DEVELOPMENT OF AN AUTONOMOUS QUADCOPTER WITH COLLISION AVOIDANCE AND LOCALIZATION CAPABILITIES USING VISION SENSORS

a project presented to

The Faculty of the Department of Aerospace Engineering

San José State University

In partial fulfillment of the requirements for the degree

Master of Science in Aerospace Engineering

by

Aldrich D'silva

May 2020

approved by

Dr. Sean Swei

Faculty Advisor

SAN JOSÉ STATE UNIVERSITY

DEVELOPMENT OF AN AUTONOMOUS QUADCOPTER WITH COLLISION AVOIDANCE AND LOCALIZATION CAPABILITIES USING VISION SENSORS

by

Aldrich Dsilva

APPROVED FOR THE DEPARTMENT OF AEROSPACE ENGINEERING

SAN JOSÉ STATE UNIVERSITY

May 2020

Dr. S. Swei Department of Aerospace Engineering Advisor

ABSTRACT

DEVELOPMENT OF AN AUTONOMOUS QUADCOPTER WITH COLLISION AVOIDANCE AND LOCALIZATION CAPABILITIES USING VISION SENSORS

by Aldrich D'silva

This paper details the process of building a fully autonomous quadcopter with vision sensors used for positioning and obstacle avoidance. This was achieved using open source hardware and software and combines efforts from two of the biggest autopilot projects: Ardupilot and PX4. A simulation containing a quadcopter that is representative of the final design was also created to aid with the development and integration process. The drone was successfully tuned using automated tools and an autonomous mission was performed using the optical flow vision sensor data. Position estimates from the flight were shown to be comparable to a similar flight aided by GPS. Collision-free path planning was also successfully achieved by the platform using stereoscopic sensor data and a companion computer.

Table of Contents

1. Introduction

1.1 Motivation

Between 2012 and 2019 over $2.6 billion has been invested into the commercial drone market. Venture capitalists are pouring millions at a time into this rapidly growing industry with the hopes of multiplying their investments in the near future. Economic models predict that by 2028 the market will grow six-fold compared to its current state[5]. Today, the vast majority of commercial use cases provide a service to the customer that, by traditional means, would either be too expensive or sometimes even impossible to achieve. These include inspections, surveillance, photography, film-making, data transmission, delivery, surveying and mapping. A large majority of these services are provided with a human in the loop, actively controlling the drone. Since this raises the cost of the service provided, automation of these tasks is a topic of interest for many robotics companies. The biggest barrier to achieving this in a commercial setting is the legislation surrounding drone operations.

On August 29, 2016, the Federal Aviation Administration (FAA) announced that the commercial use of small Unmanned Aerial Vehicles (sUAS) will be regulated by the Part 107 rules. Most of these rules conflict with autonomous operations, but by far the most prohibitive one is 107.31, which states that the sUAS must be operated within the visual line-of-sight of the operator[11]. Like all of the FAA's legislation, the focus is always safety. Although the FAA does grant waivers based on special use cases with sufficient risk-assessment and fail safes in place, these waivers are extremely hard to obtain and have a high rejection rate[16]. To date there are less than 50 waivers issued for part 107.31[11]. With this information, one can deduce that, although the autonomous capabilities might be present today, adequate failsafes and redundancies are not in place for the FAA to feel comfortable to grant these waivers.

The biggest fear of the FAA is for a sUAS to lose control and collide with a manned aircraft and result in a fatal accident. To protect against this many systems are equipped with redundant sensors and have fail-safe logic preprogrammed into the flight controller should anything go wrong. As an example, the popular open-source flight controller, the pixhawk, is capable of supporting and fusing multiple GPS modules and IMUs to protect against a single point of failure. Although this creates hardware redundancy, it does not protect against signal degredation or loss. Hence it is critical that the redundant systems operate on different principles but provide the same function so that the system is protected again hardware and signal failure.

Furthermore, since most commercially available drones are built around the idea of a human operating it, most drones do not possess any collision avoidance capabilities. This project will aim at addressing both these needs by using optical sensors and an on-board computer.

1.2 Literature Review

1.2.1 Quadcopter Dynamics

A quadcopter has 4 motors and 4 propellers that are mounted on a frame that positions in an equi-distant manner relative to each other. The two common types of frames for quadcopters are the X and the H frames. The quadcopter has 6 degrees of freedom and is able to control its attitude and position by varying the relative and absolute torques of its individual motors. In order to pitch and roll, the two motors farthest away from the desired direction of pitch or roll are spun up. To yaw the motors spinning in the same direction as that of the desired yaw direction are spun up while the other two are spun down thus creating a net torque which rotates the aircraft. Figure 1 shows how a quadcopter pitches/rolls and translates in a desired direction, while Figure 2 shows how the system changes heading.

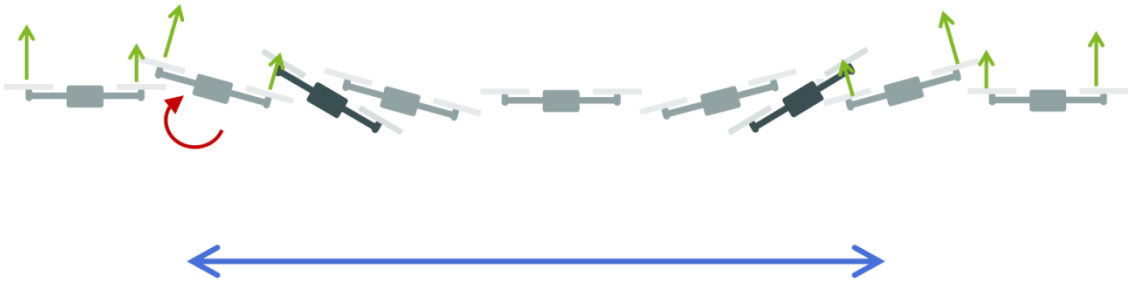

Figure 1. Quadcopter changing the relative speeds of its motors to roll and translate[12]

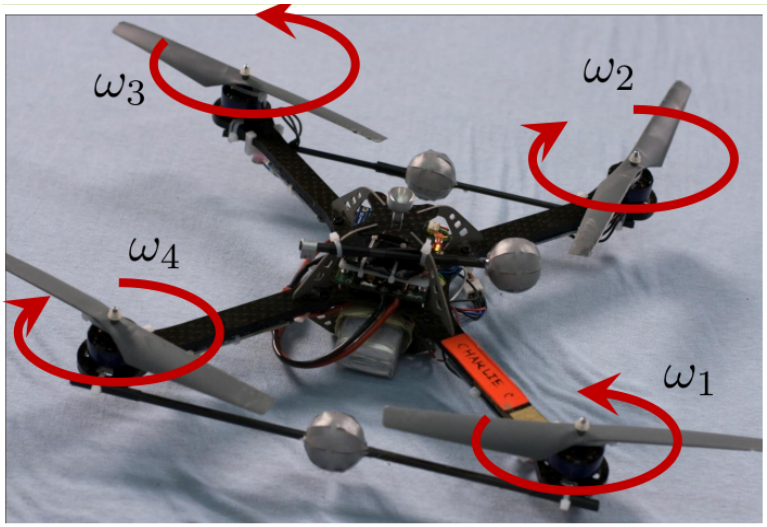

Figure 2. Quadcopter changing the relative speeds of its motors to yaw [12]

ψ, θ and φ are the the rotation angles that translate the inertial coordinate frame to the body coordinate frame of the quadcopter. The symbols u, v and w are the velocities in the x, y and z body frame. These six variables along with the force of gravity and the thrust generated by the quadcopter can be used to describe the linear acceleration in 3 axes using the following relations.

$$\begin{aligned}
\dot{u} &= -g\sin(\theta) + rv - qw \\
\dot{v} &= g\sin(\varphi)\cos(\theta) - ru + pw \\
\dot{w} &= \frac{1}{m}(-F_z) + g\cos(\varphi)\cos(\theta) + qu - pv
\end{aligned} \tag{1.1}$$

The roll, pitch and yaw rates, p, q and r, can be used to represent the angular velocities in the 3 axes of the body frame. Combined with the moments of inertia of the quadcopter and the moments induced by each motor, the rates in body frame can be written as the following.

$$\begin{aligned}
\dot{p} &= \frac{1}{I_{xx}}\left(L + (I_{yy} - I_{zz})qr\right) \\
\dot{q} &= \frac{1}{I_{yy}}\left(M + (I_{zz} - I_{xx})pr\right) \\
\dot{r} &= \frac{1}{I_{zz}}\left(N + (I_{xx} - I_{yy})pq\right)
\end{aligned} \tag{1.2}$$

The Euler angle derivatives and the translational velocities in the earth frame can be expressed in terms of all the variables above using the following relationships:

$$\begin{aligned}
\dot{\varphi} &= p + (q\sin(\varphi) + r\cos(\varphi))\tan(\theta) \\
\dot{\theta} &= q\cos\varphi - r\sin\varphi \\
\dot{\psi} &= (q\cos\varphi + r\cos\varphi)\sec\theta \\
\dot{x^E} &= c_\theta c_\varphi u^b + (-c_\varphi s_\psi + s_\varphi s_\theta c_\psi)v^b + (s_\varphi s_\psi + c_\varphi s_\theta c_\psi)w^b \\
\dot{y^E} &= c_\theta s_\varphi u^b + (-c_\varphi c_\psi + s_\varphi s_\theta s_\psi)v^b + (-s_\varphi c_\psi + c_\varphi s_\theta c_\psi)w^b \\
\dot{h^E} &= -1 * (-s_\theta u^b + s_\varphi c_\theta v^b + c_\varphi c_\theta w^b)
\end{aligned} \tag{1.3, 1.4}$$

Where x, y and h represent the translation of the system in 3-dimensional space and u, v and w represent the velocities. The superscript ‘E’ denotes the Earth reference frame while ‘b” represents the body frame. A detailed derivation of the equations of motions can be accessed on Tytler’s page cited in the References section.

1.2.2 Control Algorithms

A quadcopter being inherently underactuated requires the implementation of a control algorithm to stabilize it. The journal article by Kim entitled A Comprehensive Survey of Control Strategies for Autonomous Quadrotors was researched to stay up-to-date with the latest control efforts and gauge the feasibility of incorporating them in this project. In article Kim divides the groups the different control methods into three main categories and they are as follows:

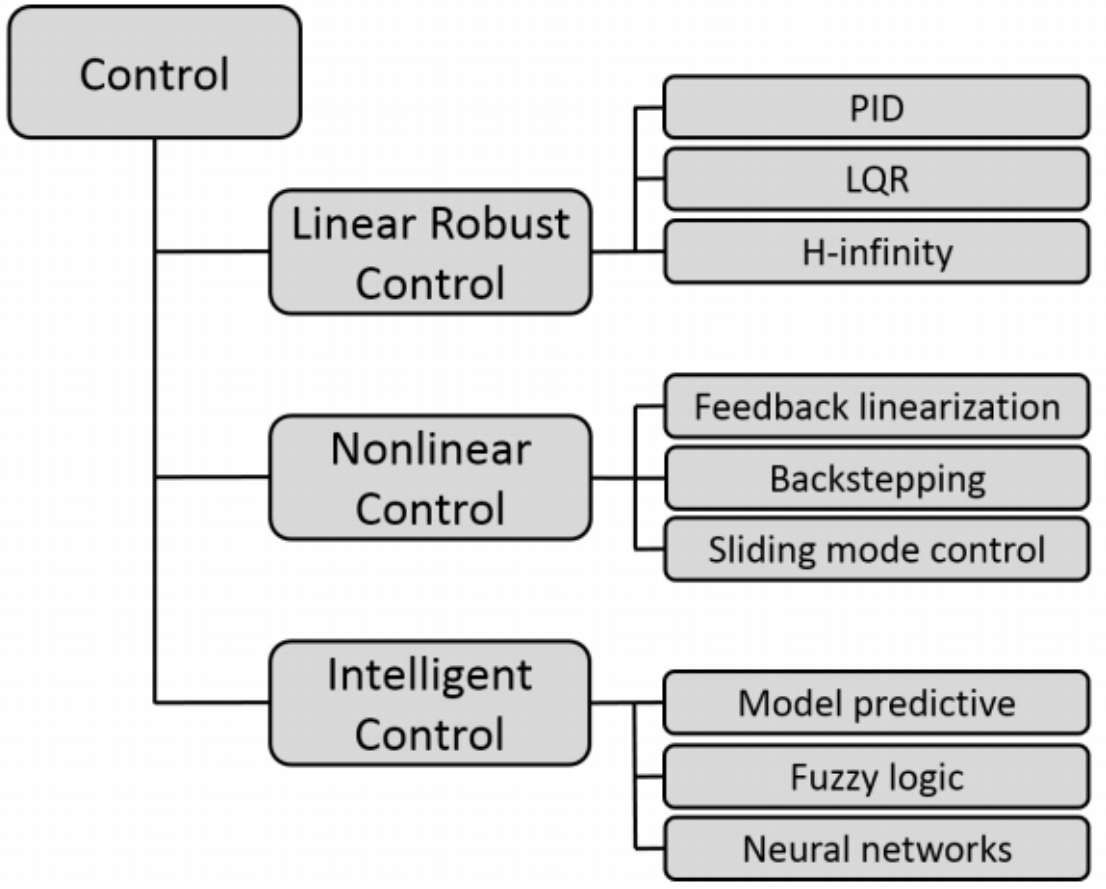

Figure 3. Different control systems that can be used to stabilize a quadcopter [2]

With the recent breakthroughs in AI and machine learning, the application of neural networks in controls is a hot research topic. Believed by many to be the control architecture of the future, the implementation of such a controller requires no tuning and the control gains are automatically optimized due to the architecture of the control law. Non-linear control is another modern approach at the classical problem of stabilizing a multirotor. The implementation of PID control is a topic that has been approached from every possible angle. Hence all the nuances that come with implementing such a controller is known and well documented. [2] Using Ogata's book

entitled *Modern control engineering* [2] one can get a deeper understanding of how PID control works.

Feedback control is the process of evaluating the error between the current and desired state and correcting it by creating an actuated response to compensate for the error. In order to control a dynamic system efficiently, accurate measurements need to be made, the right type of controller needs to be used and an accurate model of the plant's dynamics needs to be created. A controller that has proven to be robust for an application like this is the PID controller. The P, I and D stands for proportional, integral and derivative gain. The role of the PID controller is to minimize the error between the measured state and the desired state by providing inputs to achieve a specified performance metric. The error can be minimized by adjusting the control inputs in every loop.

