Vibration Monitoring of Infrastructure via Uncrewed Aerial Vehicles

by

Rijesh Augustine

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Mechanical Engineering
University of Alberta

© Rijesh Augustine, 2023

Abstract

This thesis explores vibration monitoring of infrastructure or machinery through an uncrewed aerial vehicle (UAV) also commonly referred to as a drone. The vibration signature of an object may be used to assess the health of that object. This thesis looks at the means of acquiring this vibration signature in hard-to-access and GPS-denied environments and does not focus on the actual health assessment of infrastructure and machinery. A simple lightweight vibration monitoring arm was developed and mounted onto a drone. This along with an imaging system guided the drone and vibration arm towards a fiducial marker on an artificial vibration target. The thesis work successfully was able to measure vibrations from a vibration target. Future work is required to increase automation, the performance of position and yaw control and vibration sensitivity.

Acknowledgements

I would like to make note of all the people who helped me complete this thesis. I would first like to thank my research supervisor Dr. Michael Lipsett from the Mechanical Engineering Department at the University of Alberta. He has provided me with unwavering support throughout the entirety of this thesis. In addition, I would like to thank all my fellow research group members, for their support and knowledge. This includes Jorge Marin Marcano, Mark Sherstan, and Nicolas Olmedo. I would also like to thank Dr. Duncan Elliott from the Electrical Engineering Department at the University of Alberta for his initial contributions and supervision in this work.

In addition to all the support from my colleagues, I would also like to acknowledge my parents, sister, and wife for their support through all these years of work.

Table of Contents

1 Introduction **1**

1.1 Thesis Objectives . . . 2

1.2 Thesis Outline . . . 3

2 Literature Review **4**

2.1 Robotic Vehicle Use in Infrastructure Inspections . . . 4

2.2 Control of UAVs . . . 6

2.2.1 Requirement of Position Estimation System . . . 6

2.2.2 Non-Vision Based Control Schemes . . . 7

2.2.3 Vision Based Control Schemes . . . 8

2.3 UAV based manipulators . . . 12

2.4 UAV-based vibration measurement . . . 13

2.5 Vibration Measurement Devices . . . 14

2.6 Current Technology . . . 15

3 Proposed Approach and Methodology **16**

3.1 Autopilot . . . 17

3.1.1 Paparazzi UAV Autopilot System . . . 17

3.1.2 Ardupilot . . . 17

3.1.3 PX4 . . . 18

3.1.4 Comparison between PX4 and Ardupilot . . . 18

3.1.5 Proposed Approach for Autopilot Selection . . . 19

3.2 Vision Based Position Estimation . . . 19

3.2.1 Basics of Image Aquisition . . . 19

3.2.2 Stereovision . . . 21

3.2.3 Localizing UAV from Image Characteristics . . . 21

3.2.4 Proposed Approach for Vision Position Estimation . . . 25

3.3 UAV Control . . . 25

3.3.1 Controlling the UAV into range of the Target . . . 26

3.3.2 Challenges with Making Contact with the Target 27
3.3.3 Autopilot Control Messages 29
3.3.4 Control via Attitude Setpoint 30
3.3.5 Control via Position Setpoint 32
3.3.6 Proposed Control Approach 33
3.4 Vibration Monitoring 34
3.4.1 Permanently Affixed Sensor Box 35
3.4.2 Electromagnet Based Deployment Mechanism 36
3.4.3 Simple Sensor Arm with Vibration Dampening 37
3.4.4 Proposed Vibration Monitoring Mechanism 37
3.5 Functional Requirements and Specifications of Experimental Work . . 38

4 Hardware Setup and Testing 39
4.1 Aerial Platform 39
4.1.1 Airframe 40
4.1.2 Low-Level Command and Control of the Platform 42
4.1.3 Camera Systems 44
4.1.4 Onboard Computer 45
4.2 Evaluating ArUco 45
4.2.1 Precision and Accuracy of ArUco 46
4.2.2 Marker Ambiguity Problem 48
4.3 UAV Control Implementation 51
4.3.1 Controlling UAV using Attitude Setpoint Messages 51
4.3.2 Controlling UAV using Position Setpoint Messages 54
4.4 Sensor Arm with Vibration Dampening 56
4.4.1 Arm V1 56
4.4.2 Arm V2 57
4.4.3 Arm V3 57
4.4.4 Accelerometer Data Processing 58
4.5 Test Configurations 60

5 Results and Discussion 62
5.1 Aerial Platform Performance 62
5.1.1 Aperture Hexacopter 62
5.1.2 DJI FlameWheel F450 63
5.1.3 Communications links 64
5.2 Autopilot Software 65

5.2.1 PX4 65
5.2.2 Ardupilot 66
5.3 Onboard Computer and Imaging System 67
5.4 Control Performance 68
5.4.1 Vision Position Estimate Performance with DMM 42BUC03-ML 68
5.4.2 Attitude Controller Performance 69
5.4.3 Vision Position Estimate with Intel Realsense T265 Camera . 70
5.5 Vibration Measurement 76
5.5.1 Arm V1 76
5.5.2 Arm V2 77
5.5.3 Arm V3 77

6 Conclusions and Recommendations for Future Work 82
6.1 Conclusions 82
6.2 Future Work 83

Bibliography 86

Appendix A: Bill of Materials for Configuration 5 95

Appendix B: Python UAV Control Software 96
B.1 Installation Instructions 96
B.2 Main File 97
B.3 Position Control 114
B.4 Accelerometer Logging 117
B.5 Accelerometer Driver 117

List of Tables

3.1 Different Mavlink messages sent to and from the autopilot 30

4.1 Table showing different test configurations 61

5.1 Estimate of position error of end effector during contact taken from pixel measurements 75

List of Figures

3.1 Raw and processed stereoscopic images using OpenCV stereo algorithms. 23
3.2 ArUco Fiducial Marker with ID 19 25
3.3 Control State Machine. Fig 3.3c shows the inner workings of the Position Controller block referenced in Fig 3.3a and Fig 3.3b 27
3.4 Failed detection of markers during drone approach 28
3.5 SW steps for Attitude Setpoint Controller 31
3.6 Attitude Controller Diagram 32
3.7 SW steps for Vision Position Estimate and Position Setpoint based controller 33

4.1 Aerial platform based off of DJI Flamewheel 450, Pixhawk, Intel T265, and Jetson Nano 40
4.2 Aerial platform HW block diagram 41
4.3 DJI Flamewheel 450 close up 43
4.4 Aerial Platform networking configuration 44
4.5 Depth error of an ArUco marker at different depths and marker angles 47
4.6 Angle Error vs Marker Angle for different depths 48
4.7 Translation Error in X vs Angle in Frame 49
4.8 Marker ambiguity in a stereoscopic image. Both images were taken at the same moment in time, yet show different position estimations. . . 50
4.9 UAV Sensor Deploy Program and Libraries Summary 52
4.10 Simulator Setup 54
4.11 Python vision position estimate control program. Modules in boxes are primarily where work has been done to the work done in [61]. . . 55
4.12 Arm V1 uses 2 electromagnets. This arm was used with the Aperture Hexacopter. 56
4.13 Arm V2 end effector used for test flights with the DJI Flamewheel F450. 57
4.14 Arm V3 used for final test flights. 58
4.15 Close up of Arm V3. String was used to keep the sensor from drooping 59

5.1 Aperture hexacopter at outdoor test site 64
5.2 Flight 11 Position Control 71
5.3 Flight 12 Position Control 72
5.4 Flight 13 Position Control 73
5.5 Flight 14 Position Control 74
5.6 Flight 14 Contact 75
5.7 Vibration Source 76
5.8 Flight 11 frequency measurement 78
5.9 Flight 12 frequency measurement 79
5.10 Flight 13 frequency measurement 80
5.11 Flight 14 frequency measurement 80

Abbreviations

BHMS Bridge Health Monitoring System.

BLE Bluetooth Low Energy.

EKF Extended Kalman Filter.

FFT Fast Fourier Transform.

GNSS Global navigation sattelite system.

GPS Global Positioning System.

MCU Microcontroller.

RC Radio Control.

RTK Realtime Kinematic.

SBC Single-board computer.

SHM Structural Health Monitoring.

SLAM Simultaneous Localization and Mapping.

UAV Uncrewed Aerial Vehicle.

UKF Unscented Kalman Filter.

Chapter 1

Introduction

The monitoring of infrastructure and machinery plays a crucial role in ensuring their operational reliability and safety. By employing monitoring techniques, potential failures can be detected proactively, mitigating risks before they escalate into larger, potentially catastrophic events. This field is called Structural Health Monitoring (SHM) and is expansive enough that it has a few journals including The Journal of the International Association of Structural Control and Monitoring and the Journal of Structural Health Monitoring. Health monitoring techniques employed in SHM can be highly intensive and encompass various methods such as tap tests, visual inspection, and vibration monitoring.

Vibration monitoring has received significant attention since the 1960s [1]. Researchers have sought to develop effective methods for detecting faults and anomalies in structures [2–4]. Health assessments via vibration may be made on infrastructure and machinery. A few examples include bridges [5, 6], power transformers [7], and rotating industrial machinery [8].

Many newly constructed infrastructure projects now incorporate built-in wired sensors as part of their design, enabling real-time monitoring and data collection for structural health assessment [9]. However, a significant challenge lies in the monitoring of aging infrastructure, as many older structures lack integrated wired sensor systems. Among these investigations, a significant amount of attention has been

dedicated to understanding the structural health of bridges [10]. In bridges, these systems are referred to as Bridge Health Monitoring Systems (BHMS). Inspecting aging infrastructure without built-in sensors can be both costly and hazardous due to the need for manual inspections and potential risks to personnel. The largest cause of death in the construction industry is due to falls from a height [11–13]. This makes inspecting bridges, vessels and other tall structures particularly challenging. Despite these challenges, inspecting aging infrastructure is essential to mitigate the risk of structural failures and ensure the safety and longevity of these structures.

Uncrewed Aerial Vehicles (UAVs), also sometimes referred to as drones, have proven to be valuable tools for monitoring and assessing the condition of structures [14]. Their ability to reach hard-to-access areas and capture data has revolutionized the field of infrastructure monitoring. Primarily they have been equipped with cameras that allow for visual inspections of structures, but new studies have also explored the utilization of drones in the vibration monitoring of bridges. Overall drones have potential in detecting structural changes and assessing overall health.

The use of drones for vibration monitoring of infrastructure, particularly bridges and machinery in hard-to-access areas, such as wind turbines, allows SHM of these kinds of infrastructure to be more cost-effective and safer. A drone may be equipped with tools to provide SHM work.

1.1 Thesis Objectives

This master's thesis aims to further contribute to the field of drone-based infrastructure monitoring. Specifically, the focus of this study in on developing and evaluating techniques for controlling a drone near infrastructure, obtaining vibration samples from the drone, evaluating the efficacy of using fiducial markers for drone control, and examining alternative approaches or potential challenges associated with this problem. By exploring these aspects, this research intends to propose practical solutions and insights that can enhance and allow for further development of drone-based

vibration analysis in infrastructure monitoring.

1.2 Thesis Outline

The presented work shows how a lightweight low-cost UAV may be used to acquire a vibration sample in GPS-denied environments. Chapter 2 of this thesis reviews existing solutions to the problem. Chapter 3 focuses on the proposed approach to solving this problem. Chapter 4 describes the apparatus built along with providing some preliminary results. Chapter 5 follows up with data collected and the evaluation of design choices. Finally, Chapter 6 concludes this thesis, describing the overall learnings from this work and recommendations for future work.

Chapter 2

Literature Review

2.1 Robotic Vehicle Use in Infrastructure Inspections

The use of drones for bridge inspections is becoming an increasingly commonplace part of bridge health monitoring systems (BHMS). Drones are effective tools for detecting and assessing cracks in bridges. By employing visual servoing techniques, these vehicles can maintain continuous visual contact with defects and cracks during the inspection process [15, 16].

The utilization of drones in bridge inspections offers numerous advantages. Firstly, it allows the bridge to remain open during the inspection, minimizing disruption to traffic flow. Additionally, it significantly reduces the risk of injuries associated with working at heights for personnel, as the need for manual inspections at elevated locations is minimized. Moreover, the use of drones proves to be more cost-effective compared to traditional inspection methods, as it reduces the need for specialized equipment and manpower.

Research in this field has explored various approaches. Some studies have focused on employing computer vision algorithms to automate crack detection during bridge inspections by simply flying a drone in proximity to the structure [17]. These methods leverage the capabilities of drones to capture high-resolution imagery and analyze it using advanced image-processing techniques.

Furthermore, some assessments of bridge conditions using UAVs incorporate both conventional RGB (red, green, blue) imagery and infrared (IR) imaging techniques [18]. This combination allows for a comprehensive evaluation of bridge health by capturing visible light information to locate surface cracks and IR images to locate artificial subsurface delaminations.

Additionally, drones can be utilized to generate three-dimensional (3D) scans of bridges using photogrammetry techniques [19–21]. By capturing imagery from various angles during flight, the collected data can be processed to reconstruct a detailed 3D model of the bridge. This enables engineers and inspectors to visualize the structure from multiple perspectives, aiding in the identification of potential problem areas and facilitating more accurate assessments.

UAVs may be used in conjunction with a laser tracking station to measure a bridge's beam deflection [22, 23]. This work had a multirotor to make contact with the ceiling of the bridge in different locations. Using the ceiling effect also reduces the power consumption of the multirotor while measurements are being taken.

Research on surface thickness measurements from UAVs has been performed. [24] uses a UAV, with position estimation from Vicon motion capturing cameras to measure the thickness of a plate using ultrasonic measurements. [25] is similar research with a more complicated control scheme and an over-actuated UAV. The research with an over-actuated UAV allows measurements to be taken from planes at different angles.

Uncrewed marine vehicles have also been used for visual inspection of submerged foundations [26]. Uncrewed ground vehicles may also be equipped with cameras and accelerometers for use in infrastructure inspections deemed unsafe for humans and points of measurements are accessible by a ground robot [27]. Uncrewed ground vehicles may also be equipped with ground penetrating radar [28]. The use of such vehicles can programmatically collect data, and assess areas that are too small or deemed unsafe for human access.

2.2 Control of UAVs

2.2.1 Requirement of Position Estimation System

The control of UAVs close to bridges or walls poses significant challenges. Factors such as wind patterns and turbulent airflows generated by the UAV's motors can impede precise and accurate control. When operating close to another body, the other body may additionally affect the airflow and aerodynamics of the drone. This makes maintaining stability and control even more challenging. The risk of unintentional drift increases, potentially leading to collisions with the other body and causing damage. If the UAV needs to be operated physically far away from the operator, as in many cases with large tall structures, new challenges are introduced such as command latency and reduced situational awareness.

To address these challenges, onboard reliable position estimation methods and automatic control of the UAV are essential. The most commonly used position estimation technique for UAVs is using GNSS. However, GNSS systems have inherent limitations that make it not effective for position estimation in certain environments.

In North America, the primary system used is GPS. GPS receivers can determine position by measuring the direct signal from a minimum of 4 satellites in orbit. These satellites broadcast their time and location. This allows GPS receivers to solve for their current time and spatial coordinates. The relative position of these satellites also affects how accurate the position estimate is. This is referred to as dilution of precision (DOP). If the satellites are in the same area of the sky the precision of the position estimate will be lower. To improve the position estimate it is important to have more than the minimum number of satellites and to have satellites positioned across the sky. GPS-Denied Environments are areas where GPS signals do not reach the receiver. Infrastructure that may need vibration monitoring may be in GPS-denied environments or environments with high GPS uncertainty. The inside of buildings tends to have very poor GPS signals. Large bridges may also act as a canopy blocking

out signals. Blocking out signals is not the only cause of uncertainty. If the GPS signal reaches the receivers via a reflection, it will also cause poor location estimation. This can easily occur next to buildings. Another cause of location uncertainty is an ionospheric delay. GPS in a vacuum would be very precise, but radio waves passing through the ionosphere will slow down by an undetermined amount depending on the ionospheric conditions that day.

There are two ways of combating atmospheric delay. The first method is using two different GNSS frequencies, and the second method is using an RTK base station. In the first method, a satellite broadcasts information at two different frequencies. These frequencies slow by a different amount in the ionosphere as a result the amount of ionospheric delay may be calculated. GPS is slowly modernizing its satellites to allow for a 2nd civilian frequency to be broadcast from each of its satellites. The second method uses a local datum point and compares observations at this datum point vs at the receiver to determine an accurate baseline between the receiver and the datum point. Even with either of these methods of improving the location accuracy of GPS, they would not solve the issues from multi-path interference, and GPS-denied environments. Therefore a different localization approach may be required for certain applications.

2.2.2 Non-Vision Based Control Schemes

The utilization of ultrasonic sensors and ground-based ultrasonic beacons for precise positioning of drones, as experimented in [29], presents a few impracticalities for this application. This system requires ground control beacons. The size of the infrastructure makes it challenging to set up beacons in outdoor locations, and a substantial number of beacons may be required to be set up for each survey. This adds time for setup and additional costs. The location of each of these beacons must be known and for full coverage, workers may need to still be used to install them in hard-to-access areas. Some areas may even be close to impossible to install an ultrasonic beacon.

Manyam [30], suggests employing an uncrewed ground vehicle (UGV) in conjunction with a UAV for positioning. However, this approach is also impractical for infrastructure measurements. If the target point is located on a bridge pier, it may fall outside the range of the beacon on the UGV, rendering the UGV ineffective for aiding in the positioning of the UAV.

Time-of-flight sensors can be utilized. However, these sensors often have limited range capabilities. For instance, L M González-deSantos [31] utilizes sensors with a range of 25 cm. By using two sensors and an IMU to keep the drone upright, distance and angle measurements to a planar wall may be obtained. A human operator will need to bring the contact arm of the UAV to within 25 cm of the wall manually before these sensors could work. If one sensor is out of range the UAV does not automatically know which direction it should turn for the other sensor to come into range. This work was further improved on and described in [32, 33]. The newer work incorporates Garmin LiDAR sensors with a range of 40 m. While these sensors aid in position estimation and collision avoidance, they do not provide direct position feedback to the drone. So in GPS-denied environments, the drone could potentially move parallel to a planar wall without the sensors detecting this as movement.

Research has also explored techniques for making contact with curved walls, resulting in the development of algorithms [34]. However, the mentioned paper did not involve an actual hardware implementation.

2.2.3 Vision Based Control Schemes

Vision-based control of robotics has a long history, with early works dating back to the mid to late 20th century. For example, J.T. Feddema [35] describes vision-based control using ground robotics, which shares similarities at a high level with the control objectives pursued in this thesis involving UAVs. The use of vision-based control with UAVs is extensive. [36] is a large survey paper that discusses a large array of research efforts on the subject. The paper includes a discussion on SLAM, group navigation,

visual odometry, and localization based on template matching.

Some vision systems [37] employ static ground-based cameras, which encounter similar challenges as the previously discussed ground-based ultrasonic sensors. Issues such as limited field of view, precise placement of the cameras, and size of the UAV in the image frame may arise with these camera-based systems especially.

Practical application of vision processing includes the detection of infrastructure and objects and the control of the UAV around them. [38] describes using computer vision to detect a road. Once a road is detected, images of the road are captured to analyze its health and the drone is then automatically controlled to follow the road. [39] has a UAV that detects a road sign and automatically perches onto the road signs. This is done by detecting the plane of the sign using a depth camera and a specialized perching payload. [40] uses a drone to detect and follow power lines. The purpose of that research was also to assess health. [41] uses a UAV to track another flying target and chase the target.

Fiducial markers offer a means of accurately determining the location of a drone. By utilizing a fiducial marker with a known size, shape, and camera parameters within an image frame, the camera's position in space can be determined. When the camera is rigidly mounted to a drone, the drone's position can be inferred accordingly [42, 43].

An application of fiducial markers with UAVs that have been largely researched is their use with precision landing [44]. The use of fiducial markers aids enables the drone to land precisely. However, this approach encounters the challenge of marker pose ambiguity as shown by Springer in [45]. This issue is further addressed in this thesis when evaluating fiducial markers. Springer continues to investigate the concept of nested markers [46]. Nested markers are where a smaller marker is within a larger marker. For precision landing, the drone at height needs to be able to identify the marker, so the marker needs to be large. But as it approaches the marker, the marker may be too large to fit in the camera's field of view. In this case, a nested marker

may be used to continue guiding the drone to the landing spot. This approach allows for effective guidance and position estimation at all altitudes. [47] uses fuzzy control and ArUco fiducial markers to land on a target. Their use of fuzzy control allows for some shock absorption of the landing and the landing of the UAV on a moving landing pad. [48] has done more work on using a drone to autonomously follow a moving platform with an ArUco marker on it and then land on that platform. [49] uses a custom marker for a precision landing. The contours of their custom marker are analyzed to produce a relative position of the UAV to the marker. This work is attempting to do a fully autonomous landing in an outdoor environment. They also propose to use an arm with a contact sensor to aid in the final stages of landing. [50] describes using a monocular camera to land a drone onto a moving target, that may also be pitching, rolling and heaving. This research uses the landmarks of the targets to determine the pose and motion of the landing platform. Modelling the motion of the landing platform allows them to control the drone to compensate for the movement.

Other research [51] has focused on indoor localization in industrial environments, combining Lidar data, vision, and fiducial markers. The fusion of data from multiple sensors, along with simultaneous localization and mapping (SLAM) algorithms, enables accurate position estimation in these environments.

[52] uses a series of images for path following. The drone would fly the path once and generate a series of images. Then it would try to follow this path based on images it currently is receiving. Its control loop would try to figure out how the drone should move to get the current image to correspond to the next desired image in the path. This would be repeated for the entire path. The research also explores obstacle avoidance when an obstacle is detected in the original path. [36] discusses research on localizing a UAV by matching its current view of the world from one previously taken, like from a Google map tile or a different flight taken at a different angle, time of day or camera configuration.

Optic flow is also a good means of position estimation. Optic flow works by measuring the change in the location of a feature in an image between subsequent frames. Knowing the camera parameters and the time between frames a velocity estimate may be determined [53, 54]. Kendoul in [55] shows the strong performance of optic flow systems in aerial applications. These systems can be just as good as GPS. They do have a problem where if the area they are getting information from is an area that is devoid of features, the position solution may drift. Optic flow only provides a relative change in position to a position estimation and does not provide any global position estimation.

Visual Inertial Odometry (VIO) uses the visual information of the surrounding environment, fused with inertial measurements for position estimation. This solves some issues with just optic flow and improves optic flow. [56] is some early research on the subject. [57] shows VIO in action with fisheye cameras. [58] shows different enhancement strategies for visual odometry to help improve drift reduction. The use of visual information prevents drift that an inertial-only navigation solution would experience. The use of inertial data gives clues and aids the feature-matching algorithm used on subsequent images. [59] shows how to build a VIO drone using an Intel RealSense ZR300 VI Sensor and a DJI M100 aerial platform.

In one particular solution [60], an Intel T265 camera is employed for position estimation. This approach integrates time-of-flight distance sensors to measure the distance between the drone and the wall. The manipulator in this scenario takes the form of a crawler. While no fiducial markers are utilized, the T265 camera leverages vision-based techniques to generate position estimates.

[61] is a practical example of integrating an Intel T265 camera with Ardupilot. This work with April tags shows how to use a T265 camera to generate a position estimate, send that position estimate to Ardupilot, get position information from a tag seen by the T265 camera, and control a drone to land on the target. This work could easily be adapted for vertical targets and for positioning a UAV near infrastructure.

[62] is work on integrating stereo vision data more directly into state estimation. The model they propose has better computational performance and could be useful for computation-constrained systems.

There is work that has been done on obstacle detection and avoidance. [63] is some recent work on the subject. This work uses background subtraction of subsequent camera frames to detect and object and its relative motion.

2.3 UAV based manipulators

Considerable research has been conducted on UAVs equipped with manipulators [64], demonstrating a growing interest in this field. One theoretical investigation [65] focused on evaluating the feasibility of utilizing consumer drones with rigid arms to establish contact and apply force to a surface. The force applied by the drone to a wall comes from changing the angle of the drone. This study went on to show how much force a drone may apply laterally whilst still having enough thrust to remain airborne.

The dynamics and control of a two-degree manipulator are discussed in this paper [66], highlighting the interplay between the manipulator's state and the flight control of the drone. Understanding this relationship is crucial for effective coordination between the manipulator and the UAV's flight system.

[67] is a conceptual work to mount a sensor to a wall using aerial manipulators. Two drones are conceptualized to be used in tandem. One drone sprays a surface with resin, after which a second drone will come and place a sensor on the surface.

Specific drone models have been developed to achieve contact with different types of surfaces [68]. For instance, one drone was designed to make contact with walls, while another was tailored for interacting with pipes. These specialized drones are engineered with specific task requirements in mind. The structure of the first drone is 5 rotor aircraft. The 5th rotor is farther away from the main rotors and allows for counteracting the weight of the payloads attached. The second drone described in the

paper had landing legs that straddled the pipe and 6 propellers, 2 of which allowed the drone to rotate around the pipe and inspect the pipe at a 90-degree angle.

The manipulator may also be instrumented to provide data to the flight controller. The use of springs attached to the end of the manipulator's arms enables the application of a controlled amount of force [69]. By measuring the compression of the springs, the force being applied by the drone to the surface may be measured and can be incorporated into the flight control loops of the UAV.

Additionally, a manipulator with active compliance has been investigated [70], offering the capability to adjust the applied force using an arm that can be extended using a linear screw. This design allows the manipulator to even tap the target surface with the application needed for it.

Zhou [71] presents a UAV equipped with grippers capable of relocating objects from one location to another. Fiducial markers are placed on the objects to facilitate tracking and manipulation by the gripper. He also encountered the challenge of the marker pose ambiguity problem that was previously mentioned.

2.4 UAV-based vibration measurement

Several research papers have explored the subject of vibration measurement using UAVs. Currently, there are no known commercial drones found that are specifically designed for this purpose.

Carroll [72] presents a drone specifically designed for contacting ceilings. This design is suitable for use under bridges, but cannot be used on vertical surfaces. This method reduces the risk of collision between the drone and the surface, but it does not allow for capturing vibrations from vertical surfaces. The drone in this paper utilizes an electromagnet for establishing contact, so this does not work for non-magnetic surfaces.

Garg [73] has done work with a drone that is equipped with a non-contact vibrometer for vibration measurement. The focus of this work lies in the efforts made to

compensate for the motion of the UAV in the vibrometer readings. This approach aims to address the challenge of accurately measuring vibrations while accounting for the movement of the drone.

It is worth noting that none of the research discussed in this section specifically addresses the topic of automatic position control of the UAV during vibration measurement. The emphasis in these papers is primarily on the design of the UAV and the vibration acquisition mechanism.

Overall, while research has been conducted on vibration measurement using UAVs, there are currently no readily available commercial products tailored for this application. The explored methods range from utilizing contact-based mechanisms to non-contact vibrometers, each with its limitations and considerations. Further research is needed to address automatic position control and enhance the capabilities of UAV-based vibration measurement systems.

2.5 Vibration Measurement Devices

The market offers various pre-existing vibration sensors with different functionalities and capabilities. Established companies such as Fluke and SKF manufacture wired vibration sensors for monitoring applications. These sensors are typically connected through physical wiring to the monitoring system.

In the context of railway networks, a research paper [74] discusses the utilization of wireless sensor networks for taking vibration measurements for detecting and preventing failures. Wireless sensors provide the advantage of eliminating the need for physical wiring, increasing the places they can be used and making them easier to install. These sensors do have the disadvantage of needing to replace batteries and setting up base stations within range of the wireless sensors.

Several companies specialize in manufacturing wireless sensors for various applications. Erbessd offers Bluetooth-based wireless sensors with a range of 50 meters and a battery life of up to 2 years under low usage conditions. Emerson's AMS line

of sensors provides a range of 100 meters and a battery life of 3-5 years with low usage. LORD Microstrain offers the G-Link-200 wireless sensor with a range of 800 meters and a battery life of 90 days. These sensors are designed to cater to specific monitoring needs.

In summary, the market offers a range of vibration sensors. Each of the sensors mentioned above costs upwards of 1000 CAD.

2.6 Current Technology

The current technology available has shown mechanisms for contact inspections with UAVs. These contact inspections have either not been made fully automatic, do not employ a vibration acquisition mechanism or do not allow contact on vertical surfaces. GPS free localization of UAVs have been performed using fiducial markers. Most of these localizations have been performed in the context of precise landing. The use of fiducial markers to make contact with a wall has not been throughly explored. As such this work shall focus on an UAV based contact vibration measurement device that can automatically deploy and make contact at a specific location on a vertical surface.

Chapter 3

Proposed Approach and Methodology

This work proposes a robotic method for the collection of vibration measurements from infrastructure utilizing a UAV. The primary focus of the research is on three main aspects: obtaining a reliable position estimate, developing effective control strategies based on this position estimate, and implementing a method to capture vibration data using a contact sensor. This chapter describes a trade study between different approaches considered before describing experiments for proof of concept for each key functionality.

A reliable position estimate shall be able to be achieved in daylight hours with normal lighting conditions using vision. The position estimate should provide enough information to prevent collision with the infrastructure.

Once a reliable position estimate is established, the next goal is to be able to control the drone via some sort of automatic controller. This involves an autopilot and the means of using this autopilot to control the UAV at a low level. Control algorithms are required to enable the UAV to navigate and maneuver near infrastructure, avoiding collisions.

Finally a methodology to physically capture vibration measurements from the infrastructure is developed. This involves selecting an appropriate vibration sensor and integrating it with the UAV platform.

3.1 Autopilot

Three main autopilot systems were evaluated. These were Paparazzi UAV, Ardupilot, and PX4. These three were selected because they were free to use, open source, offered low-level control, have many open source hardware options and have a developer community of some sort. Additional autopilot systems are available on the market but they are less popular or more commercial in nature [75].

3.1.1 Paparazzi UAV Autopilot System

Paparazzi UAV is an open-source autopilot system, it garnered a lot of early support in the field, but in more recent years Ardupilot and PX4 have become a lot more popular in the UAV community. The system was founded in 2003 and was one of the pioneers of a community-driven free open-source autopilot system. Prior experience with the software in fixed-wing applications led to its consideration in this project.

In addition, its modular architecture and lightweight nature make it an attractive option for customization. However, in 2016, data on Paparazzi's performance with multi-rotor aircraft was limited. Paparazzi has been slowly losing market share, but was once a leader in the field of consumer UAVs [76]. The popularity and information available on competing autopilot systems led to the ultimate decision to reject Paparazzi as a candidate for this thesis work.

3.1.2 Ardupilot

Ardupilot is another open-source autopilot system used in UAVs. It is a popular choice among the UAV community due to its vast array of features, ease of use, and reliability. Ardupilot provides a comprehensive platform for controlling UAVs with advanced features, including GPS-based navigation, waypoint following, and autonomous mission execution.

Ardupilot has a vast community of developers who continually update and improve the software, providing users with access to new features and capabilities. This also

ensures that the system is continuously optimized and updated to meet changing needs.

3.1.3 PX4

PX4 is another open-source autopilot system designed for UAVs. It is known for its advanced control algorithms, efficient processing capabilities, and support for a wide range of base platforms. Like Ardupilot, PX4 is also widely used in the UAV community and includes all the base features of Ardupilot.

3.1.4 Comparison between PX4 and Ardupilot

Both PX4 and Ardupilot are very capable autopilot systems. When comparing the two it almost comes down to preference. Some research has been conducted to compare autopilot systems [77]. Ardupilot and PX4 differ in various aspects.

Architecture: They both are relatively modular and both provide a good framework for customization and development. PX4 has better code separation than Ardupilot making it easier to develop and push new code for the community. In contrast, Ardupilot has a bit more integrated architecture, which allows for easier changes to the codebase, but it is more difficult to push these changes back to the community.

Community and Development: Both have large active communities. Ardupilot tends to skew more toward hobbyists, whilst PX4 has a larger research community

Flight Control Algorithms/Ease of Use: Out of the box, the performance of Ardupilot is superior to that of PX4.

Supported Platforms: Both support a wide range of hardware. They are also compatible with a wide range of vehicles.

Mission Planning and Interface: Ardupilot uses Mission Planner as its primary interface. PX4 on the other hand uses QGroundControl. Ardupilot may also be controlled by QGroundControl. Both autopilot systems utilize mavlink in their communication layer.

Industry Adoption: PX4 has larger industry adoption due to their license structure.

3.1.5 Proposed Approach for Autopilot Selection

Preliminary research has shown that either PX4 or Ardupilot would be an adequate choice for the application. The low-level implementation details drives what is used in the final implementation of this work. Despite the better out-of-the-box flight performance of Ardupilot, work started with PX4 due to its larger adoption by the research community.

3.2 Vision Based Position Estimation

3.2.1 Basics of Image Aquisition

The basic way a digital camera works is light enters a lens and is then focused onto an imaging sensor. Many factors affect the quality of the image that the sensor produces. Imaging sensors may be made with different technologies. The two most prominent types are CCD and CMOS [78, 79]. Of these two sensor types, CMOS sensors have largely become more popular. They have numerous advantages over CCD sensors. These advantages include lower power draw and faster readout times.

Camera sensors may have a global shutter or a rolling shutter. A global shutter captures light incident to the sensor at a single point in time. A rolling shutter captures the image sequentially across the sensor. A rolling shutter may produce image artifacts when photographing objects in relative motion to the imaging sensor. When CMOS sensors were first introduced they were entirely rolling shutter, but modern CMOS sensors are now available as global shutter. A rolling shutter may still be adequate to use for the application of visual servoing, due to modern CMOS sensors having reduced readout times, so rolling shutter artifacts are less noticeable. In addition, a camera with a mechanical shutter largely removes rolling shutter artifacts.

