicified folder with the specified file name
    """
    try:
        with open(f'{foldername}/{filename}.txt', 'a') as f:
            for point in data:
                f.write(f'{point[0]} {point[1]} {point[2] * -1}\n')
    except FileNotFoundError:
        print(f'LidarData not saved. File "./{foldername}/{filename}" not found.')
```

5.1.2 Flight path

Gathering a flight path

As mentioned Group 2 is able to show our drones flight path in the point cloud. For this to happen we need to collect some data they can visualize. This is very similar to how we save LiDAR data. We can gather the drones position once every tick and write a single new line in a *.txt* file that represents the path. This will create a file that contains points along the path that are only a couple of centimeters apart, and are therefore easy to visualize. The combination of a point cloud we have gathered, the drones path and Group 2s platform yield this result:

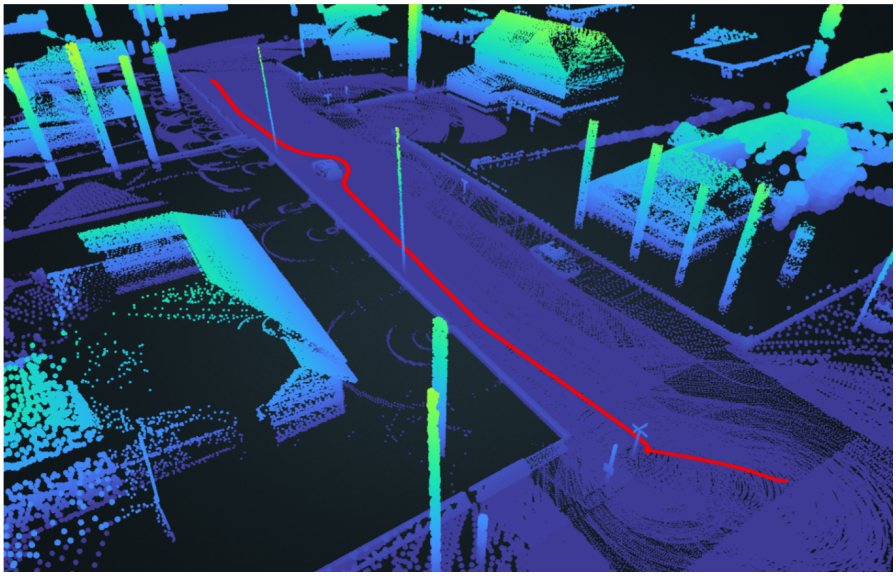


Figure 5.3: Group 2 point cloud with flight path

This image shows a neighborhood with streets, houses and cars. The red line represents the path our drone has taken from starting at the bottom right and flying to the upper left. We can see the red line diverge at a signpost close to where the drone started and dodging a parked car about halfway through the flight. All movements from the drone in this image are done by our algorithm.

Importing and following a flight path

Our algorithm also has a module to read both *.JSON* and *.txt* flight paths and follow these in a simulation. For simplicity our file reader uses the same data format as when a flight path is output. We can read this file and format it to a list of 3D points. When this list is fed to our algorithm it uses the logic explained in Chapter 4 to follow a path by flying from point to point.

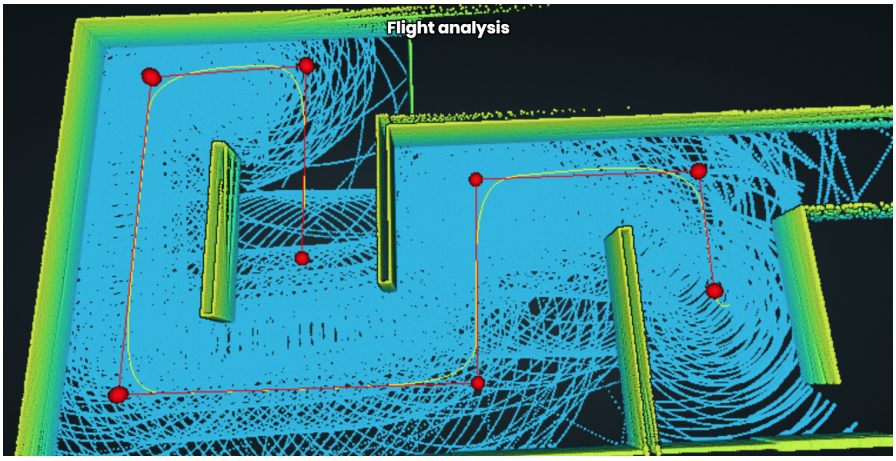


Figure 5.4: LiDAR visualization with planned path and simulated path

By using Group 2s flight analysis tool on one of our flight in a maze we can compared the planned flight path with the simulated flight path. This images shows a maze we have LiDAR scanned and flown through. Each red dot is a point in the planned path and the red line represents the shortest path between each point. The green line represents the simulated path. In this case our algorithm achieved less than one meter deviation from the planned path through the entire flight.