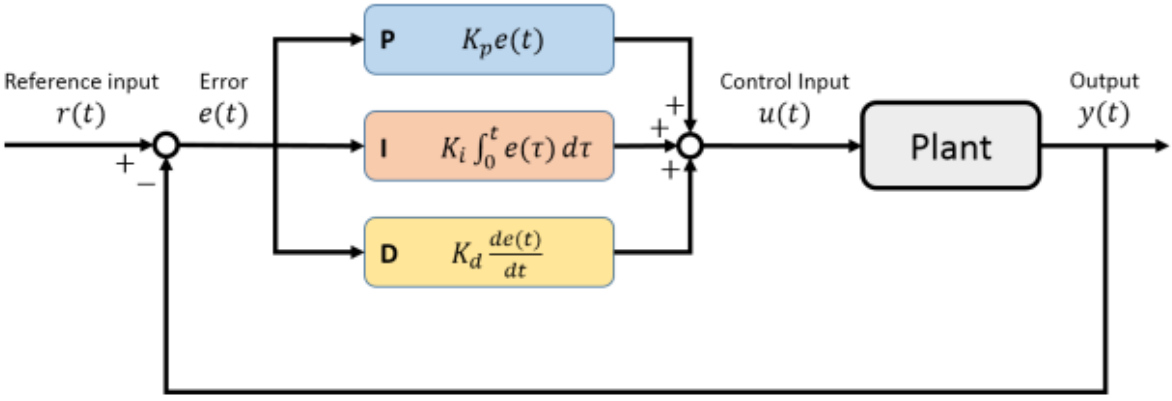

Figure 4. Block diagram of a PID controller in a feedback loop [2]

Performance metrics can be met by tuning the P, I and D gains. The 'P' term focus on minimizing the current error while the 'D' term reacts to the predicted future error. The 'I' term is the accumulation of the past input. A PID controller sums up the results of these three calculations based on the initial state and outputs the actuating signal. Nested control loops like the one depicted in Figure 4 can be used to control the UAV's position and attitude. This process is known as successive loop closure, where minimizing the error of the innermost loop is prioritized. The block diagram of such a system is shown in the following figure. The symbols r_1, r_2 and r_3 below are synonymous with x, y and h in Eq. (10) – (12). Figure 6 shows how the sensor inputs and motor outputs are related to the block diagram in Figure 5.

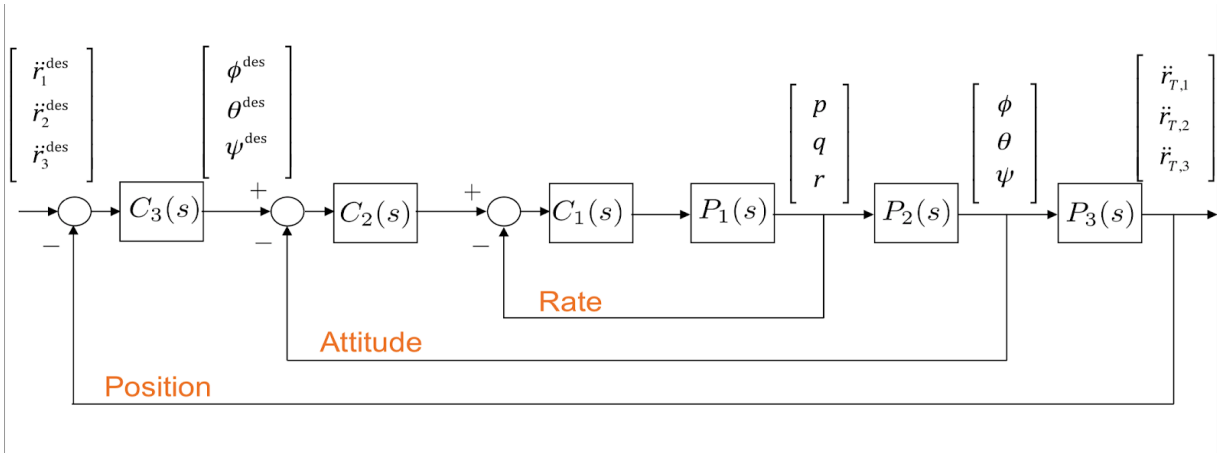

Figure 5. Block diagram of successive loop closure

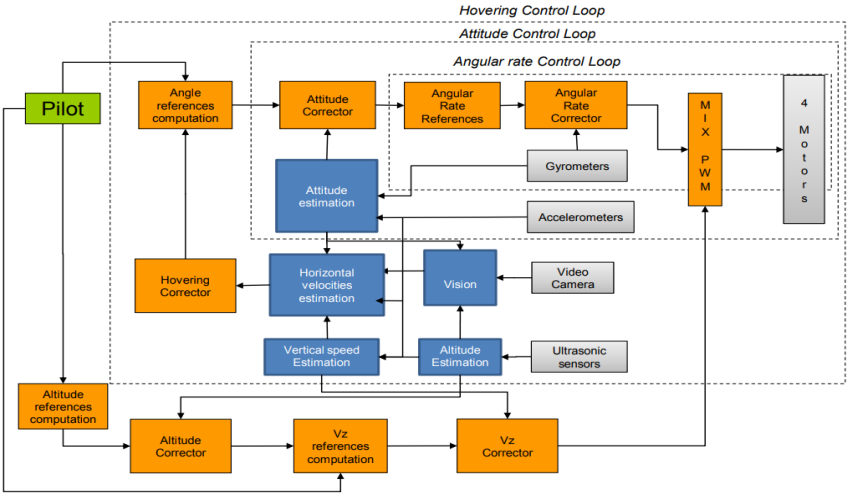

Figure 5. Block diagram of successive loop closure [17]

In the paper entitled *Development of a Low—Cost Experimental Quadcopter Testbed Using an Arduino Controller for Video Surveillance* Turkoglu and Ji put this theory to practice by designing and building a quadcopter and stabilizing with adequate P, I and D gains. The equations of motions for a generic quadcopter were used and the known constants that were unique to his aircraft were substituted in. The derived EOMs are modeled using simulink and are linearized with MATLAB around the hover condition. A state space representation of the actual system was used in simulation to find ball-park values for the P, I and D gains. System identification is a process where measured user inputs (commanded by using the transmitter) are mapped to the systems corresponding outputs. Since it is nearly impossible to accurately model a real system, the mathematical approach known as system identification is used to empirically create an accurate model of the physical system.employs the method of system identification to generate the transfer functions. Once the input/output relations were known, Turkoglu and Ji were able to successfully build a control system capable of stabilizing his quadcopter[13].

1.2.3 Path Planning Algorithms

Algorithms for path planning and collision avoidance fall under two main approaches; the global approach and the local approach. In the global approach, the entire map is known beforehand (or stored as more of it is uncovered) and the optimal path to the target is generated. In a local approach, the data that information that is immediately available to the sensor is used to plan a trajectory and avoid collisions. Consequently, it is not as efficient as the latter but the advantage is that the approach is light-weight computationally [1].

The A* algorithm is intended to be used in situations where the obstacles between the starting point and the end are known and are static. The algorithm divides the environment into equally sized nodes (usually squares), computes all possible paths to the target, then executes the shortest one. The A* algorithm is as follows:

$$f(v)=h(v)+g(v) \tag{13}$$

Here *g(v)* is the distance of the path to the goal from the initial state and *h(v)* is the heuristic distance of the cell relative to the goal. The path that produces the smallest *f(v)* is considered to be the most efficient. At the center of each cell the cost of moving up, down, left or right and diagonally across is computed and another node of that trajectory is computed. For example, below 14 is the initial state and 35 is the goal. From 14 it is possible to go to 7, 8, 9, 13, 15, 19, 20 and 21. The cells 15, 21 and 20 are favored since they move closer to the goal of 35.

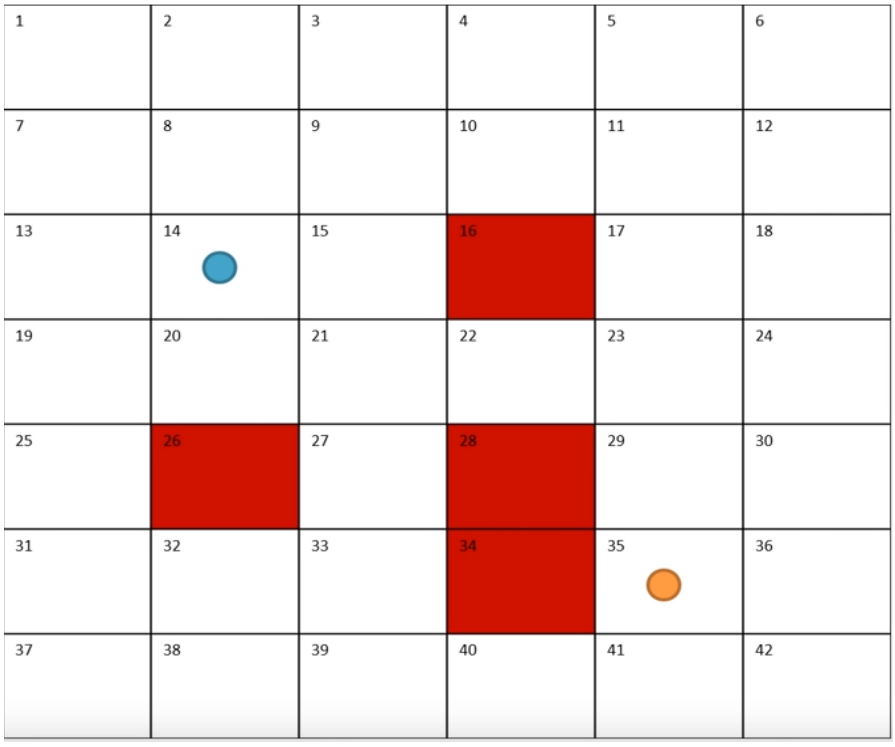

Figure 6. An example of an A* graph[4]

This is repeated until the goal is reached. The corresponding waypoints of the planned path generated by the algorithm are all at the center of the individual cells. Since movement is allowed in only 45° increments from a given cell, the resulting path may not be the most efficient. The following image depicts this flaw of A* and the advantage of its modified counterparts.

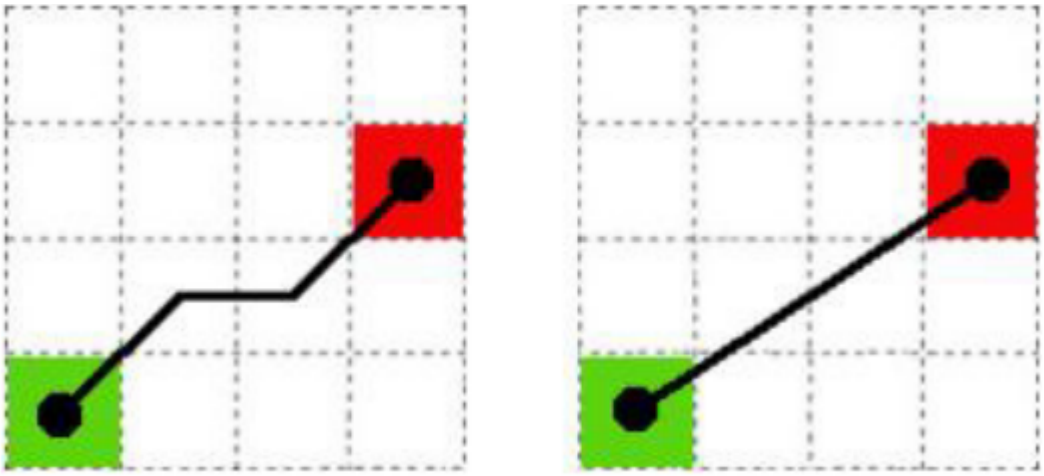

Figure 7. The solution of A* (left) and a modified version (right)[4]

Duchon et al. created a simulation where pros and cons of the A* algorithm and its modifications are studied. The resulting efficiency of the path is compared to the demanded compute time. The study explores the variant known as Basic Theta* documented by Nash et al. This algorithm searches in all angles instead of 45° increments and takes into account how often the robot needs to change its orientation and factors it into the cost. Naturally, this significantly increases compute time depending on the angle increments chosen. A further modification of the Basic Theta*, known as Phi* is also compared. Nash states that the advantage of this method is that the dynamics of the robot are factored into the solution, hence increasing the chance that the resulting trajectory will be achievable by the robot.

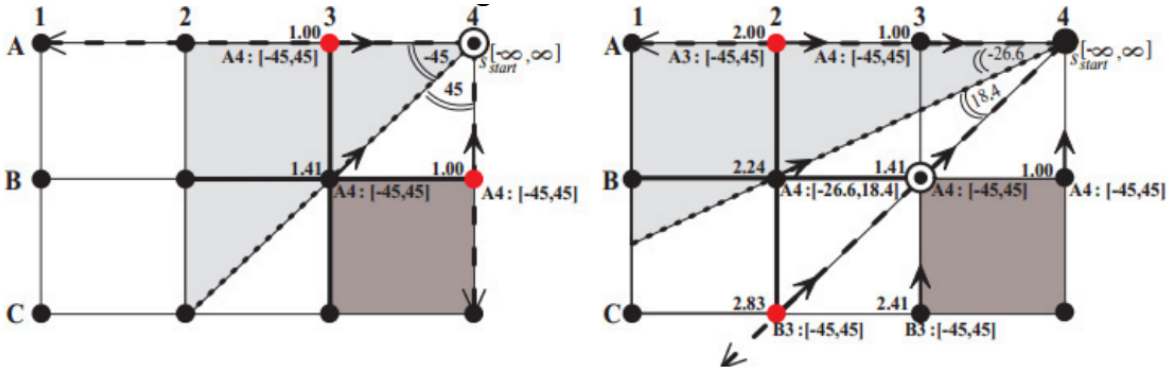

Figure 8. The solution generated by A* (left) and Phi* (right)[4]

Phi* produces two lines from the starting point and every intermediate node and these lines are representative of the physical limitations of the robot[4]. In Figure 8 since A* does not factor this

piece of information into its solution, the resulting trajectory requires the robot move at its physical limit leaving no room for error. Phi* star, on the other hand, does account for this and is able to generate a longer but safer path.

The following table shows the length of the path generated using the method along with the compute time in parenthesis.