The size and resolution of the imaging sensor also affect the image quality that is produced. The resolution of an imaging sensor is a measure of how many pixels

or data points are present in each image. A larger resolution sensor records more detailed images. These more detailed images are larger and have the downside of increased computing time when processing the image. The sensor size of an imaging sensor affects the size of lenses required. A larger sensor size requires a larger lens to focus the light across the entire sensor. A larger sensor and lens may drastically increase the size and weight of the camera. For a given aerial platform, increasing the payload size and weight is detrimental to the overall performance of the aerial platform.

The lens installed onto the imaging sensor is also important. The choice of the lens determines the focal length and aperture of the camera. The longer the focal length, the more narrow the field of view that the camera has. In addition to the narrow field of view, there are fewer distortions. The downside is that it may be difficult to put the object into the field of view and keep it there during the entire visual servoing process with larger focal length lenses. But if the object can be maintained in the field of view, a larger focal length lens maximizes the resolution of the object providing better position estimation. In the application of visual servoing, there is a balance between being able to detect an object when it is far away with high resolution and keeping it in the frame when it is close by. One solution to this is having different-sized objects be detected at different distances away. [80] has an interesting solution to this tradeoff with a marker within a marker. The aperture of the lens determines the amount of light that hits the imaging sensor and the depth of field of the lens. If the aperture is too large then an object will not be reasonably in focus for a range of distances from the camera. If the aperture is too small then the object may be too dark to detect, unless lighting conditions are sufficiently bright.

Another aspect of the camera is how the data is read out. Imaging sensors tend to either have a parallel or serial interface for reading data out. This interface connects to a computer usually through another IC on the camera that translates the data to a more common format like USB, Ethernet, or Firewire. Data readout speed and

overall latency are factors to consider for this application.

3.2.2 Stereovision

Stereovision uses two cameras to generate a 3-dimensional scene. The process is similar to the way humans perceive depth by using slightly different viewpoints. Traditionally two cameras are placed side by side, similar to how human eyes are, and a photo is captured at the same time. Usually, each camera will have the same resolution, focal length, and exposure settings. Objects that are in both images may be found in the image frame. Triangulation is the process of calculating a point in 3D space based on how it is perceived from different locations. In the traditional case of 2 cameras side by side, the difference between the locations of the object in each image frame is called the disparity. The disparity along with the baseline between the two cameras, resolution, and focal lengths of both cameras allow calculation of the 3-d point of the image relative to the image frame. The math for calculating the 3D point is relatively simple once all the calibrations and corrections have been performed for lens distortion. [81] goes into detail on how this is accomplished.

The largest challenge with stereo vision is correlating image points between the two images. Any deviation or noise, (due to poor lighting, or motion blur) may cause stereovision algorithms to fail as they may incorrectly correlate image points.

3.2.3 Localizing UAV from Image Characteristics

Once an image has been acquired, it needs to be analyzed for features that allow for knowing where the UAV is in the scene. This may be a challenge. There needs to be some reference data in the image acquired to be able to determine where the UAV is with acceptable accuracy.

A few different image characteristics were explored throughout this project.

Plane Detection

Plane detection using stereoscopic cameras was one of the first things attempted in this project. The premise of this idea was to get the UAV close enough to a wall and then use stereovision to detect the plane of the wall. With this, the UAV would then square itself with the wall and slowly move towards the wall and make contact. This method did not require any fiducial marker on the wall, as a result, it would be very much dependent on the characteristics of the wall. It required a minimum of 3 points in the images to calculate the plane of the wall. Figure 3.1 shows an example a stereoscopic image captured. OpenCV was used for camera calibration [82], detection of matching features and generation of a disparity map. Walls devoid of characteristics or with fine repeating patterns would not perform well. If there were characteristics, these characteristics may go out of the frame during the approach. Figure 3.1d shows how difficult it would be even for a human to control a decipher the disparity map. In addition, this method lacks any repeatability. There is no reference point on the wall where the vibration measurement should be taken, so this method is not adequate for taking a vibration measurement.

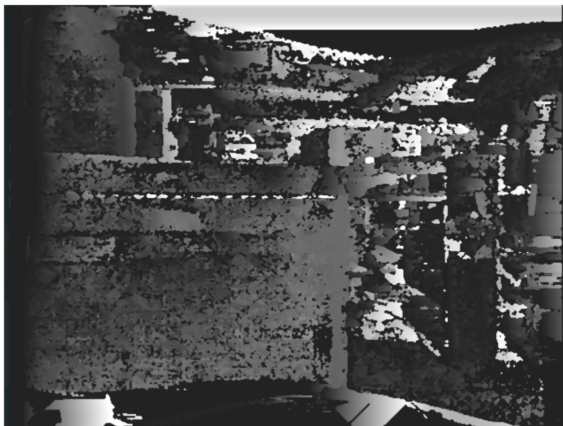
Rectangular Marker Detection

A basic rectangular marker was then explored. This marker was a black rectangle on a white background. Using OpenCV algorithms, the edges of this marker were detected. With a stereoscopic setup, this allowed the generation of vectors in 3D space for the edges. Cross-multiplying these vectors produced a plane. Using a single camera, with knowledge of the real-world size of the marker and basic trigonometry allowed calculation of the position of the UAV relative to the marker.

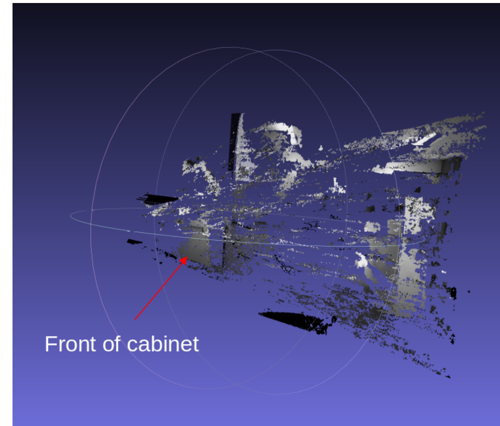
This method was adequate, but could also lead to false positives when attempting to detect the marker. Attempts were made to limit false positives based on the marker's size and aspect ratio. Due to the nature of visual servoing, the marker size would change as the camera approached the target so discriminating targets was

(a) Left Image

(b) Right Image



(c) Disparity Map



(d) 3D render

Figure 3.1: Raw and processed stereoscopic images using OpenCV stereo algorithms.

difficult. In addition, rolling and pitching of the UAV would skew the markers and corrections would need to be made for those cases.

Coloured Marker Detection

Another method of marker discrimination explored was using rectangular marker that was coloured. This was done with an orange rectangle and an RGB camera. This method performed well in quickly detecting the coloured rectangle amongst a noisy backdrop. The colour space was converted from RBG to HSV, and this allowed filtering the captured image to only retain an orange hue. The main issue with this method is poor edge identification in certain lighting conditions. Lighting conditions could cause glare on the marker and overexpose parts of the marker leading to a corner being incorrectly identified. Workarounds for this issue may exist with using a matte finish on the marker. Overall this method was rejected due to the background needing to be a different hue than the marker and this method not providing a means to differentiate between different markers at the same site.

Fiducial Marker System

There are a few ready-made fiducial marker systems. These were then explored after evaluating the manual methods described above. Libraries that were explored include ArUco Tags [83, 84] and AprilTags [85, 86]. Both systems have markers that can be generated/downloaded and printed out. An image of this marker can be loaded into its respective library, along with camera parameters and the printed size of the marker. The library then outputs the pose of the marker relative to the camera. Figure 3.2 shows an example ArUco marker. In addition, various methods have been tried to improve these marker systems [87].

Comparisons have been made between the two systems, and both would perform adequately. A paper describes the performance of both systems with various amounts of Gaussian Noise [88], with ArUco generally being superior.

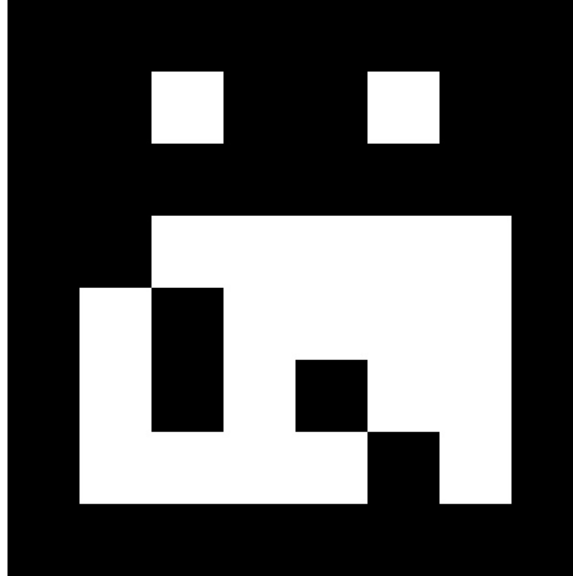


Figure 3.2: ArUco Fiducial Marker with ID 19

3.2.4 Proposed Approach for Vision Position Estimation

The use of vision is explored to generate a position estimate. Preliminary testing shows that the most reliable means is the use of a fiducial marker system. In addition, fiducial markers allows for greater repeatability when measurements are collected. Prebuilt fiducial marker systems, like ArUco and AprilTags, remove a lot of development work. Work will be shown to determine if the accuracy and precision of fiducial markers is sufficient for this application. From the literature, most research has used fiducial markers for precision landing, where the pose of the marker is flat, so additional challenges may be experienced depending on the severity of the marker ambiguity problem. This may need to be explored in the final solution.

3.3 UAV Control

This section describes the overall strategy and challenges of contacting a surface. This includes aspects of automatic navigation, low-level control messages, and control methods.

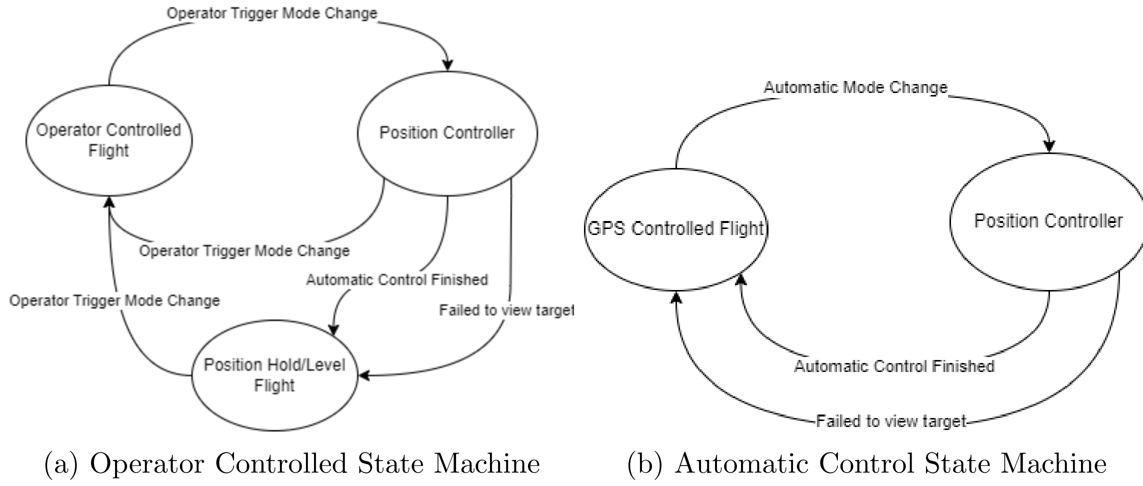
3.3.1 Controlling the UAV into range of the Target

The first step is to get the drone close enough to the marker so that its camera can detect it. This can be done in different ways, depending on the specific application and the available equipment. For example, a human operator can manually guide the drone toward the marker, adjusting its position and orientation as needed. Alternatively, GPS navigation can be used to direct the drone along a pre-determined trajectory that brings it close to the target. In either case, the drone must be positioned and oriented in such a way that the marker is within range of detection by the camera and in the field of view of the camera. For example, if the drone is positioned too close to the marker or the field of view of the camera is very narrow, then the drone may need to be pointed more precisely or the marker may be too large in the image frame. If the drone is too far away or the field of view of the camera is too large, the marker may be too small for the imaging system to detect the marker and generate an accurate position estimate. The field of view of the lens should be considered. In general, a narrow field-of-view lens may be used if precise position control is achievable otherwise a wide angle lens should be used.

Figure 3.3a shows how the drone would be controlled if an operator were to fly the drone to a target. An operator would manually trigger a mode change to enable the automatic position controller and manually regain control either directly from the position controller or from a loitering state. After the position controller has finished the drone would go to a loitering state by performing a position hold or maintain level flight. If the local position estimation of the drone is generated independently of having the target in view, then a position hold may be achieved. Otherwise, the best the drone could do is maintain level flight until an operator takes control.

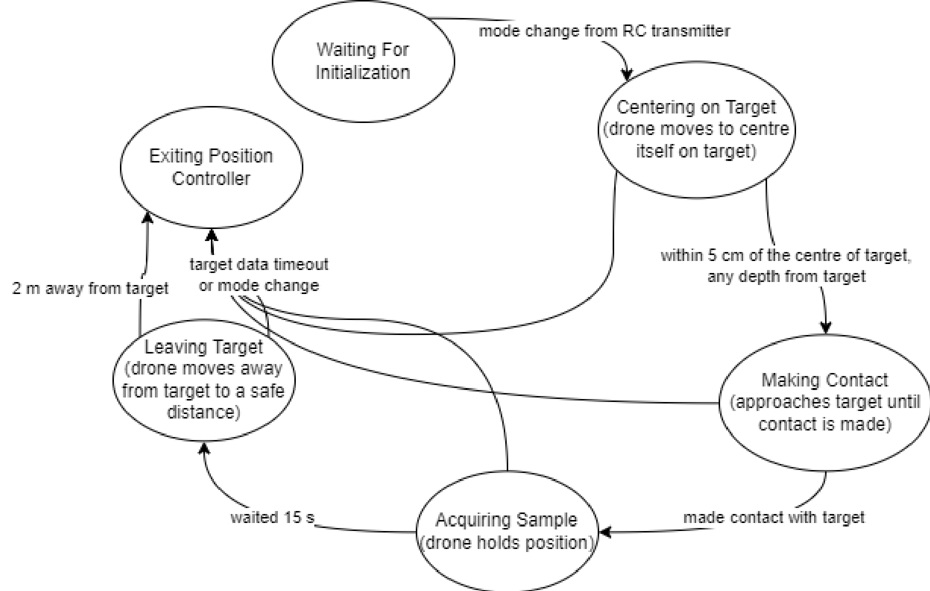
Figure 3.3b show how the drone would be controlled if GPS was available. In this case, after the position controller had finished or there was a failure in viewing the target, the GPS navigation would immediately take over and the drone would continue

the mission. This state machine in practice may need to be improved further after testing (to deal with automatic retries) but is outside the scope of this work.



(a) Operator Controlled State Machine

(b) Automatic Control State Machine



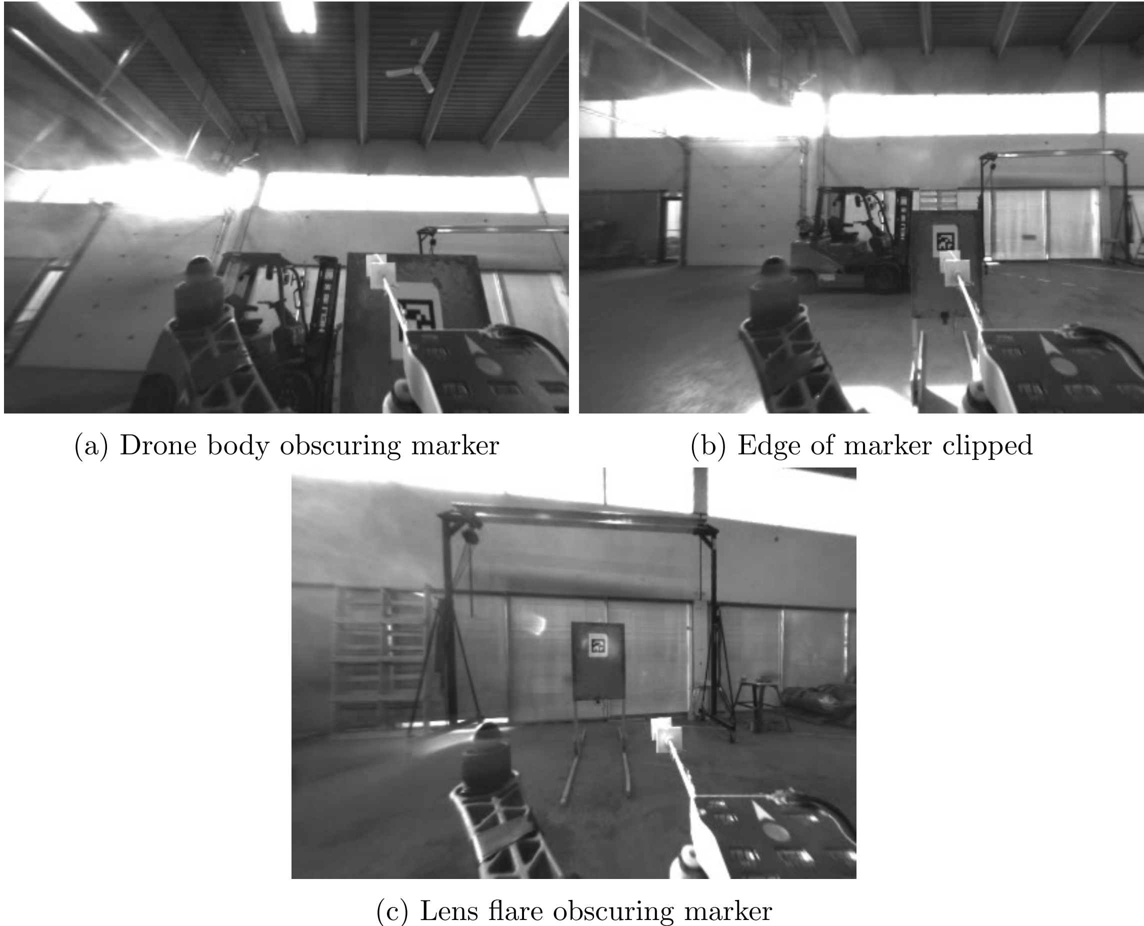
(c) State Machine to generate position setpoint

Figure 3.3: Control State Machine. Fig 3.3c shows the inner workings of the Position Controller block referenced in Fig 3.3a and Fig 3.3b

3.3.2 Challenges with Making Contact with the Target

The process of navigating and controlling a multirotor with a fiducial marker requires several steps, each of which must be executed carefully.

After the drone is pointed toward the marker, it must keep the marker in view at



(a) Drone body obscuring marker

(b) Edge of marker clipped

(c) Lens flare obscuring marker

Figure 3.4: Failed detection of markers during drone approach

all times. This requires the drone to continuously adjust its orientation in response to its movements. Specifically, the drone must keep the marker centred in the camera's frame to ensure that it remains within the camera's field of view. This can be challenging, especially if the drone is moving rapidly or erratically due to wind. In addition to keeping the marker in view, the drone must also ensure that the marker is not obscured by the vibration arm or propeller arms. This can be especially challenging if the drone's arms are large or if the camera is positioned in a way that makes it difficult to see the marker. Figure 3.4 shows the drone approaching a target with part of the drone in the camera frame.

Once the marker is located, the next step is to make contact with it. This requires careful planning and execution to ensure that the drone makes contact in a way that

is safe and effective. Specifically, it is best to make contact at a 90-degree angle to the wall. This is to maximize the force applied by the drone through the axis of the arm to the vibration sensor, prevent slippage of the sensor on the wall, and prevent the drone's propellers from contacting the wall. To square the drone up with the target, the drone's yaw may be controlled to keep the target at the centre of the image frame and then move the position of the drone to a staging position square with the wall.

Once the drone has reached the staging position, it may approach the wall slowly and make contact. It is important to approach the wall slowly to avoid damaging the drone. Approaching too quickly may cause the drone to overshoot or the sensor to not make good contact. After making contact, the drone should wait for a period to collect data. The length of the waiting period depends on the specific application and the amount of data that needs to be collected. In this research, the drone was set to wait for 15 s collecting a sample. Figure 3.3c shows the internal state machine of the position controller.

Finally, after collecting data, the drone should pull back from the wall. At this point, the drone should then continue the mission either through GPS-aided navigation or pilot control.

3.3.3 Autopilot Control Messages

The autopilot accepts and sends a handful of messages. The relevant Mavlink messages for this project are in Table 3.1.

There are two ways of controlling the UAV that this work explores. Method one is sending attitude setpoints to the UAV. Method two is sending vision position estimates and then sending position targets to the UAV. The first method is referred to as the attitude setpoint or attitude controller method. The second method is referred to as the position setpoint or vision position estimate method for this thesis work.

Table 3.1: Different Mavlink messages sent to and from the autopilot

Message Name	Sender	Description
Heartbeat	Autopilot	Heartbeat indicating that the autopilot hw is online. Also includes the current mode of the autopilot
Attitude	Autopilot	Roll, pitch and yaw information from autopilot
Set Attitude Target	Onboard Computer	Message to set desired attitude and thrust of the drone. Attitude is passed as a quaternion, thrust ranges from -WPNAV_SPEED_DN to WPNAV_SPEED_UP parameters in autopilot.
Set Position Target Local NED	Onboard Computer	Set a desired position based on EKF frame or drone body frame. The message also accepts a yaw or yaw rate.
Vision Position Estimate	Onboard Computer	Update autopilot state estimator with a position estimate and attitude from visual odometry
Vision Speed Estimate	Onboard Computer	Update autopilot state estimator with a speed estimate from visual odometry

3.3.4 Control via Attitude Setpoint

Both autopilots also offer the ability to send raw attitude setpoint messages. Figure 3.5 shows the process of generating an attitude setpoint. Using raw attitude setpoint messages is a little bit riskier than using position setpoint messages. By using attitude setpoint messages the autopilot goes to the commanded setpoint and this could easily be set outside the flight envelope of the drone.

This method uses the low-level attitude controller built into the autopilot, so there is very little tuning required on the autopilot software. The majority of the tuning would be on the onboard computer that generates the attitude setpoint. The autopilot runs an attitude controller, yaw rate controller, and vertical velocity controller.

The overall attitude setpoint controller can be seen in Figure 3.6. Once a target is captured in a frame, the drone's position is determined. This position is then needed to be filtered to remove noise in measurements. There will be a lot of high

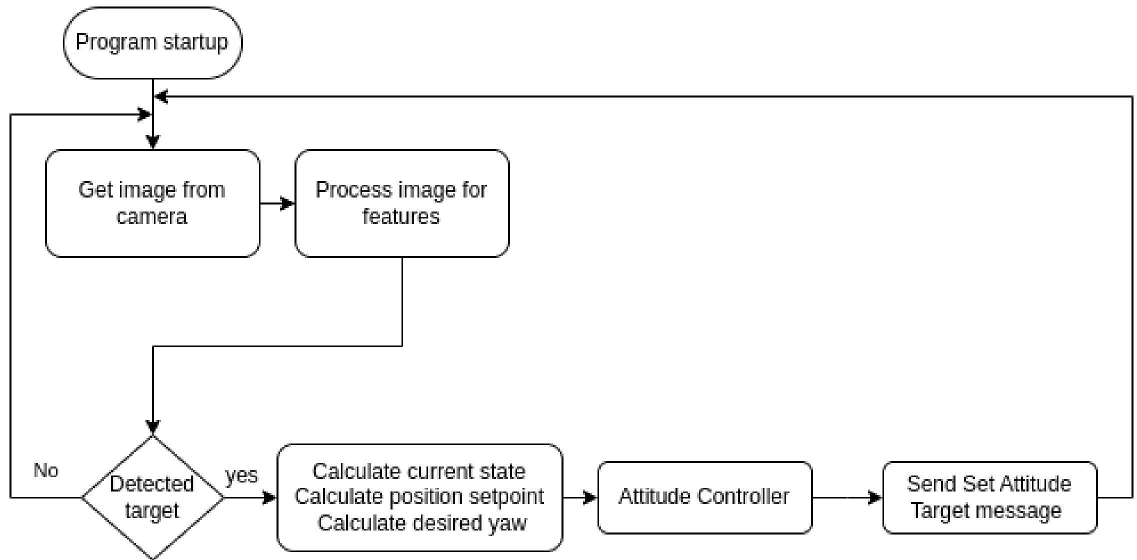


Figure 3.5: SW steps for Attitude Setpoint Controller

frequency noise in the position and if this is not removed, this noise will cause issues in position control and velocity estimation. Estimating velocity by simply finding the difference between subsequent position inputs would result in incorrect and swinging velocity estimates. A low pass filter may be sufficient to remove most of the noise in the position, but a Kalman filter is more useful. A Kalman filter generates a position estimate and a velocity estimate from the position inputs. Using a Kalman filter removes the step of manually determining a velocity estimate. An Unscented Kalman Filter (UKF) library is used as a C++ library for it was readily available.

The position estimate can now be compared with the position setpoint. This comparison produces a position error. This position error is how much the drone is commanded to move. From this point, a desired velocity is generated by multiplying the position error by a gain. The greater the position error the larger the velocity setpoint. This velocity setpoint is then compared with the velocity estimate from the Kalman Filter to produce a velocity error.

The velocity error then generates an acceleration setpoint, by applying a proportional and integral gain. One of the needs for an integral gain is to combat external effects such as wind.

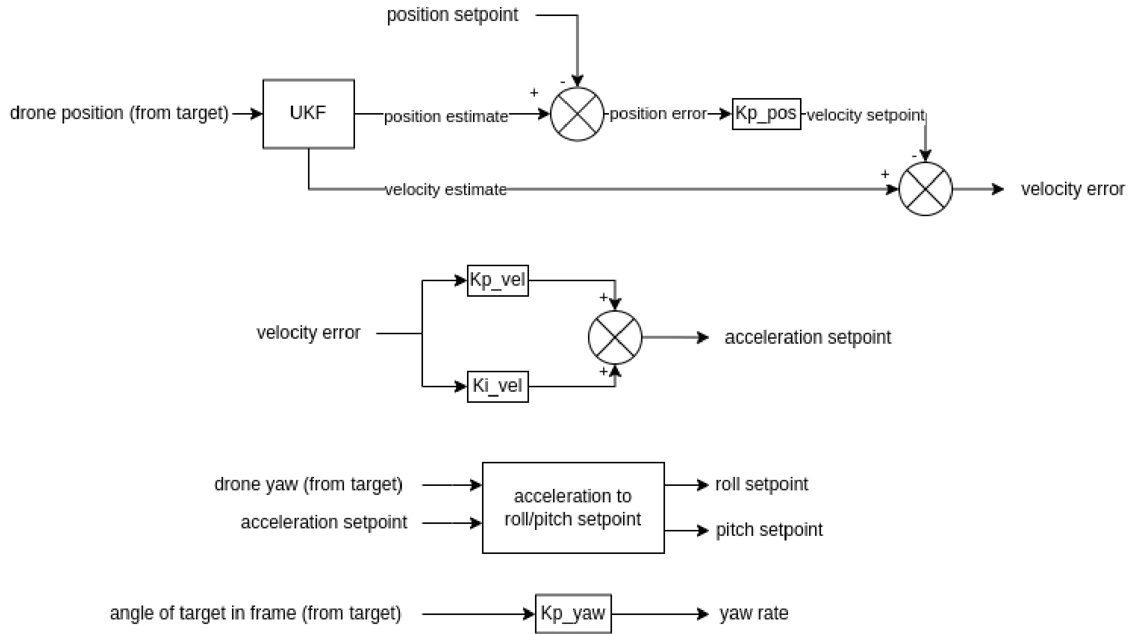


Figure 3.6: Attitude Controller Diagram

The acceleration setpoint is in the coordinate frame of the target. The drone may be yawed in that frame so a rotation needs to be applied and finally, a roll and pitch setpoint may be generated directly from the desired rotated roll and pitch.

In addition, the drone trys to keep the target at the centre of the frame by applying a proportional gain to the angle of the target in the frame to generate a yaw rate.

During each stage of this process, bounds are placed to help ensure controlled flight. The velocity setpoint is bounded to 1 m/s, the roll and pitch setpoint are bounded to 5 degrees, and the yaw rate is bounded to 30 deg/s. These bounds along with using the autopilot's attitude controller, make this controller safer to use as opposed to a far simpler PI controller that would generate an actuator setpoint directly from a position error input.

3.3.5 Control via Position Setpoint

PX4 and Ardupilot offer the capability to integrate visual position estimations into their state estimator. The state estimator operates using an Extended Kalman Filter (EKF) algorithm that integrates gyro, accelerometer, magnetometer, GPS position,

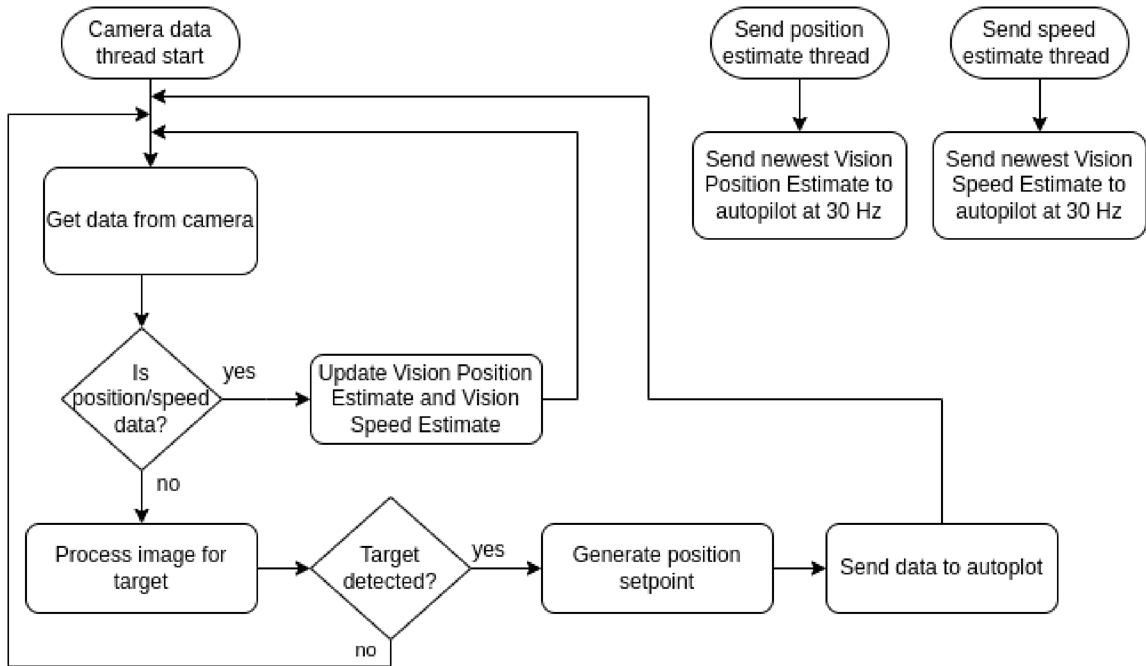


Figure 3.7: SW steps for Vision Position Estimate and Position Setpoint based controller

GPS velocity, GPS height and barometric height into the predicted state.

For an external vision system to effectively contribute to the state estimation, it is necessary for the vision system to provide position and speed estimates with uncertainties. Once the drone receives these messages, the information is fused with the data from other sensors. The autopilot uses this data for precise position estimates. Once the autopilot fuses the position and speed estimates it can do a position hold. With this position hold the set_position_target_local_NED message may be sent with a desired position. This message accepts a desired position as an offset from the drone's body frame. The software process is shown in Figure 3.7.

3.3.6 Proposed Control Approach

Both control approaches are explored in the work. The attitude controller has the advantage of giving low-level control of the UAV. This allows the ability to implement more complicated control schemes. In addition, no work needs to be done to generate and integrate a vision position estimate from the camera system into the autopilot

state estimator. The disadvantage of the attitude controller is that the proposed design localizes the UAV from seeing the fiducial marker in the frame. So if the marker is not detected or leaves the frame the system no longer has a reliable position estimate of where it is. The second control method of using the vision position estimate is also very promising. Especially if using an Intel Realsense T265, Ardupilot, and the work from [61] as a starting point. The first method allows for more complicated low-level control schemes, but this may only be needed for more complicated vibration monitoring mechanisms, i.e. ones with a changing centre of mass. Another aspect that could be explored in the future is using the position and velocity outputs from the Intel Realsense T265 and having that feed into the proposed attitude controller.

3.4 Vibration Monitoring

During the development of a vibration monitoring apparatus, several conceptual ideas are explored. The primary objectives for this project were to detect vibrations via drone but prevent the sensor from detecting vibrations generated by the drone. Whatever solution was selected needed to be developed in conjunction with an aerial platform. The size and weight of the base drone determine the weight and dimensions of the vibration-monitoring payload. The weight of the payload has to not be greater than the max payload capacity of the drone and the deployment of the payload must ensure that the centre of gravity on the drone is maintained throughout the deployment. In addition, when flying a drone next to other objects, the wash from the motors causes turbulent air making control of a drone more difficult. So keeping the drone reasonably far away from the object it is measuring is something else to prioritize.

To address these challenges, several potential solutions were considered. One approach involved permanently affixing a sensor box to the infrastructure. This would enable the sensor to be firmly coupled to the infrastructure and eliminate the possibility of detecting vibrations generated by the drone. Another option involved using

an electromagnet to secure the sensor to the infrastructure while attaching the sensor to the drone using a string. This would also allow the sensor to be strongly coupled to the infrastructure and yet remain securely attached to the drone, but decoupled from the drone's vibrations.

A third approach involved using a long arm with vibration-dampening foam to separate the sensor from the drone. The foam would absorb any vibrations generated by the multi-rotor, preventing them from being detected by the sensor. It would also allow for small positioning and attitude errors during contact. This solution would also allow the sensor to be positioned at a distance from the drone, and eliminate any changes to the centre of mass during deployment.