5.1.3 Capturing images

Group 2s platform is able to display images taken during flight. Images are captured and saved in a predefined folder. Since Group 2 displays these images in their 3D environment we also need to save the position where the image is taken. We used the same logic as when saving the drones position. This creates another *.txt* file with a single point for each image taken. By using this format Group 2 can simply link the first image in the image folder with the first position in the *.txt* file.

```
if(time.thread_time() - self.lastImageTime > self.imageFrequency):
    self.saveImage()
    self.imageNumber += 1
    self.lastImageTime = time.thread_time()
```

We only want to capture images at a certain time interval. This code is run directly in the main loop and checks if the time since we last captured an image is more than the number of seconds we want between each image. If it is we get an image from AirSim and write it to the specified folder. This image is named by using the *imageNumber* variable. After the image is saved the variable is incremented. This ensures that each image is named a single number starting at 0. When the image is saved the position file is opened and the drones position is appended to the file. When the image and position are saved we return to the main loop and set the *lastImageTime* to the current thread time. Now the algorithm is ready to continue and we can compare the thread time in the upcoming iterations to check if a new image should be saved.

```
def saveImage(self):
    img = self.client.simGetImage("0", airsim.ImageType.Scene)
    with open(f'{self.imageFolderName}/{str(self.imageNumber)}.png', 'wb') as f:
        f.write(img)
    with open(f'{self.imageFolderName}/pos.txt', 'a') as f:
        pos = self.client.simGetVehiclePose().position
        f.write(f'{self.imageNumber} {pos.x_val} {pos.y_val} {pos.z_val}\n')
```

5.2 Argument Parsing

While testing the program there arouse a need for easy switching of variables or input for the collision avoidance program so that we did not have to change the value of variables in the code ourselves and that it could be easily implemented with a user interface or something along those lines in the future.

5.2.1 Library

To accomplish the parsing of input elements from the command line interface we decided to use the library module *argparse* which is part of the *Generic Operating System Services* under the *Python Standard Library*, on pythons documentation website[18]. The library has many nice to

have features built in like easily showing help and usage for the specific arguments which makes it easier to use for a user.

5.2.2 argumentParser.py script

Most of the logic for this achieved by using a separate class to make the main script *obstacleAvoidance.py* more clean and easy to read. When the main class initiates itself the variable *args* is declared by running the separate *argumentParser.py* script. In a short sense this script interprets the input from the command line interface and makes a dictonary object out of this. To begin with the script initialises a ArgumentParser from the *argparse* module, then the next couple of lines uses the *add_argument()* function to make the parsing object capable of interpreting the different kinds of inputs. The method has a couple of arguments, like the the name of arguments. The arguments used in our script are only optional, declared by using the "-" sign before the name and since none of the arguments are positional are obligatory the main script obstacleAviodance.py will run anyway. Most of the arguments used with the *add_argument()* method is the *default=* argument which sets a starting value and the input from command line can for example change the value of a boolean. There are two arguments for controlling the drone and one of them is the *-points* argument and it has to have 3 arguments of the type integer and will send an error if for example "-points 3 5 h" is inputted. The other one is the *-path* and this one is for choosing a specific file with many points.

The optional arguments also has a string of text that will help a user decide which to use and how to format the input with the help command. An example on how this look can be seen in figure figure down below, 5.5.

```
C:\Users\odin-\Documents\Repo\Fly_tests_bin\BachelorTests>python obstacleAvoidance.py -h
usage: Parses arguments from cli [-h] [-sl] [-saveImage] [-saveImageFolder SAVEIMAGEFOLDER] [-savePos]
                                 [-savePosFile SAVEPOSFILE] [-points POINTS POINTS POINTS] [-path PATH]

optional arguments:
  -h, --help            show this help message and exit
  -sl, -save_lidar      Save lidar data to file.