Table 1. Comparison of various 2D path planning algorithms by Nash et al.[3]

		FD*	Basic Theta*	AP Theta*	Shortest Paths	A*	A* PS
100 × 100	Game Maps	41.98 (0.0126)	41.92 (0.0063)	42.01 (0.0070)	41.89 (0.6490)	43.80 (0.0029)	42.00 (0.0060)
	Random 0%	51.88 (0.0109)	51.80 (0.0026)	51.80 (0.0034)	51.80 (0.0020)	54.63 (0.0015)	51.80 (0.0057)
	Random 5%	48.83 (0.0097)	48.74 (0.0022)	48.74 (0.0038)	48.69 (0.0311)	51.24 (0.0013)	48.99 (0.0048)
	Random 10%	50.64 (0.0120)	50.53 (0.0028)	50.54 (0.0051)	50.45 (0.1173)	53.11 (0.0014)	50.91 (0.0054)
	Random 20%	48.65 (0.0135)	48.54 (0.0034)	48.55 (0.0065)	48.43 (0.4594)	50.86 (0.0019)	49.04 (0.0054)
	Random 30%	50.19 (0.0153)	50.10 (0.0045)	50.11 (0.0081)	49.98 (1.0769)	52.25 (0.0028)	50.61 (0.0058)
500 × 500	Game Maps	205.60 (0.1916)	205.28 (0.0988)	206.20 (0.1624)	N/A	214.80 (0.0661)	205.64 (0.1040)
	Random 0%	259.65 (0.1231)	259.24 (0.0288)	259.24 (0.0113)	N/A	273.11 (0.0045)	259.24 (0.1688)
	Random 5%	257.19 (0.1538)	256.58 (0.0390)	256.60 (0.0523)	N/A	270.40 (0.0053)	259.14 (0.1747)
	Random 10%	259.37 (0.1795)	258.62 (0.0577)	258.65 (0.0870)	N/A	271.77 (0.0108)	261.62 (0.1783)
	Random 20%	258.71 (0.2219)	257.88 (0.0882)	257.93 (0.1384)	N/A	270.60 (0.0273)	261.36 (0.1871)
	Random 30%	266.49 (0.3207)	265.84 (0.1244)	265.90 (0.2000)	N/A	277.57 (0.0628)	269.60 (0.1951)

Table 2. Comparison of various 2D path planning algorithms[4]

Algorithm	Computational time [ms]	Length of path [cells]	Examined cells [cells]
A*	447	595,2670	69240
Basic Theta*	3518	581,9875	76368
Phi*	4473	584,1820	75712
JPS (A*)	34	595,2670	1248

Both studies found that the variants of A* increased compute time from anywhere between 2x to 7x but did not produce paths that were proportionally equivalent.

The 3D Vector Field Histogram Star (3DVFH*) algorithm is the implementation of the A* algorithm in a 3D space represented by the octomap framework and optimized by the 3DVFH+ local planning algorithm. For a map to be usable and practical, it must contain enough detail so that a safe path can be planned but still be coarse enough to keep computation times low. The octomap framework is employed due to its ability to do both. Hornung et al. co-authored a journal article that elaborates on how this framework is able to get the best of both worlds. The region over which the trajectory needs to be calculated is divided into equal sized cubes known as voxels. If any part of the obstacle, no matter how small, is inside a unit voxel, that voxel is considered to be occupied. The resulting representation has a lower resolution but captures the adequate amount of information required to plan a safe path.

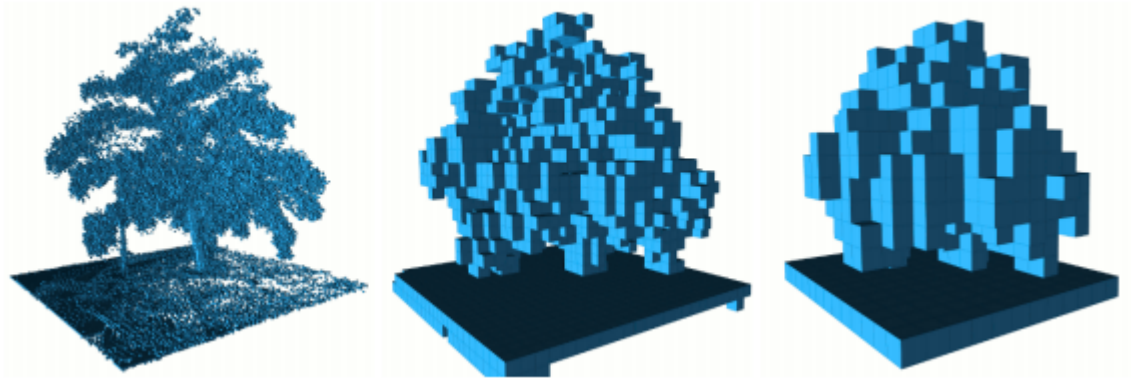

Figure 9. Comparison of various 2D path planning algorithms[6]

Another journal article published in 1998 co-authored by Ulrich and Borenstien discusses the VFH and VFH+ algorithms, which serve as the basis for the 3-dimensional implementation. While A* is aimed at generating a global solution with respect to known static obstacles, the VFH planner is aimed at generating shorter trajectories quickly in order to plan paths around dynamic obstacles. The algorithm generates a trajectory based on the obstacles identified within the field of view (FOV) of the robot's sensors. The resulting trajectory is small and once executed another one is generated with the newly obtained data. VFH+ limits the number of possible solutions by accounting for the vehicle's dynamics. The following image shows the difference between the two approaches[15].

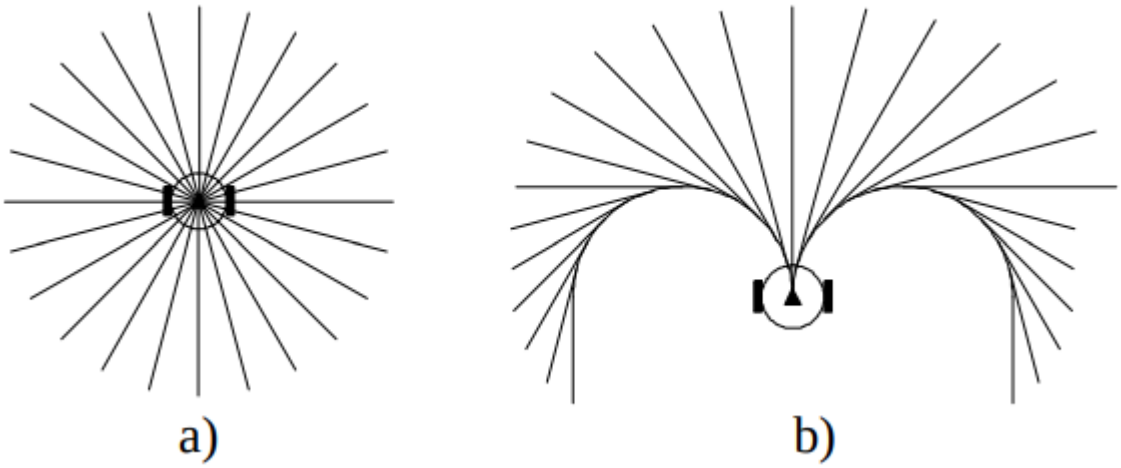

Figure 10. Comparison of various 2D path planning algorithms[15]

Notice how the solutions of VHF do not factor in the heading of the robot and the system is treated as a point instead of a rigid body. On the other hand, in the case of VHF+, the heading of the robot is accounted for and the resulting trajectories look very different from that generated by VHF. It is apparent from the figure above that accounting for the orientation of the robot and combining it with its max turn radius, a more realistic path is generated[15].

As the name suggests, 3DVFH+ applies this concept in 3 dimensions to find a safe trajectory around the octomap representation of the environment. Since it is a local planner, in complex maps there is no guarantee that the desired target can in fact be reached. To rectify this, 3DVFH* was conceived, guaranteeing that, if a solution exists, the algorithm will be able to find it. 3DVFH* takes the global path planning abilities of A* and combines it the local path planning abilities of 3DVFH+. If the map is not known beforehand, the VFH+ algorithm is used exclusively while the map is being built. Once there is enough information for an A* solution to exist, it is calculated and executed in conjunction with solutions from the local planning algorithm.

1.2.4 Sensor Options

In order to have collision avoidance capabilities, two of the most commonly used depth sensors were researched. The two sensors of interest are the sweeping LIDAR and the stereoscopic camera.

One alternative to GPS in UAVs, and robotics for that matter, is the scanning LIDAR, which stands for *Light Detection and Ranging*. It works by emitting an encoded beam of light and measuring the time it takes for the light beam to get reflected off of a surface and make it back to the sensor and are capable of generating centimeter-level accuracy. In its simplest form the sensor in mounted rigidly to the robot with a fixed orientation and is used to get distance information relative to the robot in one dimension. This is the most robust form of measuring altitude in a UAV. Mounting the sensor on a servo can give the robot distance information in a 2-dimensional plane by sweeping the sensor around in 360°. Mounting the sensor to two servos, that move it in two perpendicular planes results in a 3-dimensional map.

In the conference paper entitled 'LIDAR-Inertial Integration for UAV Localization and Mapping in Complex Environments' Opromolla et al. explore the possibility of integrating 2-D LIDAR with a UAV. The purpose was to generate accurate maps of the surroundings, especially where a GPS signal was poor or not present at all. The approach utilized a LIDAR/Inertial Odometry algorithm that fused IMU data with the sweeping LIDAR readings. Based on the initial attitude at the start of the trajectory, the IMU data was used along with adjacent scans to generate a map of its surroundings. Using inertial measurements and features in the map generated by the algorithm, position and odometry were obtained. The experiments described in the conference paper involved the team carrying the flight controller, sensor, companion computer and batteries through a maze created by office dividers. The resulting map generated by the mock setup was representative of what the 2.5-D environment it was in looked like. The LIDAR used in the setup was UTM-30LX-EW by Hokuyo, which is a $5,000 sensor[10].

A more cost-effective means of obstacle sensing can be achieved with stereoscopic optical sensors. The research presented by Niu et al. explains how these stereoscopic cameras work. Two RGB cameras are oriented in a way that their respective FOVs cross each other. Since the distance and the angle between the two cameras is known, distance from any object in the field of view of both cameras can be triangulated[8]. Common features identified by both cameras can be isolated and used to form a cluster of pixels known as point clouds to represent the obstacles ahead. This point cloud data can be used to build the octomap of the UAV's surroundings[1]. The following image visualizes the resulting sensor data when viewed from a stereo camera.

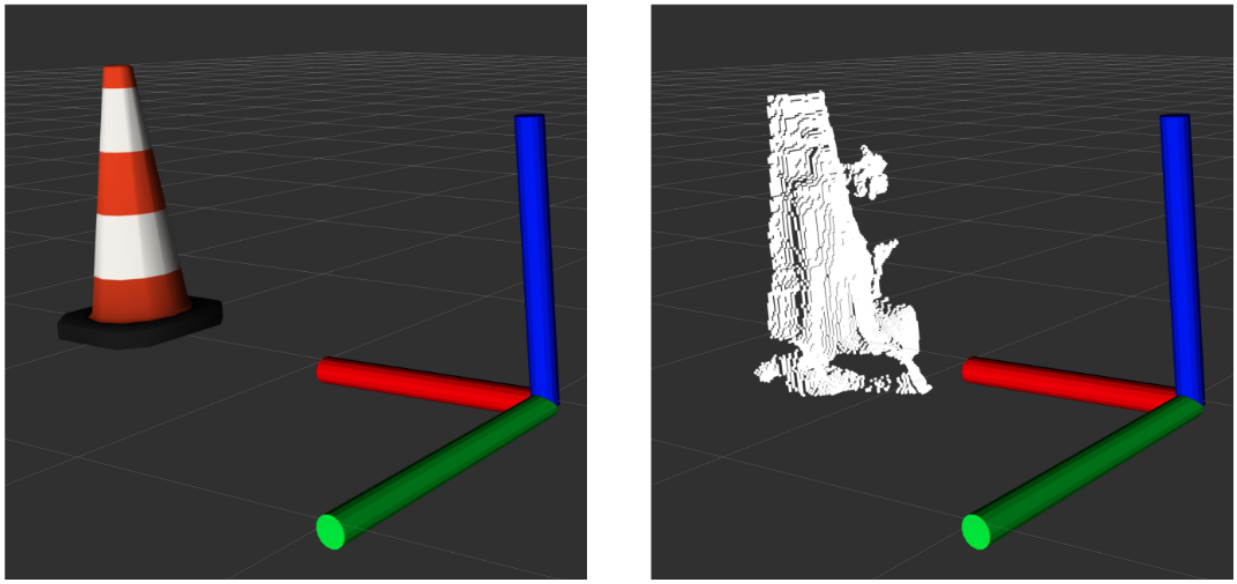

Figure 11. Comparison of various 2D path planning algorithms[1]

The conference paper authored by Hu et al. explores the possibility of using the RealSense R200 depth sensing camera to build a map of the UAV's surroundings. The hardware used is relatively inexpensive and light-weight when compared to that of the LIDAR build. The project utilizes the Jetson TK1 as a companion computer and a pixhawk2 to serve as the flight controller. Unfortunately, the only data that was available for analysis was generated through simulation and the hardware implementation was not completed in time. Nevertheless, the research still provided some useful insights in the implementation of the hardware with a UAV. The team performed various tests using the sensor and determined that stratification is required with the use of stereoscopic cameras. Proposed an approach where the scene in front of the stereoscopic camera was divided into 3 main regions:

A. The first layer: blind area layer. The blind area for RealSense r200 is 0.5 m. All objects in this area have depth value of zero. If the UAV is at this level, it should be hanged immediately.
B. The second layer: obstacle avoidance decision making layer. When the distance between the UAV and the obstacle is in the range of 0.5-2 m, the obstacle avoidance measures shall be taken.
C. The third layer: security layer. When the distance between the UAV and the obstacle is exceeds 2 meters, the UAV is in a safe flight area and can fly freely. [7]

Due to an inherent limitation of the sensor, the distance to any obstacle in the first layer (<0.5m) is not measurable. However, aside from the cases where the starting state is in this arrangement or the motion of a dynamic obstacle results in this arrangement, a properly functioning system should never get to this state. Assuming this is the case, the system needs to be programmed in a way that moves in the direction where all objects are in layer 3, stops when an object appears in layer 2 and plans a path around it to get back to a situation where obstacles are in layer 3.

1.3 Project Proposal

The project will be aimed at making drones safer by introducing sensor redundancy for positioning and building collision avoidance capabilities. To achieve this, an autonomous quadcopter will be built, tuned and programmed. The optical sensors on-board will enhance the capabilities of the system. Optical flow data will be fused into a GPS, IMU and altimiter setup for improved localization. Depth sensor data produced by stereoscopic vision will be used to detect and avoid static and dynamic obstacles. A simulation that is representative of the final system will also be built to serve as a platform to rapidly test code before deploying it on the actual system.

1.4 Methodology

To achieve the goals of this project, a quadcopter was built and tuned, then equipped with the required sensors to have the capabilities of collision avoidance and vision-based localization. Open source projects were leveraged to produce a robust and affordable means of building the system. A downward facing optical flow sensor was used to add to another means of getting positional estimates along with the GPS and IMU modules. A depth sensor was used to detect dynamic and static obstacles. To execute the high-level logic required to control the quadcopter, a companion computer was used to interface with the sensors and flight controller and tie everything together.

2. Hardware

2.1 Hardware Component Selection

2.1.1 Airframe & Power Distribution

Of the various drone configurations, the multicopter is popular due to its ability to stop its momentum instantly, hover and quickly execute collision avoidance maneuvers. Within this family of multirotors, the quadcopter configuration was chosen for its simplicity and robustness. The selection of the QAV500 frame was due to its relatively small footprint, exceptional build quality and modular design. The dimensions of the frame are 452mm x 115mm x 65mm and the frame weighs 543g.The following is an image of the frame.