3.4.1 Permanently Affixed Sensor Box

The permanently affixed sensor box is a very good solution due to being completely decoupled from the drone. The feasibility of this approach, however, was hindered by the difficulty of physically attaching the box. If this were done by a human then this may be a high upfront cost. As an alternative, a drone could attach the sensor box using a strong adhesive.

The sensor box was envisioned to contain an MCU, accelerometer, Bluetooth Low Energy module, and battery. To conserve energy, the device would remain in low power mode until a drone arrived to collect measurements. Once the drone arrived, the device would collect data for a brief period and transmit it to a collection unit on the drone. The device would then revert to low power mode, consuming current at $\sim$1 uA, and draw approximately $\sim$30 mA when collecting and transmitting data. Assuming a 60 s measurement each day, a single AA battery with 900 milliampere hours (mAh) could last for approximately five years. With proper battery selection and data collection frequency, the sensor box could be designed to have enough energy to operate for the remaining lifetime of the infrastructure being monitored.

Another alternative to the above-proposed sensor box is to not use BLE, but some-

thing more long-range, such as LoRa, so that once the box was affixed a human or drone would never have to be near the box again. The box may need to be a bit larger, to accommodate more energy storage, but the benefit would be that a drone would be needed for installation and all future acquisition could be done from a base station.

While this approach offered the benefit of complete physical decoupling of the sensor from the drone, which would enable better data quality and ease of data collection without the need for precise drone positioning, the challenges of placing the sensor box and ensuring long-term adherence to the infrastructure ultimately led to its rejection as a solution for this thesis project.

This would be an excellent solution for new projects where a battery-operated vibration sensor is required, or for projects where the installation of these sensors would be carried out during a routine maintenance event.

3.4.2 Electromagnet Based Deployment Mechanism

Another potential solution is using an electromagnet to engage the sensor in the infrastructure. By using an electromagnet, the sensor can be strongly coupled to a piece of infrastructure and can be physically decoupled from the vibrations of the drone and ensure clean data collection. The device could be designed in such a way that even wind battering the drone may not be registered by the device. The device would still be tethered to the drone so the device can be made smaller and lighter since it can rely on the drone battery and processor.

However, there are several limitations to using an electromagnet. First, the device may need to be reset after deployment, which can be accomplished either by having the drone return to the operator or by developing a mechanism to reset the device in the air. Second, the sensor may dangle from the drone, negatively affecting drone flight dynamics. This can be especially problematic if the sensor is large and heavy. In such cases, it may be necessary to design a mechanism to maintain the centre of

mass.

Another limitation of using an electromagnet is that it does not stick to non-ferromagnetic materials, such as concrete. This necessitates the use of a tacky substance to attach the sensor to the infrastructure. However, this substance may need to be cleaned and/or reapplied after each data collection event, reducing the overall ease of use of the system.

Given the challenges involved, the use of an electromagnet was considered in part with the simple sensor arm solution.

3.4.3 Simple Sensor Arm with Vibration Dampening

A fixed sensor arm with a vibration dampener may also be used to collect a measurement. The end of the arm would have a vibration dampener and the tip would have an accelerometer. An electromagnet may still be used to secure the vibration mechanism to the structure. If no electromagnet is used, the UAV would need to apply all the force and positioning for the arm to ensure good contact is made during the sampling period. The main advantage of this method is that the arm is fixed so there are no changes in the centre of mass during flight. An arm with a changing centre of mass may need to measure how the centre of mass changes and incorporate that into the control algorithms. A fixed arm is therefore lighter due to its simplicity. This means the drone has a greater power-to-weight ratio or that a smaller drone may be used. A fixed arm design would be a lot simpler and inexpensive to make, yet yield satisfactory results.

3.4.4 Proposed Vibration Monitoring Mechanism

Using a simple sensor arm with vibration dampening is the proposed method for collecting data in this project. Compared to the other options, the drone would have increased maneuverability and better battery life and the design complexity of the arm is far lower. The main disadvantages are that the position, attitude control

and vibration isolation may need to be stronger to compensate for the holding and isolation benefits that a more complicated vibration acquisition method may provide.

3.5 Functional Requirements and Specifications of Experimental Work

This thesis work is split into the four major aspects discussed in this chapter. The overall goal is to create a proof of concept platform that would be able to acquire a vibration measurement from an artificial vibration source in a GPS-denied environment.

An aerial platform is required. This platform should be able to efficiently deploy the vibration monitoring mechanism as a proof of concept. As a result of this being a proof of concept, the platform may be small enough to be deployed indoors, without the need for actually executing a full vibration monitoring mission. A GPS unit would not be required as another requirement is the ability to deploy the vibration monitoring payload in a GPS-denied environment. The computer and imaging systems are selected to provide satisfactory performance, whilst minimizing their size, weight and energy requirements.

The ArUco fiducial marker system will need to be characterized to determine the overall positioning performance when used to control and position a UAV.

Two methods of controlling the UAV were discussed in this chapter. The first was by sending attitude setpoints to the autopilot and the second was using vision position estimates and position setpoints. Both methods will need to be explored to determine if one is better suited for the application than the other. The method that allows for precise enough control to deploy the vibration monitoring payload is the method that will be used for actually acquiring vibration samples.

Finally, the experimental work shall also develop and characterize a vibration monitoring arm that will be affixed to the UAV. An artificial vibration source will be developed to aid in the characterization of the full vibration deployment mechanism.

Chapter 4

Hardware Setup and Testing

4.1 Aerial Platform

This section describes the hardware components that comprise the full aerial platform utilized in this project. The platform that was used for the majority of testing is shown in Figure 4.1. The aerial platform encompasses an array of interconnected systems, which include the airframe, autopilot system, UAV hardware, pilot control and communications system, onboard computer, and imaging system.

The base drone contains the physical components that allow the drone to fly and is meant to be a starting point for the aerial platform. This base drone includes the airframe, propulsion system, and power source, which form the backbone of the aerial platform.

The autopilot system enables the UAV to be controllable and operate automatically through the air. It incorporates a range of sensors, including accelerometers, gyroscopes, and magnetometers, to determine the aircraft's speed and direction. These sensors allow the autopilot to adjust the attitude of the UAV as needed. The autopilot system has a built-in MCU that performs the calculations required for stable flight and generates the command signals to the actuators.

The pilot control and communications system facilitates communication between the UAV and the ground station, allowing the operator to monitor the aircraft's performance, directly control the aircraft and make adjustments to its flight path

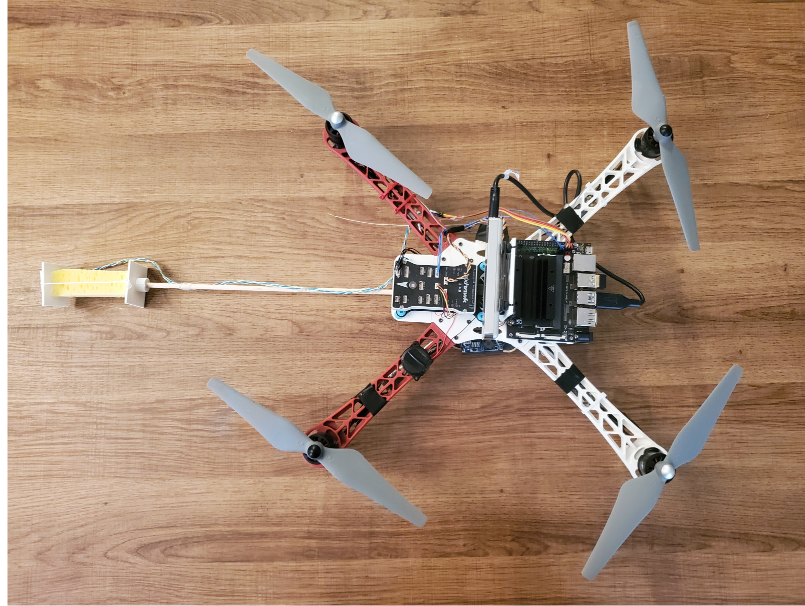


Figure 4.1: Aerial platform based off of DJI Flamewheel 450, Pixhawk, Intel T265, and Jetson Nano

as necessary. It utilizes a range of communication protocols, such as Wi-Fi, 900 MHz telemetry, and a 2.4 GHz radio control link, to ensure a reliable and redundant connection.

The camera plays a vital role in capturing visual data during flight and providing the onboard computer with images of the aircraft's surroundings. The onboard computer then processes the images for markers, generates a desired position, and then commands the autopilot to that position.

The next sections explore each of these components in more detail, highlighting their features and functionality. Figure 4.2 provides an overview of the hardware layout.

4.1.1 Airframe

The research project went through two airframes. The first was a larger hexacopter and this was followed by a smaller quadcopter.

The Hobbyking Aperture hexacopter has a wheelbase of 800 mm and is equipped with 15-inch propellers. The aerial vehicle is powered by a 16000 mAh battery that

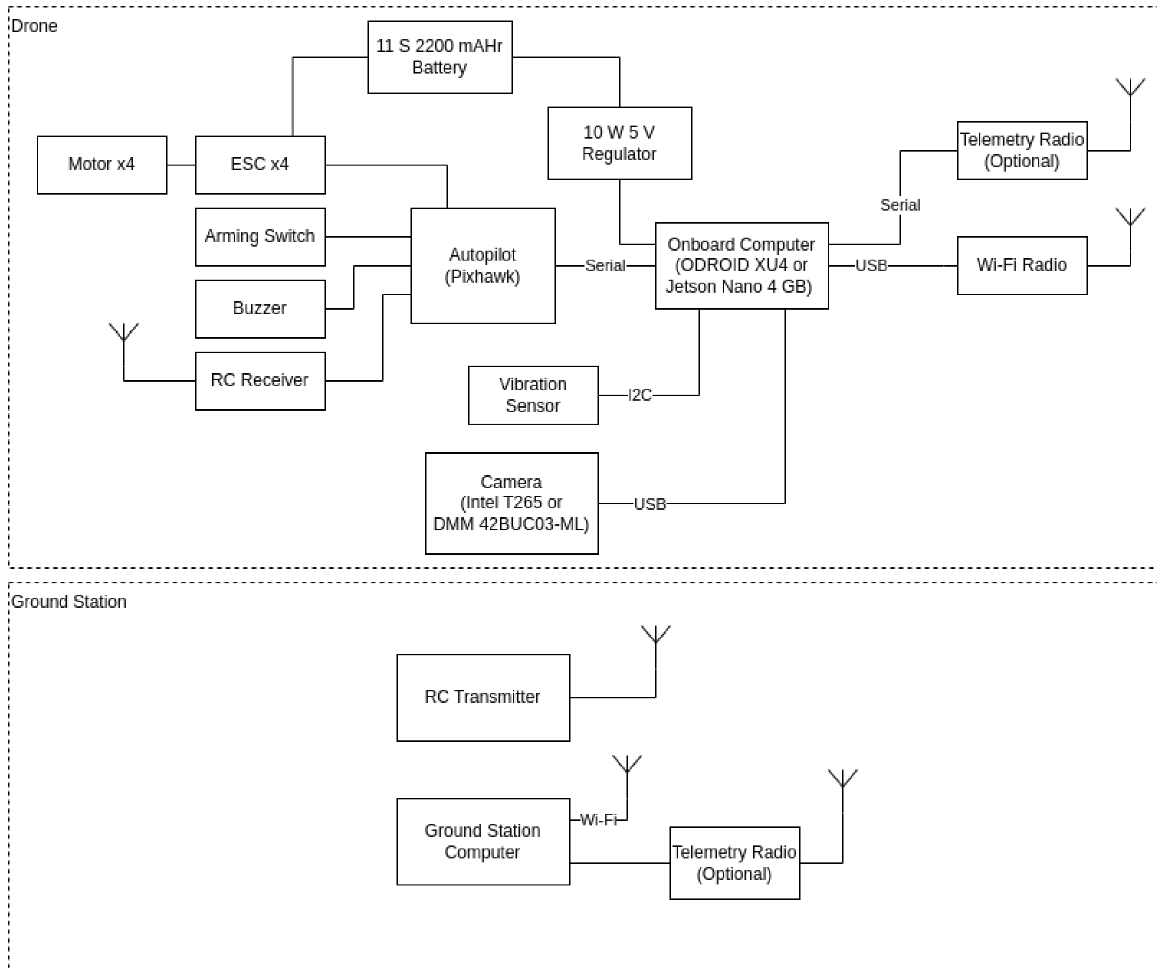


Figure 4.2: Aerial platform HW block diagram

facilitates a hover time of 25 minutes. The hexacopter has a total mass of 4.8 kg and is capable of carrying a maximum payload mass of 1.5 kg. The payload capacity of the hexacopter enabled the integration of a more complex payload. However, the larger size of the hexacopter necessitated the use of a longer arm. The elongated arm led to a substantial shift in the center of mass of the hexacopter, prompting the installation of a counterweight on the arm to achieve a balanced distribution of weight. This arm is shown in Figure 4.12.

The DJI Flamewheel F450 is a small quadcopter that is meant to be used as a customizable drone with FPV in mind. It was purchased as a kit and included all the necessary components to assemble the drone. The kit includes 9.4-inch blades, 920 kV motors, and ESCs. This propulsion system in combination with a 4S 2200 mAhr lithium-polymer (LiPo) battery, allows for a takeoff mass up to 1.6 kg. The base drone weighed 1.1 kg and allows a flight time of $\sim$8 minutes. Due to how light the arm was, repositioning the battery was all that was required to maintain the centre of gravity at the centre of the drone.

The frame of the DJI Flamewheel F450 kit comprises two pieces of carbon fibre plate that sandwich the four arms of the drone. The bottom plate incorporates the power distribution for the electronic speed controllers (ESCs) of the drone. The top plate is not utilized in this project and is replaced with a custom 3D-printed piece that is designed to mount the required components for this application. Figure 4.3 shows the custom 3D-printed top plate. This top plate houses the autopilot on vibration-dampening mounts, the onboard computer, the Intel T265 camera on vibration-dampening mounts, the RC receiver, and the arming switch.

4.1.2 Low-Level Command and Control of the Platform

The aerial systems were controlled through a variety of means. Firstly, a handheld radio controller that operated in the 2.4 GHz spectrum was utilized. This controller was available at all times to the operator and allowed the operator to take manual

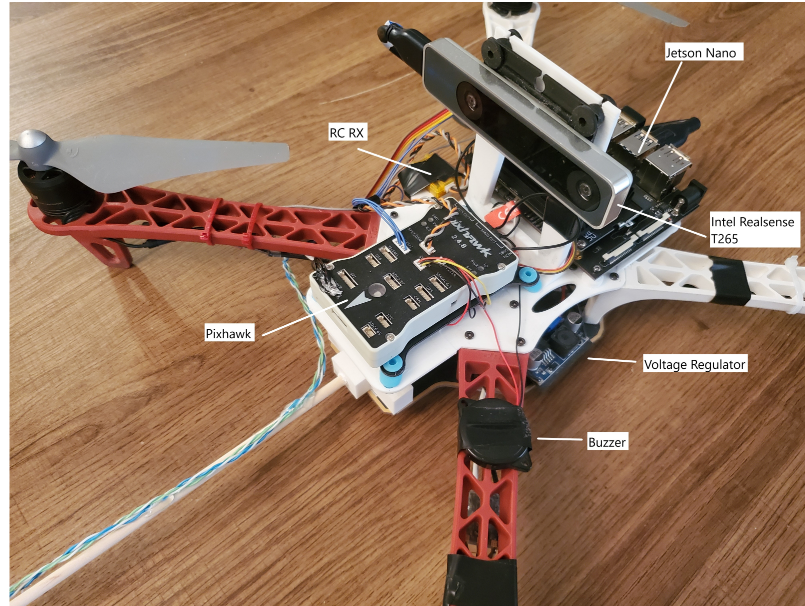


Figure 4.3: DJI Flamewheel 450 close up

control at any given moment. It was also used for take-off and bringing the aerial system within the visual range of the target.

Secondly, a 900 MHz telemetry link was established with Sik radios, which allowed communication between the autopilot and ground control station software. This link was used to change tuning variables for the autopilot, monitor the aircraft, and enable/disable the autopilot as necessary.

Thirdly, an Ethernet cable may be physically connected to the onboard computer on the drone. This allowed the imaging and navigation program to be started and stopped. Additionally, it allowed the raw and processed data to be downloaded from the computer.

In the final implementation, the 900 MHz telemetry link and requirement for the ethernet link were eliminated and replaced with a 2.4 GHz WiFi link. The imaging computer ran a mavproxy server, which allowed it to forward telemetry information from the autopilot to the ground control station laptop and the onboard navigation software. The networking and control links are shown in Figure 4.4.

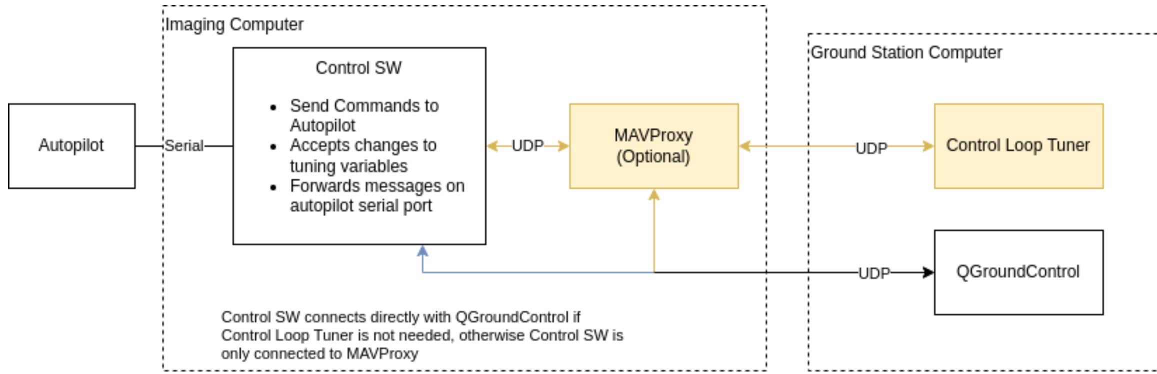


Figure 4.4: Aerial Platform networking configuration

4.1.3 Camera Systems

Two camera systems were utilized in the project. The first camera system, DMM 42BUC03-ML, was from the imaging source. This camera system was lightweight, weighing only 7 g, and had small dimensions of 30x30x15 mm. It was equipped with a global shutter and featured a 1/3-inch Onsemi CMOS MT9M021 sensor. The camera was capable of capturing 1,280×960 (1.2 Mpx images, up to 25 frames per second. It was a monochrome board camera with a USB 2.0 interface. The camera was controlled via openCV, with each frame being captured and analyzed. This camera system was more thoroughly characterized than the final camera used in the project. By taking an image of a clock on a computer screen, it was found that the rough latency with this camera was ∼102 ms for a 0.3 Mpx image and ∼169 ms for a 1.2 Mpx image.

The 2nd camera system was the Intel Realsense T265 camera. It features two fish-eye cameras that capture a wide field of view of 170 degrees, allowing for robust tracking and mapping in indoor environments. Each camera captures images at a resolution of 848x800 (0.68 Mpx). The camera system uses Intel's Visual Inertial Odometry (VIO) to accurately estimate the camera's position and orientation in 3D space. This camera system is capable of tracking the movement of the device in six degrees of freedom (6DoF), including translation along the X, Y, and Z axes, as well as rotation around those axes. It also has built-in support for hardware-accelerated

depth sensing and can output dense depth maps up to a range of 6 m. The Intel Realsense T265 camera is compact and lightweight, considering the sensor suite held within. The size makes it ideal for use in small drones or other mobile robotics platforms. It also features a USB 3.1 Gen 1 Type-C interface for easy integration with a host computer or embedded system.

4.1.4 Onboard Computer

The Odroid C2 is a single-board computer (SBC) produced by the Korean company Hardkernel. This computer was initially used and took ˜75 ms to process a 0.3 Mpx. It is lightweight and inexpensive. The newer Odroid C4 may be purchased at a retail price of USD 62.95.

The Jetson Nano 4GB is an SBC produced by NVIDIA, designed for use in AI and machine learning applications. The Jetson Nano 4GB is known for its small form factor, low power consumption, and efficient performance in AI and machine learning workloads. It supports a variety of AI frameworks and libraries, such as TensorFlow, PyTorch, and OpenCV, and is commonly used for applications such as robotics, autonomous vehicles, and smart devices. The Jetson Nano was used in the final solution as it has more processing power. It took $\sim$54 ms to process a 0.68 Mpx image. The cost of a Jetson Nano is USD 149.00.

4.2 Evaluating ArUco

The ArUco Library was developed by the University of Cordoba. The library is capable of detecting different markers including its custom marker and AprilTags. Each ArUco marker contains a marker id. In the vibration monitoring application, this allows for each vibration site to have its marker ID dedicated to it.

The process of marker detection starts by thresholding the images. Then contours are found and marker candidates are determined. These candidates are then filtered to remove markers that are too large, small, or too close together. The marker detector

library allows for modifying a multitude of parameters.

The next step in the process is to analyze the inner code of the marker. A marker, that is generated from the 6x6 marker dictionary, is subdivided into 36 smaller squares. The marker detection algorithm then determines the code inside the marker, the marker id, and then determine whether or not it is part of the dictionary. The use of dictionaries and coded bits prevents false positives that the basic methods of just a rectangle or square would fail at.

Next, the algorithm returns a list of all the markers. The marker with the correct ID is then selected and pose estimation is performed on the correct marker. The ArUco Library uses the OpenCV Solve PnP function. PnP stands for Perspective-n-Point. This function iteratively estimates the pose of a marker, based on the physical object points and their respective projected points on the image. The SolvePnP function returns a translation and rotation vector of the marker relative to the camera frame.

4.2.1 Precision and Accuracy of ArUco

It is important to evaluate and understand the functionality, accuracy, and limits of the ArUco system. To this end, a testing configuration was established, which involved the use of long rails, 8020 pieces, a 3D-printed camera holder, and target holders. The purpose of this setup was to capture images of targets at varying distances and angles to simulate real-world scenarios where the ArUco platform may not be directly in line and square with the target.

Images of the marker were taken from 1-6 m at 1 m intervals from the camera. At each location, the marker was rotated from 0 to 60 degrees at 10-degree intervals. For each measurement, 100 photographs of the target were taken. This data is represented in Figure 4.5. The data showed that depth error increased the farther away the target was from the camera. The data also showed that as the marker angle changed the depth error increased. Both of these are expected as the depth is calculated by measuring how many pixels the marker takes up in the image. The farther the marker

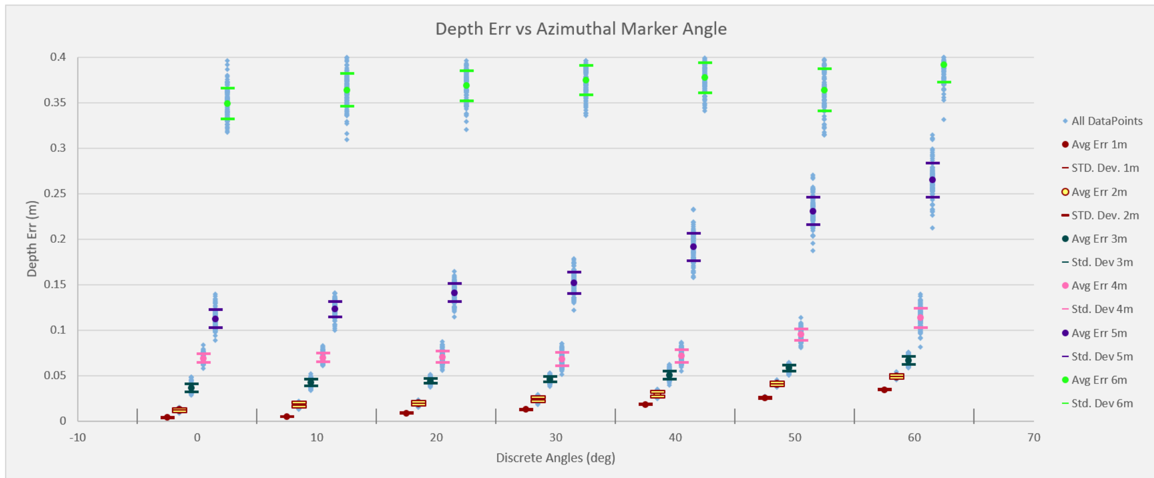


Figure 4.5: Depth error of an ArUco marker at different depths and marker angles

is away from the image the smaller it is in the image and the number of pixels it takes up in the image is smaller. Any error in measuring the contours of the marker has a larger effect on calculating depth. Similarly, markers in an angle occupy fewer pixels in the image. Overall this data indicates that the depth error within 1 meter and a 60-degree angle will be less than 5 cm. At smaller angles, this depth error is negligible. The arm that is designed should all for a few centimetres of error when positioning.

Figure 4.6 shows the angle error for each marker. This is using the same data as collected previously, but instead of measuring depth, it measures the angle of the marker in the frame. This Figure shows an interesting result that there is a larger angle error the closer a marker is square to the camera. This error also is greater the farther the marker is away from the camera, but the primary driver for marker angle error is the marker angle itself. The 6-meter data from the graph shows that when the marker is square to the camera (0 degrees), the marker can be incorrectly detected at even -10 degrees. When the marker has a larger angle to the camera like at 60 degrees, the error is less that 2 degrees. In practice, this means that a deadband may need to be implemented when doing yaw control when the marker is square to the camera.

Figure 4.7 measure the translation error in the X-axis at different distances and

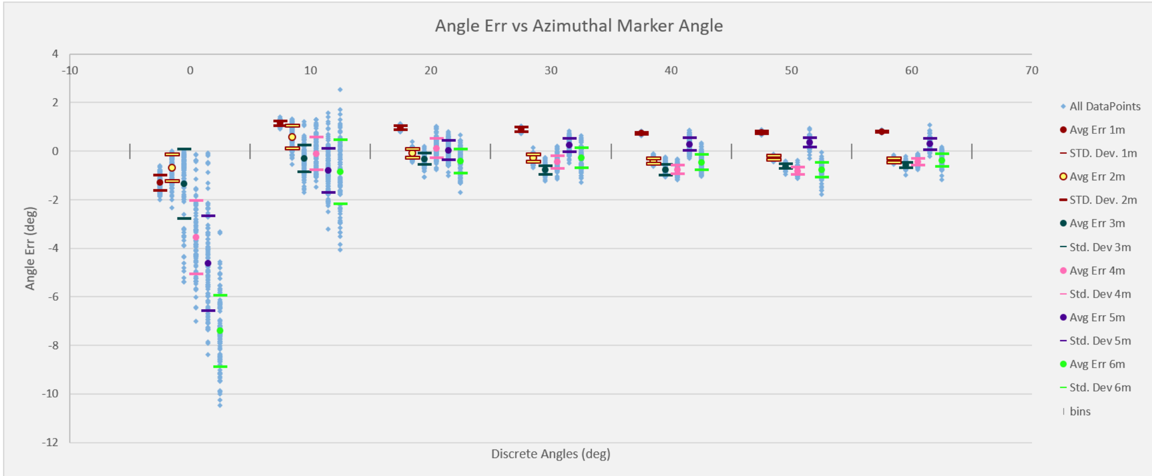


Figure 4.6: Angle Error vs Marker Angle for different depths

angle in the frame. The translation error in the Y-axis should be similar to the X-axis as such no measurements were taken for that axis. The data shows that when the target is not centre in the frame the translation error is minor, less than 5 cm when the drone is yawed 10 deg away from the target. Targets when not in the centre of the frame are exposed to more lens distortions. Later sections detail the actual control methodology of the drone and the main goal is to always keep the target in the center of the camera frame, so the drone is yawed 0 deg away from the target. The data collected continued to show that the ArUco marker system along with the camera would be adequate for visual servoing.

From the data, it was determined that the system would be able to localize the target adequately. The largest problem that occurred was pose jumping of the marker due to the marker ambiguity.

4.2.2 Marker Ambiguity Problem

The marker ambiguity problem refers to the challenge of accurately identifying the pose of a marker as there may be more than one valid solution. When the algorithm switches between two solutions, this is known as pose jumping. The problem is worsened with lower-resolution images or image blur. Accurate corner estimation

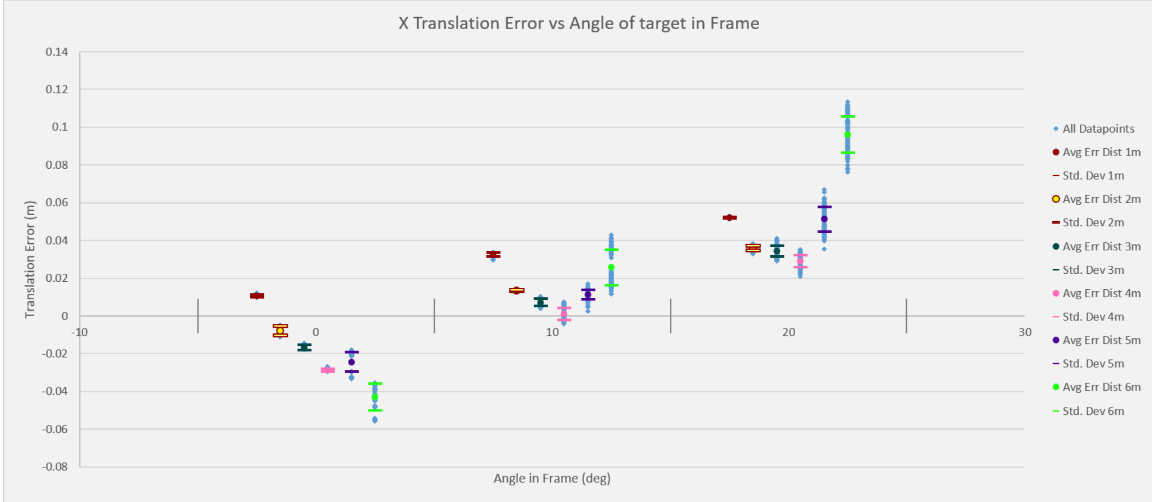


Figure 4.7: Translation Error in X vs Angle in Frame

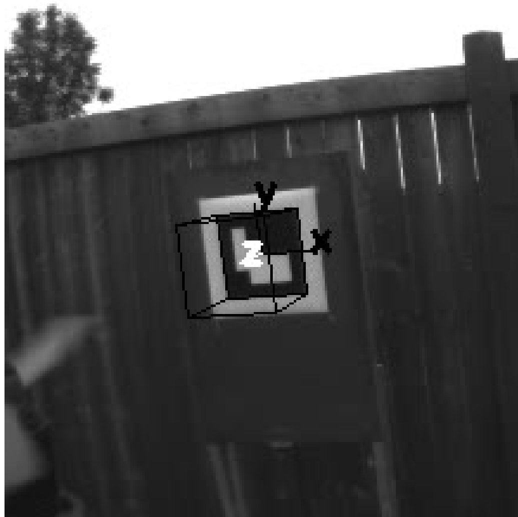
mostly alleviates the problem, but that is difficult to attain with changing lighting and motion conditions.

Figure 4.8a shows a left and right image taken at the same time in flight from a stereoscopic camera. Figure 4.8b shows the correct pose being estimated, but Figure 4.8c shows the incorrect pose being estimated. The left image predicts the drone's NED coordinates are (-1.09, 0.46,-0.12) which is correct. The right image predicts the drone's coordinates as (-1.16, -0.46, 0.12). If the drone were to try and control itself from the incorrect data it would travel in a direction that would take it farther from where it should be.

A possible solution to this problem is the MarkerPoseTracker, which employs the last marker pose as the starting point for the iterations of the solvePnP function. However, this approach may encounter issues if there is an incorrect starting point, especially when the target angle is close to 0 degrees, as shown in Figure 4.6. This can lead to the propagation of incorrect solutions, which can affect the accuracy of the tracking system.

Further solutions to eliminate the problem were attempted. The use of depth information from stereo cameras was incorporated to aid in selecting the correct solution. By analyzing the distance between the marker and the camera, the tracking system

(a)



(b) Zoomed in on the left image with correct pose

(c) Zoomed in on the right image with incorrect pose

Figure 4.8: Marker ambiguity in a stereoscopic image. Both images were taken at the same moment in time, yet show different position estimations.

can eliminate ambiguous solutions that do not correspond to the actual position and pose of the marker. Still this method would struggle when the target angle is close to 0 degrees.

Furthermore, environmental clues can also be utilized to eliminate incorrect solutions. For instance, the marker's x-axis can be set to point upwards, and the z-axis can never be directed away from the camera when visualizing servoing a multi-rotor in front of the target. By incorporating these constraints into the tracking system, the accuracy of the marker identification may be improved for some obvious erroneous solutions, but not the ones that primarily affect lateral position control.

Since the problem is directly linked to corner estimation, ArUco also generates markers that help to further improve the accuracy of corner estimation. This method was briefly tried but also produced some solutions to the marker ambiguity problem.

There are a few solutions proposed in other papers. [89] describes the problem well and [43, 90] propose some solutions. The problem was mitigated in this research by using larger targets and better lighting, but this issue has not been eliminated. The use of different style markers, multiple markers, or 3D markers may eliminate the issue and should be explored in future work.

4.3 UAV Control Implementation

There were two methods of controlling the UAV that were explored in this work. The first method was using the information from the target to generate a position estimation and then sending attitude setpoints to the drone. The second method was generating a position estimation message and then sending that to the drone along with a target position message.