  -saveImage            Save images during flight
  -saveImageFolder SAVEIMAGEFOLDER
                        Saves images to given folder.
  -savePos              Save drone position during flight to file.
  -savePosFile SAVEPOSFILE
                        Filename for drone positions.
  -points POINTS POINTS POINTS
                        Specifies a goal point. Format: X Y Z
  -path PATH            Specifies a file to get a path of points from.
```

Figure 5.5: Output from terminal when using help command

The rest of the script sets or changes some logical variables based on the dictionary object made from the parsing and then returns this object. This script is imported to the main obstacleAvoidance.py file and initiated when making the class and saved as the args variable. The dictionary is used in deciding if parts of the data will be saved and where. The main application could also just as easy use the input from a GUI in the future, like the one in part 2.

Chapter 6

Future Work

In this chapter, we will address the potential for further development of this algorithm. We will present some ideas that could be used as potential bachelors, or masters, by adding to the product that we have already made. In addition, we will address some use cases and implementations for the product that we have developed.

6.1 Further development possibilities

This section outlines some of the changes or improvements that might have been made to this product if we had taken another direction or had more time.

6.1.1 LiDAR obstacle avoidance

When constructing this algorithm, we began by trying to use a LiDAR sensor to interpret the surroundings. When we did our research and tested the AirSim APIs, we realized that we would prefer to complete this project by using only a stereo camera. It would be interesting to see if and how anyone could accomplish the same logic by using only LiDAR data instead. The demand for autonomous drones is on the rise, but related technology, autonomous cars, is already a highly debated subject of technological innovation today. Most of the big American car manufacturers that are currently building an autonomous fleet are using LiDAR sensors, while some rivals are sticking to radars and stereo cameras[27].

It would also be important to compare the difference in performance between the use of imagery and LiDAR data. It is critical to be able to choose the cost-optimal hardware for a drone / car in an algorithm like this one.

6.1.2 Simulating flight through LiDAR point clouds

AirSim is based on the Unreal Engine and all the maps we used are created in the Unreal Engine editor. Being able to travel across any point of the cloud will make this algorithm / platform more versatile. We can imagine a situation where a user scans an area manually and can simulate any flight in this area using our algorithm. This will save time by eliminating the process of manually building each environment in the Unreal Engine.

This enhancement is not a direct addition to this thesis, but rather to the usability of the entire platform including Unreal Engine, AirSim and our script.

6.1.3 Interpreting entire surroundings

As of now the algorithm is only able to interpret objects directly in front of it. Therefore it will not be able to see objects approaching from the sides or from behind. As the drone only moves forwards it will never run into an object. There is a risk that another moving object might interfere with the drone from other angles than straight in front. A good further enhancement would be to incorporate the ability to see things approaching from the side or back and stop them.

We will need logic at a higher level than what is used in this algorithm to do this. Our algorithm simply reacts every iteration of the execution loop off a single image. To be able to determine whether the objects are moving from the side or back to the drone, we would have to save some data between the loops and compare the images to see whether objects are moving closer or not.

6.1.4 Dynamic turning when avoiding obstacles

The drone is currently turning $\pi/32$ radians each tick when avoiding obstacles. This should be done dynamically, meaning that the drone should turn more whenever it is close to an object. However, as we will explain in section 7.1, a lack of cooperation stopped us from being able to implement this.

A simple, but efficient, solution would be to employ a PID-controller[35]. The dynamic turning towards goals helps counter the twitchy and static nature of avoiding objects by quickly putting the drone back on the right course. However, by inputting the drones current speed and range to an object in PID-controller one could achieve further reliability.

6.1.5 Smoother adjustment in altitude

In our current build the altitude is statically changed according to the coordinate system. By using the *moveByVelocityZAsync()* function the Z values is given as a static value rather than the algorithm controlling the speed of the altitude change. For smoother adjustments in altitude one could use the *moveByVelocityAsync()* which is similar to the method we use. The difference is that this function lets us set a velocity on the Z axis instead of setting a static height above ground. This way we could change regulate the velocity so that the changes in height were less twitchy.

As mentioned in the previous section, section 6.1.4, employing a PID-controller would help gentle the adjustment in altitude. Some example inputs for this PID-controller could be the drones current speed on the Z-axis and the range to the closes obstacle in the corresponding direction.

