TALLINN UNIVERSITY OF TECHNOLOGY
SCHOOL OF ENGINEERING
Department of Electrical Power Engineering and Mechatronics

DEVELOPMENT OF A MULTI-PLATFORM, PRECISION-LANDING SYSTEM FOR QUADCOPTERS

DROONIDE PLATVORMIÜLESE TÄPPIS-MAANDUMISSÜSTEEMI ARENDAMINE

MASTER THESIS

Üliõpilane:	Nsikak Akpan Iquaibom /nimi/
Üliõpilaskood:	184689MAHM
Juhendaja:	Mart Tamre, Prof. /nimi, amet/

Tallinn 2020

AUTHOR’S DECLARATION

Hereby I declare, that I have written this thesis independently.
No academic degree has been applied for based on this material. All works, major viewpoints and data of the other authors used in this thesis have been referenced.

25th May, 2020

Author: Nsikak Iquaibom, [illegible]
/signature /

Thesis is in accordance with terms and requirements

“.......” 2020

Supervisor: …........................
/signature/

Accepted for defence

“.......”....................2020 .

Chairman of theses defence commission: ....
/name and signature/

[1] *Non-exclusive Licence for Publication and Reproduction of Graduation Thesis is not valid during the validity period of restriction on access, except the university`s right to reproduce the thesis only for preservation purposes.*

[illegible] (*signature*)

25th May, 2020(*date*)

Department of Electrical Power Engineering and Mechatronics

THESIS TASK

Student: Nsikak Akpan Iquaibom, 184689MAHM (name, student code)

Study programme: MAHM, Mechatronics (code and title)

main speciality: Mechatronics

Supervisor(s): Professor, Mart Tamre, 620 3202 (position, name, phone)

Consultants: …………………………………………………………..(name, position)

……………………………………………………………………………… (company, phone, e-mail)

Thesis topic:

(in English) Development of a multi-platform precision-landing system

(in Estonian) Droonide platvormiülese täppis-maandumissüsteemi arendamine

Thesis main objectives:

1. Research available Precision landing technics and understand their limitations.
2. Implement a functional Precision-landing Algorithm.
3. Increase the reusability and scalability of the Precision-Landing Algorithm by using FlytOS.

Thesis tasks and time schedule:

No	Task description	Deadline
1.	Review of Related Papers	03/12/2019
2.	Study of Precision-Landing Algorithms	29/02/2020
3.	Study of foundational Drone programming	14/03/2020
4.	Study and implementation of Image-processing based landing	31/03/2020
5.	Integrating FlytOS and Image-processing control with life Demonstration	31/04/2020
6.	Compilation of Thesis Report	20/05/2020

Language: English **Deadline for submission of thesis:** 22nd May, 2020

Student: Nsikak Iquaibom [illegible] 25th May, 2020

/signature/

Supervisor: ………………… …………………….. “…….”…………………201….a

/signature/

Consultant: ………………… …………………….. “…….”…………………201….a

/signature/

Head of study programme: …………… ………………… “…….”…………………201.a

/signature/

CONTENTS

PREFACE

This Thesis work was initiated by Tallinn University of Technology (TalTech's) department of Electrical Power Engineering and Mechatronics lead by Professor Mart Tamre. The author carried out all research and studies in the laboratories and external facilities of the TalTech campus.

The author wishes to express gratitude to Prof. Tamre for emphasizing a practical-based learning in this Master's program, this hands-on approach has really boosted the authors' confidence towards career success.

This thesis seeks to address two primary issues, related to the autonomous control of UAV (Unmanned Aerial Vehicles) and Quadrotors in particular;
- It aims to make create and implement a generic precision-landing algorithm.
- It aims to increase the reusability and scalability of this algorithm by using multi-platform API (Application Programming Interface) developed by FlytOS [1].

It is expected that a ready-to use precision-landing system will be developed and implemented in TalTech campus as an output of this Master Thesis.