Figure 12. QAV500 Frame

The sandwich frame design provides two layers to mount components and the spacing allows for airflow to keep components cool. The frame has an embedded power distribution board within the bottom layer which allows for convenient and minimal wiring. The frame is separated into two sections known as the clean and dirty sections. The top two layers shown in Figure 13 make up the clean section while the arms and the carbon fiber plates at the bottom make up the dirty section of the frame. The following image shows just the dirty section by itself.

Figure 13. ‘Dirty’ section of the QAV500 Frame

The clean and dirty sections are connected with silicon bobbins (orange cylinders in Figure 13) that dampen the vibrations created by the propulsion system. This reduces the amount of noise in sensor readings, which results in better state estimation.

2.1.2 Propulsion system

The propulsion system of this UAV is comprised of a set of MN4006 380kV and 12in carbon fiber propellers with 4° of pitch. Each motor is controlled by a 30A Lumenier electronic speed controller (ESC). The quadcopter is powered by two 3 cell lithium polymer batteries in series to create one 6 cell power source. The combined voltage of the battery at full charge is 25.2 V. To estimate the total thrust created by this configuration, the motor manufacturer's data was extrapolated and the results are in table 1 .

Figure 14. Lumenier 30A ESC, T-Motor MN4006 380Kv motor and 12x4 in Carbon Fiber Propellers

Table 3. Propellor tradeoff study

50% throttle				
Prop Length (in)	Pitch (deg)	Power (W)	Thrust (N)	Torque (N*m)
12*	4	32.8	417.5	0.0605
13	4.4	48	544	0.096
14	4.8	57.6	675	0.116
15	5	74.4	805	0.152
16	5.4	88	928	0.187
75% throttle				
12*	4	80.4	835	0.115
13	4.4	122.4	1062	0.172
14	4.8	156	1286	0.222
15	5	199.2	1561	0.285
16	5.4	240	1740	0.343
100% throttle				
12*	4	169.2	1466	0.2125
13	4.4	232.8	1633	0.267
14	4.8	292.8	1975	0.338
15	5	360	2228	0.404
16	5.4	420	2309	0.447

*extrapolated data

The drone was weighed out to be 2kg(4.6lbs). At a 100% throttle, this configuration produces 5.8kgs of thrust, which results in a 2.9:1 thrust-to-weight ratio.

2.1.3 Flight Controller and IMUs

The Pixhawk is an open source hardware project that was created to be easily integrated to create an autonomous robot. The specific version of the Pixhawk controller chosen for the system was the Pixhawk2.1, which features the pixhawk ‘cube’. The cube contains two IMU sensors for redundancy and is separated from the rest of the board to minimize sensor noise and interference in readings. The board features 256 KB RAM, a 32 bit failsafe co-processor, 5 serial ports, 2 CAN ports, 2 power ports, 1 ADC port, and the capability to expand to other connectors and interfaces that are commonly used in robotics projects.

Figure 15. Pixhawk2.1 flight controller with the cube

2.1.4 Sensor Suite

The PX4FLOW optical flow module was chosen to provide position information and odometry. The module combines a 752x480 image sensor with an IMU of its own and an embedded processor to produce precise sensor data.

Figure 16. PX4FLOW module

The optical flow sensor uses altitude data to determine the scale of the features in every frame. The relative movement of features between frames is a measurement of odometry, hence accurate altitude data is needed for the optical flow sensor to work optimally. Laser rangefinder data is fused with the built-in barometer inside the pixhawk. The rangefinder is a time-of-flight sensor that measures the amount of time taken for a beam of light to be emitted, travel to a surface and get reflected back to the sensor. The sensor takes 360 readings in a second, has a range of 50m and has an accuracy of +/- 10cm.

Figure 17. LW20 laser rangefinder

To have obstacle avoidance capabilities, the RealSense D435i sensor was chosen. The camera is part of the line of depth sensing sensors sold by Intel which use the concept of stereoscopic vision to sense objects in its field-of-view. The unit features two image sensors with a resolution of 1280x720, along with a separate dedicated RGB sensor and an infrared projector to produce usable data even in low light conditions. To compute all incoming data in real-time, the sensor has a Intel's D4 Vision Processor integrated into it.

Figure 18. Intel RealSense D435i depth sensor

2.1.5 Companion Computer

The Jetson Nano is a relatively new single board computer developed by Nvidia for robotics and AI applications. Compared to its counterparts, the TX2 and Xavier, the Nano has the lowest power requirements, possess the smallest footprint and is the cheapest of the three. This comes at the cost of reduced performance The system only requires 5V and the computational module is 70x45mm and as thin as the average credit card. The development kit is comprised of the jetson nano, a massive heatsink and interfaces like USB 3.0, gigabit ethernet and general purpose input/output (GPIO). The computer is built for the linux environment and the standard image is based on Ubuntu 18.04.

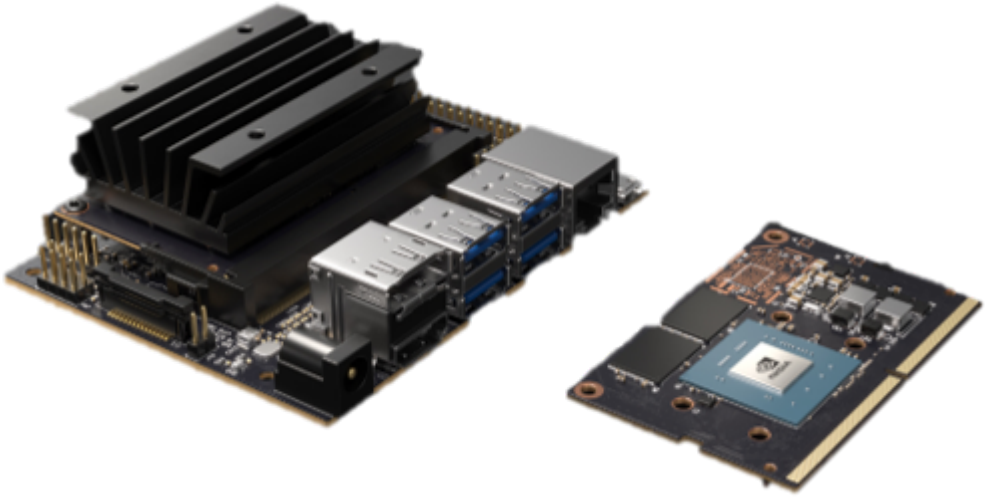

Figure 19. Jetson nano dev kit (left) and jetson nano module (right)

2.1.6 Power

Due to its energy density and ease of handling and integration, Lithium Polymer (LiPo) batteries are commonly used electronics and robotics applications. Unlike the liquid electrolyte in traditional batteries, LiPo batteries have a solid polymer film that serves as the electrolyte for the chemical reaction. This makes it easy to handle and ideal for this use case. Moreover, these types of batteries also possess high gravimetric and volumetric energy densities. The gravimetric energy density is a measure of the energy contained per unit mass while the volumetric energy density is the energy per unit volume. The following figure compares the gravimetric and volumetric energy density of commonly used batteries.

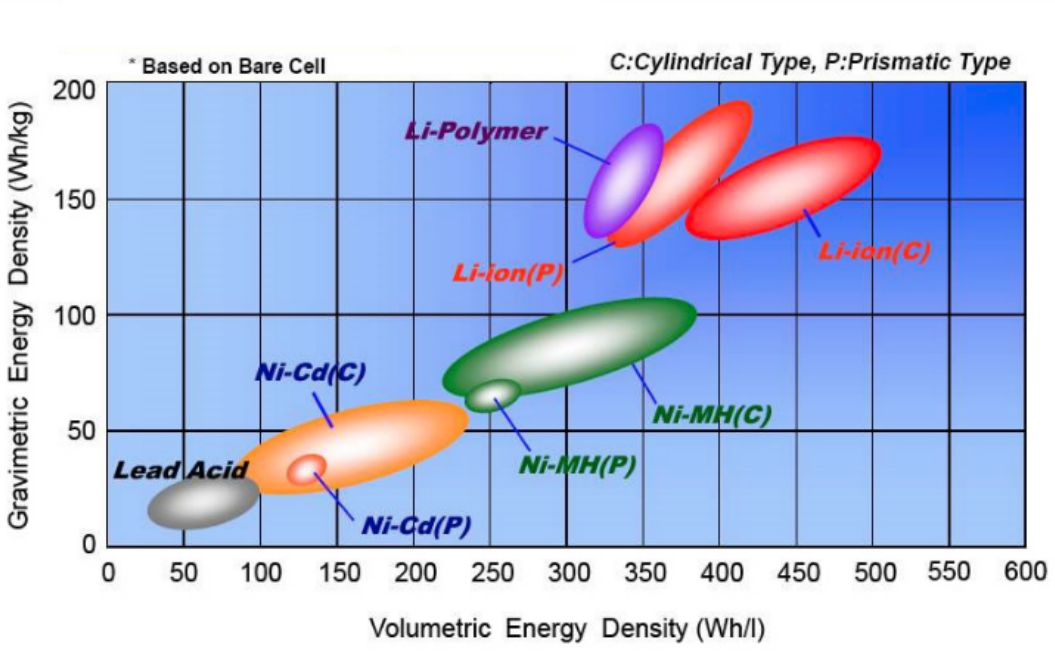

Figure 20. Comparison of volumetric and gravimetric energy density of different battery chemistries

From the figure above it is evident that Lithium based batteries have the highest gravimetric and volumetric energy density among the options. Since LiPo batteries were readily available, they were selected for this project. Two 3 cell, 5400 mAh batteries were connected in series to produce a 6 cell battery with a combined voltage of 25.2 volts. The batteries are Lithium Polymer (LiPo) and have a combined weight of 656 grams.

Figure 21. 3 cell LiPo battery

2.2 Sensor Calibration and Testing

2.2.1 Optical Flow Sensor

The lens of the optical flow module was focused by connecting the module to a PC and streaming the image from the sensor while rotating the threaded lens. A high-contrast, well-lit scene was chosen and the lens was adjusted till a sharp image was visible. The distance between the camera and the object used for focusing was 5m to match the altitude of intended flight.

Figure 22. Calibrated optical flow image

When connected to the flight controller, the sensor measures the average movement of features in the scene in the x and y directions, factors in the resolution of the camera and the distance from the scene, and estimates the relative odometry. The sensor also evaluates the quality of the incoming image and makes a decision on whether or not to trust it. This feature was built in to prevent erroneous readings in low light conditions. All of this data is supplied to the flight controller in the form of OPTICAL_FLOW MAVlink messages. The following shows the format of this message and a description of its payload.

OBJ OBJ

Table 4. OPTICAL_FLOW MAVlink message format and description

Field Name	Type	Units	Description
time_usec	uint64_t	us	Time Stamp
sensor_id	uint8_t		Sensor ID
flow_x	int16_t	dpix	Flow in x-sensor direction
flow_y	int16_t	dpix	Flow in y-sensor direction
flow_comp_m_x	float	m/s	Flow in x-sensor direction, angular-speed compensated
flow_comp_m_y	float	m/s	Flow in y-sensor direction, angular-speed compensated
quality	uint8_t		Optical flow quality / confidence. 0: bad, 255: maximum quality
ground_distance	float	m	Ground distance. Positive value: distance known. Negative value: Unknown distance

2.2.2 Stereoscopic Sensor

The D435i camera did not require any calibration and worked right out of the box. Intel's realsense viewer was used to visualize the incoming data. The following image shows the RGB and corresponding depth image from the sensor. In the depth image, the distance from the objects in the field of view of the sensor is represented in a range of colors from red to blue. In the example below, the two extremes represent 0 and 2m but the sensor is capable of sensing objects up to 4m.

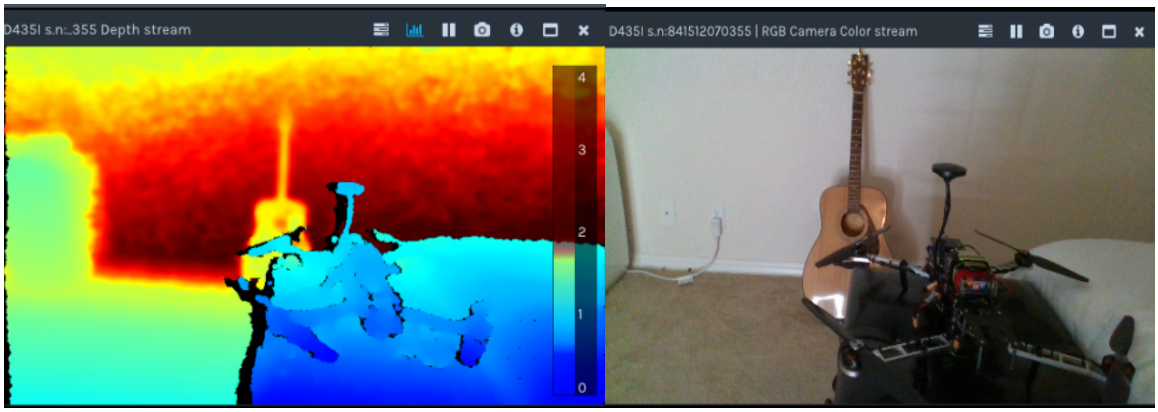

Figure 23. Depth image (left) and RGB image (right) from the D435i camera

Utilizing ROS' librealsense2 package the data can be streamed wireless from the drone a ground station PC. The following image is a rendering of the point cloud generated by the depth sensor.

Figure 24. Point cloud (left) from the D435i sensor and an RGB image taken from a camera phone (right)

On the ground station computer, the incoming frames can be stitched together based on common features and IMU data and a map can be built with this information. This is demonstrated in the figure below where the sensor is moved in a scanning motion around the guitar. The image from a single frame (top left) only shows a portion of the guitar but the combined image (right) shows the complete guitar and part of its surroundings. The yellow dots overlayed on the single frame (bottom left) depicts the features recognized in that frame and these features are used to locate all frames relative to each other.