6.1.6 Cloud computation

The problem that often has to be addressed when designing software to operate on a small computer, such as a drone, is the limitations of computing capacity.We would prefer to have cheaper hardware on a real drone and to make calculations on a larger computer off-drone. If we were to use cloud computing, there would be virtually no limit on how much data we might use to compute further flight. We would be able to incorporate LiDAR sensors or multiple cameras without having to think about the time of the computation. There would, however, arise other issues. One would, for example, have to implement a communication protocol fast and secure enough to efficiently transfer data between the cloud and the drone. The drone would also have to be connected to this communication protocol at all times, as it would not be able to

navigate by itself. At the same time, introducing cloud computation would be a brilliant learning oppertunity for bachelors students as the industry progresses towards cloud at a rapid pace[36].

6.1.7 Machine Learning

A way to develop the algorithm would be to implement some kind of neural network with inputs from the simulation and use them to accomplish better decision making in general. With the setup now in place it could look something like this 6.1 and could be trained by machine learning.

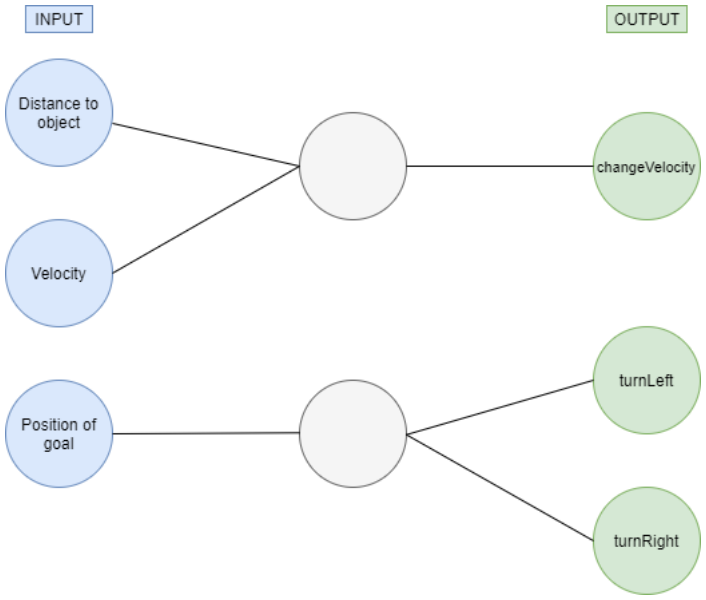


Figure 6.1: Visual representation of possible neural network

Machine learning is often done by making different variations of the same code or adjusting the variables somewhat and test how they stack up against each other. Different actions or goals could give scores that would quantify which of the variations did best in each generation and bring these forward for further iteration. Since the goal of the algorithm is for the drone to make its way to one or different points this could be one of the main incentives and give a lot of points. The same thing would apply to crashing since this could destroy the equipment in real life or ruin some of the essential sensory or cargo. AirSim shows a collision counter when colliding with an object so this data would not be hard to acquire. After tons of incremental improvements this could foster a better algorithm than we developed, but would

These are methods not yet learned by students at our level of bachelors, and an approach more grounded in machine learning would probably more suiting for students in the masters or higher tiers since it would be within their skill set. At this tier of knowledge things like machine vision for identifying objects and taking actions based on this information may be more suited. In the future this work could be replicated or advanced upon by other people and these people should consider machine learning for a more overall suited product.

6.2 Possible Implementations

Our algorithm can be employed on any drone with a front-facing stereo camera and a similar movement API. In this section we will discuss some potential implementations for our algorithm. With just a few modifications, this algorithm, in some cases combined with Group 2s platform, can form a product that can add value to a business or work as a business' core product.

6.2.1 Nordic Unmanned Inspection Service

As mentioned in the agreement with Nordic Unmanned, this algorithm is intended to be further implemented in inspection software. This algorithm has the ability to fly from point to point without crashing into something in the way. By utilizing this Nordic Unmanned could plan a path along an inspection target and move safely between each point.

Currently our algorithm only returns images with a set interval and all images are taken directly in front of the drone. By adding some logic to the image capture one would be able to take images at specified locations instead of at specified times. Additionally it would be useful to be able to make the drone wait while taking images and take images in any angle from the drone.

By using a combination of our thesis and the Group 2s application, we will be able to analyze the flight path and images, and send additional flight paths back to the drone. These flight paths can then include new places to take pictures for further analysis of specific areas.

The core functionality for a product like this is already implemented through the two thesis'. With some simple additions this could become a useful product for Nordic Unmanned and their customers.