List of abbreviations and symbols

1. API- Application Programming Interface
2. GCP- Ground Control Point
3. GNSS- Global Navigation Satellite system
4. GPS- Global Positioning System
5. MPC – Model Predictive Control
6. PPK- Post-Processed Kinematics
7. PPP- Precise-Point Positioning
8. RTK- Real-Time Kinematics
9. UAV – Unmanned Aerial Vehicle
10. URI – Uniform Resource Identifier

1 INTRODUCTION

Over the past decade, research into the application of UAVs (popularly called drones) technologies have witnessed an exponential increase, this is because Drones have huge potential to increase productivity both at a consumer level and in key industrial applications.

At the consumer level, parcel delivery for 'last-mile' of goods delivery is a key application area, however the highest commercial value of drone applications are in industrial and enterprise settings. Drones are widely used in the civil industry to inspect infrastructure such as Bridges, roads, windmills and so on. They are also used by architects as well as artists for building 3D visual maps of large areas. There is a growing trend for warehouse logistics firms to employ drones for automated inventory management. Government and security agencies use UAVs for surveillance of territories and hazardous zones.

The list of applications could proceed infinitely but drones presently are not as ubiquitous as would be hoped, considering their potential. This is because of many technical, social and legal issues. In this study, focus is made on the technical issues that limit the application of drones, chief of which is precision-landing.

For instance, for parcel delivery to be practical as well as safe, it must be guaranteed that the drone will land on its designated landing pad 100 percent of the time within a small tolerance, and it must be able to do so in varying environmental situations including rain, snow, dust, mud and even smoke, achieving this level of performance technically can become quite challenging.

In particular, for this study, an application of UAVs for parcel deliveries is the focus. TalTech desires a drone parcel delivery infrastructure that involves the use of one or more of her available drones to deliver parcels from building to building within the campus. The environmental conditions vary from cold and snowy winters, rainy seasons and fair summers. It is necessary that a precision-landing solution targeted at parcel delivery within the Campus must be able to operate in these wide variety of environmental situations, if possible, and must also be safe and convenient for public use.

Thus, it is the Author's desire to actualize a low-cost, low-maintenance, and open-source, precise-landing system which is critical to accomplishing inter-campus deliveries. This, the Author believes, will improve the quality of life of its users and as well inspire creativity in the general public.

1.1 Thesis Focus and Expected Outcomes

As mentioned previously, much work has been done in the field of Autonomous control of UAVs, this Study however, will focus on two key issues that the Author wishes to investigate;

- **Create a generic precision-landing Algorithm**: Many researchers have achieved precision-landings using various techniques, one of the most impressive [2] achieved landing accuracies of ca. 10 cm using a custom control algorithm. The Author aims to create a generic algorithm based on visual-servoing that will be easily adapted on many multi-rotors.
- **Make the developed algorithm executable on the widest range of Multi-rotor platforms using FlytOS**: As will be later seen in the literature review section, many good solutions have been implemented for precision-landing. The problem with these highly specialized solutions that have unique hardware and software is that, they are difficult to maintain from a practical point-of-view.

The Author will make use of the recently developed *FlytOS*, which is a ROS [3] (Robot Operating System)-based Operating System (OS), to develop a precision-landing application because FlytOS has the widest compatibility with commercial autopilots. The intention is to make it easier to scale the developed precision-landing algorithm across different drones, and since it will be open-source the algorithm can be developed further instead of new researchers having to start from the beginning every time. The Author believes this will help improve the quality and speed of research in this field.

Thus, in summary, **the expected outcomes** of this Study are;

- Have landing tolerance (ca. ±20 cm).
- Be consistently accurate.
- Have low infrastructural set-up cost.
- Be easily scalable even on different Drone platforms.
- Be able to operate in a wide variety of environmental conditions.
- Land in ca. 46 seconds.