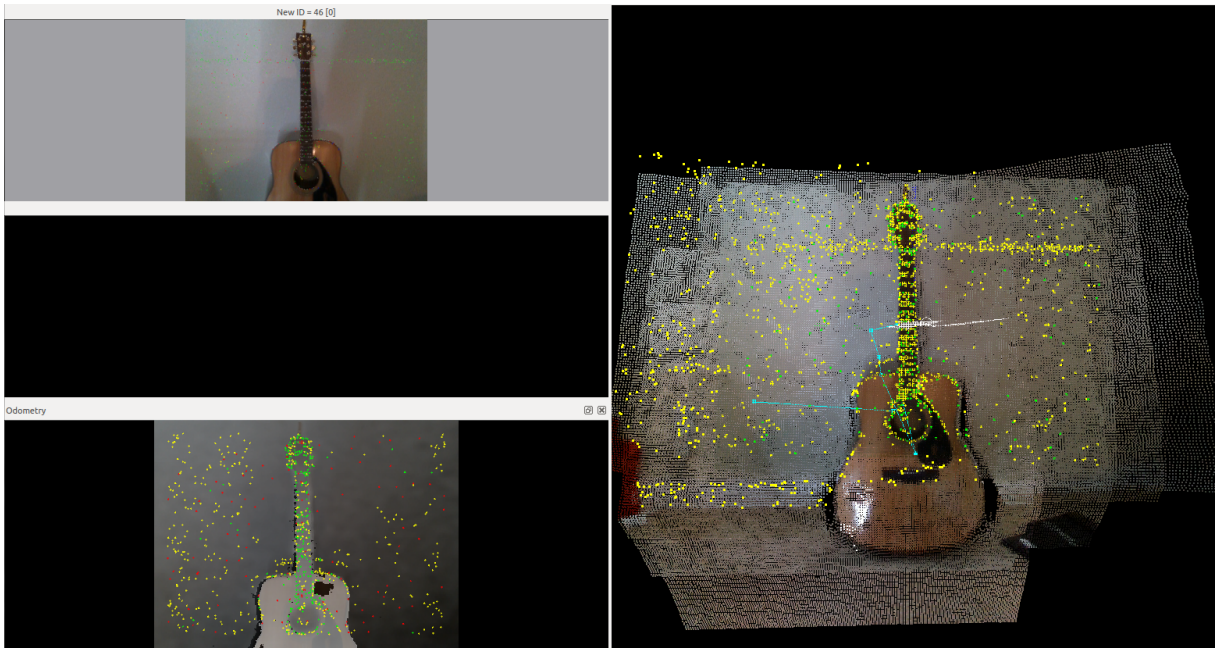

Figure 25. 3D map created from multiple adjacent images of a guitar and IMU data from the D435i camera

3. Software

The ArduCopter and PX4 autopilot projects are two of the most active and mature projects in the open-source robotics community, especially when it comes to drone hardware and software. Both autopilot stacks use the MAVLink protocol to pass messages, are compatible with ROS and are capable of running interchangeably on the same hardware. However, the focus of the two stacks are noticeably different and the choice of picking one over the other is ultimately dependent on the application. It is unclear as to why there has not been much collaboration between the two projects considering the fact that they are both open source and support the same type of internal communication. To explore whether or not combining aspects of the two projects could yeild better results, the decision was made to combine aspects of the two projects together.

Both autopilots implement PID controllers to stabilize the aircraft but currently only ArduCopter has a self-tuning feature. The autotune process, which is outlined later in this report, was the main reason why the ArduCopter firmware was chosen to run on the Pixhawk flight controller. The PX4 documentation hints that tuning aircraft is currently a manual process and takes some trial-and-error to get right. Where the PX4 stack excels in is its implementation of computer vision with UAVs and its simulations. To possess path planning and collision avoidance capabilities the PX4 local planner node was modified and ran on the companion computer. Communication between the local planner and the ArduCopter firmware was achieved with the implementation of ROS; more specifically, the MAVLink implementation of ROS known as mavros. The individual software components are detailed in the following subsections.

3.1 Autopilot

3.1.1 Firmware

ArduCopter was the autopilot software chosen for this project due to its robustness, maturity and ease of customization. It is the multirotor implementation of the ArduPilot project, which is aimed at creating open source software for autonomous robots. Development of the project has been ongoing for over 5 years and the software is used by many businesses to provide professional services. The autopilot software is hardware-agnostic and can be used in conjunction with a companion computers to achieve autonomy. The ArduCopter firmware version 3.6.11 was used to flash the flight controller.

To allow for safe operation, the subset of all the instructions that the software is allowed to receive and execute is limited by the flight mode the system is operating in. These flight modes also determine which sensors are being used by the software for state estimation. This is useful as it gives the user/companion computer the ability to intervene in the event of a sensor failure or if the underlying code has faulty logic. The autopilot allows for pitch, roll, yaw and altitude control to be mapped to PWM (pulse width modulation) inputs of 4 channels of a radio transmitter. Mimicking these inputs via code is also permitted and this process is known as overriding the RCs. This allows the user to program the desired motion based on incoming sensor data. Higher level instructions like taking off, navigating to desired way points and landing autonomously can also be sent by the user or programmed into a companion computer to be executed based on the desired logic. The autopilot also allows for direct integration of optical flow sensors and indirect integration of depth sensors with the help of a companion computer. Sending telemetry, receiving instructions and interfacing with other components connected to the flight controller are all achieved by the autopilot software by using a standardized protocol known as the MAVLink protocol.

3.1.2 Communication Protocol

The MAVLink protocol is a standardized message format adopted by ArduPilot and other projects like it. This lets the components to be shared between projects and even collaboration between drones using different autopilots. The MAVLink message itself is in an XML format but libraries from most of the popular programming languages have already been created for easy translation.

In the context of this project, MAVLink messages will be used to communicate the following things to and from the flight controller:

- Send high-level and low-level instructions to the flight controller
- Relay depth sensor information from the companion computer to the flight controller
- Get odometry from the optical flow data
- Receive telemetry data to evaluate overall system health3.2 Robotics Software

The local planner node gets depth sensor data and drone state information in the form of ROS messages. In the case of the D435i, the node receives data in the form of a pointcloud message to evaluate the obstacles in the field-of-view of the system. The node then uses this information, along with the drone's current position and it desired position to plan a path. The drone's position is obtained from the flight controller via the mavros node and the resulting trajectory is communicated using the SET_POSITION_TARGET_LOCAL_NED MAVLink message. Like the name suggests, the message is a desired position target described in the drone's local coordinate frame and that frame is oriented in the North-East-Down convention. The payload of the message contains either a position, velocity or acceleration target for the flight controller to execute. Both the PX4 and ArduCopter stack support this type of message so the output of local planner node from PX4 can be interpreted by the flight controller running the ArduCopter firmware without needing any translation or conversion. While performing collision avoidance, the PX4 stack only supports position and velocity set points. To keep the implementation in this project simple and safe, only position set points will be provided to the flight controller. The following table describes all the components of this MAVLink message in detail.

3.1.3 Flight Controller Tuning

To achieve stable flight, the control loops of the flight controller needed to be tuned with the ideal gains for the given configuration. The ardupilot flight stack automates this process with a flight mode called ‘Autotune’. This flight mode tunes the PID values of the flight controller by commanding a roll, pitch or yaw rate and recording the maximum rate and bounce back based on IMU readings. If the lean or the yaw angle is exceeded by a predetermined amount or 1 second has passed, a level flight state is commanded. The flight controller tunes the P gain first, followed by I and D gains. The following is an example of how arducopter tunes roll:

1) Invokes 90 deg/sec rate request
2) Records maximum "forward" roll rate and bounce back rate
3) When copter reaches 20 degrees or 1 second has passed, it commands level
4) Tries to keep max rotation rate between 80% ~ 100% of requested rate (90 deg/sec) by adjusting rate P
5) Increases rate D until the bounce back becomes greater than 10% of requested rate (90 deg/sec)
6) Decreases rate D until the bounce back becomes less than 10% of requested rate (90 deg/sec)
7) Increases rate P until the max rotate rate becomes greater than the request rate (90 deg/sec)
8) Invokes a 20 deg angle request on roll or pitch
9) Increases stab P until the maximum angle becomes greater than 110% of the requested angle (20 deg)
10) Decreases stab P by 25%

3.2 Companion Computer

3.2.1 Robotics Software

To interface with the autopilot and achieve the desired behavior, ROS (Robot Operating System) will be used on the companion computer. ROS is a robotics framework that provides tools called ROS packages that can be used to interface with sensors and actuators to achieve a common goal. Similar to the MAVLink protocol, ROS passes sensor information and executes actuator control using ROS messages. This gives the user the opportunity to create their own customized logic and build the required interaction between components.

A ROS system is made up of a ROS master and multiple nodes. The ROS master manages all the nodes and facilitates communication between them. Nodes are tasked with executing some low-level logic and communicating its state to other nodes by publishing to a common topic in the form of a ROS message. Nodes can be categorized into three main groups; publishers, subscribers and both. A publisher reads a sensor state or controls and actuator based on the programmed logic, and publishes this information to a topic. A subscriber gets information from a topic, applies its programmed logic and performs the resulting instructions.

For this project, the ROS package known as mavros will be used to convert MAVLink data into ROS messages to be easily interpreted and vice-versa. The mavros package was originally built as part of another autopilot project called PX4, but since that uses the MAVLink protocol as well, it is usable with reduced functionality with ArduCopter. The librealsense2 ROS package will also be used to convert the depth sensor data into ROS messages for easy transportation and visualization on the ground station. The path planning mechanism that will be developed later in this project will tie the two together so that the drone will be able to sense obstacles and instruct the flight controller to traverse the desired trajectory.

3.2.2 Planning Algorithm

The PX4 local planner node employs the 3DVFH* algorithm and uses the point cloud data from a depth sensor to compute a safe trajectory. The trajectory is broken up into waypoints that are then communicated to the flight controller via the mavros node. Like the name suggests, the local planner node uses only the information that is immediately in the field of view of the depth sensor, along with some data from the immediate past, to plan its path. This keeps computational and memory costs low as the system does not need to store and use the entire map to compute its path. This approach is ideal for systems that perform the computation on an embedded device. The downside to this approach is that a successful path to the goal is not always guaranteed and the system is also susceptible to getting stuck at dead ends, constantly planning paths away and towards it. This is mostly an issue in complex maze-like environments. The flight The PX4 documentation states the maximum speed to be around 3 m/s. Along with computational

limitations, sensor range plays a huge role in how fast an aircraft can move while performing collision avoidance calculations and maneuvers. Most stereoscopic cameras have a sensing range of 10~15 m depending on ambient lighting so 3 m/s is a reasonable speed for the range limitation.

3.3 Simulation

3.3.1 Gazebo

Gazebo is a physics engine developed by the same community that created ROS. It allows for the creation of a sandboxed environment for a ROS system to be simulated and developed in. For this project it was used to simulate the quadcopter, its dynamics and the obstacles in its environment. The sensor data produced from the simulation was reasonably similar to the actual data so served as a first pass at verifying compatibility of software components and data formats.

To simulate the dynamics of the aircraft, Gazebo's lift-to-drag plugin will be used. The plugin uses linear approximations of an aerodynamic body's cross section to estimate the amount of lift and drag created. This approximation is combined with the rotational/translational speed of the simulated body and the aerodynamic forces generated by the robot are simulated in real-time[6].

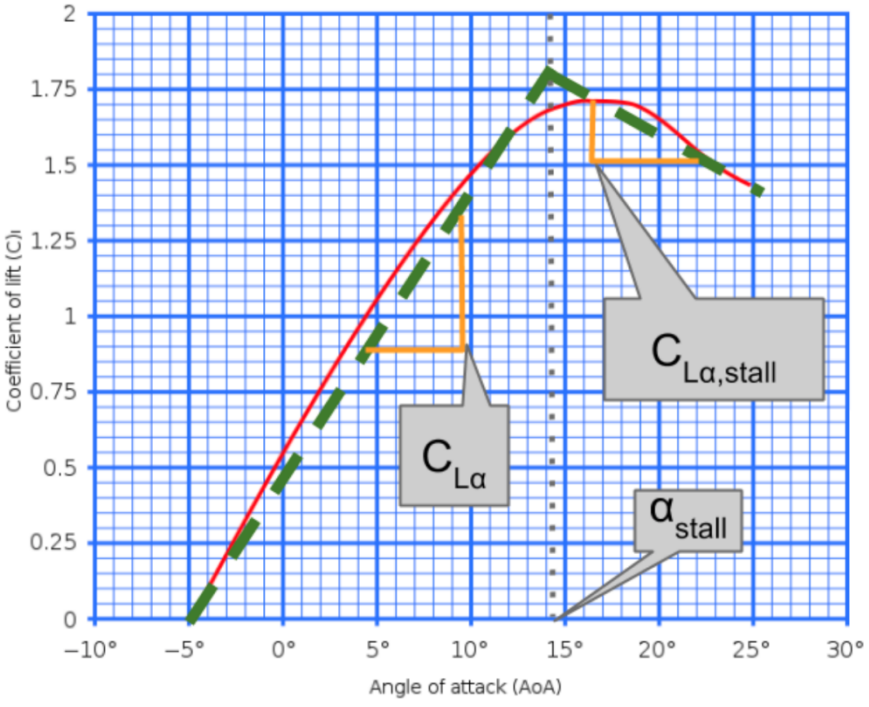

Figure 26. C_L vs AOA of an airfoil. Actual (red) and linear approximation (green)

The linear approximation of the aerodynamic coefficients and the fluid characteristics are all user configurable, making it possible to simulate most simple aerodynamic bodies. In the case of a

spinning rotor blade generating thrust, these parameters can be combined with the rotation speed to calculate the lift generated by each motor. Furthermore, Gazebo provides the ability to add sensors and actuators to the robot to simulate the entire system. The community has already created plugins to produce optical flow and depth sensor data that is representative of what the PX4FLOW and D435i is able to produce so the focus can be placed on using this information as actionable data.

Every gazebo model is comprised of two main aspects: the visual and the collision model. Like the names suggest, the visual model is the more realistic looking of the two as it contains small details, different colors and textures of the object. The collision model is a rough representation of the shape of the object and contains no color or texture information. The model is used to determine if two bodies are interfering with each other so that it can produce the resulting physical response. To optimize the speed of the simulation, the collision model is set to a rough equivalent of the visual.