4.3.1 Controlling UAV using Attitude Setpoint Messages

The attitude controller as proposed in section 3.3.4 was implemented and run on the onboard computer. The program was implemented in cpp and split into a few sub-

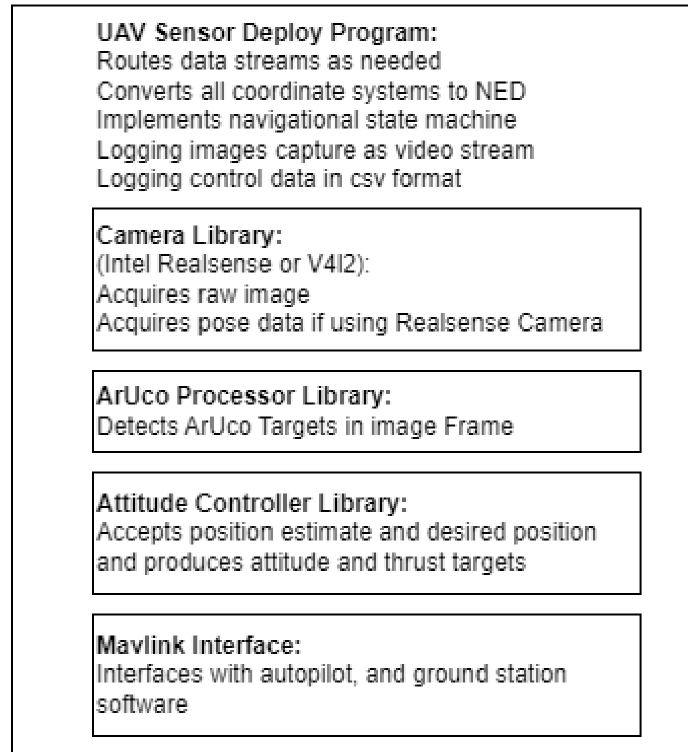


Figure 4.9: UAV Sensor Deploy Program and Libraries Summary

libraries for added modularity and testability. A summary of the duties of the main program and library is shown in Figure 4.9.

An ArUco processor library was implemented. This library is initialized with camera calibration parameters and marker size. The ArUco processor library is then called each time an image is received. The library takes the image, applies any required camera calibrations, finds a marker in the image, calculates the pose of the marker and then returns the found marker with its pose.

Two different camera libraries were used. A library to implement the Intel Realsense API was created. This library triggered a callback in the main program whenever a full data frame composed of 2 image frames and a pose frame was received from the Realsense camera. The second camera library was used to interface with any V4L2 camera. This library was already available online and just needed a few modifications to make it CMAKE-compatible.

A multithreaded Mavlink interface library was implemented. This library allowed

multiple threads to send and receive Mavlink messages over serial or UDP interfaces.

An attitude controller library was implemented. This library implemented the controller in Figure 3.5.

Finally, all of these programs were bundled together in a singular program that would collect, process and route data between the camera, autopilot and ground station. The program would also implement the navigational state machine as seen in Figure 3.3c.

The attitude controller has parameters that need to be adjusted. In addition, the attitude controller has feedback that is valuable for debugging. Three custom Mavlink messages were created; A control gains message for modifying the control gains; a target NED message for where the target is in the frame and where the controller is trying to send it; a control targets message that details the current estimated velocity, desired velocity, desired acceleration, desired roll, and desired pitch. These custom messages were available on the ground station using a Python program. This program allowed the viewing of feedback messages and adjusting the tuning gains.

In addition, a simulator was used to verify that the position controller worked. Ardupilot has a Python-based simulator with a generic UAV model. Since the attitude controller library was separate from the main program, a separate wrapper program was developed for the attitude controller library. This wrapper would place a target at coordinates (0,0,0) in NED. It would then take the local position of the UAV as reported by the Ardupilot simulator and would report what the expected raw pose data from the target would be. This raw pose data would then be used along with user-entered desired position data to generate attitude and thrust targets by the attitude controller library. This simulation was in a completely noise-free and wind-free environment. The simulation always knew where the target was with complete precision and accuracy. The attitude controller worked well with this simulator and was able to control the UAV to the desired locations. The simulator and simulator wrapper worked well to prove that the underlying math would work but did not take

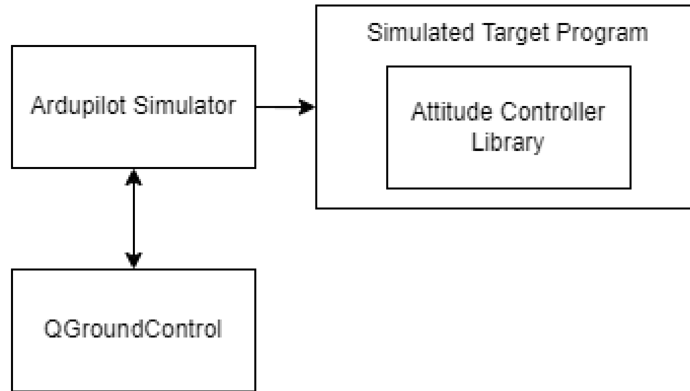


Figure 4.10: Simulator Setup

into account real-world noise and issues. The simulator was most valuable in finding issues with coordinate transformations and correct signs. A block diagram of the simulator setup can be seen in Figure 4.10.

In addition, support programs and scripts were also generated. An ArUco camera calibration program was developed and a test data collection program was used for evaluating the ArUco marker system.

4.3.2 Controlling UAV using Position Setpoint Messages

Vision Position Estimate with Imaging Source DMM 42BUC03-ML Camera

To generate a vision position estimate using the DMM 42BUC03-ML camera a fiducial marker needs to be in the frame. When the marker comes into the frame a position estimate may be generated. This position estimate is then sent to the autopilot. This method with this camera alone proved to be ineffective and is described in the next chapter. As such the low-level implementation of this method is not detailed here as a complete solution was not developed.

Vision Position Estimate with Intel Realsense T265

The Intel Realsense T265 generates a position estimate regardless of having a fiducial marker in the frame. Nguyen in [61] already has a good starting point to build off from. This work is done in Python and already has the work done to obtain a

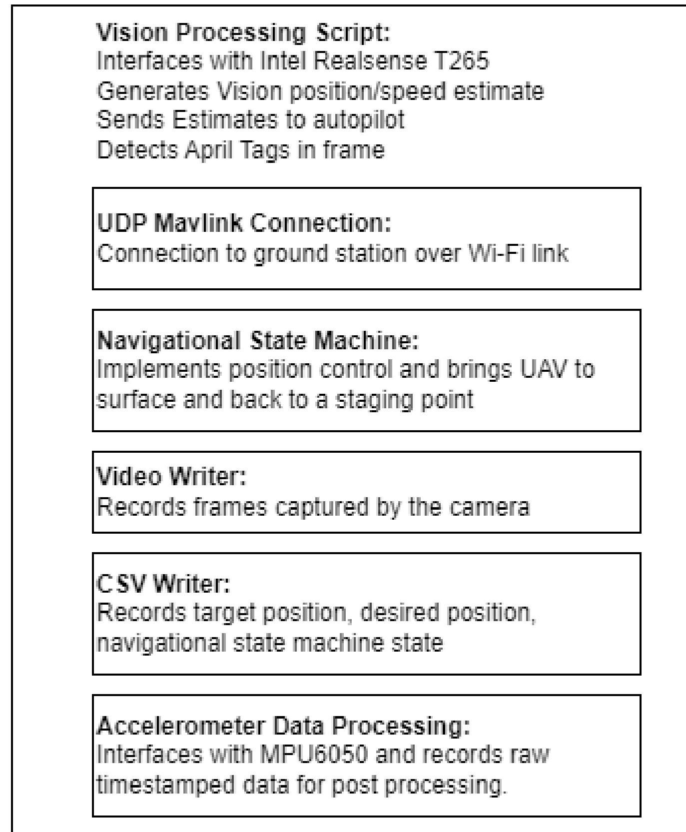


Figure 4.11: Python vision position estimate control program. Modules in boxes are primarily where work has been done to the work done in [61].

vision_position_estimate and vision_speed_estimate from the Intel Realsense camera and send it at 30 Hz to the autopilot. Nguyen also has done the work of obtaining a frame from the camera, applying camera calibration parameters and detecting an April Tag in the frame.

To make this work compatible with this thesis work a few elements needed to be implemented. Nguyen's original work connected directly to the autopilot, so a Mavlink connection through the Wi-Fi link needed to be implemented. This involved creating a Mavlink UDP object in the script and doing the appropriate message routing. The position controller and navigational state machine as shown in Figure 3.3c was implemented. Logging was also added to the script to log in flight video, target position, desired position and accelerometer data. Figure 4.11 shows an overview of the elements of the Python vision position estimate control program.

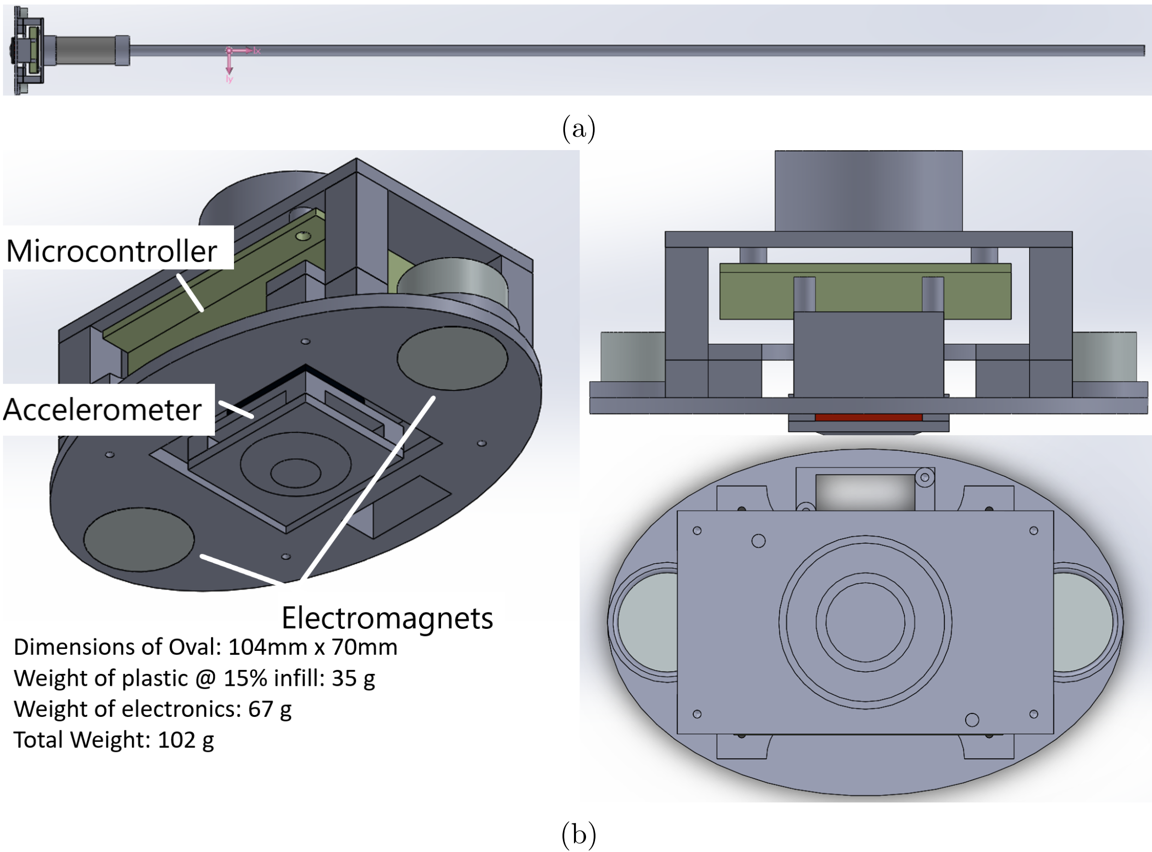


Figure 4.12: Arm V1 uses 2 electromagnets. This arm was used with the Aperture Hexacopter.

4.4 Sensor Arm with Vibration Dampening

Using a sensor affixed to a vibration arm was the method used for collecting data in this project. One of the main advantages of this method is that the arm is fixed, so there are no changes in the centre of mass during flight. This helps remove motion that could be imparted into the arm from the drone's turbulent motion.

4.4.1 Arm V1

An early version of the arm was built and tested with the Aperture Hexacopter. Figure 4.12 shows this arm. It housed an MPU 6050 accelerometer and a Teensy 3.2 on the sensor head. This was powered by the onboard computer and communicated with the onboard computer via serial. This arm was not fully tested with a vibration

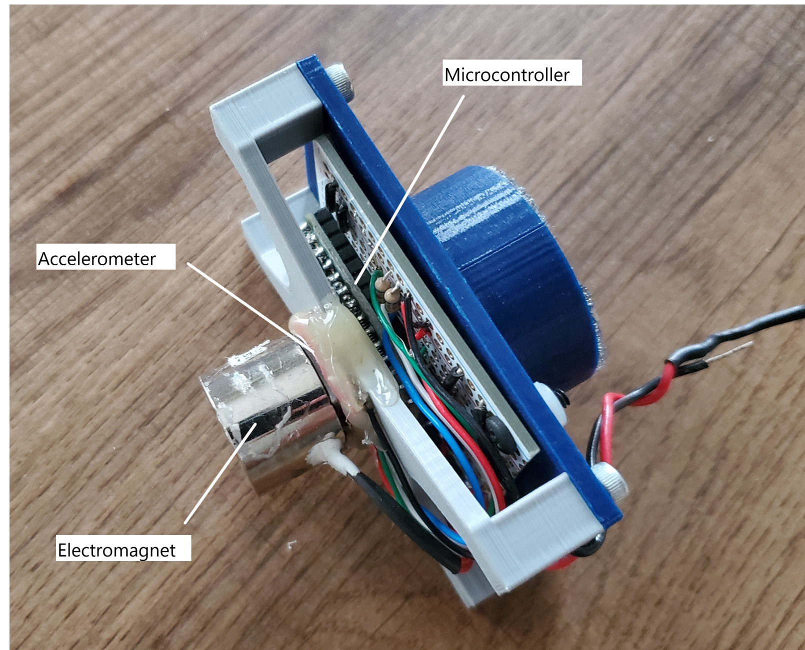


Figure 4.13: Arm V2 end effector used for test flights with the DJI Flamewheel F450.

target.

4.4.2 Arm V2

Arm V2 was used on the DJI Flamewheel F450. It is very similar in electrical hardware to Arm V1, but smaller so as to fit onto the smaller platform. This arm has a single electromagnet and the end effector of this arm is shown in Figure 4.13.

4.4.3 Arm V3

The vibration arm used for the final test flights is shown in Figure 4.14. This arm was more lightweight than the previous one and could be installed onto a smaller drone. To ensure reliable coupling with the infrastructure, the end of the arm has a rubber pad. An MPU6050 accelerometer is glued directly to the rubber pad. This pad helps to maintain contact with the infrastructure, even in situations where the surface is irregular or uneven. In addition, foam from a soft car-washing sponge is used to keep the sensor connected to the drone but decoupled from the vibrations of the drone. This helps to ensure that the data collected is accurate and not distorted

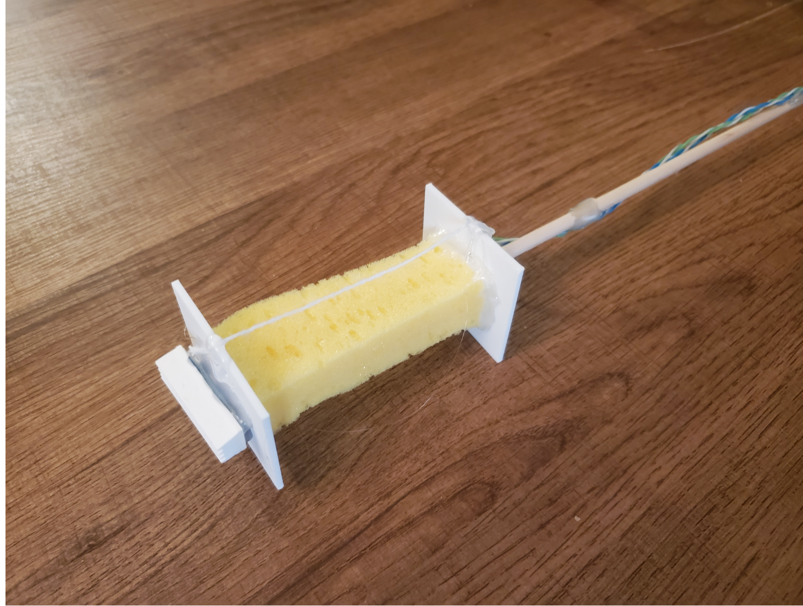


Figure 4.14: Arm V3 used for final test flights.

by the drone's motion.

To prevent the weight of the end of the arm from causing the sensor to droop away from the axis of the arm, a string is used to keep the end of the arm upright as shown in Figure 4.15. Once contact is made the strings are no longer in tension and the sponge is under compression. The MPU 6050 was connected via I2C directly to the drone's onboard computer.

This sensing arm requires that the drone be relatively steady and able to apply constant pressure to the wall through the arm.

4.4.4 Accelerometer Data Processing

The accelerometer data from the sensor affixed to the vibration arm was collected using the onboard computer on the drone. While the data was post-processed for this thesis, real-time processing would be easily achievable by the navigation/imaging computer. This would allow for immediate feedback and analysis of the data, enabling the drone operator to make real-time decisions about the inspection and maintenance of the infrastructure.

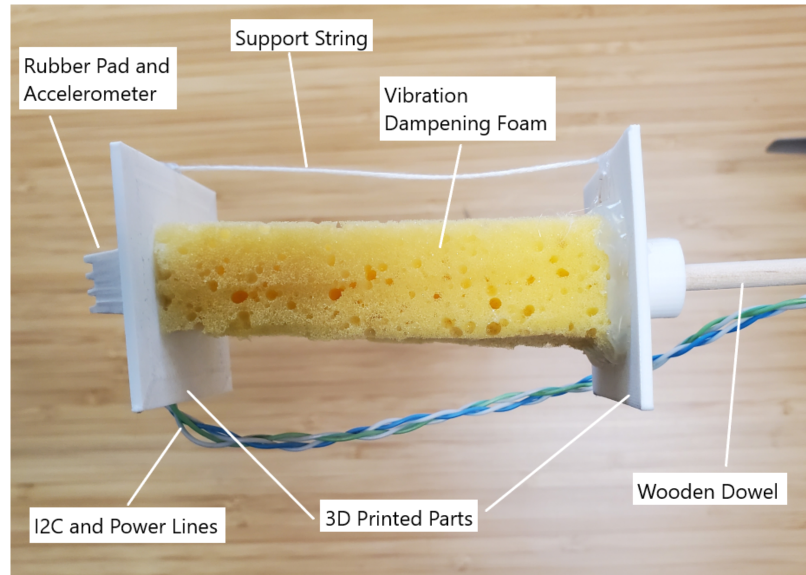


Figure 4.15: Close up of Arm V3. String was used to keep the sensor from drooping

The MPU6050 was accessed via I2C with the hardware's max bus rate of 400 kHz. Since the data ready line of the accelerometer was not connected and since the onboard computer was running other software, getting real-time periodic data was difficult. The setup had the MPU6050 being polled for the three-axis accelerometer data. Saving the accelerometer data would also cause irregularities in the frequency of acquired raw data. As a result, a method of collecting data was used where the sensor would collect 1 second worth of data as fast as possible. This data comes in timestamped and ranged between 650 and 700 samples. After collecting 1 second worth of data, the data is saved and then more data is collected at the next second. This means that a 20 s contact time would result in 10 s of vibration data.

The collected samples were then post-processed. The first step was interpolating the data to get 700 samples for 1 second at an even sampling rate. This was done using a linear interpolation. This data is now ready to be analyzed in the frequency domain. A Fast Fourier Transform (FFT) is performed on each axis. The FFT is a mathematical algorithm that converts a time-domain sampled signal into its frequency-domain representation. The data that has gone through the FFT allows us to identify the dominant frequencies and amplitudes of the vibrations. Sampling

the data at 700 samples per second allows for a frequency analysis of up to 350 Hz. The resolution of the data is 1 Hz as the data has been sampled for 1 second.

4.5 Test Configurations

Table 4.1 describes all of the test configurations attempted for this project. All of the configurations are discussed in depth in the following chapter. Configuration 1 is using vision position estimate from the DMM42BUC03-ML and the aperture hexacopter platform. Configuration 2 uses the Aperture hexacopter with an attitude controller. Configuration 3 uses the smaller DJI Flamewheel 450 platform with a newer lightweight arm with an electromagnet. Configurations 4 and 5 were large HW changes. They still used the Flamewheel 450, but now used the Intel Realsense T265 camera, the arm shown in Figure 4.14, and the Jetson Nano as the onboard computer. Configurations 4 and 5 differ only in the control scheme utilized. Configuration 5 is the only one that proved viable for the acquisition of vibration measurements. The performance of the vibration arm from configuration 5 is the only one detailed in the following chapter. The flight performance of configuration 5 is discussed in depth.

Table 4.1: Table showing different test configurations

	Config 1	Config 2	Config 3	Config 4	Config 5
Airframe	Aperture Hexacopter	Aperture Hexacopter	DJI Flame-wheel F450	DJI Flame-wheel F450	DJI Flame-wheel F450
Arm	Arm V1	Arm V1	Arm V2	Arm V3	Arm V3
Camera	Imaging Source DMM 42BUC03-ML	Imaging Source DMM 42BUC03-ML	Imaging Source DMM 42BUC03-ML	Intel T265	Intel T265
Computer	Odroid XU4	Odroid XU4	Odroid XU4	Jetson Nano 4 GB	Jetson Nano 4 GB
Autopilot	PX4	Ardupilot	Ardupilot	Ardupilot	Ardupilot
Control Scheme	Vision Position Estimate and Position Controller	Attitude Controller	Attitude Controller	Attitude Controller	Vision Position Estimate and Position Controller
Comms	900 MHz Telem, 2.4 GHz RC	900 MHz Telem, 2.4 GHz RC	900 MHz Telem, 2.4 GHz RC	2.4 GHz WiFi, 2.4 GHz RC	2.4 GHz WiFi, 2.4 GHz RC

Chapter 5

Results and Discussion

Many test flights were conducted with the configurations listed in Table 4.1. The only flights that proved successful in being able to satisfactorily control the UAV up to the wall and obtain a vibration measurement were with Configuration 5. Configuration 1 had issues where fusing image data alone based on the position of a fiducial marker was too noisy causing the state estimation on the autopilot to fail. Configurations 2-4 had issues with incorrect pose estimation, poor tuning gains, and a problem where the target would easily leave the frame causing the controller to fail. As a result vibration data was only acquired from Configuration 5 and that data is examined in this chapter. This chapter describes the hardware and control variations that were investigated throughout this work. This includes the aerial platform, autopilot system, onboard computing and imaging system, control via sending the autopilot a vision position estimate and control via sending attitude control messages.

5.1 Aerial Platform Performance

5.1.1 Aperture Hexacopter

The Aperture hexacopter was utilized in Configuration 1 and Configuration 2. This hexacopter served as the initial larger platform for conducting the experiments. However, it should be noted that the platform's utilization necessitated access to an outdoor flying field and/or an SFOC, making the experimentation process considerably

more expensive in terms of time. Debugging issues outside of the lab is more challenging, due to power, internet, ergonomic, and tool constraints. An image of the Aperture Hexacopter at the outdoor testing site is shown in Figure 5.1.

One key aspect of the Aperture hexacopter is its larger wheelbase, which in turn required the utilization/development of the larger vibration monitoring arm V1. The larger wheelbase meant that the arm needed to be longer, so it needed to be made of heavier/sturdier material. Additionally, the larger size of the drone introduced heightened risks if it went out of control during experimentation. This increased risk stems from several factors, such as the presence of more propellers and larger propellers on the platform. The larger propellers have higher inertia within each propeller when they are spun up to speed as compared to the propellers utilized on a smaller UAV. Moreover, the hexacopter's overall weight is greater, so there is more inertia in the frame in the event of loss of control. The larger inertia of the drone also meant that more force is required for it to be pushed around by the wind. This benefit is very quickly offset by the larger size of the UAV creating more surface area that the wind can affect and the reduced power-to-weight ratio of the drone.

Despite its adequacy in terms of performance, the Aperture hexacopter was deemed unsuitable for proof of concept experimentation for this particular research work. This limitation is primarily attributed to the requirement for a large indoor flight area, an outdoor flying field or SFOC, which considerably restricted its practicality and accessibility for conducting experiments.

5.1.2 DJI FlameWheel F450

In the context of the research application, the DJI Flamewheel F450 proved to be an exceptionally suitable platform and was consequently employed in configurations 3 to 5. One of the most significant advantages of this platform was its small size, which allowed for indoor experimentation within a controlled indoor laboratory environment. To ensure safety during indoor tests, the drone was operated within an

Figure 5.1: Aperture hexacopter at outdoor test site

area, enclosed by a net, mitigating the risk of any untoward incidents. The ease of customizability of the F450, with the custom 3D printed top plate, allowed for two different imaging and computing systems to be easily implemented.

The small size of the DJI Flamewheel F450 had a direct impact on the design of the platform, particularly concerning the vibration-sensing arm. The platform utilized smaller vibration-sensing arms to accommodate its compact form factor. Notably, the V3 arm, being the smallest of these arms, weighed 200 grams.

Vibration data analyzed later in this chapter were acquired from the DJI Flamewheel F450 platform.

5.1.3 Communications links

Initially, the communication setup for the UAV involved a 900 MHz telemetry link and a 2.4 GHz RC link. This combination of communication links was selected for configurations 1 to 3 due to its effectiveness in low bandwidth and long-range applications. This was done as it is a standard setup.

For configurations 4 and 5, a modification was introduced by replacing the 900 MHz telemetry link with a 2.4 GHz WiFi link. This change was achieved by integrating a low-profile USB WiFi module with the UAV's onboard computer. The decision to switch to a WiFi link was primarily driven by the absence of high-range communication requirements in the proof of concept phase.

One significant advantage of implementing the 2.4 GHz WiFi link was its ability to simplify the ground station setup. With the introduction of the WiFi link, all that was needed for the ground station was a computer with WiFi capabilities. This streamlined setup contributed to increased convenience and portability during experiments and reduced the need for specialized equipment.

The WiFi link not only served as a means of transmitting telemetry data but also offered additional functionalities. The WiFi link offered direct ssh access to the onboard computer. This provided greater flexibility during experimentation, allowing adjustments to be made in real-time without the need for physical connections or reprogramming the onboard computer directly. Additionally, log files could be downloaded wirelessly.

The transition from the initial 900 MHz telemetry link to the 2.4 GHz WiFi link for configurations 4 and 5 was a strategic decision driven by the specific requirements of the proof of concept phase. By leveraging the advantages of the WiFi link, the proof of concept added convenient access to the onboard computer, streamlined ground station setup, allowed quick control program modifications, and wireless data transfer.

5.2 Autopilot Software

5.2.1 PX4

The PX4 flight control system was initially used in Configuration 1. However, it was not utilized in subsequent configurations for the remainder of the project. The decision to discontinue the use of PX4 was primarily influenced by the unsatisfactory

out-of-the-box performance observed during the initial experiments.

Despite attempts to improve the performance of PX4 by tuning the autopilot control loops and exponentials of the RC transmitter, challenges were encountered in maintaining a stable altitude using the RC transmitter in manual mode. This difficulty in altitude control was mainly due to a result of inadequate tuning and configuration, but the platform could have had more intuitive configuration steps.

Moreover, during one outdoor test flight, there was a notable instance of very poor manual UAV control with the PX4 system. While it remains uncertain whether this poor control was solely due to the system's poor configuration or other factors, it raised doubts about the platform's suitability for the project's objectives.

In light of the difficulties faced with PX4, the switch was made to explore Ardupilot and see if these issues would immediately be resolved and to reduce time in trying to debug the configuration.

5.2.2 Ardupilot

The transition from PX4 to Ardupilot proved to be a positive change for the research project. Ardupilot offered an excellent out-of-the-box experience and the hardware compatibility made it seamless to switch between the two different flight control systems. From the operator's perspective, Ardupilot resolved the manual control issues that were experienced when using PX4. Ardupilot, with little to no configuration tuning, allowed for more precise flight and altitude control.

Ardupilot was used in Configurations 3 to 5. The flight control system demonstrated was used in the vibration data that was acquired and that is presented later in this chapter.

The documentation provided for Ardupilot, was clear and comprehensive with lots of community-generated forum information if needed. The simulator provided by the Ardupilot team was another valuable feature that contributed to the success of the project. Setting up and using the simulator was straightforward and user-friendly.

This capability not only saved time and resources but also provided a safe environment for testing and refining control algorithms and flight strategies before deploying them in real-world experiments.

5.3 Onboard Computer and Imaging System

In configurations 1 to 3, the inexpensive imaging source camera was utilized. This camera was found to be highly functional in detecting markers within images, making it a suitable choice for the initial stages of the experimentation. However, after experiments with Configuration 3 had concluded, the camera was no longer accessible. Rather than continuing to utilize the same imaging and computing system, this was an opportunity to implement a different system with improved performance and features. To address the limitations observed in using the attitude controller in Configuration 3 and to improve the overall imaging system's processing time and latency, the Intel Realsense camera and Jetson Nano were implemented for configurations 4 and 5.

The introduction of the Intel Realsense camera and Jetson Nano brought about some improvements in the system's performance. The new camera system reduced image processing time and latency, thereby enhancing the UAV's responsiveness and overall control during flight. These enhancements were expected to contribute to better experimental outcomes. By utilizing stereoscopic images that the Intel Realsense camera provided there was an opportunity for experimentation with trying to improve marker pose ambiguity. Configuration 4 proved to be ineffective in controlling the UAV, so the position data from the Intel Realsense camera was utilized in Configuration 5.

5.4 Control Performance

5.4.1 Vision Position Estimate Performance with DMM 42BUC03-ML

In Configuration 1, the imaging source camera was utilized to generate a vision position estimate, which aimed to provide the UAV with an accurate estimation of its position. However, this control system encountered significant challenges and ultimately failed to meet expectations.

The fundamental issue with the vision position estimate in Configuration 1 was its heavy reliance on having a marker present within the camera field of view. The control system was designed to utilize the marker's information to calculate the UAV's position relative to the marker. Consequently, any misidentification or loss of the marker in the camera frame would lead to a breakdown in the position estimation process. Whenever the marker was misidentified or no longer visible in the camera frame, the position estimate would fail to be generated. Consequently, the autopilot, which received these vision position estimate messages, would completely reject them as they were deemed invalid. When the correct messages were again generated, the resulting position jump would necessitate a need to reset the PX4 autopilot state estimator.

While additional filtering techniques could have been applied to improve the robustness of the marker detection process, the fundamental approach of relying solely on the marker in the frame proved to be unsuitable for providing a consistent and accurate position estimation.

Configuration 1's vision position estimate based on the imaging source camera failed due to its heavy reliance on viewing a single marker in the frame. The lack of robustness and the susceptibility to marker misidentification led to inconsistent position estimates and, ultimately, unsatisfactory performance.

5.4.2 Attitude Controller Performance

In configurations 2 to 4, the attitude controller was utilized as the primary control system for the UAV. However, real-world testing using this method yielded poor results for several reasons.

One significant issue was the limited image acquisition rate of the Imaging source camera, which operated at less than 10 Hz. This slow image acquisition rate and the image processing latency introduced lagging inputs to the control system. As a result, the UAV's response to control inputs became sluggish, and it struggled to track the target accurately in real time.

The slow image acquisition rate, combined with the latency, caused challenges in maintaining precise control over the UAV's position. If the controller gains were set too high, the drone frequently overshot the target position due to the delayed response. Conversely, if the gains were too low, the UAV would drift away from the target, failing to keep the target within the camera's field of view.

Additionally, the issues with the attitude controller were further exacerbated by other factors. First, the system was sensitive to misidentifications of the target, especially when there was pose ambiguity. This meant that if the camera incorrectly identified the target or experienced difficulties in discerning its pose accurately, it would lead to incorrect attitude setpoints. Subsequently, the UAV would attempt to follow these incorrect setpoints, causing erratic movements and deviations from the target position. Furthermore, the control system was susceptible to noise, which could be introduced by various factors, such as environmental conditions or camera artifacts. The presence of noise in the sensor data further contributed to inaccurate attitude setpoints, leading to erratic behaviour and challenging control performance.

All of these things, slow image acquisition, misidentification of the target, and noise in the target position data, poor control gains without wind estimation, created a problematic situation where the target would quickly move out of the camera's field

of view. This resulted in a constant struggle to maintain tracking and precise control over the UAV during real-world testing.

While more tuning efforts could have potentially improved the performance of the attitude controller to some extent, the fundamental limitations of using only the fiducial marker as a reference for position estimation as the basis for the control strategy would remain.

5.4.3 Vision Position Estimate with Intel Realsense T265 Camera

The introduction of the Intel Realsense T265 camera position for vision position estimation marked a significant improvement in the UAV's control system. The vision position estimate provided by the Intel Realsense camera proved to be highly reliable and accurate, enabling the drone to perform loitering and position hold maneuvers even in challenging environments, such as when flying close to a wall with turbulent air.

Once the drone was in a position hold state, the generation and transmission of commands for relative movements became straightforward. With the confidence of knowing its precise position, and having a fiducial marker in the camera's field of view, the controller could easily generate relative desired position commands to its current location and enable the UAV to navigate and perform the desired movements.

In Configuration 5 the drone must be 0.43 m away from the target for the arm to make contact with the wall. Any farther away and the vibration sensor will not make good contact with the wall. The drone is also programmed to make contact 0.10 m below the target so that the arm does not occlude the fiducial marker.