1.2 Thesis Structure

Here, is an overview of the present Study;

- Chapter 1 talks broadly about the potential of Drones and how this potential can be unlocked by easily-applicable precision-landing technology.

- Chapter 2 discusses the state-of-the-art of precision landing technologies for UAVs, and then discusses key features that an implemented solution should achieve.
- Chapter 3 will discuss details of how the solution is intended to be executed.
- Chapter 4 goes into details as to how the methology operates, the hardware and software employed, the infrastructure it relies on and how environmental factors affect results.
- Chapter 5 then presents an analysis of the study. It discusses the goal and compares it with actual outcomes. The performance of the implemented precision-landing technology is also discussed and compared with existing solutions, then conclusions are made. The chapter then closes by discussing further directions that this study can evolve into.

2 LITERATURE REVIEW AND BACKGROUND

This chapter discusses the problem of precision landing in more detail, then a review is made of the state-of-the-art in terms of the implemented solutions for this problem, their limitations are discussed, then some approaches are considered for implementation in this study.

2.1 Current understanding of the problem

Precise Landing is fundamental to many applications such as parcel delivery, automatic charging of UAVs and thus is of interest to many stakeholders. Many companies including Amazon, Google and UPS already have implemented landing solutions, in various contexts.

The main challenge is that landing solutions are very context-sensitive, for instance, landing in fair weather, with sunlight and mild wind, would require different approach when landing in night conditions or snowy conditions. Also, the type of UAV being used will affect the quality of the landing, generally, larger UAVs have better performance.

For this study however, the focus will be on the use of quadrotors, and the target is to develop a robust precise-landing system that can work in windy, snowy, night and day conditions, to be used for parcel delivery.

2.2 Existing solutions

- **Use of GNSS (Global Navigation Satellite System) derived solutions**: Precise landing has been attempted by applying RTK (Real-Time Kinematics) [4], PPK (Post-Processed Kinematic) [5], PPP (Precise Point Positioning) [6] and GCP (Ground Control Points) [7] to conventional GNSSs. With these technologies, accuracy of landing can be brought to about 20cm tolerances.

The limitations of using GNSS based technologies are:

(i) They are quite expensive,

(ii) Since it relies on radio waves, it can easily be jammed or be interfered with,

(iii) It is largely dependent on the drone platform and cannot be easily scaled.

- **Use of Infra-red (IR) Sensors**: IR transmitters and receivers installed on the landing platform and the drone respectively, have been proven to be quite effective for precise-landing applications (More details concerning IR technics are explained in the next secion).

The limitations of IR-based solutions are:

(i) It requires a power-source on the landing platform, which implies greater infrastructural cost.

(ii) It can be affected by ambient sunlight.

(iii) Cannot be easily applied to many drone platforms

- **Use of Image Processing**: This method uses a camera mounted on the drone to read ArUco markers (ArUco Marker - ArUco markers are a series of coded grey images deployed in a two-dimensional area [8]), and then uses this feedback to accurately guide the drone till it lands. It has the lowest infrastructural costs, it can work in the widest varieties of environmental conditions (assuming appropriate apparatus are used), and be scaled on all major drone platforms. This holds the greatest promise for this project.

Some limitations of this system however are:

(i) Demands very good-level of technical know-how to implement- Unlike GNSS and IR implementation, using image-processing is not a ‘plug-and-play’ solution and demands a lot of software development.

(ii) Image processing become affected by precipitation and when the landing platform is covered by dust, mud, snow, leaves or any kind of obstacle that can obstruct the view of the Drone camera.

2.3 Review of Pertinent Literature

Prominent literature that covers this area of research are highlighted and discussed in this section, all references can be accessed below.

Alvika Gautam, et al. [9], provided a broad perspective on the status of the landing control problem. This literature pointed out that technics for tackling the landing problem for UAVs can be classified broadly thus:

- **GPS-based Landing**: involves the use of control systems to modify the motion of UAVs based on GPS data.