Figure 27. Collision models of the trees and the terrain depicted in orange

3.3.2 PX4 Avoidance Simulator

The PX4 obstacle avoidance simulator was built and studied to understand the inner workings of the local planner. In the simulation, a generic quadcopter is used to represent the robot navigating its environment. The robot is equipped with gazebo plugins to simulate the data generated by the sensors of the quadcopter while its moving. These plugins are as follows:

- IMU plugin to provide the system with attitude and heading
- Rangefinder plugin to determine the altitude of the drone
- Odometry plugin to simulate optical flow data measurements
- Stereo camera plugin to sense obstacles

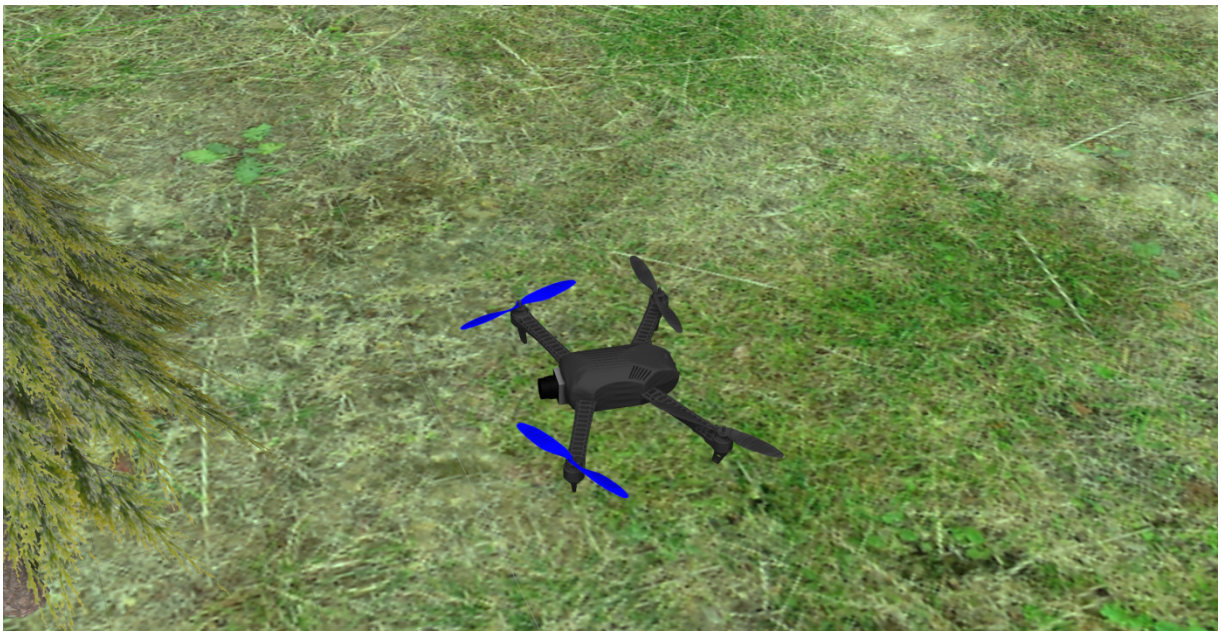

Figure 28. Simulated quadcopter with stereo camera in the front (blue side)

The drone is in a virtual environment with trees that makes navigation challenging and really stresses the drone's path planning and collision avoidance capabilities. The terrain of environment is also irregular and is of varying heights to add more complexity to the environment. The following image shows this irregular terrain and obstacles.

Figure 29. Gazebo simulation environment with irregular terrain and obstacles

Depth data is generated by comparing the frames of two cameras that are placed adjacent to each other on the model. The streams of these cameras, along with the disparity image can be visualized using the gazebo tool image_view.

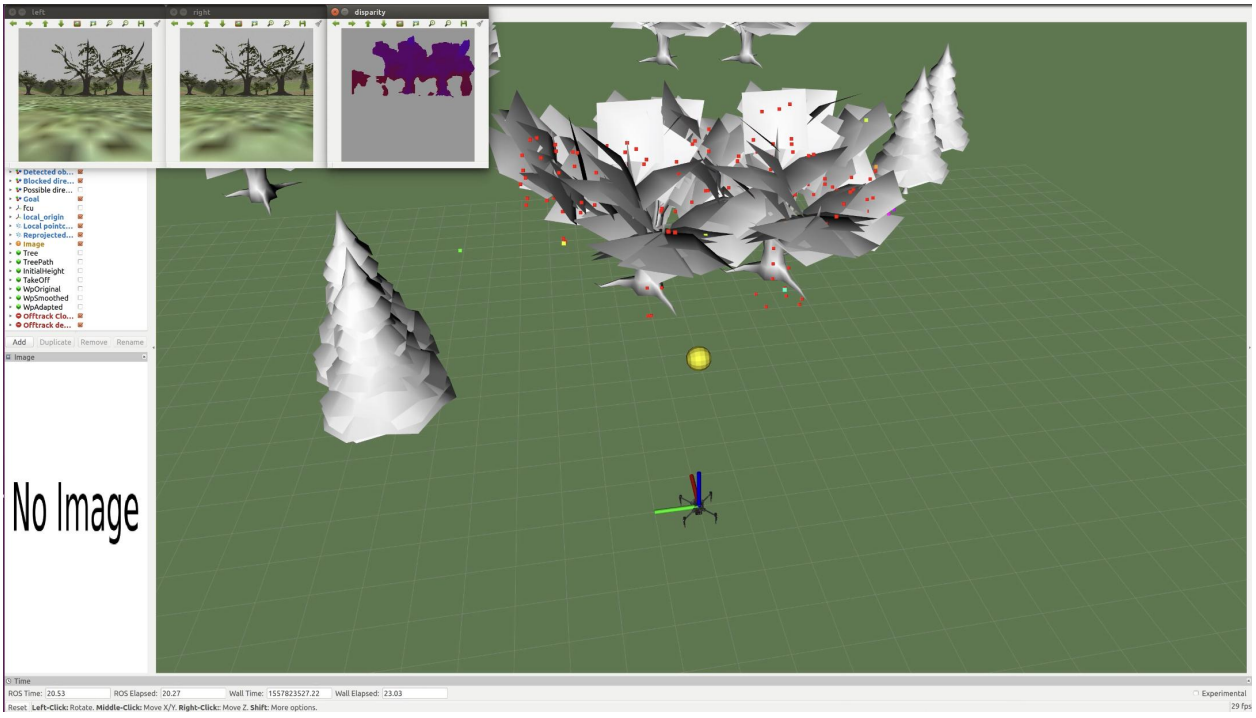

Figure 30. RVIZ rendering of sensor information

The following is an example of what the incoming frames look like, along with a visual representation of the associated disparity.

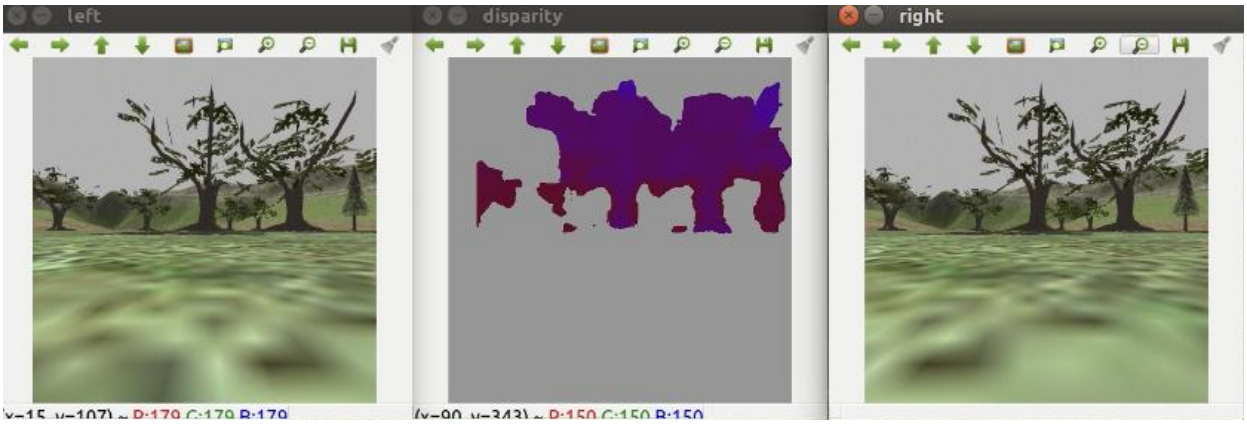

Figure 43. Left and right frames from the stereo camera with their combined disparity

Figure 31. Snapshot of the simulation environment with left and right camera streams

4. Final Design

4.1 Hardware

A simple L-bracket mount was designed and 3D printed to fasten the depth sensor near the front of the frame. A case for the Jetson Nano was also printed to secure the companion computer to the back of the frame. A 2.4 GHz wireless adapter was also connected to the Jetson Nano and setup as a hot spot to connect to the drone and receive telemetry on the ground station.

Figure 33. Top view

Figure 34. Front view

Figure 35. Side view

4.2 Planner Node Modifications

As mentioned earlier in the literature review section 1.2.3 of this report, the 3DVHF+ algorithm factors in vehicle dynamics to build its trajectory so that the resulting solution is achievable by the system. The local planner, which runs the 3DVHF+ algorithm under the hood, learns about the physical limitations of the system by querying flight controller parameters. Unlike the MAVLink messages that the ArduCopter and PX4 stacks share, the flight controller parameters are not interchangeable and require translation for this interaction to work.

When communicating with a PX4 flight controller, the planning node queries the vehicle's position controller parameters to gauge its dynamic capabilities. These parameters are set while tuning the vehicle's response and differ between vehicles. These parameters are prefixed by 'MPC_', which stands for Multicopter Position Controller. This controller is comprised of a P controller for the outer position loop and a PID controller for the inner velocity loop. The following figure represents the controller setup in the form of a block diagram.

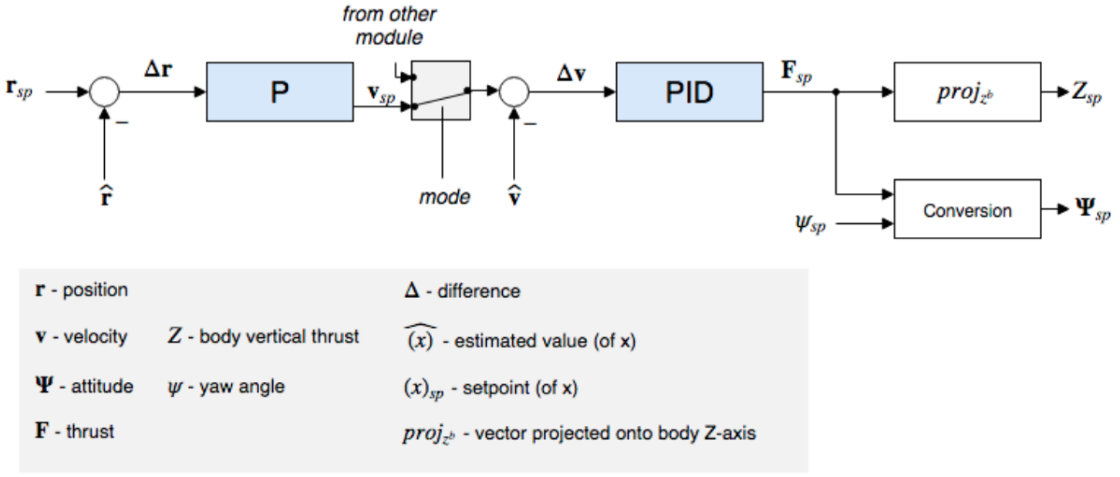

Figure 46. PX4 position controller block diagram

The planning node retrieves parameters that describe the maximum allowed acceleration and velocity in the X/Y plane and in the Z direction. It also factors in user-constrained parameters that indicate how close the vehicle is allowed to get to an obstacle. Two parameters that are unique to the PX4 code base are the MPC_JERK_MIN and MPC_JERK_MAX parameters. These values only come into play when the vehicle is transitioning from velocity-controlled to position-controlled instructions and they determine how aggressive the vehicle's transition is. Further more, these parameters are gated by the MPC_AUTO_MODE setting where a value of 0 turns it off and a value of 1 turns jerk-limited control on. The following table lists out all the queried parameters, their descriptions and the accepted range of values.

Table 6. Description of PX4 parameters requested by local planner node

No	Parameter	Description	Range	Units
1	MPC_ACC_DOWN_MAX	Maximum vertical acceleration in velocity controlled modes down	2.0 ~ 15.0	m/s/s
2	MPC_ACC_HOR	Acceleration for auto and for manual	2.0 ~ 15.0	m/s/s
3	MPC_ACC_UP_MAX	Maximum vertical acceleration in velocity controlled modes upward	2.0 ~ 15.0	m/s/s
4	MPC_AUTO_MODE	Auto sub-mode	0/1	
5	MPC_JERK_MIN	Velocity-based jerk limit	0 ~ 30.0	m/s/s/s
6	MPC_JERK_MAX	Maximum jerk limit	0.5 ~ 500.0	m/s/s/s
7	MPC_LAND_SPEED	Landing descent rate	0.6 ~ NA	m/s
8	MPC_TKO_SPEED	Takeoff climb rate	1.0 ~ 5.0	m/s
9	MPC_XY_CRUISE	Maximum horizontal velocity in mission	3.00 ~ 20.0	m/s
10	MPC_Z_VEL_MAX_DN	Maximum vertical descent velocity	0.5 ~ 4.0	m/s
11	MPC_Z_VEL_MAX_UP	Maximum vertical ascent velocity	0.5 ~ 8.0	m/s
12	CP_DIST	the closest distance that the vehicle can approach the obstacle	-1.0 ~ 15.0	m
13	NAV_ACC_RAD	Acceptance Radius	0.05 ~ 200.0	m

Where applicable, the ArduCopter equivalents of these parameters were found and are listed below.

Table 7. ArduCopter equivalents of requested parameters from the local planner node

No	Parameter	Description	Range	Units
1	WPNAV_ACCEL_Z	Defines the vertical acceleration used during missions	50 ~ 500	cm/s/s
2	WPNAV_ACCEL	Defines the horizontal acceleration used during missions	50 ~ 500	cm/s/s
3	(same as 1)			
7	LAND SPEED	The descent speed for the final stage of landing	30 ~ 200	cm/s
8	WPNAV_SPEED_UP	Defines the speed in which the aircraft will attempt to maintain while climbing during a WP mission	10 ~ 1000	cm/s
9	WPNAV_SPEED	Defines the speed in cm/s which the aircraft will attempt to maintain horizontally during a WP mission	20 ~ 2000	cm/s
10	WPNAV_SPEED_DN	Defines the speed in cm/s which the aircraft will attempt to maintain while descending during a WP mission	10 ~ 500	cm/s
11	(same as 8)			
12	AVOID_DIST_MAX	Distance from object at which obstacle avoidance will begin in non-GPS modes	1 ~ 30	m
13	N/A			

Upon comparing tables 6 and 7, it was evident that the two code bases preferred different units and a scaling factor would be necessary for communication between the two to make sense. The local planner code was modified to request the ArduCopter version of the parameters and apply the appropriate scaling factor, after which it is sent to the planning node.

5. Results

5.1 Tuning

The following figures show the desired and measured roll, pitch and yaw from the Autotune flight performed.