The following data shows the drone's position and control setpoint for the duration of the flight. The drone is making contact when the current forward position is 0.43 m. The setpoint line is only graphed when the drone is being commanded to a position. When the setpoint line is not there, the drone is being flown by an operator, aided

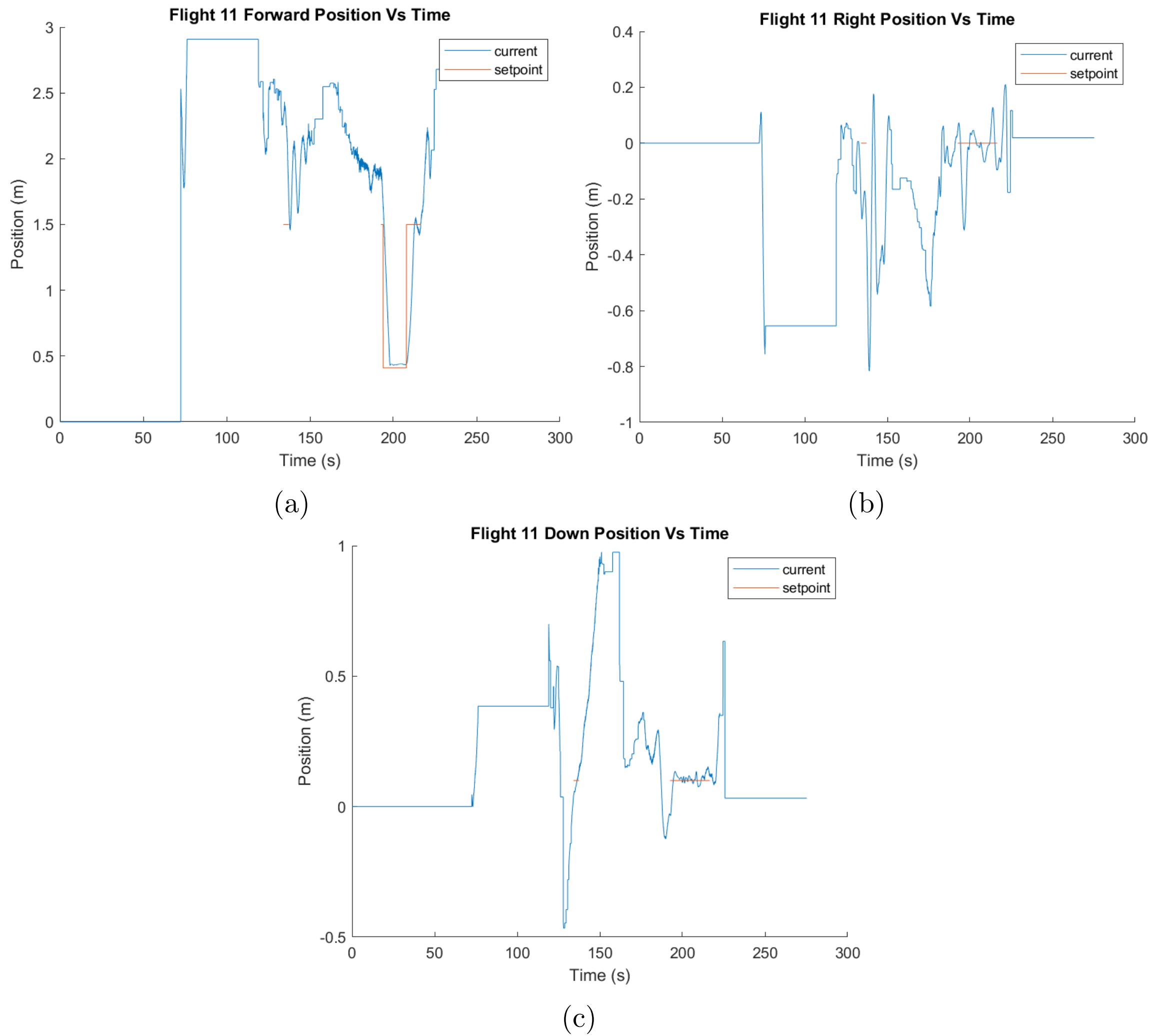


Figure 5.2: Flight 11 Position Control

by the position estimate being sent by the onboard computer. This means that when the operator lets go of the sticks the drone will remain in its current position.

Figure 5.2 shows flight 11. During this flight, the position controller was enabled once. The drone made contact with the wall at ∼200 s into the flight. When the controller is enabled the drone first tries to move from ∼1.8 m away from the wall to 1.5 m. As soon as it meets the staging condition (see Figure 3.3c), it then goes to make contact and holds the position at the wall for ∼10 s. The forward setpoint for the drone is 0.41 m. After making contact for ∼10 s it then goes back to the staging area until the operator takes control again at ∼140s.

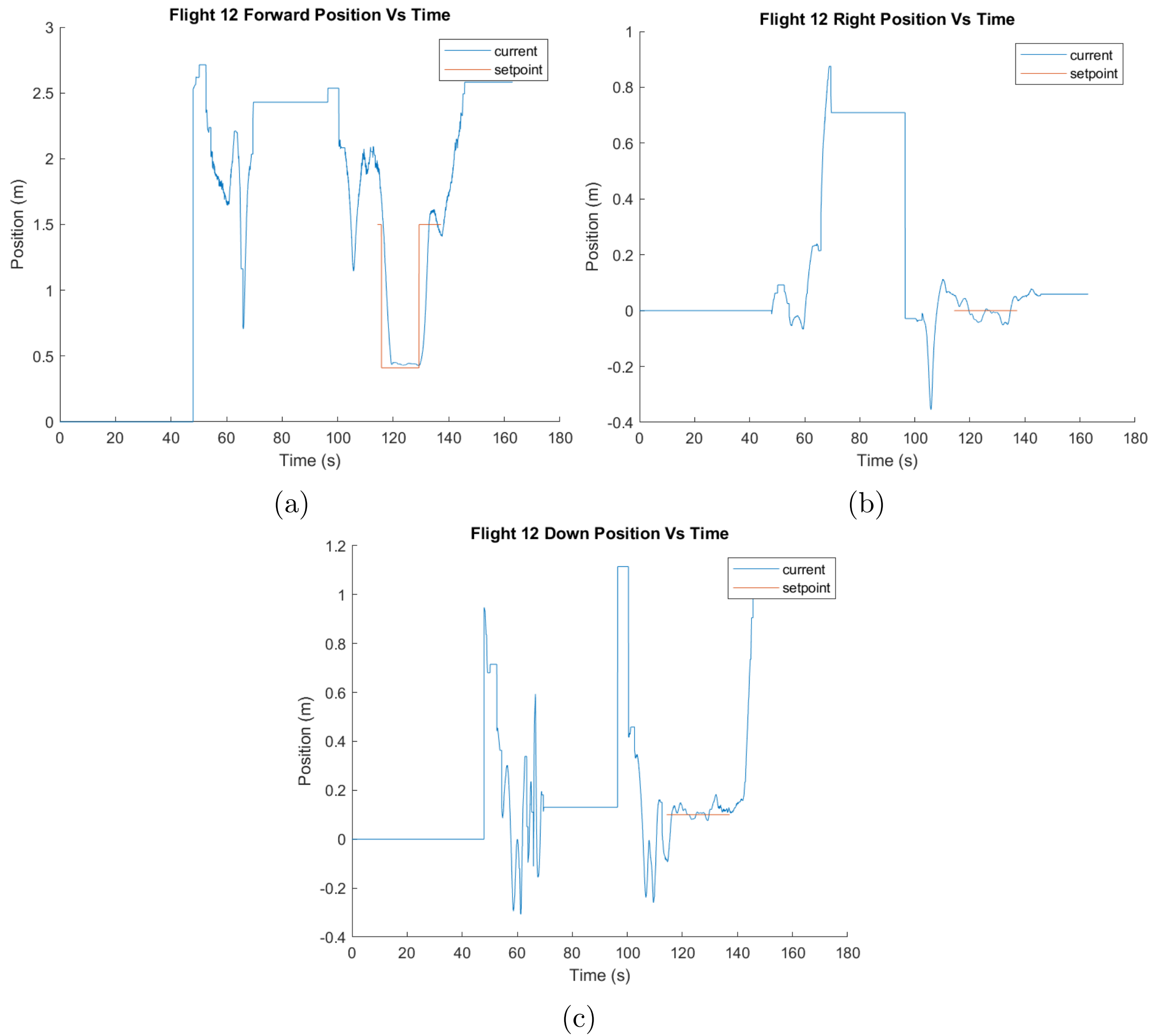


Figure 5.3: Flight 12 Position Control

Flight 12 as shown in Figure 5.3 its results are similar to flight 11. The position controller was enabled once and shows a flight where the position controller was enabled once at ∼115 s. The drone made contact with the wall at ∼120 s into the flight, by being commanded to go to 0.41 m. When comparing Figure 5.2a and Figure 5.3a, the oscillations can be seen at the point of contact. Flight 12 seemed to have more oscillations when making contact. To reduce these oscillations, one strategy is to not apply as much pressure to the arm.

Flight 13 as shown in Figure 5.4 shows a flight where the forward position setpoint was set to 0.43 m when contact is made. In Figure 5.4a, there are some oscillations

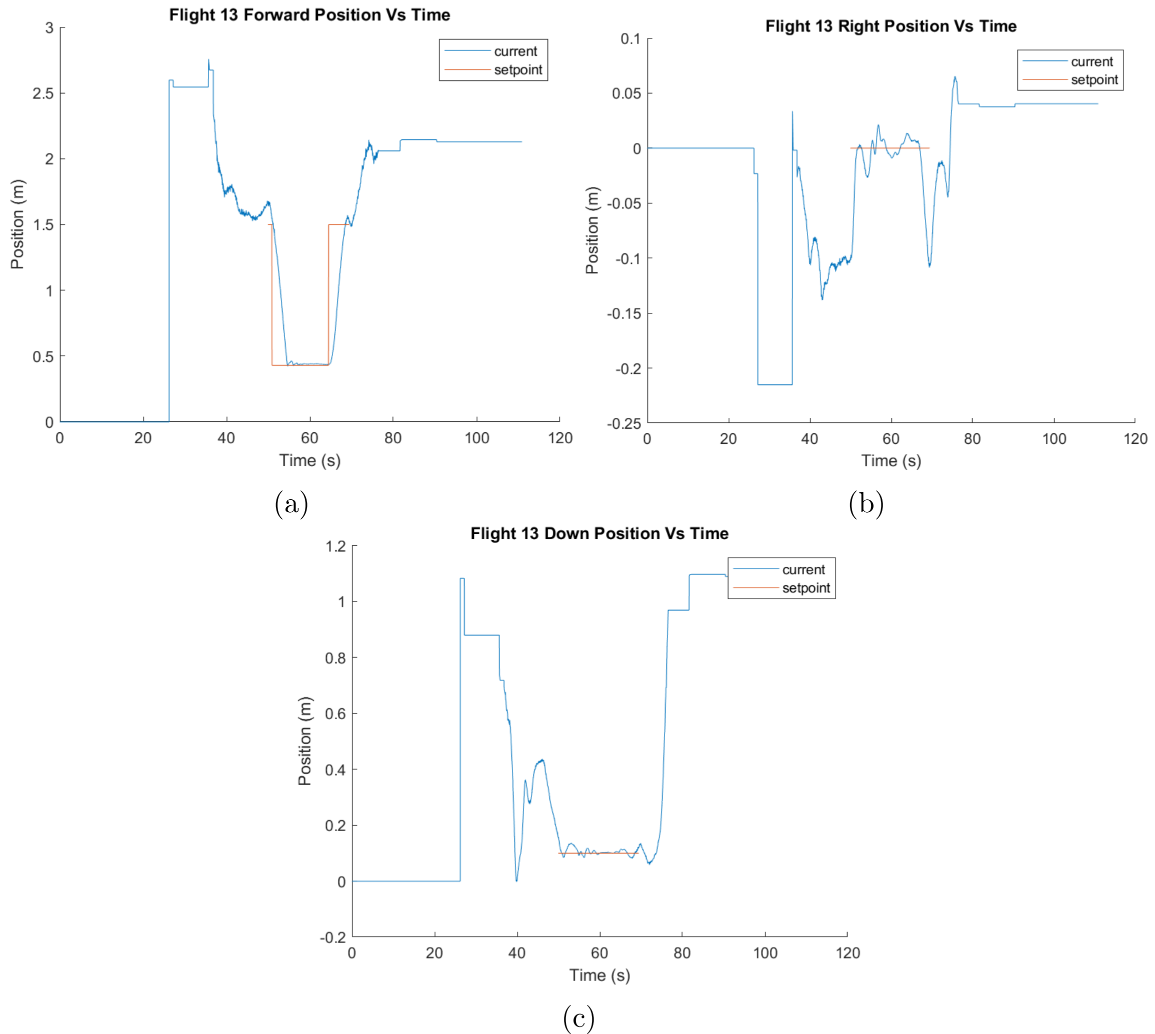


Figure 5.4: Flight 13 Position Control

when contact is first made but they die down and are smooth shortly after.

Flight 14 as shown in Figure 5.5 again had the forward setpoint set to 0.43 m for when contact is made. On this flight, contact was made 3 times. Figure 5.6 shows the variation in arm position relative to the fiducial marker. This variation is primarily caused by the lack of yaw control being implemented in Configuration 5. The images also tend to show a more exaggerated effect on the drone position due to the wide-angle lens. The first contact made was the poorest in the series with the end effector of the arm off by ~14 cm. Having the arm at such an angle also means that the drone is not pushing as axially through the arm as it could be. This can be

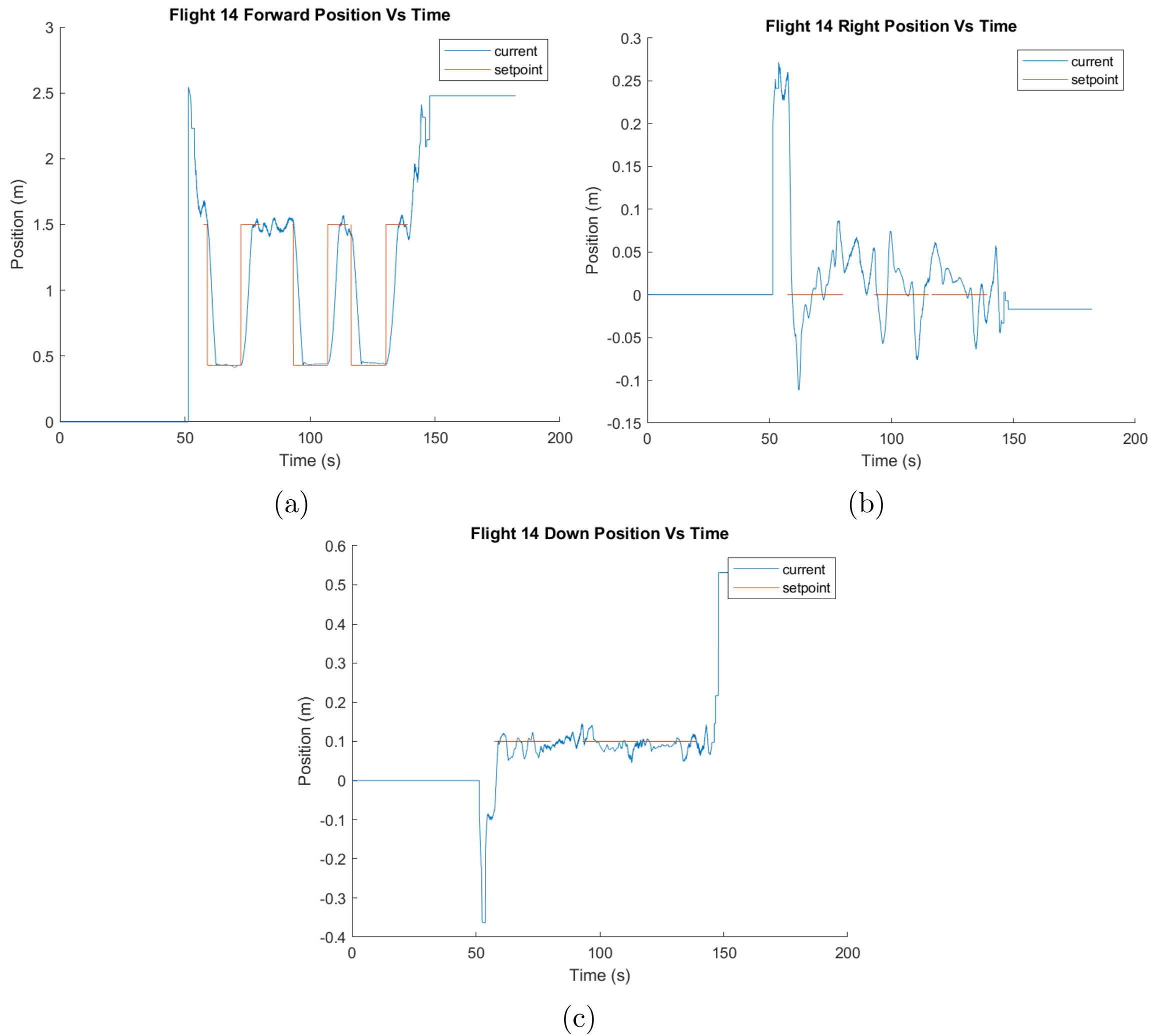
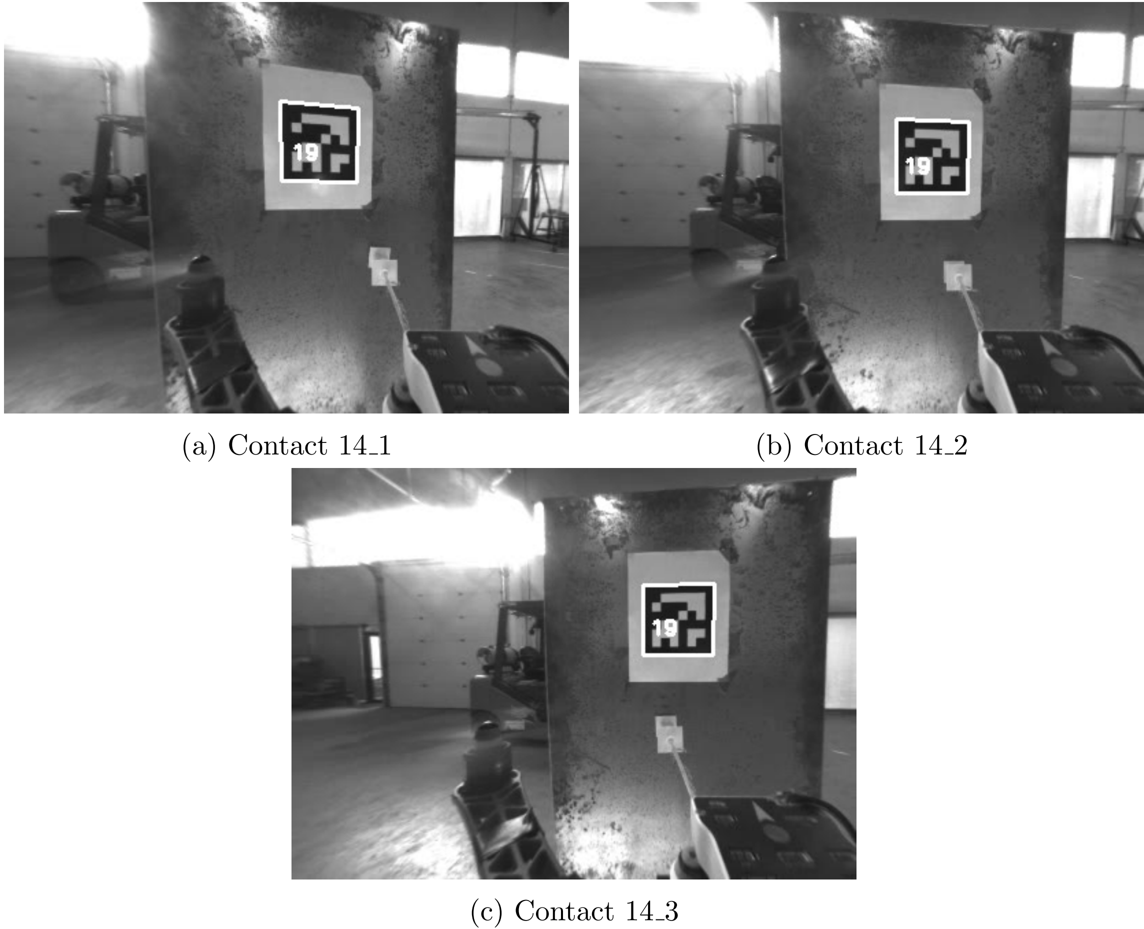


Figure 5.5: Flight 14 Position Control

seen when looking at the frequency graph in Figure 5.11, where a very poor vibration measurement is captured.

It is difficult to measure the location of the drone relative to the target in flight as no other position verification mechanism was used. By taking images of each contact, pixel measurements can be done to determine the error in the end effector position. Table 5.1 shows the position errors for each contact, with the average error from all 6 contacts being ∼74 mm.



(a) Contact 14_1
(b) Contact 14_2
(c) Contact 14_3

Figure 5.6: Flight 14 Contact

Table 5.1: Estimate of position error of end effector during contact taken from pixel measurements

	X	Y	Total
Contact 11	81 mm	-12 mm	82 mm
Contact 12	-12 mm	-45 mm	47 mm
Contact 13	85 mm	-30 mm	90 mm
Contact 14_1	145 mm	24 mm	147 mm
Contact 14_2	54 mm	-12 mm	56 mm
Contact 14_3	-24 mm	3 mm	24 mm
Average	55 mm	-12 mm	74 mm
Standard Deviation	58 mm	22 mm	39 mm

5.5 Vibration Measurement

The same stand that held the fiducial marker also was a vibration target. Vibrations were generated using a brushless motor with a 3D-printed unbalanced rotor. An Arduino sent PWM signals to a speed controller to vary the speed of the brushless motor and allow for different frequencies to be generated. Figure 5.7 shows the vibration source.

Figure 5.7: Vibration Source

5.5.1 Arm V1

During the research project, Arm V1 was developed for the Aperture Hexacopter and used in configurations 1 and 2. The arm was not used to collect any vibration data as no working position estimation system was developed with the Aperture Hexacopter.

The vibration characteristics and performance of Arm V1 are not fully unexplored in the context of the research project.

5.5.2 Arm V2

In Configuration 3, Arm V2 was employed as a direct successor to Arm V1 in the research project. Arm V2 was specifically designed to be smaller and more lightweight, making it suitable for installation onto the DJI Flamewheel F450.

However, the use of Arm V2 with the electromagnet had some drawbacks on the smaller platform. The main notable issues were a reduction in the UAV's overall runtime and maneuverability. The weight of Arm V2, along with the energy consumption of the electromagnet, contributed to the decreased flight time of the vehicle. Longer flight durations are desirable for extended experimentation and data collection with the UAV and also to reduce the amount of power cycling the drone when a new battery is needed to be installed.

Moreover, during experimentation, it was discovered that the electromagnet's functionality was ultimately unnecessary for creating contact with surfaces. Despite having the electromagnet, the UAV still needed to approach the wall close to perpendicular to ensure proper attachment. This meant that the UAV's position control had to be effective enough to establish perpendicular contact with the wall. The electromagnet would provide strong coupling with the wall, but the UAV position control already needed to be adequate to provide a satisfactory level of coupling with the wall.

5.5.3 Arm V3

Arm V3 was lightweight and simpler in design. It is close to the smallest, lightest and simplest that a vibration arm for the Flamewheel F450 could be. Figure 5.8 shows a flight to a vibrating target. The target is vibrating at a lower frequency than the drone motors. The point of contact is made at $\sim$200 s. In Figure 5.9, the vibrating target is vibrating at a frequency very similar to the drone motors. At between $\sim$120 s and $\sim$130 s the drone makes contact with the target. When the drone makes contact with the target it can be seen that the vibration signature from the motors goes away

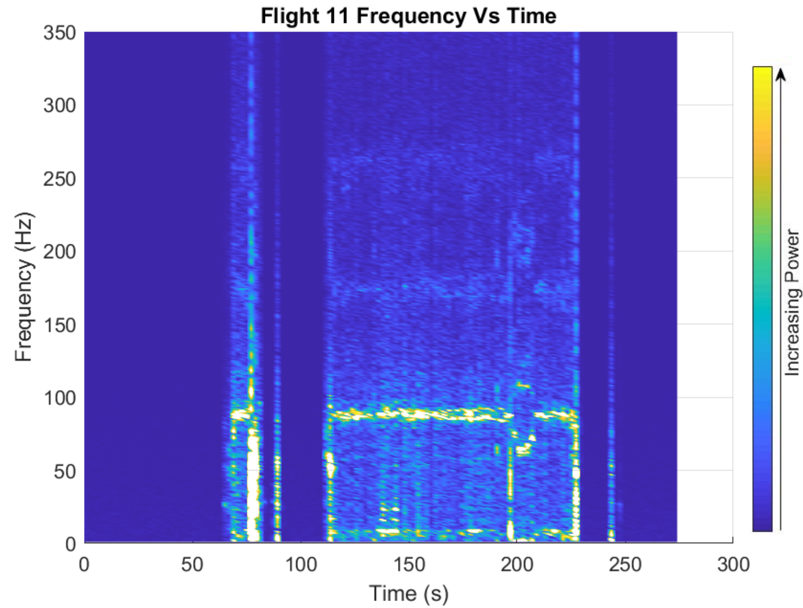


Figure 5.8: Flight 11 frequency measurement

and the vibration from the target is measured. The broadband frequency elements in these frequency graphs are primarily from the motors spooling up and down. During both of these flights, the drone made poor contact with the target, nevertheless, the frequency graphs can still be used to identify when the drone made contact with the target. These two graphs may be compared to Figure 5.2 and Figure 5.3, to correlate the change in frequency with the moment contact was made.

Flight 13 and Flight 14 took place with less force being applied on the vibration arm. This was done by setting the forward position of the drone to 0.43 m instead of 0.41 m. These flights can be seen in Figure 5.10 and Figure 5.11. In both of these flights, the frequency was set higher than the drone motors. Flight 13 seems to have made adequate contact, but it could have been stronger as there are still some elements from the drone motors while the contact was made. Flight 14 made contact with the target 3 different times in one flight. The first contact was poor, but the higher frequency of the vibrating target can be seen. The first 2 contacts were made relatively well, with the 2nd contact having no frequency elements from the motors preset. The 3rd contact was mostly clean, but some elements of the drone's motors

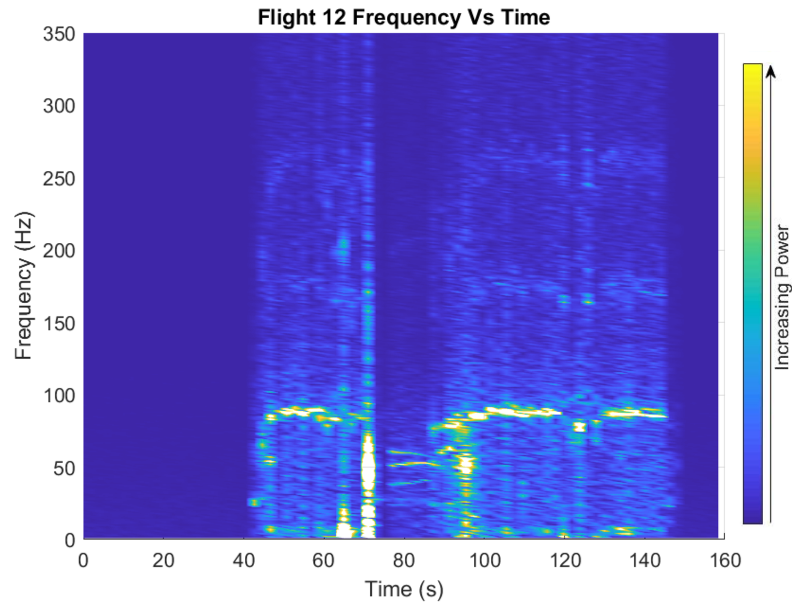


Figure 5.9: Flight 12 frequency measurement

are visible between 120-130 s.

The vibration data acquired during the research project demonstrated relatively good performance of the vibration monitoring arm. The arm's capabilities allowed for the detection of four different frequencies. Notably, two of these frequencies were higher than the frequency of the drone motors, while the other two frequencies were lower than the drone motor frequency.

The ability to distinguish between different frequencies is crucial for accurately assessing the UAV's vibration characteristics and identifying potential sources of vibration. The vibration monitoring arm's capacity to capture multiple frequencies provided valuable insights into the dynamic behaviour of the drone during flight.

Additionally, when the vibration monitoring arm achieved good contact with the UAV, it exhibited a damping effect on the vibrations generated by the drone motor. This damping effect is highly beneficial as it helps reduce the impact of vibrations on the UAV's stability and performance. By mitigating vibration levels, the monitoring arm contributes to a smoother and more controlled flight experience.

Improvement efforts could focus on increasing the arm's sensitivity to capture

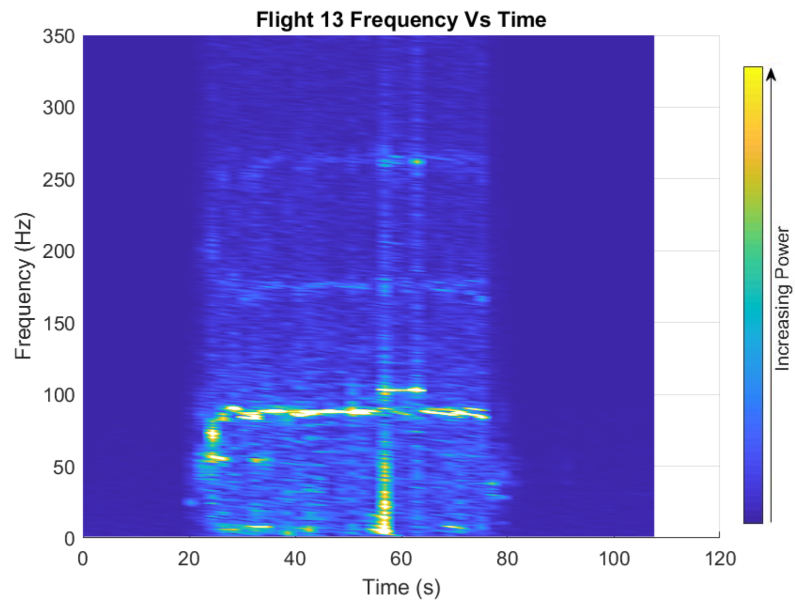


Figure 5.10: Flight 13 frequency measurement

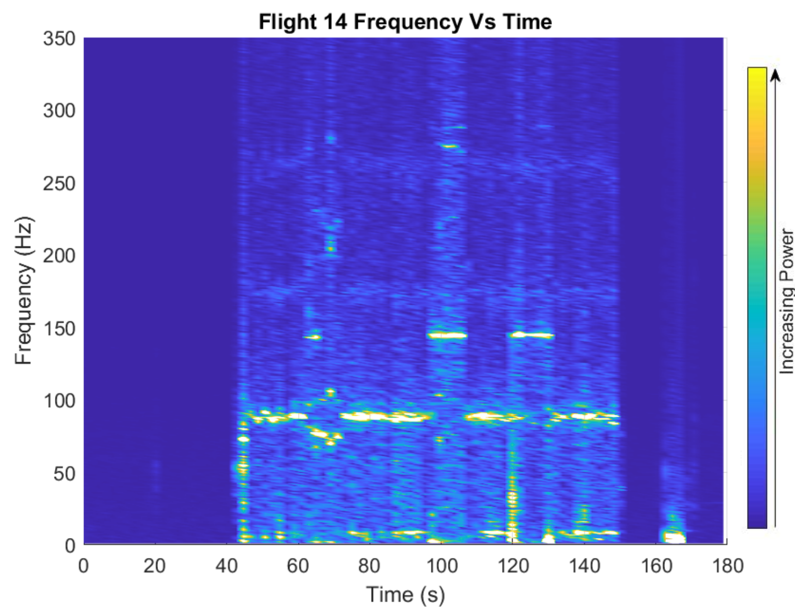


Figure 5.11: Flight 14 frequency measurement

a broader range of frequencies, especially those that may be indicative of specific UAV subsystems or mechanical components. Additionally, refining the arm's contact mechanism with the drone could lead to more consistent and reliable data acquisition, which in turn would improve the accuracy of vibration analysis.

Moreover, exploring ways to optimize the data processing and analysis algorithms could lead to more detailed and comprehensive insights into the UAV's vibration patterns. By fine-tuning the data analysis procedures, researchers can extract valuable information related to the drone's performance, health, and potential areas for optimization.

In conclusion, the vibration monitoring arm showcased relatively good performance in the research project, enabling the detection of four different frequencies, two of which were higher and two lower than the drone motor frequency. The arm's ability to dampen vibrations when in good contact with the drone was a significant advantage. However, further work and enhancements are recommended to fully leverage the arm's potential and achieve even more insightful and comprehensive vibration analysis results.

Chapter 6

Conclusions and Recommendations for Future Work

6.1 Conclusions

In conclusion, this thesis has explored the utilization of drones for vibration monitoring, focusing on the development of a small, lightweight aerial platform capable of integrating an imaging system, control system, and vibration acquisition system. By employing fiducial markers on a vibration target, the drone was able to approach the target and successfully collect data from the vibration target. The vibration target was set to different frequencies and these were able to be detected and distinguished from vibrations generated by the drone.

Throughout the research, various solutions were investigated, leading to the realization that the use/installation of fiducial markers may make the explored solution less attractive than just mounting permanently affixed sensors instead. If the markers need to be installed then permanently affixed sensors should be installed instead. This alternative approach offers a practical solution for long-term monitoring requirements, with good vibration coupling to the infrastructure, and eliminates the need for a drone and repeated flights. That being said more computer vision may be used to eliminate the need for a fiducial marker depending on the specific environment.

If it is infeasible for a human to affix a fiducial marker or a permanently affixed sensor box, then it is recommended to employ a drone utilizing a vision-based position

estimator to permanently affix a sensor box. This strategy eliminates the need for a fiducial target and repeated drone flights each time a measurement is required. The use of a vision base position estimator allows a human to easily control a drone in GPS-denied environments, in turbulent air next to infrastructure. In addition, an ultrasonic range finder may also allow for additional automation for a drone attaching a sensor to a wall.

This work was a proof of concept system. There are limitations to the study. The vibration measurements were not fully characterized to determine their accuracy and real-world usability. The UAV was only flown close to the pilot at close to ground level. The test flights in the final configuration were all indoor.