- **Vision-based Landing (Infra-Red (IR) and image processing technics)**: involves the use of a camera to scan a reference point (IR beacon or Image) and use the information acquired to trigger a landing sequence.

- **Guidance-based Landing (Proportional and Pursuit Guidance)**: Alvika Gautam, et al. also explained that; Guidance refers to the determination of desired trajectory from vehicle's current location to a target, as well as direction, rotation and acceleration for following that trajectory.

The absence of literature that shows how precision landing can be achieved by GNSS systems alone, further buttress the fact that GNSS-based solutions are in most cases, not practical or reliable, however, A. Cesetti, et al. [10] showed that, in the case of unmanned helicopters GPS and INS (Inertial Navigation Sensors) systems are suitable for long range and low precision flights but fall short for precise and close proximity flights.

Guidance-based Landing also relies often on vision-based sensors, H. Bang, et al. [11] designed a guidance law for automatic landing of UAV using vision sensors for both fixed wing and rotary wing UAV. The method iteratively estimated a time-to-go until target intercept and modified the acceleration command based upon the revised time-to-go estimate. Thus, it is obvious at this point that Vision-based technics must be applied to achieved precise, reliable and consistent landings.

Jiri Janousek and Petr Marcon [12] compared GPS based landing with IR and Lidar assisted landing, and observed that IR and Lidar sensors provided consistent high accuracy and high precision landing, while GPS landing accuracy varied significantly.

Ephraim Nowak, et al. [13] executed an indoor IR-based precise landing system using a larger IR-beacon, to eliminate the need for an external Lidar to estimate the precise height of the UAV. This consisted their ‘plug-and-play’ solution for precise landing in GPS-denied environments, but this solution is highly platform dependent and was not exposed to external wind and ambient lighting conditions. Most other literature reviewed, show that the application of image processing to the problem of precise landing is the key to the solution, as discussed in the following paragraphs;

Mohammad Fattahi SANI, et al. [14] used an AR.Drone 2.0 to also execute precise landing in a GPS-denied environment. Their solution involved using a Kalman Filter to fuse the Inertial Sensor data of the Drone with the Vision data acquired from an onboard monocular camera and thus controlling and landing the drone smoothly and quickly with a landing accuracy of 3 cm. Another key component of their control algorithm is its ability to overcome the oscillation of the drone when it is above the landing target. This, they accomplished by their ‘movement-slicing’ method, which partitioned the moving time around the target into ‘moving’ and ‘halting’ time, by controlling the drone for this specific time-bursts, stability of the drone was achieved with significant landing speeds.

Phong Ha Nguyen, et al. [15] accomplished remote tracking and control of a multi-rotor also in GPS-denied environments. However, rather than using conventional landing Markers, such as ArUco, they developed a unique marker. This unique marker, they showed in their report, significantly out-performs state-of-the art object trackers in terms of both accuracy and processing time [15]. This method however requires undisclosed specifications needed for printing this marker properly.

Further work was done by Mohammad Fattahi Sani, et al. [16], where they used two cameras (one, forward facing, and one downward facing) and fused the visual data with the drone’s inertial data, similar to the trend followed by most researchers investigating this problem. This solution however achieved a low-cost solution that is effective over a broad range from a landing target, in an indoor-environment.

Most research work up to this point, highlight the use of fiducial markers such as ArUCo markers, vision sensors, control algorithms and filters to localize and control mobile robots. Ho Chuen Kam, et al. [17], however focused on improving the marker-method of localization of mobile robots using a linear Kalman Filter to compensate for real-world conditions when the marker is sometimes briefly obstructed, thus improving the robustness (noise insensitivity) of the control system.

A similar technique has also been applied to a fixed-wing UAV to smoothen the landing of the UAV [18]. It was shown that by lining-up ArUco markers on a landing strip and using a long-range camera, the UAV could derive its height and pose gradually and thus land in a generally smooth gradient.