6.2.2 Delivery

One of the first concepts that led us to start working with drones was the distribution of smaller packages. The idea stems from a situation where we wanted to order food, but there was no way we could get food delivered. We explored the possibility of getting food delivered by drones and all the advantages that would offer us over the options available. In Stavanger today, you can get food delivered by bike or by car directly from the restaurant. By hiring a drone to do this work, it would be possible to deliver the food without the need for an employee to carry the food to the place of delivery.

This would be a large implementation and could be the main product for an entire business. In this work, drones will have versatility by being able to deliver to windows in high-rise buildings, boats at sea and areas that do not have road connections. In addition, drones can bypass traffic and unoptimized roads by flying directly up and going straight from point to point.

There are companies that employ similar technology, such as Amazon [23], and others are looking to implement this in the near future, such as Stavangers Main Hospital [24]. Our point-to-point algorithm has already addressed a significant part of the issue in designing these solutions.

6.2.3 Simulation of real life flight

On the Group 2s site, the user will be able to design a new route and send it to a drone. It would be helpful to be able to test this route in a simulated pre-flight environment. We have already had a look at an analysis like in this Figure 5.4. It would be helpful to learn how to model a flight before doing it physically. We can see the deviations from the expected path and plan the positions from which to gather the data.

An obstacle that would have to be overcome in order to do this with the current AirSim setup is that the environment would have to be created in Unreal Engine to allow us to fly through it. With the extra implementation from section 6.1.2 this could be a powerful tool for saving time and capital by making accurate simulations available at any time

6.2.4 Avium projects

As stated in section 1.1, a large part of the personal interest in this project stems from the potential of our student organization Avium to use what we learn in future projects. Avium is working to develop software that can participate in the International Aerial Robotic Competition, IARC [25]. This competition changes the mission every couple of years, after the previous mission has been solved. Some of these missions will involve a drone capable of avoiding obstacles.

For example, IARC Mission 5 [26] asks competitors to fly through many windows and corridors. Our algorithm is capable of doing this and can be implemented almost directly as part of the solution to this task.

Chapter 7

Discussion

In this chapter we will address what challenges we have faced, what we have learned and how this thesis is relevant for the current autonomous UAV service market. We will look specifically at our algorithm as a product, but also the combination of algorithm and Group 2s state-of-the-art web-platform.

7.1 Challenges and solutions

As we plunged into the first project that we have encountered of this caliber, we expected to face a range of challenges. In this segment we will not only discuss what obstacles we have encountered, but clarify if and how we have overcome them.

7.1.1 Physical drones

As described in "Redefining the thesis for simulation.", in section 1.2, the first major challenge we faced was the difficulty of testing our product with a physical drone. We would prefer if we could build directly on a physical drone, because this would be an even more important experience for us personally and provide shorter time to market for possible users of our algorithm.

We already had semi-compatible drones through personal investment, but chose not to use them. Acquiring stereo-image camera similar to the one simulated by AirSim is cheap and accessible for a business that wants to utilize this algorithm as a core product. For students with little time to wait for delivery and extremely limited resources, the acquisition of a stereo sensor was another justification to simply simulate the testing of the algorithm.

If we were to redo this project or launch a similar project we should have studied and determined whether we wanted to use a real drone before beginning the development process. The ongoing debate about using our drone or not slowed down the production of the product we have created. This experience can also be used for future software development projects by being more succinct when deciding on a project description and a minimum viable product.

7.1.2 Time management and collaboration

The year this report was written (2020) we were struck with a Coronavirus pandemic. this stopped us from being able to physically work together on the product and demanded an increase in self-discipline for us to achieve our goals. During this time different work ethic, aspirations and motivation resulted in a significant deviation in contribution to both the product and this report.

To make this distinction clear we will present a table over the work done by each participant. This list provides a summary of what chapters have been written by each participant, and implies that the part of the product that is explained is also created by the author of each section.

Participant	**Work**
Marius Sørensen	Chapter 1 Chapter 2 Chapter 3 with the exception of: 3.1.1 & 3.3.2 Chapter 4 Chapter 5 with the exception of: 5.2 Chapter 6 with the exception of: 6.1.5 Chapter 7
Odin Bjørnebo	Section 3.1.1 & 3.3.2 Section 5.2 Section 6.1.5 Appendix A.1

Table 7.1: Work overview

In addition, the direct contributions to the code are available through the Insight tools in our GitHub repository[28].