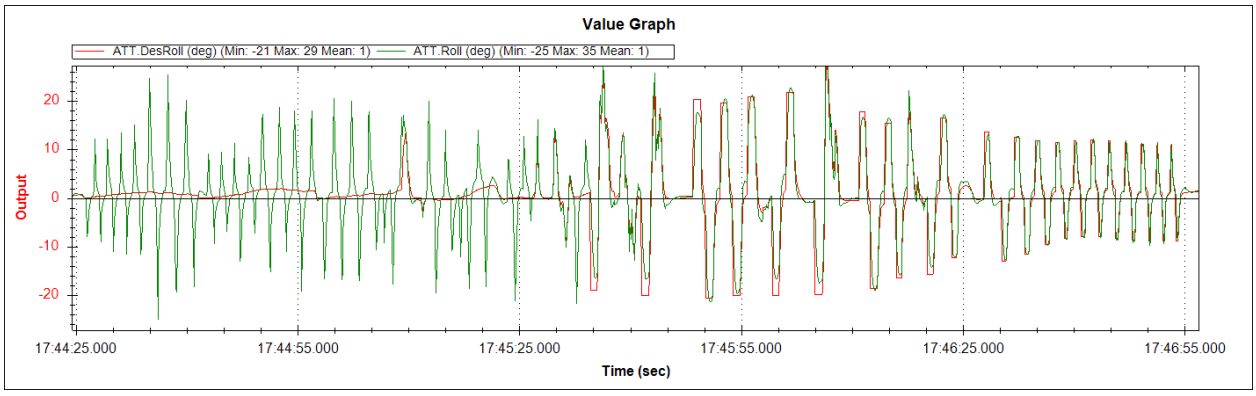

Figure 36. Desired vs measured roll

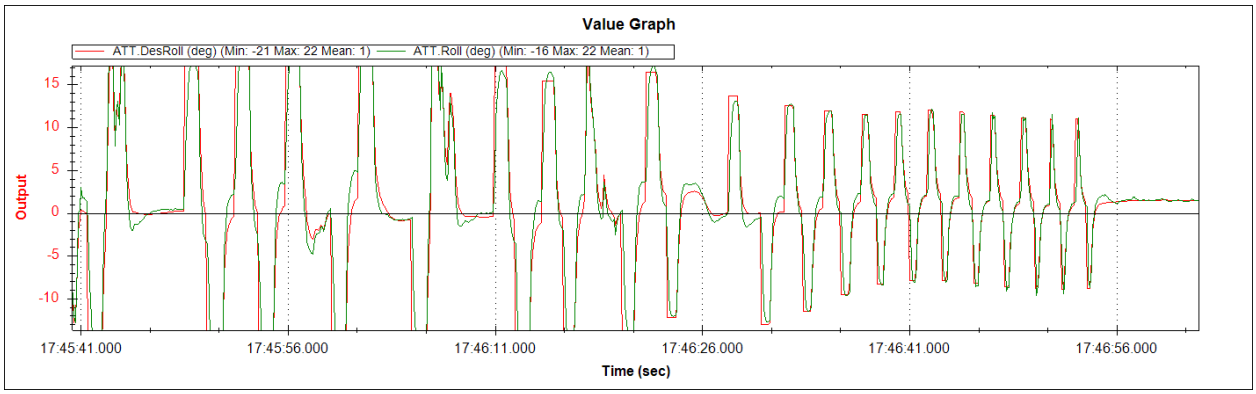

Figure 37. Desired vs measured roll tuned response

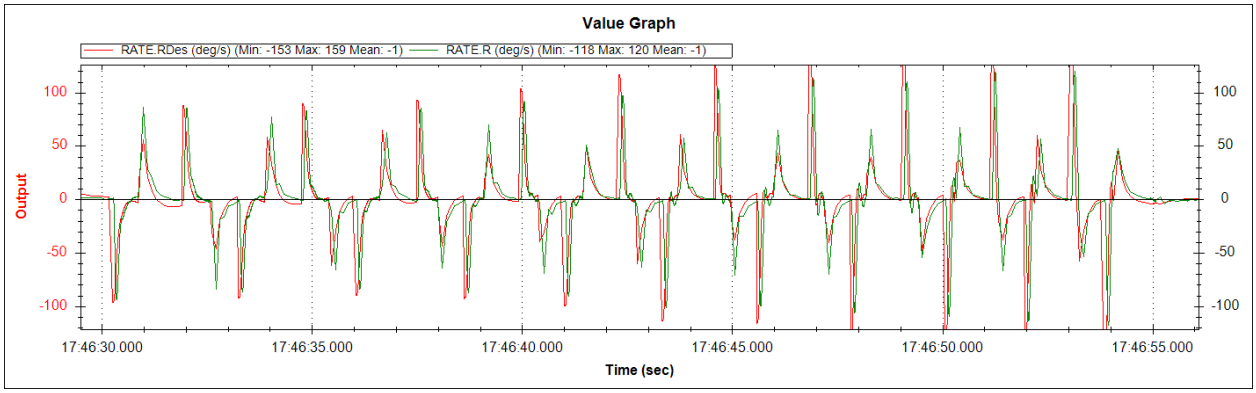

Figure 38. Desired vs measured roll rate tuned response

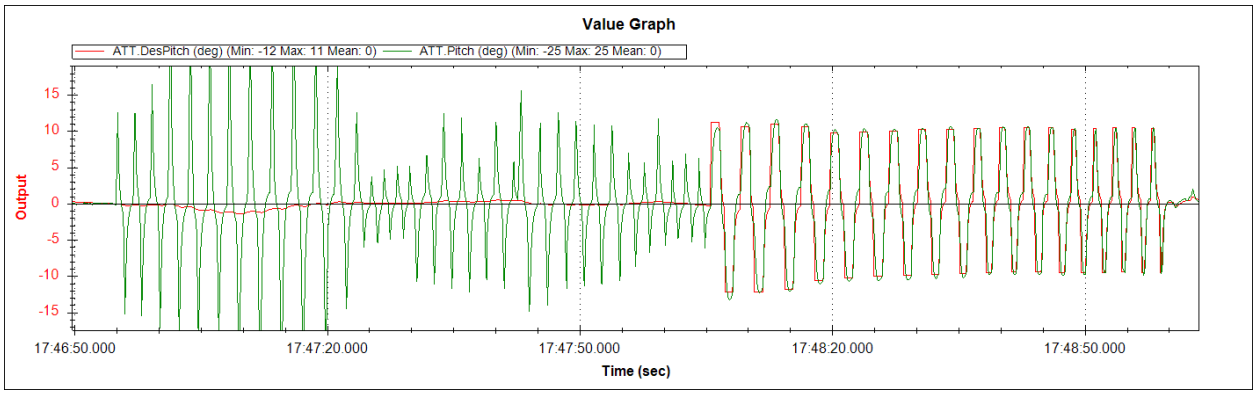

Figure 39. Desired vs measured pitch

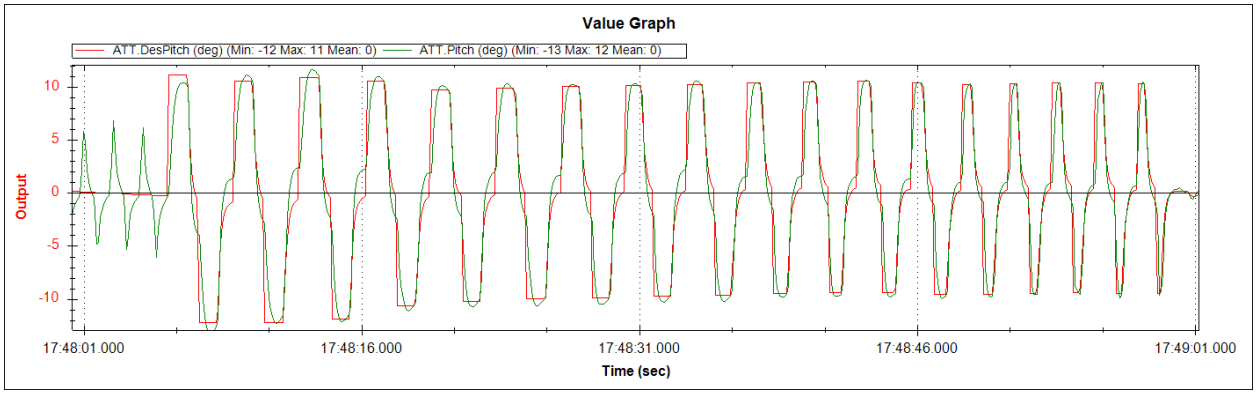

Figure 40. Desired vs measured pitch tuned response

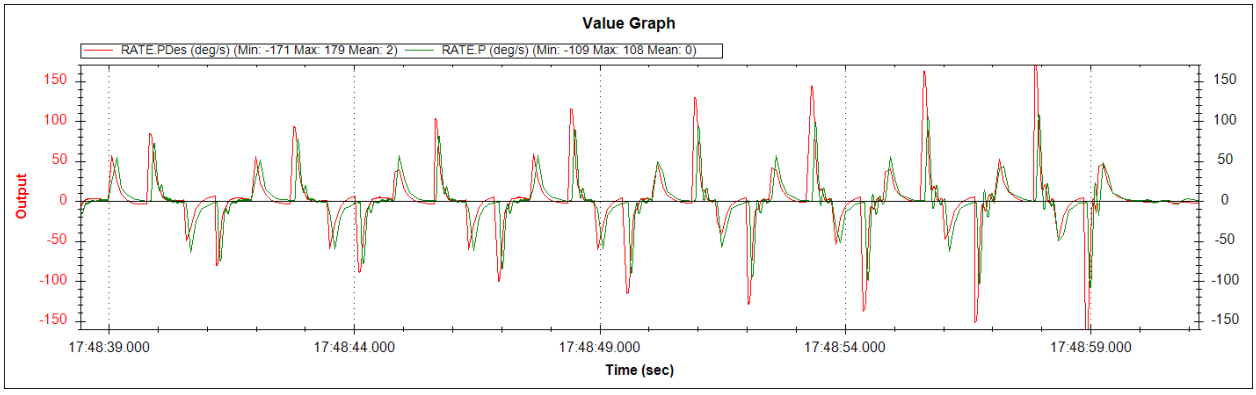

Figure 41. Desired vs measured pitch rate tuned response

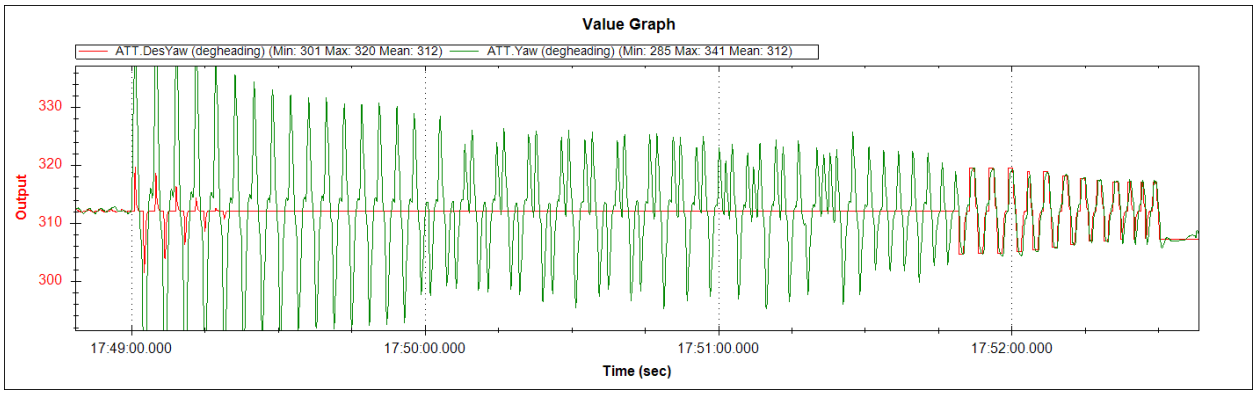

Figure 42. Desired vs measured yaw

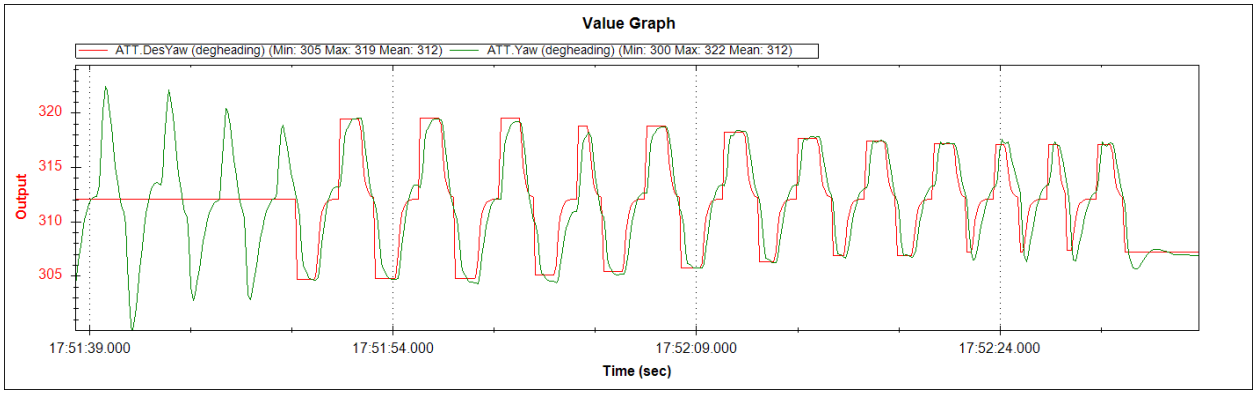

Figure 43. Desired vs measured yaw tuned response

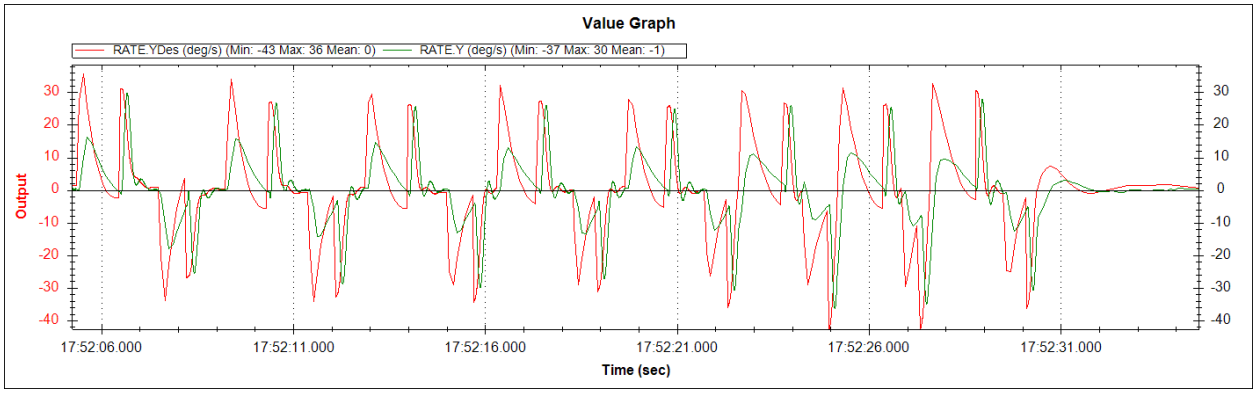

Figure 44. Desired vs measured yaw rate tuned response

From Figures 25, 28 and 31 it is evident that the starting PID parameters do not result in the desired response, but as the gains are changed, the measured values are able to track that of the desired. Comparing Figures 27 and 30 to Figure 33 we can see that roll and pitch are tuned with more stringent requirements than yaw. The following to figures show the gains before and after the Autotune process