In summary, this thesis has contributed to the field of vibration monitoring by demonstrating the feasibility and effectiveness of using drones for vibration data acquisition. The recommended path for commercialization involves further improving the system and using other methods, such as permanently affixed sensor boxes when feasible.

6.2 Future Work

This future work section is focused on how to make the explored solution, a drone with a vibration arm, ready for real-world applications.

Firstly, the implementation of yaw control, for the drone was not realized in the final test phase due to time constraints. However, incorporating yaw control would enable more precise position control of the drone, allowing for better test repeatability and contact with the infrastructure.

Furthermore, additional work needs to be conducted to optimize and test the drone for outdoor real-world applications. Up until now, the test flights that have made contact with the vibration target, have only taken place indoors, and thus, adjustments and tuning specific to outdoor conditions are required. Doing this ensures that the drone operates reliably and accurately in real-world scenarios, where environmental

factors, such as wind and lighting, and other challenges may differ from controlled indoor environments. Operating the UAV from a distance may also be challenging especially when operating close to infrastructure. A real-time feed from the camera at the ground station, range finders and increased marker detection range will an operator to more easily control the UAV at range. This will involve range finding hardware, a higher bandwidth and range data link, increased resolution cameras or multiple cameras.

Regarding the vibration sensor, a couple of issues were encountered. One of the problems was the presence of noise in the sampling period, while the other issue was related to sensitivity. To address the noise problem, a straightforward solution is to utilize a dedicated MCU for recording data from the accelerometer to ensure periodic data collection. This approach would help mitigate noise when performing frequency analysis on the data. In addition the sampling period of the accelerometer only allows measurements up to ~350 Hz, so a higher rate accelerometer would improve the data collected and allow for measurements of higher frequencies.

The sensitivity of the vibration sensor requires further investigation, as the thesis work did not delve into determining the necessary sensitivity levels for real-world applications. Future research should focus on determining the optimal sensitivity requirements for various real-world scenarios. The current setup may adequately fulfill the needs of specific real-world applications, but further work should be done regardless.

A fully developed system has the potential for increased automation. Expanding the system's automation capabilities would contribute to its usability and practicality in monitoring infrastructure and machinery. GPS may be incorporated into the system and automated path planning may be used for bringing the drone close to each target.

This work has provided a base for using a lightweight UAV to obtain a vibration sample from infrastructure. Further work on this base will improve and develop a

low cost reliable vibration acquisition mechanism that may be used for future commercialization.

Bibliography

[1] G. C. Rouse and J. G. Bouwkamp, “Vibration studies of monticello dam,” *A Water Resources Technical Publication*, 1967.

[2] S. W. Doebling, C. R. Farrar, and M. B. Prime, “A summary review of vibration-based damage identification methods,” *The Shock and Vibration Digest*, vol. 30, no. 2, pp. 91–105, Mar. 1998. DOI: 10.1177/058310249803000201.

[3] E. P. Carden and P. Fanning, “Vibration based condition monitoring: A review,” *Structural Health Monitoring*, vol. 3, no. 4, pp. 355–377, Dec. 2004. DOI: 10.1177/1475921704047500.

[4] O. Salawu, “Detection of structural damage through changes in frequency: A review,” *Engineering Structures*, vol. 19, no. 9, pp. 718–723, Sep. 1997. DOI: 10.1016/s0141-0296(96)00149-6.

[5] J. Caicedo, J. Marulanda, P. Thomson, and S. Dyke, “Monitoring of bridges to detect changes in structural health,” in *Proceedings of the 2001 American Control Conference. (Cat. No.01CH37148)*, IEEE, 2001. DOI: 10.1109/acc.2001.945586.

[6] S. S. Saidin, A. Jamadin, S. A. Kudus, N. M. Amin, and M. A. Anuar, “An overview: The application of vibration-based techniques in bridge structural health monitoring,” *International Journal of Concrete Structures and Materials*, vol. 16, no. 1, Dec. 2022. DOI: 10.1186/s40069-022-00557-1.

[7] S. A. Rikardo, C. B. Bambang, C. Sumaryadi, T. D. Yulian, S. E. Arief, and S. F. I. Kharil, “Vibration monitoring on power transformer,” in *2008 International Conference on Condition Monitoring and Diagnosis*, IEEE, 2008. DOI: 10.1109/cmd.2008.4580454.

[8] M. N. H. Chikuruwo, L. Maregedze, and T. Garikayi, “Design of an automated vibration monitoring system for condition based maintenance of a lathe machine (case study),” in *2016 International Conference on System Reliability and Science (ICSRS)*, IEEE, Nov. 2016. DOI: 10.1109/icsrs.2016.7815838.

[9] K.-Y. Wong, “Instrumentation and health monitoring of cable-supported bridges,” *Structural Control and Health Monitoring*, vol. 11, no. 2, pp. 91–124, Apr. 2004. DOI: 10.1002/stc.33.

[10] P. C. Chang, A. Flatau, and S. C. Liu, “Review paper: Health monitoring of civil infrastructure,” *Structural Health Monitoring*, vol. 2, no. 3, pp. 257–267, Sep. 2003. DOI: 10.1177/1475921703036169.

[11] B. Y. Jeong, "Occupational deaths and injuries in the construction industry," *Applied Ergonomics*, vol. 29, no. 5, pp. 355–360, Oct. 1998. DOI: 10.1016/s0003-6870(97)00077-x.

[12] O. Rozenfeld, R. Sacks, Y. Rosenfeld, and H. Baum, "Construction job safety analysis," *Safety Science*, vol. 48, no. 4, pp. 491–498, Apr. 2010. DOI: 10.1016/j.ssci.2009.12.017.

[13] S. S. Oliveira, W. de Albuquerque Soares, and B. M. Vasconcelos, "Fatal fall-from-height accidents: Statistical treatment using the human factors analysis and classification system – HFACS," *Journal of Safety Research*, May 2023. DOI: 10.1016/j.jsr.2023.05.004.

[14] Z. Ameli, Y. Aremanda, W. A. Friess, and E. N. Landis, "Impact of UAV hardware options on bridge inspection mission capabilities," *Drones*, vol. 6, no. 3, p. 64, Feb. 2022. DOI: 10.3390/drones6030064.

[15] N. Metni and T. Hamel, "A UAV for bridge inspection: Visual servoing control law with orientation limits," *Automation in Construction*, vol. 17, no. 1, pp. 3–10, Nov. 2007. DOI: 10.1016/j.autcon.2006.12.010.

[16] Y. Benkhoui, T. E. Korchi, and L. Reinhold, "UAS-based crack detection using stereo cameras: A comparative study," in *2019 International Conference on Unmanned Aircraft Systems (ICUAS)*, IEEE, Jun. 2019. DOI: 10.1109/icuas.2019.8798311.

[17] C. M. Yeum and S. J. Dyke, "Vision-based automated crack detection for bridge inspection," *Computer-Aided Civil and Infrastructure Engineering*, vol. 30, no. 10, pp. 759–770, May 2015. DOI: 10.1111/mice.12141.

[18] F. Khan *et al.*, "Investigation on bridge assessment using unmanned aerial systems," in *Structures Congress 2015*, American Society of Civil Engineers, Apr. 2015. DOI: 10.1061/9780784479117.035.

[19] W. Chiu, W. Ong, T. Kuen, and F. Courtney, "Large structures monitoring using unmanned aerial vehicles," *Procedia Engineering*, vol. 188, pp. 415–423, 2017. DOI: 10.1016/j.proeng.2017.04.503.

[20] L. Palombi *et al.*, "Fluorescence LIDAR experiments and their integration in a user-friendly platform to support inspection of railway bridges," in *Remote Sensing Technologies and Applications in Urban Environments VII*, N. Chrysoulakis, T. Erbertseder, and Y. Zhang, Eds., SPIE, Oct. 2022. DOI: 10.1117/12.2638533.

[21] S. Feroz and S. A. Dabous, "UAV-based remote sensing applications for bridge condition assessment," *Remote Sensing*, vol. 13, no. 9, p. 1809, May 2021. DOI: 10.3390/rs13091809.

[22] P. J. Sanchez-Cuevas, G. Heredia, and A. Ollero, "Multirotor UAS for bridge inspection by contact using the ceiling effect," in *2017 International Conference on Unmanned Aircraft Systems (ICUAS)*, IEEE, Jun. 2017. DOI: 10.1109/icuas.2017.7991412.

[23] P. Sanchez-Cuevas, P. Ramon-Soria, B. Arrue, A. Ollero, and G. Heredia, "Robotic system for inspection by contact of bridge beams using UAVs," *Sensors*, vol. 19, no. 2, p. 305, Jan. 2019. DOI: 10.3390/s19020305.

[24] D. Zhang, R. Watson, C. MacLeod, G. Dobie, W. Galbraith, and G. Pierce, "Implementation and evaluation of an autonomous airborne ultrasound inspection system," *Nondestructive Testing and Evaluation*, vol. 37, no. 1, pp. 1–21, Feb. 2021. DOI: 10.1080/10589759.2021.1889546.

[25] R. Watson *et al.*, "Dry coupled ultrasonic non-destructive evaluation using an over-actuated unmanned aerial vehicle," *IEEE Transactions on Automation Science and Engineering*, vol. 19, no. 4, pp. 2874–2889, Oct. 2022. DOI: 10.1109/tase.2021.3094966.

[26] R. R. Murphy *et al.*, "Robot-assisted bridge inspection," *Journal of Intelligent & Robotic Systems*, vol. 64, no. 1, pp. 77–95, Jan. 2011. DOI: 10.1007/s10846-010-9514-8.

[27] M. Benndorf *et al.*, "Robotic bridge statics assessment within strategic flood evacuation planning using low-cost sensors," in *2017 IEEE International Symposium on Safety, Security and Rescue Robotics (SSRR)*, IEEE, Oct. 2017. DOI: 10.1109/ssrr.2017.8088133.

[28] T. Le, S. Gibb, N. Pham, H. M. La, L. Falk, and T. Berendsen, "Autonomous robotic system using non-destructive evaluation methods for bridge deck inspection," in *2017 IEEE International Conference on Robotics and Automation (ICRA)*, IEEE, May 2017. DOI: 10.1109/icra.2017.7989421.

[29] D. Kang and Y.-J. Cha, "Autonomous UAVs for structural health monitoring using deep learning and an ultrasonic beacon system with geo-tagging," *Computer-Aided Civil and Infrastructure Engineering*, vol. 33, no. 10, pp. 885–902, May 2018. DOI: 10.1111/mice.12375.

[30] S. G. Manyam, S. Rathinam, S. Darbha, D. Casbeer, Y. Cao, and P. Chandler, "GPS denied UAV routing with communication constraints," *Journal of Intelligent & Robotic Systems*, vol. 84, no. 1-4, pp. 691–703, Feb. 2016. DOI: 10.1007/s10846-016-0343-2.

[31] L. M. G. de Santos, J. Martínez-Sánchez, H. González-Jorge, A. Novo, and P. Arias, "FIRST APPROACH TO UAV-BASED CONTACT INSPECTION: A SMART PAYLOAD FOR NAVIGATION IN THE NEIGHBOURHOOD OF STRUCTURES," *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, vol. XLII-2/W13, pp. 323–328, Jun. 2019. DOI: 10.5194/isprs-archives-xlii-2-w13-323-2019.

[32] L. M. González-deSantos, J. Martínez-Sánchez, H. González-Jorge, M. Ribeiro, J. B. de Sousa, and P. Arias, "Payload for contact inspection tasks with UAV systems," *Sensors*, vol. 19, no. 17, p. 3752, Aug. 2019. DOI: 10.3390/s19173752.

[33] L. González-deSantos, J. Martínez-Sánchez, H. González-Jorge, F. Navarro-Medina, and P. Arias, "UAV payload with collision mitigation for contact inspection," *Automation in Construction*, vol. 115, p. 103 200, Jul. 2020. DOI: 10.1016/j.autcon.2020.103200.

[34] S. il Lee, H. Kim, U. Kim, and H. Lee, "Concave wall surface tracking for aerial manipulator using contact force estimation algorithm," in *2020 20th International Conference on Control, Automation and Systems (ICCAS)*, IEEE, Oct. 2020. DOI: 10.23919/iccas50221.2020.9268298.

[35] J. Feddema and O. Mitchell, "Vision-guided servoing with feature-based trajectory generation (for robots)," *IEEE Transactions on Robotics and Automation*, vol. 5, no. 5, pp. 691–700, 1989. DOI: 10.1109/70.88086.

[36] P. Tong, X. Yang, Y. Yang, W. Liu, and P. Wu, "Multi-UAV collaborative absolute vision positioning and navigation: A survey and discussion," *Drones*, vol. 7, no. 4, p. 261, Apr. 2023. DOI: 10.3390/drones7040261.

[37] P.-H. Chu, Y. T. Huang, C.-H. Pi, and S. Cheng, "Autonomous landing system of a VTOL UAV on an upward docking station using visual servoing," *IFAC-PapersOnLine*, vol. 55, no. 27, pp. 108–113, 2022. DOI: 10.1016/j.ifacol.2022.10.496.

[38] H. Ranjbar, P. Forsythe, A. A. F. Fini, M. Maghrebi, and T. S. Waller, "Addressing practical challenge of using autopilot drone for asphalt surface monitoring: Road detection, segmentation, and following," *Results in Engineering*, vol. 18, p. 101 130, Jun. 2023. DOI: 10.1016/j.rineng.2023.101130.

[39] T. Kominami, H. Paul, and K. Shimonomura, "Detection and localization of thin vertical board for UAV perching," *Journal of Robotics and Mechatronics*, vol. 35, no. 2, pp. 398–407, Apr. 2023. DOI: 10.20965/jrm.2023.p0398.

[40] L. F. Diniz, M. F. Pinto, A. G. Melo, and L. M. Honório, "Visual-based assistive method for UAV power line inspection and landing," *Journal of Intelligent & Robotic Systems*, vol. 106, no. 2, Oct. 2022. DOI: 10.1007/s10846-022-01725-x.

[41] G. Wang, J. Qin, Q. Liu, Q. Ma, and C. Zhang, "Image-based visual servoing of quadrotors to arbitrary flight targets," *IEEE Robotics and Automation Letters*, vol. 8, no. 4, pp. 2022–2029, Apr. 2023. DOI: 10.1109/lra.2023.3245416.

[42] M. S. Amiri and R. Ramli, "Visual navigation system for autonomous drone using fiducial marker detection," *International Journal of Advanced Computer Science and Applications*, vol. 13, no. 9, 2022. DOI: 10.14569/ijacsa.2022.0130981.

[43] E. Mraz, J. Rodina, and A. Babinec, "Using fiducial markers to improve localization of a drone," in *2020 23rd International Symposium on Measurement and Control in Robotics (ISMCR)*, IEEE, Oct. 2020. DOI: 10.1109/ismcr51255.2020.9263754.

[44] A. S. Priambodo, F Arifin, A Nasuha, Muslikhin, and A Winursito, "A vision and GPS based system for autonomous precision vertical landing of UAV quadcopter," *Journal of Physics: Conference Series*, vol. 2406, no. 1, p. 012 004, Dec. 2022. DOI: 10.1088/1742-6596/2406/1/012004.

[45] J. Springer and M. Kyas, "Evaluation of orientation ambiguity and detection rate in april tag and WhyCode," in *2022 Sixth IEEE International Conference on Robotic Computing (IRC)*, IEEE, Dec. 2022. DOI: 10.1109/irc55401.2022.00054.

[46] J. Springer and M. Kyas, "Autonomous drone landing with fiducial markers and a gimbal-mounted camera for active tracking," in *2022 Sixth IEEE International Conference on Robotic Computing (IRC)*, IEEE, Dec. 2022. DOI: 10.1109/irc55401.2022.00047.

[47] O. Bouaiss, R. Mechgoug, and A. Taleb-Ahmed, "Visual soft landing of an autonomous quadrotor on a moving pad using a combined fuzzy velocity control with model predictive control," *Signal, Image and Video Processing*, vol. 17, no. 1, pp. 21–30, Apr. 2022. DOI: 10.1007/s11760-022-02199-y.

[48] J. Morales, I. Castelo, R. Serra, P. U. Lima, and M. Basiri, "Vision-based autonomous following of a moving platform and landing for an unmanned aerial vehicle," *Sensors*, vol. 23, no. 2, p. 829, Jan. 2023. DOI: 10.3390/s23020829.

[49] X. Meng, H. Xi, J. Wei, Y. He, J. Han, and A. Song, "Rotorcraft aerial vehicle's contact-based landing and vision-based localization research," *Robotica*, vol. 41, no. 4, pp. 1127–1144, Nov. 2022. DOI: 10.1017/s0263574722001552.

[50] A. Arif, H. Wang, H. Castañeda, and Y. Wang, "Finite-time tracking of moving platform with single camera for quadrotor autonomous landing," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 70, no. 6, pp. 2573–2586, Jun. 2023. DOI: 10.1109/tcsi.2023.3256063.

[51] P. Troll, K. Szipka, and A. Archenti, "Indoor localization of quadcopters in industrial environment," in *Advances in Transdisciplinary Engineering*, IOS Press, Dec. 2020. DOI: 10.3233/atde200183.

[52] C. A. Toro-Arcila, H. M. Becerra, and G. Arechavaleta, "Visual path following with obstacle avoidance for quadcopters in indoor environments," *Control Engineering Practice*, vol. 135, p. 105 493, Jun. 2023. DOI: 10.1016/j.conengprac.2023.105493.

[53] R. Luo, R. Mullen, and D. Wessell, "An adaptive robotic tracking system using optical flow," in *Proceedings. 1988 IEEE International Conference on Robotics and Automation*, IEEE Comput. Soc. Press. DOI: 10.1109/robot.1988.12112.

[54] A. J. Bray, "Tracking objects using image disparities," in *Procedings of the Alvey Vision Conference 1989*, Alvey Vision Club, 1989. DOI: 10.5244/c.3.14.

[55] F. Kendoul, I. Fantoni, and K. Nonami, *Optic flow-based vision system for autonomous 3d localization and control of small aerial vehicles*, Feb. 2013. DOI: 10.1002/9781118599938.ch11.

[56] T. Oskiper, Z. Zhu, S. Samarasekera, and R. Kumar, "Visual odometry system using multiple stereo cameras and inertial measurement unit," in *2007 IEEE Conference on Computer Vision and Pattern Recognition*, IEEE, Jun. 2007. DOI: 10.1109/cvpr.2007.383087.

[57] L. Heng and B. Choi, "Semi-direct visual odometry for a fisheye-stereo camera," in *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, IEEE, Oct. 2016. DOI: 10.1109/iros.2016.7759600.

[58] J. Kersten and V. Rodehorst, "ENHANCEMENT STRATEGIES FOR FRAME-TO-FRAME UAS STEREO VISUAL ODOMETRY," *ISPRS - International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, vol. XLI-B3, pp. 511–518, Jun. 2016. DOI: 10.5194/isprsarchives-xli-b3-511-2016.

[59] I. Sa *et al.*, "Build your own visual-inertial drone: A cost-effective and open-source autonomous drone," *IEEE Robotics & Automation Magazine*, vol. 25, no. 1, pp. 89–103, Mar. 2018. DOI: 10.1109/mra.2017.2771326.

[60] A. C. Fauli, P. R. Soria, C. M. D. D. Espada, M. Trujillo, A. Viguria, and A. Ollero, "Assisted flight control for aerial contact UAV s in industrial environments," in *2021 Aerial Robotic Systems Physically Interacting with the Environment (AIRPHARO)*, IEEE, Oct. 2021. DOI: 10.1109/airpharo52252.2021.9571048.

[61] T. Nguyen. "Precision landing with realsense t265 camera and apriltag." (Nov. 7, 2019), [Online]. Available: https://discuss.ardupilot.org/t/precision-landing-with-realsense-t265-camera-and-apriltag-part-1-2/48978.

[62] S. Chen, Y. Feng, C.-Y. Wen, Y. Zou, and W. Chen, "Stereo visual inertial pose estimation based on feedforward and feedbacks," *IEEE/ASME Transactions on Mechatronics*, pp. 1–11, 2023. DOI: 10.1109/tmech.2023.3272208.

[63] J. Park and A. J. Choi, "Vision-based in-flight collision avoidance control based on background subtraction using embedded system," *Sensors*, vol. 23, no. 14, p. 6297, Jul. 2023. DOI: 10.3390/s23146297.

[64] A. Ronzhin, T. Ngo, Q. Vu, and V. Nguyen, "Analysis of approaches to the control of air manipulation systems," in *Ground and Air Robotic Manipulation Systems in Agriculture*, Springer International Publishing, Sep. 2021, pp. 179–204. DOI: 10.1007/978-3-030-86826-0_9.

[65] P. Lassen and M. Fumagalli, "Can your drone touch? exploring the boundaries of consumer-grade multirotors for physical interaction," in *2022 International Conference on Robotics and Automation (ICRA)*, IEEE, May 2022. DOI: 10.1109/icra46639.2022.9812187.

[66] P. D. Suthar and V. Sangwan, "Contact force-velocity control for a planar aerial manipulator," *IFAC-PapersOnLine*, vol. 55, no. 1, pp. 1–7, 2022. DOI: 10.1016/j.ifacol.2022.04.001.

[67] A. Ivanovic, L. Markovic, M. Car, I. Duvnjak, and M. Orsag, "Towards autonomous bridge inspection: Sensor mounting using aerial manipulators," *Applied Sciences*, vol. 11, no. 18, p. 8279, Sep. 2021. DOI: 10.3390/app11188279.

[68] R. WATSON *et al.*, "TECHNIQUES FOR CONTACT-BASED STRUCTURAL HEALTH MONITORING WITH MULTIROTOR UNMANNED AERIAL VEHICLES," in *Proceedings of the 13th International Workshop on Structural Health Monitoring*, Destech Publications, Inc., Mar. 2022. DOI: 10.12783/shm2021/36236.

[69] D. Smrcka, T. Baca, T. Nascimento, and M. Saska, "Admittance force-based UAV-wall stabilization and press exertion for documentation and inspection of historical buildings," in *2021 International Conference on Unmanned Aircraft Systems (ICUAS)*, IEEE, Jun. 2021. DOI: 10.1109/icuas51884.2021.9476873.

[70] S. Hamaza, I. Georgilas, and T. Richardson, "Towards an adaptive-compliance aerial manipulator for contact- based interaction," in *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, IEEE, Oct. 2018. DOI: 10.1109/iros.2018.8593576.

[71] H. Zhou, J. P. Lynch, and D. Zekkos, "Vision-based precision localization of UAVs for sensor payload placement and pickup for field monitoring applications," in *Sensors and Smart Structures Technologies for Civil, Mechanical, and Aerospace Systems 2019*, K.-W. Wang, H. Sohn, H. Huang, and J. P. Lynch, Eds., SPIE, Mar. 2019. DOI: 10.1117/12.2516049.

[72] S. Carroll, J. Satme, S. Alkharusi, N. Vitzilaios, A. Downey, and D. Rizos, "Drone-based vibration monitoring and assessment of structures," *Applied Sciences*, vol. 11, no. 18, p. 8560, Sep. 2021. DOI: 10.3390/app11188560.

[73] P. Garg, F. Moreu, A. Ozdagli, M. R. Taha, and D. Mascareñas, "Noncontact dynamic displacement measurement of structures using a moving laser doppler vibrometer," *Journal of Bridge Engineering*, vol. 24, no. 9, Sep. 2019. DOI: 10.1061/(asce)be.1943-5592.0001472.

[74] V. J. Hodge, S. O'Keefe, M. Weeks, and A. Moulds, "Wireless sensor networks for condition monitoring in the railway industry: A survey," *IEEE Transactions on Intelligent Transportation Systems*, vol. 16, no. 3, pp. 1088–1106, Jun. 2015. DOI: 10.1109/tits.2014.2366512.

[75] O. D. Dantsker and R. Mancuso, "Flight data acquisition platform development, integration, and operation on small- to medium-sized unmanned aircraft," in *AIAA Scitech 2019 Forum*, American Institute of Aeronautics and Astronautics, Jan. 2019. DOI: 10.2514/6.2019-1262.

[76] H. Lim, J. Park, D. Lee, and H. Kim, "Build your own quadrotor: Open-source projects on unmanned aerial vehicles," *IEEE Robotics & Automation Magazine*, vol. 19, no. 3, pp. 33–45, Sep. 2012. DOI: 10.1109/mra.2012.2205629.

[77] E. Ebeid, M. Skriver, and J. Jin, "A survey on open-source flight control platforms of unmanned aerial vehicle," in *2017 Euromicro Conference on Digital System Design (DSD)*, IEEE, Aug. 2017. DOI: 10.1109/dsd.2017.30.

[78] S. Mehta, A. Patel, and J. Mehta, "CCD or CMOS image sensor for photography," in *2015 International Conference on Communications and Signal Processing (ICCSP)*, IEEE, Apr. 2015. DOI: 10.1109/iccsp.2015.7322890.

[79] M. Wany and G. Israel, "CMOS image sensor with NMOS-only global shutter and enhanced responsivity," *IEEE Transactions on Electron Devices*, vol. 50, no. 1, pp. 57–62, Jan. 2003. DOI: 10.1109/ted.2002.807253.

[80] A. Khazetdinov, A. Zakiev, T. Tsoy, M. Svinin, and E. Magid, "Embedded ArUco: A novel approach for high precision UAV landing," in *2021 International Siberian Conference on Control and Communications (SIBCON)*, IEEE, May 2021. DOI: 10.1109/sibcon50419.2021.9438855.

[81] E. S. Mcvey and J. W. Lee, "Some accuracy and resolution aspects of computer vision distance measurements," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-4, no. 6, pp. 646–649, Nov. 1982. DOI: 10.1109/tpami.1982.4767319.

[82] K. Lu, X. Wang, Z. Wang, and L. Wang, "Binocular stereo vision based on OpenCV," in *IET International Conference on Smart and Sustainable City (ICSSC 2011)*, IET, 2011. DOI: 10.1049/cp.2011.0312.

[83] F. J. Romero-Ramirez, R. Muñoz-Salinas, and R. Medina-Carnicer, "Speeded up detection of squared fiducial markers," *Image and Vision Computing*, vol. 76, pp. 38–47, Aug. 2018. DOI: 10.1016/j.imavis.2018.05.004.

[84] S. Garrido-Jurado, R. Muñoz-Salinas, F. Madrid-Cuevas, and R. Medina-Carnicer, "Generation of fiducial marker dictionaries using mixed integer linear programming," *Pattern Recognition*, vol. 51, pp. 481–491, Mar. 2016. DOI: 10.1016/j.patcog.2015.09.023.

[85] E. Olson, "AprilTag: A robust and flexible visual fiducial system," in *2011 IEEE International Conference on Robotics and Automation*, IEEE, May 2011. DOI: 10.1109/icra.2011.5979561.

[86] J. Wang and E. Olson, "AprilTag 2: Efficient and robust fiducial detection," in *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, IEEE, Oct. 2016. DOI: 10.1109/iros.2016.7759617.

[87] Y. Wang, Z. Zheng, Z. Su, G. Yang, Z. Wang, and Y. Luo, "An improved ArUco marker for monocular vision ranging," in *2020 Chinese Control And Decision Conference (CCDC)*, IEEE, Aug. 2020. DOI: 10.1109/ccdc49329.2020.9164176.

[88] A. Zakiev, T. Tsoy, K. Shabalina, E. Magid, and S. K. Saha, "Virtual experiments on ArUco and AprilTag systems comparison for fiducial marker rotation resistance under noisy sensory data," in *2020 International Joint Conference on Neural Networks (IJCNN)*, IEEE, Jul. 2020. DOI: 10.1109/ijcnn48605.2020.9207701.

[89] P.-C. Wu, Y.-H. Tsai, and S.-Y. Chien, "Stable pose tracking from a planar target with an analytical motion model in real-time applications," in *2014 IEEE 16th International Workshop on Multimedia Signal Processing (MMSP)*, IEEE, Sep. 2014. DOI: 10.1109/mmsp.2014.6958793.

[90] H. C. Kam, Y. K. Yu, and K. H. Wong, "An improvement on ArUco marker for pose tracking using kalman filter," in *2018 19th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, IEEE, Jun. 2018. DOI: 10.1109/snpd.2018.8441049.

Appendix A: Bill of Materials for Configuration 5

Item	Description	Cost Estimate	Notes
DJI Flamewheel F450 Kit	450 Sized Quadcopter kit, with ESC, motors, and propellers	$314.69	Discontinued, Newer kits are available
Orange RX DSM2 RC Receiver	Sattelite receiver used for RC Control	$21.04	
2200 mAh 3S LiPo Battery	11.1 V battery	$14.72	
Pixhawk 2 Kit	Includes flight controller, buzzer, arming switch	$199.99	
Intel Realsense T265	Camera with position estimate	$759.99	Discontinued, Newer cameras available
Jetson Nano 4G	Onboard Linux computer	$267.96	
3D Printed Top Plate	-	$3.00	Cost of PLA. Overestimate
5V 2A DC DC Regulator	-	$2.25	Used to power Jetson from batteries
TP-Link Nano AC600	WiFi Module for Jetson	$14.99	
Arm 3D printed parts	-	$1.00	Cost of PLA. Overestimate
Generic Carwash Sponge	-	$3.99	
String	1 mm diameter, 10 cm long	-	
MPU6050 Breakout board	6 axis IMU	$3.33	Discontinued
Wooden Dowel	0.5 mm diamater, 30 cm long	$4.99	Cost for 30 pack
Rubber bumper		-	Taken from IKEA curtain
	Total	$1,611.94	

Appendix B: Python UAV Control Software

B.1 Installation Instructions

Listing B.1: Installation instruction be run from terminal

```
## Installation
sudo apt install python3 python3-pip python3-dev python3-venv cmake libssl-dev ↩
    xorg-dev libglu1-mesa-dev libudev-dev


upgrade to at least python 3.8
build pyrealsesnse2 from source

###
sudo apt-get update && sudo apt-get -y upgrade
sudo apt-get install -y --no-install-recommends \
    python3 \
    python3-setuptools \
    python3-pip \
    python3-dev

sudo apt-get install -y git libssl-dev libusb-1.0-0-dev pkg-config libgtk--dev
sudo apt-get install -y libglfw3-dev libgl1-mesa-dev libglu1-mesa-dev libxml2 ↩
    libxml2-dev libxslt-dev

git clone https://github.com/IntelRealSense/librealsense.git
cd ./librealsense

./scripts/setup_udev_rules.sh

mkdir build && cd build
cmake ../ -DBUILD_PYTHON_BINDINGS:bool=true

sudo make uninstall && sudo make clean && sudo make -j4 && sudo make install

##Export pyrealsense2 to your PYTHONPATH so `import pyrealsense2` works (can add ↩
    to bashrc)
export PYTHONPATH=$PYTHONPATH:/usr/local/lib/python3.8/pyrealsense2


python3.8 -m venv .venv

source .venv/bin/activate
pip install --upgrade pip
pip install cython
pip install wheel numpy
pip install transformations pyserial scikit-build
pip install dronekit apscheduler==3.8.0 py-imu-mpu6050
pip install opencv-python dt-apriltags
```

```
on each startup
sudo echo 400000 > /sys/bus/i2c/devices/i2c-1/bus_clk_rate
```