7.2 Learning experience

Even though the Bachelors thesis is the magnum opus of each third-year student, there is no doubt that we have learned more this semester than every previous semester. Throughout this segment we will explore what we have learned and how this will help us in further development and studies.

7.2.1 Learning and planning before developing

In each project we have worked with through our studies we have received a well defined task. The first difference from these projects that we had to work through was identifying our own mission. This process went hand in hand with NU, but we learned that we should have finished this earlier, as mentioned in section 7.1.1.

Once our project description was established and we were ready to start developing, we realized that the starting point for a project like this is radically different from any previous school project. With each school project the aim has been to prove that we understand the use of the technologies or concepts that we have been previously taught. Each hand-in has a specified theme and every problem should be solvable simply from the knowledge we should already possess. The start of this project was the opposite. We had studied the programming and the logical way of solving problems, but the concrete task we needed to solve had no connection to our earlier studies. We needed to learn how to handle images, move drones in a 3D space and interpret LiDAR data. This lead us to look at our objective from a wider perspective. We found that we had to know more about similar approached and solutions before we could start coding our own answer. This lead us into a learning period which lasted for the first 1-2 months of the project. During this time we learned about AirSim, how to handle and interpret images in Python and researched different possible approaches for our algorithm. Working in this way

was very different and therefore unsettling. We got the feeling that we had used close to one third of the available time without any apparent progress.

After we had learned what we felt were enough we went back to trying start programming the algorithm. This time the development process was faster and more structured. Every time we faced a problem we were able to solve it by using the new technologies we had learned and looking to our plans. We followed a step by step list we had created to get from nothing to our goal. At first we created a naive algorithm which was only able to follow a wall. This algorithm was further developed to turn both ways and fly a faster. Then we added the ability to turn towards a point and follow a path, ending with an algorithm close to what we have now. Through fine tuning and testing we changed some of the values our algorithm uses to achieve safe flight. Finally we added the data collection and extraction functions to combine our algorithm with Group 2s platform.

Learning a new work method was not something we had considered would be the first obstacle or learning experience during this thesis. However it is now obvious to us why there are extensive planning phases before every major software development project.

7.3 Wider context of this thesis

Each new technological advancement needs a place in the market or in society to be useful. In this section we will look at how the development of this algorithm compares to the evolution in the market and drone industry.

7.3.1 Autonomous drone market and use cases

The last 5-10 years have shown a rise in sale of UAV technology. We have seen a meteoric rise in the use of commercial photography and video drones. The Federal Aviation Administration (FAA)[29] reports close to a million recreational drone operators in the United States in 2019[30]. The FAA also reports more than 120 thousand commercial pilots as early as January of 2018[31]. All these numbers point to a huge increase in the use of both commercial and recreational drones in the coming decade.

Current businesses, like NU, mainly employ manual drones and pilots. In this thesis we are being a part of the next step. Moving away from the manual labor of flying the drone and analysing the data by hand. We want the technology to work for us, so we do not have to. Some businesses are already making use of technology similar to our algorithm to cut labor costs.

Amazon Prime Air delivers parcels in some US states by using their autonomous drones, SkySpecs uses similar technology to autonomously inspect wind mills[32] and NASA has recently launched a program for drones as a disaster response[33], but how does our solution fit into this industry evolution?

Our algorithm aims to produce the same results as the autonomous obstacle avoidance systems used by major corporations such as Amazon. There is no way for a few bachelor students to compete with the entire research and development department at either Amazon, SkySpecs or NASA. Our offering, however, is a step in the right direction. The problem of secure drone flight is being tackled by a multitude of organizations around the world, but in slightly different ways. Through this thesis, our work for Avium and NUs continuous effort to automate, we are supporting the industry in the best way we can. By competing.

7.3.2 Challenges

The future of the autonomous drone industry is an area of infinite possibilities, but there are always challenges and hurdles for technological advancement. As exemplified by Uber and AirBnB, one of the biggest problems when launching new technology is dealing with government regulations, or the lack thereof[3]. The regulations for drone flight in Norway are well defined with overly cautious rules for RPAS (Remotely Piloted Aircraft Systems). For example a commercial pilot is not allowed to fly within 50 meters of any vehicle, human or building that is not under the operators control. The problem for the further implementation of our algorithm is the distinct lack of regulation for autonomous UAVs.