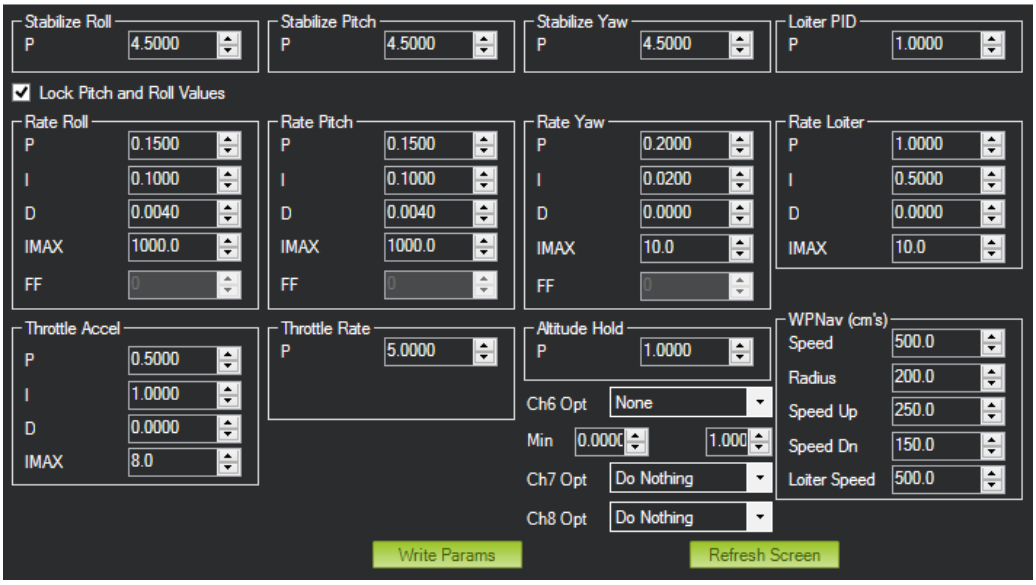

Figure 45. Control gains before Autotune

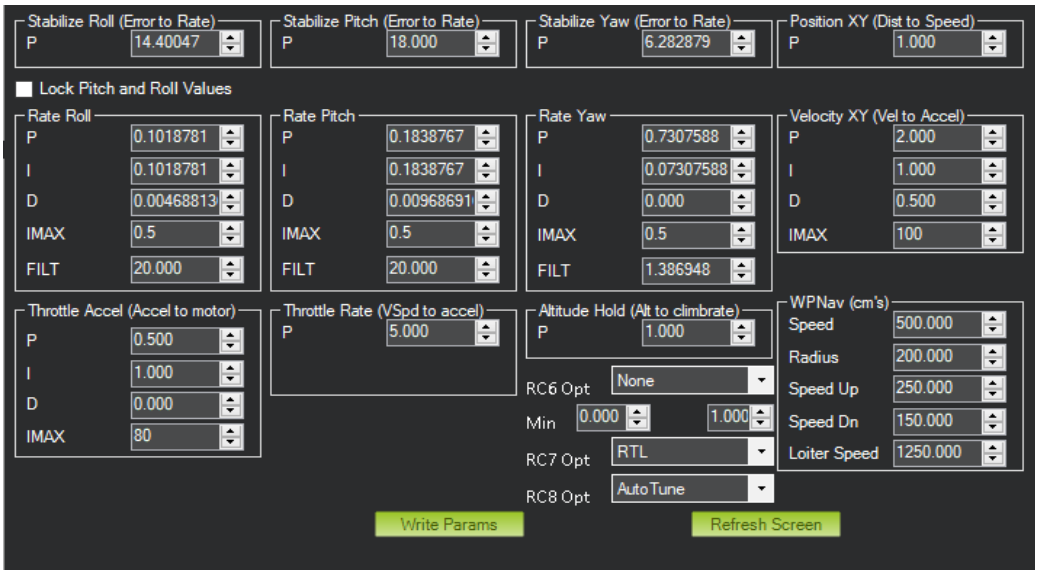

Figure 46. Control gains after Autotune

5.2 Autonomous Mission

A simple mission with 4 waypoints was constructed in python and executed using services and topics from Mavros. The scripts used ROS communication to query the state, position and location data from the flight controller, then used a combination of rosservices and message publishing to achieve an autonomous flight. The ArduCopter autopilot is capable of executing autonomous mission in the ‘GUIDED’ flight mode. While in this mode, the flight controller ignores all inputs from the RC transmitter and executes high level commands from a companion computer or ground control station. The script used a series of rosservice calls to change the flight mode to GUIDED, arm the drone then takeoff to the desired height. Once at altitude, the node sent 4 position set points defined in the drone’s local coordinate frame. Between each setpoint, there was 30 second pause in the program to give the flight controller time to reach the set point. Upon reaching the 4th set point, the node was programmed to land and disarm using the ‘land’ rosservice.

To compare the ability of the drone to keep accurate odometry while performing the preplanned mission, the mission was tested under two conditions: once with GPS and optical flow and again with optical flow only. The following figure graphs the x and y position against time and traces out the path traversed by the system.

Figure 47. X and Y position with Time with GPS (top) and Optical Flow (bottom)

In both flights both the GPS and optical flow modules were connected to the flight controller but the determination of which was contributing to the position estimate of the drone was governed by flight controller parameters setting. The red lines on the map show the position estimate of the drone throughout the flight and the blue ones show the GPS location. Looking at the bottom half of Fig. 47 we can deduce that the position estimate generated using optical flow is comparable to that received from the GPS module for short flights.

5.3 Sensor Data Visualization

Using the wireless link to the drone, the sensor data was streamed and visualized on the ground station. The following image is the raw depth sensor data visualized using RVIZ during a hover test**.**

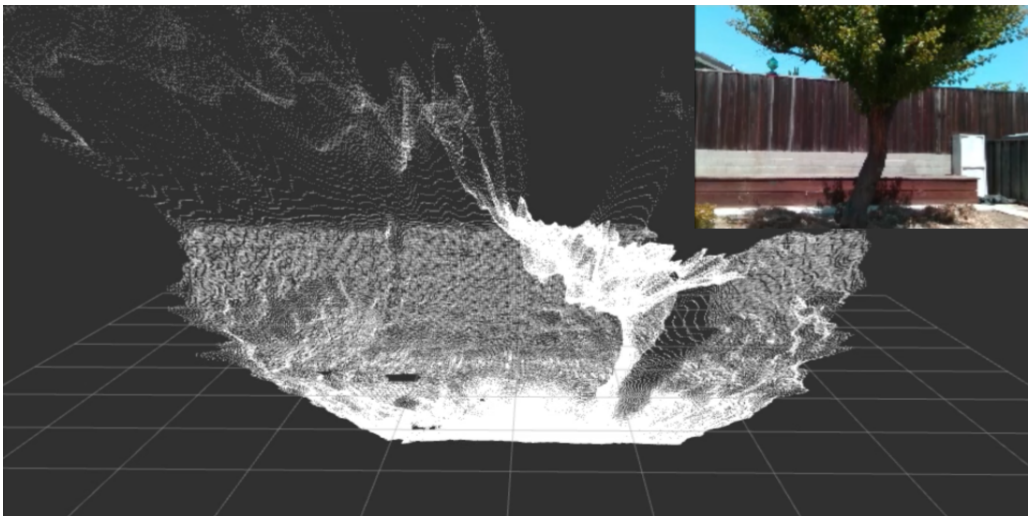

Figure 48. Raw depth sensor data from the D435i sensor

To use this data efficiently, the local planner node filters out points and produces a sparse cloud to perform its computation on. The image blow shows the same tree visualized in RVIZ but downsampled by the local planner node so it can run the planning algorithm in real time. The image also shows the chosen path (red) to the goal (yellow) along with the tree paths (purple) that were used to produce the final path.

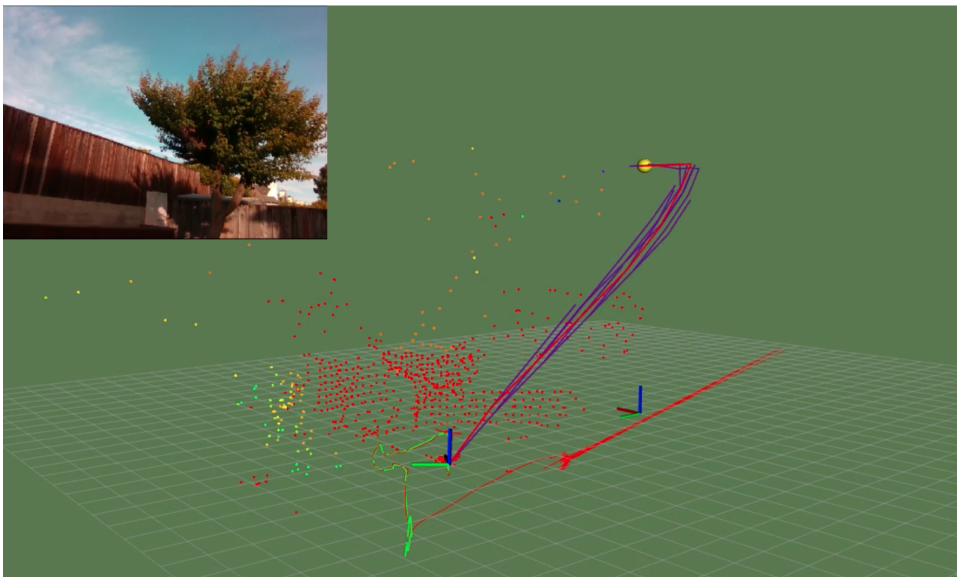

Figure 49. Reduced pointcloud and calculated trajectory visualization

Another operation the local planner performs on the pointcloud is it generates a 2D laser scan message from the point cloud. The subset of points that are directly in front of the drone and are aligned with the body frame of the drone are used to construct the laser scan message. The image in Fig. 50 shows the pointcloud (yellow) generated from facing a wall along with the corresponding laser scan (red) message. This compact message is ideal for use with the MAVLink protocol and can be sent to the flight controller as an OBSTACLE_DISTANCE message so it it can be used with its own collision prevention libraries.

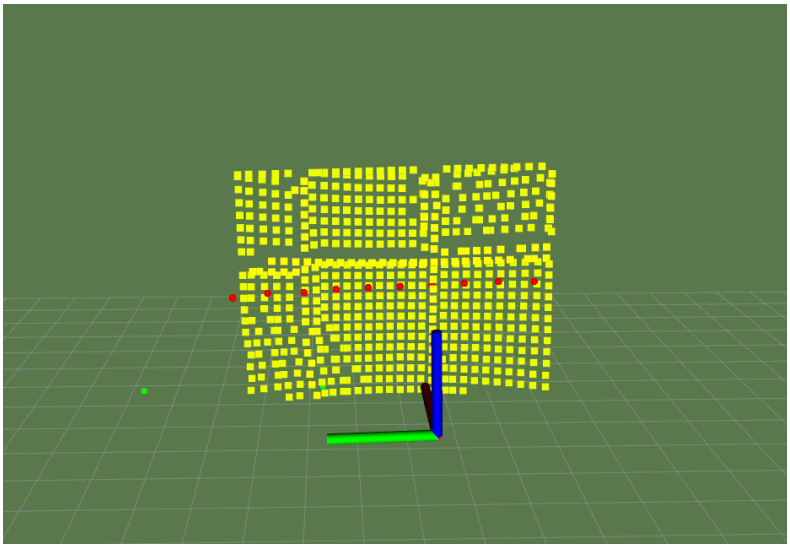

Figure 50. Pointcloud (yellow) and laser scan (red) message visualization

References

[1] Baumann, T., “Obstacle Avoidance for Drones Using a 3DVFH* Algorithm”, 2018.

[2] “A Comprehensive Survey of Control Strategies for Autonomous Quadrotors,” *CANADIAN JOURNAL OF ELECTRICAL AND COMPUTER ENGINEERING*, 2019.

[3] Daniel, K., Nash, A., Koenig, S., and Felner, A., “Theta*: Any-Angle Path Planning on Grids,” *Journal of Artificial Intelligence Research*, vol. 39, 2010, pp. 533–579.

[4] Duchon, F., Babinec, A., Kajan, M., Beno, P., Floreck, M., Fico, T., and Jurisica, L., “Path planning with modified A star algorithm for a mobile robot ,” *Modelling of Mechanical and Mechatronic Systems*, 2014, pp. 59–69.

[5] Finnegan, P., “VC Funding For Drones Surges In 2019, With More Focused Bets,” *Forbes*Available: https://www.forbes.com/sites/philipfinnegan/2019/07/22/vc-funding-for-drones-surges-in-2019-with-more-focused-bets/#4593793834d3.

[6] Hornung, A., Wurm, K. M., Bennewitz, M., Stachniss, C., and Burgard, W., “OctoMap: an efficient probabilistic 3D mapping framework based on octrees,” *Autonomous Robots*, vol. 34, Jul. 2013, pp. 189–206.

[7] Hu, J., Niu, Y., and Wang, Z., “Obstacle avoidance methods for rotor UAVs using RealSense camera,” *2017 Chinese Automation Congress (CAC)*, 2017.

[8] Niu, L., Smirnov, S., Mattila, J., Gotchev, A., and Ruiz, E., “Robust Pose Estimation with a Stereoscopic Camera in Harsh Environments,” *Electronic Imaging*, vol. 2018, 2018.

[9] Ogata, K., *Modern control engineering*, Delhi: Pearson, 2016.

[10] Opromolla, R., Fasano, G., Rufino, G., Grassi, M., and Savvaris, A., “LIDAR-inertial integration for UAV localization and mapping in complex environments,” *2016 International Conference on Unmanned Aircraft Systems (ICUAS)*, 2016.

[11] “Part 107 Waivers,” *FAA seal*Available: https://www.faa.gov/uas/commercial_operators/part_107_waivers/.

[12] Schoellig, A., “AER1216: Fundamentals of UAVs,” *AER1216: Fundamentals of UAVs*.

[13] Turkoglu, K., and Ji, A., “Development of a Low-Cost Experimental Quadcopter Testbed using an Arduino controller for Video Surveillance,” *AIAA Infotech @ Aerospace*, Feb. 2015.

[14] Tytler, C., “Modeling Vehicle Dynamics - Quadcopter Equations of Motion,” *Autonomy in Motion*Available: https://charlestytler.com/quadcopter-equations-motion/.

[15] Ulrich, I., and Borenstein, J., “VFH : Reliable Obstacle Avoidance for Fast Mobile Robots,” Belgium: , pp. 1572–1577.

[16] Zazulia, N., “FAA: Why Most Drone Rule Waiver Applications Crash and Burn,” *Avionics*Available: https://www.aviationtoday.com/2018/08/07/faa-drone-rule-waiver-applications-crash-burn/.

[17] Bristeau, P.-J., Callou, F., Vissière, D., & Petit, N. (2011). The Navigation and Control technology inside the AR.Drone micro UAV. *IFAC Proceedings Volumes*, *44*(1), 1477–1484. doi: 10.3182/20110828-6-it-1002.02327