B.2 Main File

Listing B.2: This file is the main file for the program and spawns all objects and threads

```
#!/usr/bin/env python3

######################################################
##          librealsense T265 to MAVLink          ##
######################################################
# This script assumes pyrealsense2.[].so file is found under the same directory as ↩
    this script
# Install required packages:
#   pip3 install pyrealsense2
#   pip3 install transformations
#   pip3 install pymavlink
#   pip3 install apscheduler
#   pip3 install pyserial

# File based of off work by Thien Nguyen https://github.com/thien94/ ↩
    vision_to_mavros

# Set the path for IDLE
import sys
sys.path.append("/usr/local/lib/")

# Set MAVLink protocol to 2.
import os
os.environ["MAVLINK20"] = "1"

# Import the libraries
import pyrealsense2 as rs
import numpy as np
import transformations as tf
import math as m
import time
import argparse
import threading
import signal

from time import sleep
from apscheduler.schedulers.background import BackgroundScheduler
from dronekit import connect, VehicleMode
from pymavlink import mavutil

import cv2
import dt_apriltags
import mpu6050_logger
import vision_control

# Replacement of the standard print() function to flush the output
def progress(string):
    print(string, file=sys.stdout)
    sys.stdout.flush()

#######################################
# Parameters
#######################################

# Default configurations for connection to the FCU
connection_string_default = '/dev/ttyTHS1'
```

```
connection_baudrate_default =  460800 #921600
connection_timeout_sec_default = 5

# Transformation to convert different camera orientations to NED convention. Replace camera_orientation_default for your configuration.
#   0: Forward, USB port to the right
#   1: Downfacing, USB port to the right
#   2: Forward, 45 degree tilted down
# Important note for downfacing camera: you need to tilt the vehicle's nose up a little - not flat - before you run the script, otherwise the initial yaw will be randomized, read here for more details: https://github.com/IntelRealSense/librealsense/issues/4080. Tilt the vehicle to any other sides and the yaw might not be as stable.
camera_orientation_default = 0

# https://mavlink.io/en/messages/common.html#VISION_POSITION_ESTIMATE
enable_msg_vision_position_estimate = True
vision_position_estimate_msg_hz_default = 30.0

# https://mavlink.io/en/messages/ardupilotmega.html#VISION_POSITION_DELTA
enable_msg_vision_position_delta = False
vision_position_delta_msg_hz_default = 30.0

# https://mavlink.io/en/messages/common.html#VISION_SPEED_ESTIMATE
enable_msg_vision_speed_estimate = True
vision_speed_estimate_msg_hz_default = 30.0

# https://mavlink.io/en/messages/common.html#STATUSTEXT
enable_update_tracking_confidence_to_gcs = True
update_tracking_confidence_to_gcs_hz_default = 1.0

# Monitor user's online input via keyboard, can only be used when runs from terminal
enable_user_keyboard_input = True

# Default global position for EKF home/ origin
enable_auto_set_ekf_home = False
home_lat = 535608820     # Somewhere random
home_lon = -1138528910      # Somewhere random
home_alt = 600000        # Somewhere random

# TODO: Taken care of by ArduPilot, so can be removed (once the handling on AP side is confirmed stable)
# In NED frame, offset from the IMU or the center of gravity to the camera's origin point
body_offset_enabled = 0
body_offset_x = 0  # In meters (m)
body_offset_y = 0  # In meters (m)
body_offset_z = 0  # In meters (m)

# Global scale factor, position x y z will be scaled up/down by this factor
scale_factor = 1.0

# Enable using yaw from compass to align north (zero degree is facing north)
compass_enabled = 0

# pose data confidence: 0x0 - Failed / 0x1 - Low / 0x2 - Medium / 0x3 - High
pose_data_confidence_level = ('FAILED', 'Low', 'Medium', 'High')

# lock for thread synchronization
lock = threading.Lock()
mavlink_thread_should_exit = False

# default exit code is failure - a graceful termination with a
# terminate signal is possible.
exit_code = 1
```

```
115 ######################################
    # Functions for OpenCV
    ######################################
    """
    In this section, we will set up the functions that will translate the camera
120 intrinsics and extrinsics from librealsense into parameters that can be used
    with OpenCV.
    The T265 uses very wide angle lenses, so the distortion is modeled using a four
    parameter distortion model known as Kanalla-Brandt. OpenCV supports this
    distortion model in their "fisheye" module, more details can be found here:
125 https://docs.opencv.org/3.4/db/d58/group__calib3d__fisheye.html
    """

    """
    Returns R, T transform from src to dst
130 """
    def get_extrinsics(src, dst):
         extrinsics = src.get_extrinsics_to(dst)
        R = np.reshape(extrinsics.rotation, [3,3]).T
        T = np.array(extrinsics.translation)
135     return (R, T)

    """
    Returns a camera matrix K from librealsense intrinsics
    """
140 def camera_matrix(intrinsics):
        return np.array([[intrinsics.fx,             0, intrinsics.ppx],
                         [            0, intrinsics.fy, intrinsics.ppy],
                         [            0,             0,              1]])

145 """
    Returns the fisheye distortion from librealsense intrinsics
    """
    def fisheye_distortion(intrinsics):
        return np.array(intrinsics.coeffs[:4])
150
    ######################################
    # Functions for AprilTag detection
    ######################################
    tag_landing_id = 19
155 tag_landing_size = 0.151           # tag's border size, measured in meter
    tag_image_source = "left"   # for Realsense T265, we can use "left" or "right"

    at_detector = dt_apriltags.Detector(searchpath=['apriltags'],
                           families='tag36h11',
160                        nthreads=1,
                           quad_decimate=1.0,
                           quad_sigma=0.0,
                           refine_edges=1,
                           decode_sharpening=0.25,
165                        debug=0)
    ######################################
    # Global variables
    ######################################

170 # FCU connection variables

    # Camera-related variables
    pipe = None
    pose_sensor = None
175 linear_accel_cov = 0.01
    angular_vel_cov  = 0.01

    # Data variables
    data = None
180 prev_data = None
    H_aeroRef_aeroBody = None
    V_aeroRef_aeroBody = None
```

```
heading_north_yaw = None
current_confidence_level = None
current_time_us = 0

# Increment everytime pose_jumping or relocalization happens
# See here: https://github.com/IntelRealSense/librealsense/blob/master/doc/t265.md#are-there-any-t265-specific-options
# For AP, a non-zero "reset_counter" would mean that we could be sure that the user's setup was using mavlink2
reset_counter = 1

#######################################
# Parsing user' inputs
#######################################

parser = argparse.ArgumentParser(description='Reboots vehicle')
parser.add_argument('--connect',
                    help="Vehicle connection target string. If not specified, a default string will be used.")
parser.add_argument('--baudrate', type=float,
                    help="Vehicle connection baudrate. If not specified, a default value will be used.")
parser.add_argument('--vision_position_estimate_msg_hz', type=float,
                    help="Update frequency for VISION_POSITION_ESTIMATE message. If not specified, a default value will be used.")
parser.add_argument('--vision_position_delta_msg_hz', type=float,
                    help="Update frequency for VISION_POSITION_DELTA message. If not specified, a default value will be used.")
parser.add_argument('--vision_speed_estimate_msg_hz', type=float,
                    help="Update frequency for VISION_SPEED_DELTA message. If not specified, a default value will be used.")
parser.add_argument('--scale_calib_enable', default=False, action='store_true',
                    help="Scale calibration. Only run while NOT in flight")
parser.add_argument('--camera_orientation', type=int,
                    help="Configuration for camera orientation. Currently supported: forward, usb port to the right - 0; downward, usb port to the right - 1, 2: forward tilted down 45deg")
parser.add_argument('--debug_enable',type=int,
                    help="Enable debug messages on terminal")
parser.add_argument('--visualization',type=int,
                    help="Enable visualization. Ensure that a monitor is connected")
parser.add_argument('--log_file_name',type=str,
                    help="string for file name")

args = parser.parse_args()

connection_string = args.connect
connection_baudrate = args.baudrate
vision_position_estimate_msg_hz = args.vision_position_estimate_msg_hz
vision_position_delta_msg_hz = args.vision_position_delta_msg_hz
vision_speed_estimate_msg_hz = args.vision_speed_estimate_msg_hz
scale_calib_enable = args.scale_calib_enable
camera_orientation = args.camera_orientation
debug_enable = args.debug_enable
visualization = args.visualization
log_file_name = args.log_file_name

# Using default values if no specified inputs
if not connection_string:
    connection_string = connection_string_default
    progress("INFO: Using default connection_string %s" % connection_string)
else:
    progress("INFO: Using connection_string %s" % connection_string)

if not connection_baudrate:
    connection_baudrate = connection_baudrate_default
    progress("INFO: Using default connection_baudrate %s" % connection_baudrate)
```

```
else:
    progress("INFO: Using connection_baudrate %s" % connection_baudrate)

if not vision_position_estimate_msg_hz:
245     vision_position_estimate_msg_hz = vision_position_estimate_msg_hz_default
    progress("INFO: Using default vision_position_estimate_msg_hz %s" % ↩
        vision_position_estimate_msg_hz)
else:
    progress("INFO: Using vision_position_estimate_msg_hz %s" % ↩
        vision_position_estimate_msg_hz)

250 if not vision_position_delta_msg_hz:
    vision_position_delta_msg_hz = vision_position_delta_msg_hz_default
    progress("INFO: Using default vision_position_delta_msg_hz %s" % ↩
        vision_position_delta_msg_hz)
else:
    progress("INFO: Using vision_position_delta_msg_hz %s" % ↩
        vision_position_delta_msg_hz)
255
if not vision_speed_estimate_msg_hz:
    vision_speed_estimate_msg_hz = vision_speed_estimate_msg_hz_default
    progress("INFO: Using default vision_speed_estimate_msg_hz %s" % ↩
        vision_speed_estimate_msg_hz)
else:
260     progress("INFO: Using vision_speed_estimate_msg_hz %s" % ↩
        vision_speed_estimate_msg_hz)

if body_offset_enabled == 1:
    progress("INFO: Using camera position offset: Enabled, x y z is %s %s %s" % ( ↩
        body_offset_x, body_offset_y, body_offset_z))
else:
265     progress("INFO: Using camera position offset: Disabled")

if compass_enabled == 1:
    progress("INFO: Using compass: Enabled. Heading will be aligned to north.")
else:
270     progress("INFO: Using compass: Disabled")

if scale_calib_enable == True:
    progress("\nINFO: SCALE CALIBRATION PROCESS. DO NOT RUN DURING FLIGHT.\nINFO: ↩
        TYPE IN NEW SCALE IN FLOATING POINT FORMAT\n")
else:
275     if scale_factor == 1.0:
        progress("INFO: Using default scale factor %s" % scale_factor)
    else:
        progress("INFO: Using scale factor %s" % scale_factor)

280 if not camera_orientation:
    camera_orientation = camera_orientation_default
    progress("INFO: Using default camera orientation %s" % camera_orientation)
else:
    progress("INFO: Using camera orientation %s" % camera_orientation)
285
if camera_orientation == 0:     # Forward, USB port to the right
    H_aeroRef_T265Ref   = np.array([[0,0,-1,0],[1,0,0,0],[0,-1,0,0],[0,0,0,1]])
    H_T265body_aeroBody = np.linalg.inv(H_aeroRef_T265Ref)
elif camera_orientation == 1:   # Downfacing, USB port to the right
290     H_aeroRef_T265Ref   = np.array([[0,0,-1,0],[1,0,0,0],[0,-1,0,0],[0,0,0,1]])
    H_T265body_aeroBody = np.array([[0,1,0,0],[1,0,0,0],[0,0,-1,0],[0,0,0,1]])
elif camera_orientation == 2:   # 45degree forward
    H_aeroRef_T265Ref   = np.array([[0,0,-1,0],[1,0,0,0],[0,-1,0,0],[0,0,0,1]])
    H_T265body_aeroBody = (tf.euler_matrix(m.pi/4, 0, 0)).dot(np.linalg.inv( ↩
        H_aeroRef_T265Ref))
295 else:                           # Default is facing forward, USB port to the right
    H_aeroRef_T265Ref   = np.array([[0,0,-1,0],[1,0,0,0],[0,-1,0,0],[0,0,0,1]])
    H_T265body_aeroBody = np.linalg.inv(H_aeroRef_T265Ref)

if not debug_enable:
```

```python
    debug_enable = 0
else:
    debug_enable = 1
    np.set_printoptions(precision=4, suppress=True) # Format output on terminal
    progress("INFO: Debug messages enabled.")

if not log_file_name:
    log_file_name = "test_log"
    print("INFO: Logging to test_log")

if not visualization:
    visualization = 0
    print("INFO: Visualization: Disabled")
else:
    visualization = 1
    print("INFO: Visualization: Enabled. Checking if monitor is connected...")
    WINDOW_TITLE = 'Apriltag detection from T265 images'
    cv2.namedWindow(WINDOW_TITLE, cv2.WINDOW_AUTOSIZE)
    print("INFO: Monitor is connected. Press `q` to exit.")
    display_mode = "stack"

#######################################
# Functions - MAVLink
#######################################

def forwarding_func(conn, msg):
    if msg.get_type() == "PARAM_VALUE":
        msg.param_id = str.encode(msg.param_id)
    elif msg.get_type() == "PARAM_SET":
        msg.param_id = str.encode(msg.param_id)
    elif msg.get_type() == "PARAM_REQUEST_READ":
        msg.param_id = str.encode(msg.param_id)
    elif msg.get_type() == "STATUSTEXT":
        msg.text = str.encode(msg.text)

    conn.mav.send(msg)

def reboot():
    with lock:
        conn.mav.command_long_send(
            0, 0,  # target_system, target_component
            mavutil.mavlink.MAV_CMD_PREFLIGHT_REBOOT_SHUTDOWN,  # command
            0,  # confirmation
            1,  # param 1, autopilot (reboot)
            0,  # param 2, onboard computer (do nothing)
            0,  # param 3, camera (do nothing)
            0,  # param 4, mount (do nothing)
            0, 0, 0)  # param 5 ~ 7 not used

def send_hb():
    with lock:
        conn.mav.heartbeat_send(mavutil.mavlink.MAV_TYPE_ONBOARD_CONTROLLER,
                                mavutil.mavlink.MAV_AUTOPILOT_GENERIC,
                                0,
                                0,
                                0)

def mavlink_loop(conn, callbacks, fwd_conn):
    '''a main routine for a thread; reads data from a mavlink connection,
    calling callbacks based on message type received.
    '''
    while not mavlink_thread_should_exit:
        m = conn.recv_match(timeout=1, blocking=True)
        if m is None:
            continue
        try:
            for c in callbacks:
                c(m)
```

```
                forwarding_func(fwd_conn, m)
        except Exception as e:
370             print(e)
                print("got-an-error-with-msg-type:-" + str(m.get_type()))


    # https://mavlink.io/en/messages/common.html#VISION_POSITION_ESTIMATE
375 def send_vision_position_estimate_message():
        global current_time_us, H_aeroRef_aeroBody, reset_counter
        with lock:
            if H_aeroRef_aeroBody is not None:
                # Setup angle data
380             rpy_rad = np.array( tf.euler_from_matrix(H_aeroRef_aeroBody, 'sxyz'))

                # Setup covariance data, which is the upper right triangle of the ↩
                    covariance matrix, see here: https://files.gitter.im/ArduPilot/ ↩
                    VisionProjects/1DpU/image.png
                # Attemp #01: following this formula https://github.com/IntelRealSense ↩
                    /realsense-ros/blob/development/realsense2_camera/src/ ↩
                    base_realsense_node.cpp#L1406-L1411
                cov_pose    = linear_accel_cov * pow(10, 3 - int(data. ↩
                    tracker_confidence))
385             cov_twist   = angular_vel_cov  * pow(10, 1 - int(data. ↩
                    tracker_confidence))
                covariance  = np.array([cov_pose, 0, 0, 0, 0, 0,
                                           cov_pose, 0, 0, 0, 0,
                                              cov_pose, 0, 0, 0,
                                                cov_twist, 0, 0,
390                                                cov_twist, 0,
                                                      cov_twist])

                # Send the message
                conn.mav.vision_position_estimate_send(
395                 current_time_us,            # us Timestamp (UNIX time or time  ↩
                        since system boot)
                    H_aeroRef_aeroBody[0][3],   # Global X position
                    H_aeroRef_aeroBody[1][3],   # Global Y position
                    H_aeroRef_aeroBody[2][3],   # Global Z position
                    rpy_rad[0],                 # Roll angle
400                 rpy_rad[1],                 # Pitch angle
                    rpy_rad[2],                 # Yaw angle
                    covariance,                 # Row-major representation of pose 6x6 ↩
                         cross-covariance matrix
                    reset_counter               # Estimate reset counter. Increment  ↩
                        every time pose estimate jumps.
                )
405
    # https://mavlink.io/en/messages/ardupilotmega.html#VISION_POSITION_DELTA
    def send_vision_position_delta_message():
        global current_time_us, current_confidence_level, H_aeroRef_aeroBody
        with lock:
410         if H_aeroRef_aeroBody is not None:
                # Calculate the deltas in position, attitude and time from the  ↩
                    previous to current orientation
                H_aeroRef_PrevAeroBody      = send_vision_position_delta_message. ↩
                    H_aeroRef_PrevAeroBody
                H_PrevAeroBody_CurrAeroBody = (np.linalg.inv(H_aeroRef_PrevAeroBody)). ↩
                    dot(H_aeroRef_aeroBody)

415             delta_time_us    = current_time_us - ↩
                    send_vision_position_delta_message.prev_time_us
                delta_position_m = [H_PrevAeroBody_CurrAeroBody[0][3],  ↩
                    H_PrevAeroBody_CurrAeroBody[1][3], H_PrevAeroBody_CurrAeroBody ↩
                    [2][3]]
                delta_angle_rad  = np.array( tf.euler_from_matrix( ↩
                    H_PrevAeroBody_CurrAeroBody, 'sxyz'))

                # Send the message
```

```
420             conn.mav.vision_position_delta_send(
                    current_time_us,    # us: Timestamp (UNIX time or time since ↩
                        system boot)
                    delta_time_us,      # us: Time since last reported camera frame
                    delta_angle_rad,    # float[3] in radian: Defines a rotation ↩
                        vector in body frame that rotates the vehicle from the ↩
                        previous to the current orientation
                    delta_position_m,   # float[3] in m: Change in position from ↩
                        previous to current frame rotated into body frame (0=forward, ↩
                        1=right, 2=down)
425                 current_confidence_level # Normalized confidence value from 0 to ↩
                        100.
                )

                # Save static variables
                send_vision_position_delta_message.H_aeroRef_PrevAeroBody = ↩
                    H_aeroRef_aeroBody
430             send_vision_position_delta_message.prev_time_us = current_time_us

    # https://mavlink.io/en/messages/common.html#VISION_SPEED_ESTIMATE
    def send_vision_speed_estimate_message():
        global current_time_us, V_aeroRef_aeroBody, reset_counter
435     with lock:
            if V_aeroRef_aeroBody is not None:

                # Attemp #01: following this formula https://github.com/IntelRealSense ↩
                    /realsense-ros/blob/development/realsense2_camera/src/ ↩
                    base_realsense_node.cpp#L1406-L1411
                cov_pose    = linear_accel_cov * pow(10, 3 - int(data. ↩
                    tracker_confidence))
440             covariance  = np.array([cov_pose,   0,          0,
                                        0,          cov_pose,   0,
                                        0,          0,          cov_pose])

                # Send the message
445             conn.mav.vision_speed_estimate_send(
                    current_time_us,            # us Timestamp (UNIX time or time ↩
                        since system boot)
                    V_aeroRef_aeroBody[0][3],   # Global X speed
                    V_aeroRef_aeroBody[1][3],   # Global Y speed
                    V_aeroRef_aeroBody[2][3],   # Global Z speed
450                 covariance,                 # covariance
                    reset_counter               # Estimate reset counter. Increment ↩
                        every time pose estimate jumps.
                )


455 def goto_position_target_local_ned(forward, right, down, yaw):
        """
        Send SET_POSITION_TARGET_LOCAL_NED command to request the vehicle fly to a ↩
            specified
        location in the North, East, Down frame.
        It is important to remember that in this frame, positive altitudes are entered ↩
            as negative
460     "Down" values. So if down is "10", this will be 10 metres below the home ↩
            altitude.
        """
        with lock:
            conn.mav.set_position_target_local_ned_send(
                0,       # time_boot_ms (not used)
465             0, 0,    # target system, target component
                mavutil.mavlink.MAV_FRAME_BODY_OFFSET_NED, # frame
                0b100111111000, # type_mask (only positions enabled)
                forward, right, down, # x, y, z positions (or North, East, Down in the ↩
                    MAV_FRAME_BODY_NED frame
                0, 0, 0, # x, y, z velocity in m/s  (not used)
470             0, 0, 0, # x, y, z acceleration (not supported yet, ignored in ↩
                    GCS_Mavlink)
```

```python
            yaw, 0)    # yaw, yaw_rate (not supported yet, ignored in GCS_Mavlink)

# Update the changes of confidence level on GCS and terminal
def update_tracking_confidence_to_gcs():
    if data is not None and update_tracking_confidence_to_gcs.prev_confidence_level != data.tracker_confidence:
        confidence_status_string = 'Tracking confidence: ' + pose_data_confidence_level[data.tracker_confidence]
        send_msg_to_gcs(confidence_status_string)
        update_tracking_confidence_to_gcs.prev_confidence_level = data.tracker_confidence

# https://mavlink.io/en/messages/common.html#STATUSTEXT
def send_msg_to_gcs(text_to_be_sent):
    # MAV_SEVERITY: 0=EMERGENCY 1=ALERT 2=CRITICAL 3=ERROR, 4=WARNING, 5=NOTICE, 6=INFO, 7=DEBUG, 8=ENUM_END
    text_msg = 'T265: ' + text_to_be_sent
    conn.mav.statustext_send(mavutil.mavlink.MAV_SEVERITY_INFO, text_msg.encode())
    progress("INFO: %s" % text_to_be_sent)


# Send a mavlink SET_GPS_GLOBAL_ORIGIN message (http://mavlink.org/messages/common#SET_GPS_GLOBAL_ORIGIN), which allows us to use local position information without a GPS.
def set_default_global_origin():
    conn.mav.set_gps_global_origin_send(
        1,
        home_lat,
        home_lon,
        home_alt
    )

# Send a mavlink SET_HOME_POSITION message (http://mavlink.org/messages/common#SET_HOME_POSITION), which allows us to use local position information without a GPS.
def set_default_home_position():
    x = 0
    y = 0
    z = 0
    q = [1, 0, 0, 0]   # w x y z

    approach_x = 0
    approach_y = 0
    approach_z = 1

    conn.mav.set_home_position_send(
        1,
        home_lat,
        home_lon,
        home_alt,
        x,
        y,
        z,
        q,
        approach_x,
        approach_y,
        approach_z
    )


# Request a timesync update from the flight controller, for future work.
# TODO: Inspect the usage of timesync_update
def update_timesync(ts=0, tc=0):
    if ts == 0:
        ts = int(round(time.time() * 1000))
    conn.mav.timesync_send(
        tc,     # tc1
        ts      # ts1
```

```
    )

# Listen to attitude data to acquire heading when compass data is enabled
def att_msg_callback(value):
    if value.get_type() == "ATTITUDE":
        global heading_north_yaw
        if heading_north_yaw is None:
            heading_north_yaw = value.yaw
            progress("INFO: Received first ATTITUDE message with heading yaw %.2f degrees" % m.degrees(heading_north_yaw))

#######################################
# Functions - T265
#######################################

def increment_reset_counter():
    global reset_counter
    if reset_counter >= 255:
        reset_counter = 1
    reset_counter += 1

# List of notification events: https://github.com/IntelRealSense/librealsense/blob/development/include/librealsense2/h/rs_types.h
# List of notification API: https://github.com/IntelRealSense/librealsense/blob/development/common/notifications.cpp
def realsense_notification_callback(notif):
    progress("INFO: T265 event: " + notif)
    if notif.get_category() is rs.notification_category.pose_relocalization:
        increment_reset_counter()
        send_msg_to_gcs('Relocalization detected')

def realsense_connect():
    global pipe, pose_sensor

    # Declare RealSense pipeline, encapsulating the actual device and sensors
    pipe = rs.pipeline()

    # Build config object before requesting data
    cfg = rs.config()

    # Enable the stream we are interested in
    cfg.enable_stream(rs.stream.pose) # Positional data
    cfg.enable_stream(rs.stream.fisheye, 1)         # Image stream left
    cfg.enable_stream(rs.stream.fisheye, 2)         # Image stream right

    # Configure callback for relocalization event
    device = cfg.resolve(pipe).get_device()
    pose_sensor = device.first_pose_sensor()
    pose_sensor.set_notifications_callback(realsense_notification_callback)


    # Start streaming with requested config
    pipe.start(cfg)

#######################################
# Functions - Miscellaneous
#######################################

# Monitor user input from the terminal and perform action accordingly
def user_input_monitor():
    global scale_factor
    while True:
        # Special case: updating scale
        if scale_calib_enable == True:
            scale_factor = float(input("INFO: Type in new scale as float number\n"))
            progress("INFO: New scale is %s" % scale_factor)
```

```
595         if enable_auto_set_ekf_home:
                send_msg_to_gcs('Set EKF home with default GPS location')
                set_default_global_origin()
                set_default_home_position()
                time.sleep(1) # Wait a short while for FCU to start working
600
            # Add new action here according to the key pressed.
            # Enter: Set EKF home when user press enter
            try:
                c = input()
605             if c == "":
                    send_msg_to_gcs('Set EKF home with default GPS location')
                    set_default_global_origin()
                    set_default_home_position()
                elif c=="r":
610                 reboot()
                elif c == "q":
                    main_loop_should_quit = True
                else:
                    progress("Got keyboard input %s" % c)
615         except IOError: pass


    #######################################
    # Main code starts here
620 #######################################
    vc = vision_control.VisionControl(goto_position_target_local_ned, log_file_name + ↩
        "_vision_controller")

    try:
        progress("INFO: pyrealsense2 version: %s" % str(rs.__version__))
625 except Exception:
        # fail silently
        pass

    progress("INFO: Starting Vehicle communications")
630 conn = mavutil.mavlink_connection(
        connection_string,
        autoreconnect = True,
        source_system = 1,
        source_component = 93,
635     baud=connection_baudrate,
        force_connected=True,
    )

    #udp_conn = mavutil.mavlink_connection('udpout:192.168.1.73:15667', source_system ↩
        =1, source_component=1)
640 #udp_conn = mavutil.mavlink_connection('udpout:10.42.0.1:15667', source_system=1, ↩
        source_component=1)
    udp_conn = mavutil.mavlink_connection('udpout:192.168.166.94:15667', source_system ↩
        =1, source_component=1)

    mavlink_callbacks = [att_msg_callback, vc.update_mavlink_msg]

645 rover_thread = threading.Thread(target=mavlink_loop, args=(conn, mavlink_callbacks ↩
        , udp_conn))
    rover_thread.start()

    gcs_thread = threading.Thread(target=mavlink_loop, args=(udp_conn, [], conn))
    gcs_thread.start()
650
    # connecting and configuring the camera is a little hit-and-miss.
    # Start a timer and rely on a restart of the script to get it working.
    # Configuring the camera appears to block all threads, so we can't do
    # this internally.
655
    # send_msg_to_gcs('Setting timer...')
    signal.setitimer(signal.ITIMER_REAL, 5)  # seconds...
```

```python
send_msg_to_gcs('Connecting to camera...')
realsense_connect()
send_msg_to_gcs('Camera connected.')

signal.setitimer(signal.ITIMER_REAL, 0)  # cancel alarm

# Send MAVlink messages in the background at pre-determined frequencies
sched = BackgroundScheduler()

if enable_msg_vision_position_estimate:
    sched.add_job(send_vision_position_estimate_message, 'interval', seconds = 1/vision_position_estimate_msg_hz)

if enable_msg_vision_position_delta:
    sched.add_job(send_vision_position_delta_message, 'interval', seconds = 1/vision_position_delta_msg_hz)
    send_vision_position_delta_message.H_aeroRef_PrevAeroBody = tf.quaternion_matrix([1,0,0,0])
    send_vision_position_delta_message.prev_time_us = int(round(time.time() * 1000000))

if enable_msg_vision_speed_estimate:
    sched.add_job(send_vision_speed_estimate_message, 'interval', seconds = 1/vision_speed_estimate_msg_hz)

if enable_update_tracking_confidence_to_gcs:
    sched.add_job(update_tracking_confidence_to_gcs, 'interval', seconds = 1/update_tracking_confidence_to_gcs_hz_default)
    update_tracking_confidence_to_gcs.prev_confidence_level = -1

# A separate thread to monitor user input
if enable_user_keyboard_input:
    user_keyboard_input_thread = threading.Thread(target=user_input_monitor)
    user_keyboard_input_thread.daemon = True
    user_keyboard_input_thread.start()
    progress("INFO: Press Enter to set EKF home at default location")

sched.add_job(send_hb, 'interval', seconds = 1/10.0)

sched.start()

# gracefully terminate the script if an interrupt signal (e.g. ctrl-c)
# is received.  This is considered to be abnormal termination.
main_loop_should_quit = False
def sigint_handler(sig, frame):
    global main_loop_should_quit
    main_loop_should_quit = True
signal.signal(signal.SIGINT, sigint_handler)

# gracefully terminate the script if a terminate signal is received
# (e.g. kill -TERM).
def sigterm_handler(sig, frame):
    global main_loop_should_quit
    main_loop_should_quit = True
    global exit_code
    exit_code = 0

signal.signal(signal.SIGTERM, sigterm_handler)

if compass_enabled == 1:
    time.sleep(1) # Wait a short while for yaw to be correctly initiated

send_msg_to_gcs('Sending vision messages to FCU')
accel_logger =  mpu6050_logger.mpu6050_logger(log_file_name + "_acceleration")
video_writer = cv2.VideoWriter(log_file_name + ".avi", cv2.VideoWriter_fourcc(*'MJPG'),30, (412,300), 0)
```

```
    try:
720     # Configure the OpenCV stereo algorithm. See
        # https://docs.opencv.org/3.4/d2/d85/classcv_1_1StereoSGBM.html for a
        # description of the parameters
        window_size = 5
        min_disp = 16
725     # must be divisible by 16
        num_disp = 112 - min_disp
        max_disp = min_disp + num_disp
        stereo = cv2.StereoSGBM_create(minDisparity = min_disp,
                                       numDisparities = num_disp,
730                                    blockSize = 16,
                                       P1 = 8*3*window_size**2,
                                       P2 = 32*3*window_size**2,
                                       disp12MaxDiff = 1,
                                       uniquenessRatio = 10,
735                                    speckleWindowSize = 100,
                                       speckleRange = 32)

        # Retreive the stream and intrinsic properties for both cameras
        profiles = pipe.get_active_profile()
740
        streams = {"left"  : profiles.get_stream(rs.stream.fisheye, 1). ↩
            as_video_stream_profile(),
                   "right" : profiles.get_stream(rs.stream.fisheye, 2). ↩
                       as_video_stream_profile()}
        intrinsics = {"left"  : streams["left"].get_intrinsics(),
745                   "right" : streams["right"].get_intrinsics()}

        # Print information about both cameras
        print("INFO: Using stereo fisheye cameras")
        if debug_enable == 1:
            print("INFO: T265 Left camera:",  intrinsics["left"])
750         print("INFO: T265 Right camera:", intrinsics["right"])

        # Translate the intrinsics from librealsense into OpenCV
        K_left  = camera_matrix(intrinsics["left"])
        D_left  = fisheye_distortion(intrinsics["left"])
755     K_right = camera_matrix(intrinsics["right"])
        D_right = fisheye_distortion(intrinsics["right"])
        (width, height) = (intrinsics["left"].width, intrinsics["left"].height)

        # Get the relative extrinsics between the left and right camera
760     (R, T) = get_extrinsics(streams["left"], streams["right"])

        # We need to determine what focal length our undistorted images should have
        # in order to set up the camera matrices for initUndistortRectifyMap.  We
        # could use stereoRectify, but here we show how to derive these projection
765     # matrices from the calibration and a desired height and field of view

        # We calculate the undistorted focal length:
        #
        #         h
770     # -------------------
        #  \       |       /
        #    \     |  f  /
        #     \    |    /
        #      \  fov  /
775     #        \|/
        stereo_fov_rad = 90 * (m.pi/180)    # desired fov degree, 90 seems to work ok
        stereo_height_px = 300              # 300x300 pixel stereo output
        stereo_focal_px = stereo_height_px/2 / m.tan(stereo_fov_rad/2)

780     # We set the left rotation to identity and the right rotation
        # the rotation between the cameras
        R_left = np.eye(3)
        R_right = R
```

```
785     # The stereo algorithm needs max_disp extra pixels in order to produce valid
        # disparity on the desired output region. This changes the width, but the
        # center of projection should be on the center of the cropped image
        stereo_width_px = stereo_height_px + max_disp
        stereo_size = (stereo_width_px, stereo_height_px)
790     stereo_cx = (stereo_height_px - 1)/2 + max_disp
        stereo_cy = (stereo_height_px - 1)/2

        # Construct the left and right projection matrices, the only difference is
        # that the right projection matrix should have a shift along the x axis of
795     # baseline * focal_length
        P_left = np.array([[stereo_focal_px, 0, stereo_cx, 0],
                           [0, stereo_focal_px, stereo_cy, 0],
                           [0,               0,         1, 0]])
        P_right = P_left.copy()
800     P_right[0][3] = T[0] * stereo_focal_px

        # Construct Q for use with cv2.reprojectImageTo3D. Subtract max_disp from x
        # since we will crop the disparity later
        Q = np.array([[1, 0,       0, -(stereo_cx - max_disp)],
805                   [0, 1,       0, -stereo_cy],
                      [0, 0,       0, stereo_focal_px],
                      [0, 0, -1/T[0], 0]])

        # Create an undistortion map for the left and right camera which applies the
810     # rectification and undoes the camera distortion. This only has to be done
        # once
        m1type = cv2.CV_32FC1
        (lm1, lm2) = cv2.fisheye.initUndistortRectifyMap(K_left, D_left, R_left, ↩
            P_left, stereo_size, m1type)
        (rm1, rm2) = cv2.fisheye.initUndistortRectifyMap(K_right, D_right, R_right, ↩
            P_right, stereo_size, m1type)
815     undistort_rectify = {"left"  : (lm1, lm2),
                             "right" : (rm1, rm2)}

        # For AprilTag detection
        camera_params = [stereo_focal_px, stereo_focal_px, stereo_cx, stereo_cy]
820     loop_time = time.time()
        vc.start()

        while not main_loop_should_quit:
            # Wait for the next set of frames from the camera
825         frames = pipe.wait_for_frames()

            # Fetch pose frame
            pose = frames.get_pose_frame()

830         # Process data
            if pose:
                with lock:
                    #print(str(time.time()-loop_time) + "s since last loop")
                    loop_time = time.time()
835
                    # Store the timestamp for MAVLink messages
                    current_time_us = int(round(time.time() * 1000000))

                    # Pose data consists of translation and rotation
840                 data = pose.get_pose_data()

                    # Confidence level value from T265: 0-3, remapped to 0 - 100: 0% - ↩
                        Failed / 33.3% - Low / 66.6% - Medium / 100% - High
                    current_confidence_level = float(data.tracker_confidence * 100 / ↩
                        3)

845                 # In transformations, Quaternions w+ix+jy+kz are represented as [w ↩
                        , x, y, z]!
                    H_T265Ref_T265body = tf.quaternion_matrix([data.rotation.w, data. ↩
                        rotation.x, data.rotation.y, data.rotation.z])
```

```python
                H_T265Ref_T265body[0][3] = data.translation.x * scale_factor
                H_T265Ref_T265body[1][3] = data.translation.y * scale_factor
                H_T265Ref_T265body[2][3] = data.translation.z * scale_factor

                # Transform to aeronautic coordinates (body AND reference frame!)
                H_aeroRef_aeroBody = H_aeroRef_T265Ref.dot( H_T265Ref_T265body.dot( H_T265body_aeroBody))

                # Calculate GLOBAL XYZ speed (speed from T265 is already GLOBAL)
                V_aeroRef_aeroBody = tf.quaternion_matrix([1,0,0,0])
                V_aeroRef_aeroBody[0][3] = data.velocity.x
                V_aeroRef_aeroBody[1][3] = data.velocity.y
                V_aeroRef_aeroBody[2][3] = data.velocity.z
                V_aeroRef_aeroBody = H_aeroRef_T265Ref.dot(V_aeroRef_aeroBody)

                # Check for pose jump and increment reset_counter
                if prev_data != None:
                    delta_translation = [data.translation.x - prev_data.translation.x, data.translation.y - prev_data.translation.y, data.translation.z - prev_data.translation.z]
                    delta_velocity = [data.velocity.x - prev_data.velocity.x, data.velocity.y - prev_data.velocity.y, data.velocity.z - prev_data.velocity.z]
                    position_displacement = np.linalg.norm(delta_translation)
                    speed_delta = np.linalg.norm(delta_velocity)

                    # Pose jump is indicated when position changes abruptly. The behavior is not well documented yet (as of librealsense 2.34.0)
                    jump_threshold = 0.1 # in meters, from trials and errors, should be relative to how frequent is the position data obtained (200Hz for the T265)
                    jump_speed_threshold = 20.0 # in m/s from trials and errors, should be relative to how frequent is the velocity data obtained (200Hz for the T265)
                    if (position_displacement > jump_threshold) or (speed_delta > jump_speed_threshold):
                        send_msg_to_gcs('VISO jump detected')
                        if position_displacement > jump_threshold:
                            progress("Position jumped by: %s" % position_displacement)
                        elif speed_delta > jump_speed_threshold:
                            progress("Speed jumped by: %s" % speed_delta)
                        increment_reset_counter()

                prev_data = data

                # Take offsets from body's center of gravity (or IMU) to camera's origin into account
                if body_offset_enabled == 1:
                    H_body_camera = tf.euler_matrix(0, 0, 0, 'sxyz')
                    H_body_camera[0][3] = body_offset_x
                    H_body_camera[1][3] = body_offset_y
                    H_body_camera[2][3] = body_offset_z
                    H_camera_body = np.linalg.inv(H_body_camera)
                    H_aeroRef_aeroBody = H_body_camera.dot(H_aeroRef_aeroBody.dot(H_camera_body))

                # Realign heading to face north using initial compass data
                if compass_enabled == 1:
                    H_aeroRef_aeroBody = H_aeroRef_aeroBody.dot( tf.euler_matrix(0, 0, heading_north_yaw, 'sxyz'))

                # Show debug messages here
                if debug_enable == 1:
                    os.system('clear') # This helps in displaying the messages to be more readable
```

```
                    progress("DEBUG: Raw RPY[deg]: {}".format( np.array( tf. ↩
                        euler_from_matrix( H_T265Ref_T265body, 'sxyz')) * 180 / m. ↩
                        pi))
                    progress("DEBUG: NED RPY[deg]: {}".format( np.array( tf. ↩
                        euler_from_matrix( H_aeroRef_aeroBody, 'sxyz')) * 180 / m. ↩
                        pi))
                    progress("DEBUG: Raw pos xyz : {}".format( np.array( [data. ↩
                        translation.x, data.translation.y, data.translation.z])))
900                 progress("DEBUG: NED pos xyz : {}".format( np.array( tf. ↩
                        translation_from_matrix( H_aeroRef_aeroBody))))

        # Fetch raw fisheye image frames
        f1 = frames.get_fisheye_frame(1).as_video_frame()
        left_data = np.asanyarray(f1.get_data())
905
        # Process image streams
        frame_copy = {"left" : left_data}

        # Undistort and crop the center of the frames
910     center_undistorted = {"left" : cv2.remap(src = frame_copy["left"],
                                      map1 = undistort_rectify["left"][0],
                                      map2 = undistort_rectify["left"][1],
                                      interpolation = cv2.INTER_LINEAR)}

915     # Run AprilTag detection algorithm on rectified image.
        # Params:
        #   tag_image_source for "left" or "right"
        #   tag_landing_size for actual size of the tag
        tags = at_detector.detect(center_undistorted[tag_image_source], True, ↩
            camera_params, tag_landing_size)
920     if tags != []:
            for tag in tags:
                # Check for the tag that we want to land on
                if tag.tag_id == tag_landing_id:
                    is_landing_tag_detected = True
925                 H_camera_tag = tf.euler_matrix(0, 0, 0, 'sxyz')
                    H_camera_tag[0][3] = tag.pose_t[2]
                    H_camera_tag[1][3] = tag.pose_t[0]
                    H_camera_tag[2][3] = tag.pose_t[1]
                    temp_arr = H_aeroRef_aeroBody
930                 temp_arr[0][3]= 0
                    temp_arr[1][3]= 0
                    temp_arr[2][3]= 0
                    rotated_pose = temp_arr.dot(H_camera_tag)
                    vc.update_target_position(rotated_pose[0][3], rotated_pose ↩
                        [1][3],rotated_pose[2][3])
935                 #vc.update_target_position(tag.pose_t[2], tag.pose_t[0],tag. ↩
                        pose_t[1])

                    #rotated_pose = H_camera_tag.dot(temp_arr)
                    #rotated_pose = np.linalg.inv(temp_arr).dot(H_camera_tag) # ↩
                        does nothing
                    #np.linalg.inv(
940                 #print(H_camera_tag)
                    #print(H_aeroRef_aeroBody)
                    #print(rotated_pose)

                    #print("INFO: Detected landing tag", str(tag.tag_id), " ↩
                        relative to camera at x:", tag.pose_t[2], " ", ↩
                        rotated_pose[0][3], ", y:", tag.pose_t[0], " " , ↩
                        rotated_pose[1][3], ", z:", tag.pose_t[1], " ", ↩
                        rotated_pose[2][3])
945     else:
            # print("INFO: No tag detected")
            is_landing_tag_detected = False
        # If enabled, display tag-detected image in a pop-up window, required a ↩
            monitor to be connected
        if True:
```

```
950             # Create color image from source
                tags_img = center_undistorted[tag_image_source]

                # For each detected tag, draw a bounding box and put the id of the tag ↩
                    in the center
                for tag in tags:
955                 # Setup bounding box
                    for idx in range(len(tag.corners)):
                        cv2.line(tags_img,
                                tuple(tag.corners[idx-1, :].astype(int)),
                                tuple(tag.corners[idx, :].astype(int)),
960                             thickness = 2,
                                color = (255, 0, 0))

                    # The text to be put in the image, here we simply put the id of  ↩
                        the detected tag
                    text = str(tag.tag_id)
965
                    # get boundary of this text
                    textsize = cv2.getTextSize(text, cv2.FONT_HERSHEY_SIMPLEX, 1, 2) ↩
                        [0]

                    # Put the text in the middle of the image
970                 cv2.putText(tags_img,
                                text,
                                org = (((tag.corners[0, 0] + tag.corners[2, 0] -  ↩
                                    textsize[0])/2).astype(int),
                                        ((tag.corners[0, 1] + tag.corners[2, 1] +  ↩
                                            textsize[1])/2).astype(int)),
                                fontFace = cv2.FONT_HERSHEY_SIMPLEX,
975                             fontScale = 0.5,
                                thickness = 2,
                                color = (255, 0, 0))

                # Display the image in a window
980             video_writer.write(tags_img)
                if visualization == 1:
                    cv2.imshow(WINDOW_TITLE, tags_img)

                    # Read keyboard input on the image window
985                 key = cv2.waitKey(1)
                    if key == ord('q') or cv2.getWindowProperty(WINDOW_TITLE, cv2. ↩
                        WND_PROP_VISIBLE) < 1:
                        break

except Exception as e:
990     progress(e)

except:
    send_msg_to_gcs('ERROR IN SCRIPT')
    progress("Unexpected error: %s" % sys.exc_info()[0])
995
finally:
    progress('Closing the script...')
    # start a timer in case stopping everything nicely doesn't work.
    signal.setitimer(signal.ITIMER_REAL, 5)  # seconds...
1000    pipe.stop()
    mavlink_thread_should_exit = True
    rover_thread.join()
    gcs_thread.join()
    conn.close()
1005    udp_conn.close()
    accel_logger.stop()
    video_writer.release()
    vc.stop()
    progress("INFO: Realsense pipeline and vehicle object closed.")
1010    sys.exit(exit_code)
```