In order for a drone pilot to be able to fly commercially, the pilot would first require a license from the Norwegian Civil Aviation Authority (CAA Norway). The law is simply not specified for drones that are operated solely by an automated algorithm. There is currently no way we can license our algorithm, as the Norwegian license only applies to pilots. CAA Norway provides no testing utilities for classifying the safety of autonomous drones, like the levels of autonomy for autonomous cars [34].

A lack of regulations slows down the road to technological development. Developing and testing technology that does not have specific regulations will be slowed down. At the same time, services that are already mature enough to serve their function in a safe manner will remain grounded until authorities define when and how they can be used.

7.4 Conclusion

As a conclusion to this thesis, we have successfully developed an obstacle avoidance algorithm for NU and Avium. Our algorithm can reliably circumvent obstacles in any drones path between two points. We also introduced extra data extraction and ease-of-use tools so that NU and Avium can further incorporate this algorithm into their future work. In collaboration with Group 2 we have built a complete platform for the simulation and analysis of drone inspection flights.

In order to achieve our goal we have primarily used a single simulator and built our product from the bottom, with our own hands. We learned how to interpret image data and use this information to safely move a drone without human intervention. We had to learn efficient ways of saving, communicating and using data across two platforms, and finally we had to get an overview of the existing drone industry and figure out how our technology compares and competes.

We are going to be better prepared for our next project of this scale. The preparation and learning process will be the first natural stage. When we plan, we will make concrete decisions about how to work together and how to divide the workload. All things considered, this thesis has been a valued learning experience and resulted in a solid technological development we are proud of.

Bibliography

[1] Parker, J. (2016). *Python : An Introduction to Programming.* Bloomfield: Mercury Learning & Information.

[2] Adams, R., & Essex, C. (2018). *Calculus : A complete course (9th ed.).* Don Mills, Ont: Pearson.

[3] Stone, B. (2018) *The Upstarts: Uber, Airbnb and the Battle for the New Silicon Valley.* Black Swan/Carousel/Corgi

[4] LiDAR basics:
`https://www.neonscience.org/lidar-basics`
(Accessed: 30/04/2020)

[5] GitHub language statistics:
`https://octoverse.github.com/`
(Accessed: 10/02/2020)

[6] Numpy website:
`https://numpy.org/`
(Accessed: 30/04/2020)

[7] Python package popularity:
`https://hugovk.github.io/top-pypi-packages/`
(Accessed: 30/04/2020)

[8] Git website:
`https://git-scm.com/`
(Accessed: 30/04/2020)

[9] GitHub website:
`https://github.com/`
(Accessed: 30/04/2020)

[10] Trello website:
`https://trello.com/`
(Accessed: 30/04/2020)

[11] Unreal Engine website:
`https://unrealengine.com/`
(Accessed: 30/04/2020)

[12] AirSim Python client examples:
`https://github.com/microsoft/AirSim/tree/master/PythonClient`
(Accessed: 29/04/2020)

[13] Github:AirSim/ReadME.md:
`https://github.com/Microsoft/AirSim/blob/master/README.md`
(Accessed: 10/02/2020)

[14] AirSim documentation:
`https://microsoft.github.io/AirSim/docs/apis/#vehicle-specific-apis`
(Accessed: 10/02/2020)

[15] AirSim binary files:
`https://github.com/Microsoft/AirSim/releases`
(Accessed: 02/04/2020)

[16] Dji Phantom simulator:
`https://www.dji.com/no/simulator`
(Accessed: 03/02/2020)

[17] Python Documentation, Classes:
`https://docs.python.org/3/tutorial/classes.html`
(Accessed: 07/04/2020)

[18] Python Documentation, argparse:
`https://docs.python.org/3/library/argparse.html`
Accessed: 27/03/2020

[19] Unreal Engine non-game examples:
`https://docs.unrealengine.com/en-US/Resources/ContentExamples/index.html`
(Accessed: 11/02/2020)

[20] Nordic Unmanned website:
`https://nordicunmanned.com/`
(Accessed: 12/02/2020)

[21] Nordic Unmanned website about page:
`https://nordicunmanned.com/about/`
(Accessed: 19/03/2020)

[22] Coordinate Systems:
`https://www.basicairdata.eu/knowledge-center/background-topics/coordinate-system/`
(Accessed: 26/04/2020)

[23] Amazon Prime Air:
`https://www.amazon.com/Amazon-Prime-Air/`
(Accessed: 26/04/2020)

[24] Stavanger Hospital using drones for transportation (in Norwegian):
`https://www.nrk.no/rogaland/na-vil-sjukehusa-frakte-blodprover-med-drone-1.14396158`
(Accessed: 26/04/2020)

[25] International Aerial Robotics Competition:
http://www.aerialroboticscompetition.org/
(Accessed: 30/04/2020)

[26] IARC - Mission 5:
http://www.aerialroboticscompetition.org/mission5.php
(Accessed: 30/04/2020)

[27] Article discussing the use of LiDAR sensors in autonomous cars:
https://www.therobotreport.com/researchers-back-teslas-non-lidar-approach-to-self-driving-cars/
(Accessed: 30/04/2020)

[28] Our GitHub repository for all code related to this thesis:
https://github.com/sorensenmarius/Bachelor
(Accessed: 05/05/2020)

[29] Federal Aviation Administration: https://www.faa.gov/
(Accessed: 06/05/2020)

[30] FAA Unmanned Aircraft Systems aerospace forecast: https://www.faa.gov/data_research/aviation/aerospace_forecasts/media/Unmanned_Aircraft_Systems.pdf
(Accessed: 06/05/2020)

[31] FAA Drone Registry Tops One Million: https://www.transportation.gov/briefing-room/faa-drone-registry-tops-one-million
(Accessed: 06/05/2020)

[32] SkySpecs website: https://skyspecs.com/
(Accessed: 06/05/2020)

[33] Drones for Disaster Response: NASA STEReO Project Kicks Off: https://www.nasa.gov/feature/ames/drones-for-disaster-response-nasa-stereo-project-kicks-off
(Accessed: 06/05/2020)

[34] Levels of autonomy for autonomous cars: https://www.synopsys.com/automotive/autonomous-driving-levels.html *(Accessed: 06/05/2020)*

[35] PID controllers for dummies: https://www.csimn.com/CSI_pages/PIDforDummies.html
(Accessed: 08/05/2020)

[36] Gartner Forecasts Worldwide Public Cloud Revenue to Grow 17% in 2020: https://www.gartner.com/en/newsroom/press-releases/2019-11-13-gartner-forecasts-worldwide-public-cloud-revenue-to-grow-17-percent-in-2020 *(Accessed: 11/05/2020)*

Appendices

Appendix A

AirSim

A.1 Installing AirSim

There are two different ways of using AirSim as of now, one of the options are downloading already compiled maps with all the necessary data for running it in binaries and running them. Then just use a script of your choice to connect with AirSim's API.

The other option is to download the whole program from github, build it in visual studio, run it with unreal's game engine, and then use unreal's editor as the workspace. First step would be to install the **Epic Games Launcher** and then use the launcher to install the 4.18 version of the **Unreal Engine** and check off the box that says *Editor symbols for debugging*. Continue by downloading *Visual Studio 2017* or newer, and include the *Desktop Development with C++* and the *Windows 10 SDK* in the installation or modify an already installed edition.

Now to continue just start up the *Developer Command Promt* for Visual Studio with admin rights and clone the repository, https://github.com/Microsoft/AirSim.git and them change directory to "AirSim" afterwards. In the folder "AirSim" is a file called *build.cmd* and to build the program just run this through the developer command promt. Change the directory to \AirSim\Unreal\Environments\Blocks and run the *update_from_git* file and afterwards just open the .snl-file in visual studio and run it with DebugGame_Editor and Win64. Now it is possible to edit a map and/or environment in the unreal editor and try different programs by connecting to the AirSim API.

A.2 The *.join()* function

The *.join()* function is useful in a lot of cases. As explained in Taking off and moving the drone we will have to use both asynchronous and synchronous functions during this project. An example of a synchronous use case is taking off. In this case we do not want any other function calls to interrupt the drone while it is safely taking off. To achieve this we simply add the *.join()* function call.

When we look at the *.join()* function from this angle it might not be immediately apparent why we need asynchronous function calls at all. However there are a lot of helpful functions in the AirSim API that do not need to take over control of the drone to run. For example we can access cameras or LiDAR sensors while taking off or get the speed, rotation and height of the drone while executing a flying command. During this thesis the *.join()* function will be very helpful, but when collecting data the *.join()* function will be purposefully omitted.