B.3 Position Control

Listing B.3: This file contains the state machine for controlling the UAV position

```
import os
import threading
import numpy as np
import time
from enum import Enum
import queue


end_position_z = .43
end_position_down = .1
staging_position_z = 1.5
finished_z = 1.5
final_yaw = 0

class ControlState(Enum):
    WAITING_FOR_INIT = 1
    CENTERING_ON_TARGET = 2
    MAKING_CONTACT = 3
    ACQUIRING_SAMPLE = 4
    LEAVING_TARGET = 5
    FINISHED = 6

class VisionControl:
    last_mode = 0
    time_start_acquring = time.time()
    data = queue.Queue()
    last_control_state = ControlState.WAITING_FOR_INIT
    control_state = ControlState.WAITING_FOR_INIT
    lock = threading.Lock()

    def __init__(self, got_to_position_func, filename):
        self.shutdown = False
        self.got_to_position_func = got_to_position_func

        outfile = filename + '.csv'
        self.chunksize = 10
        self.csvfile = open(outfile, 'w+')
        self.csvfile.writelines("time,right,down,forward,yaw,des_r,des_d,des_f, ↩
            r_sp, d_sp, f_sp, des_y, mode, drone_mode \n".format("Time", "Xa","Ya" ↩
            ,"Za"))

        self.last_tar_update_time = 0
        self.last_yaw_update_time = 0
        self.pos = {'time':0, 'right':0, 'down':0, 'forward':0, 'mode':0, ' ↩
            drone_mode':0, 'des_r':0, 'des_d':0, 'des_f':0, 'r_sp':0, 'd_sp':0, ' ↩
            f_sp':0,  'yaw':0, 'des_y': final_yaw}
        self.yaw = 0

        self.writer_th = threading.Thread(target=self.writer)
        self.runner_th = threading.Thread(target=self.timeout)

    def start(self):
        print("INFO: Starting Vision Control")
        self.runner_th.start()
        self.writer_th.start()

    def stop(self):
        self.shutdown = True
        self.runner_th.join()
        self.writer_th.join()

    def update_mavlink_msg(self, msg):
```

```
        if msg.get_type() == "ATTITUDE":
            self.last_yaw_update_time = time.time()
            self.yaw = msg.yaw
            self.pos['yaw'] = self.yaw
        elif msg.get_type() == "HEARTBEAT":
            self.update_mode(msg.custom_mode)

    def update_target_position(self, north, east, down):
        self.last_tar_update_time = time.time()
        self.pos['right'] = east
        self.pos['down'] = down
        self.pos['forward'] = north
        self.runner()

    def update_mode(self, mode):
        if (time.time() - self.last_tar_update_time) < .5 and self.last_mode != 4 and mode == 4:
            print("Drone changing state to centring on target")
            self.control_state = ControlState.CENTERING_ON_TARGET
        elif mode != 4:
            if self.last_mode == 4:
                print("Drone no longer in guided")
            self.control_state = ControlState.WAITING_FOR_INIT
            self.send_desired_position(0,0,0, final_yaw,0,0,0)
        self.pos['drone_mode'] = mode
        self.last_mode = mode

    def send_desired_position(self, forward, right, down, yaw, f_sp, r_sp, d_sp):
        self.pos['des_r'] = right
        self.pos['des_d'] = down
        self.pos['des_f'] = forward
        self.pos['f_sp'] = f_sp
        self.pos['r_sp'] = r_sp
        self.pos['d_sp'] = d_sp
        self.pos['time'] = time.time()
        self.got_to_position_func(forward, right, down, yaw)
        self.data.put_nowait(self.pos)

    def waiting_for_init(self):
        self.send_desired_position(0,0,0, final_yaw,0,0,0)
        return False

    def centring_on_target(self):

        self.send_desired_position(self.pos['forward'] - staging_position_z,self.pos['right'],self.pos['down']+ end_position_down,final_yaw, staging_position_z, 0, end_position_down)
        if abs(self.pos['right']) < 0.05 and abs(self.pos['down'] + end_position_down) < 0.05:
            return True
        return False

    def making_contact(self):
        self.send_desired_position(self.pos['forward'] - end_position_z,self.pos['right'],self.pos['down']+ end_position_down,final_yaw, end_position_z, 0, end_position_down)
        if abs(self.pos['forward'] - 0.05) < end_position_z :
            self.time_start_acquring = time.time()
            return True
        return False

    def acquiring_sample(self):
        self.send_desired_position(self.pos['forward'] - end_position_z,self.pos['right'],self.pos['down']+ end_position_down,final_yaw, end_position_z, 0, end_position_down)
        if (time.time() - self.time_start_acquring) > 10:
            return True
        return False
```

```
120
    def leaving_target(self):
        self.send_desired_position(self.pos['forward'] - finished_z,self.pos['right'],self.pos['down']+ end_position_down,final_yaw, finished_z, 0, end_position_down)
        if self.pos['forward'] > (finished_z - 0.05):
            return True
125     return False

    def writer(self):
        while not self.shutdown:
            d = self.data.get()
130         self.csvfile.write(" {d1[time]} , {d1[right]} , {d1[down]} , {d1[forward]}, {d1[yaw]} , {d1[des_r]} , {d1[des_d]} , {d1[des_f]}, {d1[r_sp]} , {d1[d_sp]} , {d1[f_sp]}, {d1[des_y]}, {d1[mode]}, {d1[drone_mode]} \n".format(d1=d) )

    def timeout(self):
        while not self.shutdown:
            if (time.time() - self.last_tar_update_time) > .5:
135             if self.control_state != ControlState.WAITING_FOR_INIT:
                    print("[LOST TARGET] Resetting state")
                self.control_state = ControlState.WAITING_FOR_INIT
                self.runner()
            time.sleep(1.0/20)
140

    def runner(self):
        if self.last_control_state != self.control_state:
            print("Chaging States from: " + str(self.last_control_state) + " to: " + str(self.control_state) )
145         self.last_control_state = self.control_state
            self.pos['mode'] = self.control_state.value
        if self.control_state is ControlState.WAITING_FOR_INIT:
            #print("[LOOP] waiting for init")
            if self.waiting_for_init():
150             self.control_state = ControlState.CENTERING_ON_TARGET
            else:
                pass
        elif self.control_state == ControlState.CENTERING_ON_TARGET:
            #print("[LOOP] centring_state")
155         if self.centring_on_target():
                self.control_state = ControlState.MAKING_CONTACT
            else:
                pass
        elif self.control_state == ControlState.MAKING_CONTACT:
160         #print("[LOOP] making contact")
            if self.making_contact():
                self.control_state = ControlState.ACQUIRING_SAMPLE
            else:
                pass
165     elif self.control_state == ControlState.ACQUIRING_SAMPLE:
            #print("[LOOP] sampling")
            if self.acquiring_sample():
                self.control_state = ControlState.LEAVING_TARGET
            else:
170             pass
        elif self.control_state == ControlState.LEAVING_TARGET:
            #print("[LOOP] leaving")
            if self.leaving_target():
                self.control_state = ControlState.LEAVING_TARGET
175         else:
                pass
```

B.4 Accelerometer Logging

Listing B.4: This file logs data from the IMU

```
from board import SDA,SCL
from custom_imu_mpu6050 import MPU6050
import busio
import os
import threading
import numpy as np
import csv
import time

class mpu6050_logger:
    def __init__(self, filename):
        self.outfile = filename + '.csv'
        self.csvfile = open(self.outfile, 'w+')
        self.csvWriter = csv.writer(self.csvfile,delimiter=',')

        self.shutdown = False
        self.runner_th = threading.Thread(target=self.runner)
        self.saver_th = threading.Thread(target=self.saver)
        self.saver_th.start()
        self.runner_th.start()

    def stop(self):
        self.shutdown = True
        self.runner_th.join()
        self.saver_th.join()
        self.csvfile.close()
        print("closed all vibration threads")

    def saver(self):
        pass

    def runner(self):
        self.i2c = busio.I2C(SCL,SDA)
        self.IMU = MPU6050(self.i2c)

        print ("Identification 0x{:X} ".format(self.IMU.whoami))
        time_start = time.time()
        data = np.empty((2000, 4))
        count = 0
        while not self.shutdown:
            data[count] = self.IMU.get_accel_data_fast(g=True)
            time.sleep(1.0/1500)
            count = count + 1

            if (time.time() - time_start) > 1:
                self.csvWriter.writerows(data)
                time.sleep(1)
                data = np.empty((2000, 4))
                count = 0
                time_start = time.time()
```

B.5 Accelerometer Driver

Listing B.5: Low level driver code for the MPU6050

```
"""
# Creative Commons Zero v1.0 Universal
#
# Copyright (c) 2020 Romy Bompart
```

```
#
# Permission is hereby granted, free of charge, to any person obtaining a copy
# of this software and associated documentation files (the "Software"), to deal
# in the Software without restriction, including without limitation the rights
# to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
# copies of the Software, and to permit persons to whom the Software is
# furnished to do so, subject to the following conditions:
#
# The above copyright notice and this permission notice shall be included in
# all copies or substantial portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
# IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
# AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
# LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
# OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
# THE SOFTWARE.

================================================================================

    This class helps the user to use the MPU6050 from Invensense in a
    wide variety of platform listed in the Adafruit_Blinka repository:
    https://github.com/adafruit/Adafruit_Blinka/blob/master/src/board.py

    Raspberry pi, Beaglebone, Jetson Boards, etc.

    The code is inspired in the mpu6050-raspberrypi created by Mr Tijn
    https://pypi.org/project/mpu6050-raspberrypi/

    And uses the libraries from Adafruit to make this library more robust and
        reliable
================================================================================


* Author(s): Romy Bompart
"""
__version__ = "v0.0"
__repo__ = "git@github.com:romybompart/py_imu_mpu6050.git"


from adafruit_register.i2c_struct import UnaryStruct, ROUnaryStruct
from adafruit_bus_device.i2c_device import I2CDevice
import time

class MPU6050:
    """
    Init the MPU chip at ``address`` on ``i2c_bus``

    :param i2c object bus   -> i2c_bus: The i2c bus to use for the communication
    :param int type      -> address: The MPU I2C address
    """

    """
    Class General Variables
    """
    GRAVITIY_MS2 = 9.80665

    #Registers:
    #General Registers
    PWR_MGMT_1   = UnaryStruct(0x6B, ">B")   # 0x6B
    WHO_AM_I     = ROUnaryStruct(0x75, ">B") # 0X75
    FSYC_DLP_CONFIG = UnaryStruct(0x1A,">B")
    #Temp Sensor Registers
    TEMP_OUTH    = ROUnaryStruct(0x41, ">b") # 0x41
    TEMP_OUTL    = ROUnaryStruct(0x42, ">b") # 0x41
    #Accelerometer Registers
    ACCEL_CONFIG = UnaryStruct(0x1C, ">B")  # 0x1C
```

```
    ACCEL_XOUTH = ROUnaryStruct(0x3B, ">B") # 0x3B
    ACCEL_XOUTL = ROUnaryStruct(0x3C, ">B") # 0x3B
    ACCEL_YOUTH = ROUnaryStruct(0x3D, ">B") # 0x3D
    ACCEL_YOUTL = ROUnaryStruct(0x3E, ">B") # 0x3D
    ACCEL_ZOUTH = ROUnaryStruct(0x3F, ">B") # 0x3F
    ACCEL_ZOUTL = ROUnaryStruct(0x40, ">B") # 0x3F

    #Gyroscope Registers
    GYRO_CONFIG = UnaryStruct(0x1B, ">B")   # 0x1B
    GYRO_XOUTH = ROUnaryStruct(0x43, ">B")  # 0x43
    GYRO_XOUTL = ROUnaryStruct(0x44, ">B")  # 0x44
    GYRO_YOUTH = ROUnaryStruct(0x45, ">B")  # 0x45
    GYRO_YOUTL = ROUnaryStruct(0x46, ">B")  # 0x46
    GYRO_ZOUTH = ROUnaryStruct(0x47, ">B")  # 0x47
    GYRO_ZOUTL = ROUnaryStruct(0x48, ">B")  # 0x48

    #Full scale Range ACCEL:
    """ from register 1C bit 4 to 3"""
    ACCEL_RANGE_2G = 0x00
    ACCEL_RANGE_4G = 0x08
    ACCEL_RANGE_8G = 0x10
    ACCEL_RANGE_16G = 0x18

    # LSB Sensitivity
    ACCEL_LSB_SENS_2G = 16384.0
    ACCEL_LSB_SENS_4G = 8192.0
    ACCEL_LSB_SENS_8G = 4096.0
    ACCEL_LSB_SENS_16G = 2048.0

    #Full scale Range GYRO:
    """ from register 1B bit 4 to 3"""
    GYRO_RANGE_250DEG = 0x00
    GYRO_RANGE_500DEG = 0x08
    GYRO_RANGE_1000DEG = 0x10
    GYRO_RANGE_2000DEG = 0x18

    #LSB Sensitivity
    GYRO_LSB_SENS_250DEG = 131.0
    GYRO_LSB_SENS_500DEG = 65.5
    GYRO_LSB_SENS_1000DEG = 32.8
    GYRO_LSB_SENS_2000DEG = 16.4

    #MPU Address
    MPU_ADDRESS = 0x68

    """
    Constructor
    """
    def __init__(self, i2c_bus=None, address=MPU_ADDRESS):

        """ Create an i2c device from the MPU6050"""
        self.i2c = i2c_bus

        if self.i2c == None:
            import busio
            from board import SDA,SCL
            self.i2c = busio.I2C(SCL,SDA)

        self.i2c_device = I2CDevice(self.i2c,address)

        """ Wake up the MPU-6050 since it starts in sleep mode """
        self.wakeup()
        print ('MPU already awaked')
        """ verify the accel range and get the accel scale modifier"""
        print ('accelerometer range set: {} g'.format(self.read_accel_range()))
        self.accel_scale_modifier = self.get_accel_scale_modifier()
        """ verify the gyro range and get the gyro scale modifier"""
```

```
/\*     print ( 'gyroscope range set: {} deg per s'.format(self.read_gyro_range()) ↩
    )
        self.gyro_scale_modifier = self.get_gyro_scale_modifier()
        """ configuring the Digital Low Pass Filter """
        #by default:
        # Bandwith of 21Hz and delay of 8.5ms, Sampling Freq 1KHz -> Accelerometer
        # Bandwith of 20Hz and delay of 8.3ms, Sampling Freq 1KHz -> Gyroscope
        self.filter_sensor = 0x04
        print ('digital filter configure to be: {}'.format(self.filter_sensor))

    """
        Class Properties
    """
    @property
    def whoami(self):
        """ MPU6050 I2C Address """
        return self.WHO_AM_I

    @property
    def get_temp(self):
        """ MPU6050 Temperature """
        H = self.TEMP_OUTH
        L = self.TEMP_OUTL

        raw_temp = self.raw_data_format( ( H << 8 )+ L )

        # Get the actual temperature using the formule given in the
        # MPU-6050 Register Map and Descriptions revision 4.2, page 30
        actual_temp = (raw_temp / 340.0) + 36.53
        return actual_temp

    @property
    def filter_sensor(self):
        """
        Return the filter sensor
        """
        return self.FSYC_DLP_CONFIG


    """ Class Setter """
    @filter_sensor.setter
    def filter_sensor(self, filter_value = 0x04):
        """
        Configuration Register

        In case FYNC pin is used, the bit of the sampling will be reported at
        any of the following registers. For more information consult MPU-6000- ↩
            Register
        Map at page 13.

        EXT_SYNC_SET, bit 5:3, Values:
        000: Input Disable
        001: Temp_Out_L
        010: GYRO_XOUT_L[0]
        011: GYRO_YOUT_L[0]
        100: GYRO_ZOUT_L[0]
        101: ACCEL_XOUT_L[0]
        110: ACCEL_YOUT_L[0]
        111: ACCEL_ZOUT_L[0]

        In case to set accelerometer and gyroscope are filtered according to the ↩
            following
        table.
        DLPF_CFG, bit 2:0, Values:
                    accelerometer                   gyroscope
                Bandwidth   Delays  Fs
        000:    260 Hz      0   ms  1KHz    256 Hz      0.98ms  8KHz
        001:    184 Hz      2   ms  1KHz    188 Hz      1.9 ms  1KHz
```

```
        010:    94  Hz      3   ms  1KHz    98  Hz      2.8 ms  1KHz
        011:    44  Hz      4.9 ms  1KHz    42  Hz      4.8 ms  1KHz
        100:    21  Hz      8.5 ms  1KHz    20  Hz      8.3 ms  1KHz
205     101:    10  Hz      13.8ms  1KHz    10  Hz      13.4ms  1KHz
        110:    5   Hz      19  ms  1KHz    5   Hz      18.6ms  1KHz
        111:    RESERVED        RESERVED        RESERVED    RESERVED 8KHz
        """
        if ( filter_value < 0x00 and filter_value > 0x07):
210         filter_value = 0x04

        self.FSYC_DLP_CONFIG = filter_value

    """
215     function members
    """
    def wakeup(self):
        """ waking up the MPU−6050 by writing at the POWER MANAGEMENT REGISTER 1

220     Device_Reset, bit 7, values:
        0: Nothing
        1: the device will reset all internal registers to their default values.

        Sleep, bit 6, values:
225     0: disables the sleep mode
        1: enables the sleep mode

        Cycle, bit 5 values:
        0: Nothing
230     1: The device will cycle between sleep mode and waking up to take a single ↩
             sample of data
        from active sensors at a rate determined by LP_WAKE_CTRL (register 108)

        Reserverd bit 4, value = 0

235     Temp_dis, bit 3, values
        0: Nothing
        1: Disables the temperature sensor

        ClkSel, bit 2:0, values:
240     000: Internal 8MHz oscillator
        001: PLL with X axis gyroscope reference
        010: PLL with y axis gyroscope reference
        011: PLL with z axis gyroscope reference
        100: PLL with external 32.768kHz reference
245     101: PLL with external 19.2MHz reference
        110: Reserverd
        111: Stops the clock and keeps the timing generator in reset

        """
250     self.PWR_MGMT_1 = 0x00  # BINARY 01001111

    def sleep(self):
        """ entenring into sleep mode
        Deactivate the internal clock generator and enter into sleep mode
255     """
        self.PWR_MGMT_1 = 0x4F  # BINARY 01001111

    def deinit(self):
        """ stop using the MPU−6050 """
260     self.sleep()

    def raw_data_format(self, raw_data):
        """ formating data that comes from the I2C bus"""
        """ This helps to provide the results between −1 and 1 along with the ↩
            accel or gyro modifier"""
265     if (raw_data >= 0x8000):
            raw_data = −((65535 − raw_data) + 1)
        return raw_data
```

```
    """ Accelerometer """
270 def read_accel_range(self, raw = False):
        """Reads the range the accelerometer is set to.

        If raw is True, it will return the raw value from the ACCEL_CONFIG
        register
275     If raw is False, it will return an integer: -1, 2, 4, 8 or 16. When it
        returns -1 something went wrong.
        """
        raw_data = self.ACCEL_CONFIG

280     if raw is True:
            return raw_data
        elif raw is False:
            if raw_data == self.ACCEL_RANGE_2G:
                return 2
285         elif raw_data == self.ACCEL_RANGE_4G:
                return 4
            elif raw_data == self.ACCEL_RANGE_8G:
                return 8
            elif raw_data == self.ACCEL_RANGE_16G:
290             return 16
        else:
            return -1

    def set_accel_range(self, value):
295     """
        set accel range
        """
        cond = (self.ACCEL_RANGE_2G == value) or (self.ACCEL_RANGE_4G == value) \
               (self.ACCEL_RANGE_8G == value) or (self.ACCEL_RANGE_16G)
300
        if cond == False:
            value = self.ACCEL_RANGE_2G

        self.ACCEL_CONFIG = value
305     self.accel_scale_modifier = self.get_accel_scale_modifier()

    def get_accel_scale_modifier(self):

        accel_range = self.read_accel_range(True)
310
        if accel_range == self.ACCEL_RANGE_2G:
            accel_scale_modifier = self.ACCEL_LSB_SENS_2G
        elif accel_range == self.ACCEL_RANGE_4G:
            accel_scale_modifier = self.ACCEL_LSB_SENS_4G
315     elif accel_range == self.ACCEL_RANGE_8G:
            accel_scale_modifier = self.ACCEL_LSB_SENS_8G
        elif accel_range == self.ACCEL_RANGE_16G:
            accel_scale_modifier = self.ACCEL_LSB_SENS_16G
        else:
320         print("Unkown range - accel_scale_modifier set to self. ↩
                ACCEL_SCALE_MODIFIER_2G")
            accel_scale_modifier = self.ACCEL_LSB_SENS_2G

        return accel_scale_modifier

325 def get_accel_data_fast(self, g = False):
        """Gets and returns the X, Y and Z values from the accelerometer.
        If g is True, it will return the data in g
        If g is False, it will return the data in m/s^2
        Returns a dictionary with the measurement results.
330     """
        buf = bytearray(7)
        buf[0] = 0x3B
        self.i2c_device.write_then_readinto(buf, buf, out_end=1, in_start=1)
```

```python
        x = self.raw_data_format(( buf[1] << 8 ) + buf[2])
        y = self.raw_data_format(( buf[3] << 8 ) + buf[4])
        z = self.raw_data_format(( buf[5] << 8 ) + buf[6])

        x = x / self.accel_scale_modifier
        y = y / self.accel_scale_modifier
        z = z / self.accel_scale_modifier

        if g is False:
            x = x * self.GRAVITIY_MS2
            y = y * self.GRAVITIY_MS2
            z = z * self.GRAVITIY_MS2

        return [time.time(), x, y, z]

    def get_accel_data(self, g = False):
        """Gets and returns the X, Y and Z values from the accelerometer.
        If g is True, it will return the data in g
        If g is False, it will return the data in m/s^2
        Returns a dictionary with the measurement results.
        """
        XH = self.ACCEL_XOUTH
        XL = self.ACCEL_XOUTL
        YH = self.ACCEL_YOUTH
        YL = self.ACCEL_YOUTL
        ZH = self.ACCEL_ZOUTH
        ZL = self.ACCEL_ZOUTL

        x = self.raw_data_format(( XH << 8 ) + XL)
        y = self.raw_data_format(( YH << 8 ) + YL)
        z = self.raw_data_format(( ZH << 8 ) + ZL)

        x = x / self.accel_scale_modifier
        y = y / self.accel_scale_modifier
        z = z / self.accel_scale_modifier

        if g is False:
            x = x * self.GRAVITIY_MS2
            y = y * self.GRAVITIY_MS2
            z = z * self.GRAVITIY_MS2

        return {'x': x, 'y': y, 'z': z}

    """ Gyroscope """
    def read_gyro_range(self, raw = False):
        """Reads the range the gyroscope is set to.
        If raw is True, it will return the raw value from the GYRO_CONFIG
        register.
        If raw is False, it will return 250, 500, 1000, 2000 or -1. If the
        returned value is equal to -1 something went wrong.
        """
        raw_data = self.GYRO_CONFIG

        if raw is True:
            return raw_data
        elif raw is False:
            if raw_data == self.GYRO_RANGE_250DEG:
                return 250
            elif raw_data == self.GYRO_RANGE_500DEG:
                return 500
            elif raw_data == self.GYRO_RANGE_1000DEG:
                return 1000
            elif raw_data == self.GYRO_RANGE_2000DEG:
                return 2000
            else:
                return -1

    def set_gyro_range(self, value):
```

```
        """
        set gyro range
405     """
        cond = (self.GYRO_RANGE_250DEG == value) or (self.GYRO_RANGE_500DEG == ↩
            value) \
               (self.GYRO_RANGE_1000DEG == value) or (self.GYRO_RANGE_2000DEG)

        if cond == False:
410         value = self.GYRO_RANGE_250DEG

        self.GYRO_CONFIG = value
        self.gyro_scale_modifier = self.get_gyro_scale_modifier()

415 def get_gyro_scale_modifier(self):
        """
        Get gyro scale modifier from reading the gyro range
        """
        gyro_range = self.read_gyro_range(True)
420
        if gyro_range == self.GYRO_RANGE_250DEG:
            gyro_scale_modifier = self.GYRO_LSB_SENS_250DEG
        elif gyro_range == self.GYRO_RANGE_500DEG:
            gyro_scale_modifier = self.GYRO_LSB_SENS_500DEG
425     elif gyro_range == self.GYRO_RANGE_1000DEG:
            gyro_scale_modifier = self.GYRO_LSB_SENS_1000DEG
        elif gyro_range == self.GYRO_RANGE_2000DEG:
            gyro_scale_modifier = self.GYRO_LSB_SENS_2000DEG
        else:
430         print("Unkown range - gyro_scale_modifier set to self. ↩
                GYRO_LSB_SENS_250DEG")
            gyro_scale_modifier = self.GYRO_LSB_SENS_250DEG

        return gyro_scale_modifier

435 def get_gyro_data(self):
        """
        Gets and returns the X, Y and Z values from the gyroscope.
        """
        XH = self.GYRO_XOUTH
440     XL = self.GYRO_XOUTL
        YH = self.GYRO_YOUTH
        YL = self.GYRO_YOUTL
        ZH = self.GYRO_ZOUTH
        ZL = self.GYRO_ZOUTL
445
        x = self.raw_data_format(( XH << 8 ) + XL)
        y = self.raw_data_format(( YH << 8 ) + YL)
        z = self.raw_data_format(( ZH << 8 ) + ZL)

450     x = x / self.gyro_scale_modifier
        y = y / self.gyro_scale_modifier
        z = z / self.gyro_scale_modifier

        return {'x': x, 'y': y, 'z': z}
455
    """
        magic methos helps to make easy the use of with
    """
    def __enter__(self):
460     """ to make easy the use of with """
        return self

    def __exit__(self, exceptio_typ, exception_value, traceback):
        """ to make easy the use of with """
465     self.deinit()

""" __end__ """
```