Nuno Pessanha Santos, et al. [19] used ground-based control system instead of the onboard system used by most other researchers to guide a fixed-wing UAV to land on a moving ship. The advantage of this method is lower payload for the UAV, also, since GPS signals are vulnerable to jamming, a camera is installed on the ship to visually track the UAV. This research used a Particle Filter (PF) for pose estimation of the UAV and used an Unscented Kalman Filter (UKF) for temporal filtering, the results of this research proved precision levels that were considered appropriate for Automated landing.

Popular infrastructure such as autopilot and on-board computer as well as open-source software used for drone application development are highlighted by Hyunwoong Choi [20] and he describes how they were successfully employed to develop a vision-based guidance system using an RGB camera. In his guidance system, a way-point was defined at which the drone was to execute a given motion.

Jesse and Timothy [2] achieved landing precision of less than 10 cm using what they call Variable Degree of Freedom, Image-based visual servoing (VBOF IBVS). They employed a nested Aruco Marker, which was also back-lit with IR LEDs, and maintained said landing accuracies in both day and night conditions, with a landing speed of about 26 seconds.

The concept of automatic landing of Fixed-wing UAVs using a Stereo-camera and an orientation sensor was explored by Montika Sereewattana, et al. [21]. The focus of their work was to use the stereo-camera to create depth-map of four detachable markers on a landing strip and thus estimate the pose as well as other data required for proper landing of the UAV.

V. F. Vidal, et al. [22] were able to successfully land an EMI-resistant Quadrotor quite precisely on a 1.4 m square area of a simple landmark, using relatively simple control methods such as PID relatively quickly too (about 15 seconds). A Go-pro camera was used in this research, but the image had to be processed using standard python libraries in order to be useful for actuating the drone.

Also, M. F. Silva, et al. [23] proved that low-cost equipment and fundamental control algorithms such as PID where sufficient to control the angular dynamics of a UAV. The 'Saturation Constraints and Performance Technique (SCPT)' tuning method was used for this study.

A fundamentally different approach to precision landing of UAV's was proposed by E. I. Shirokova, et al. [24]. A system consisting of a light-weight microwave reader, attached to a drone and an array of microwave transponders on the landing platform (which also serves as a charging station) was studied and simulated. The Flight controller adjusted the pose of the Drone based on feedback from the transponders. This study suggested that such a system would have an accuracy of about 1 mm, however this has not been tested on hardware in real-life conditions.

Thien Hoang Nguyen, et al. [25] gave more insight about the retrieval process of a UAV after it has completed its autonomous mission, suggesting that even though vision-based navigation technologies are mature, a UAV may fail to target its landing pad if GPS error limits its field-of-view (FOV) from seeing the target. To overcome this, an Ultra-wide band (UWB) system was installed on the drone to ensure that it is led towards the target landing pad, within the FOV of the drone, then visual navigation takes over and concludes the process. The solution worked out quite effectively, however, it demanded the use of 2 on-board computers, 1 mini-lidar sensor and a RealSense Camera, to name just a few, which demand a significant amount of power, hence in-practice, this may not be a feasible solution.

An application of a UAV and UGV (Unmanned Ground Vehicle) was made by Ivan Kalinov, et al. [26]. The study involved the UAV making precision landings on the moving UGV. The UAV was equipped with a couple of sensors including a 2D-Lidar sensor as well as an Ultrasonic sensor and the data was fused together. The study was targeted towards automated inventory management in a warehouse.

Liu Shungui, et al. [27] proposed a precision landing concept for UAVs to be used for powerline inspection. This paper emphasized the use of an iterative algorithm to know the best pose estimation for the Drone as it approaches the powerline, and then, the use of virtual control points to improve the accuracy of this pose estimation. This concept was proven in simulation environments.

Siri Holthe Mathisen, et al. [28] used a model to achieve precision landing of a fixed-wing UAV when it was undergoing 'Deep-Stall Landing'- very steep descent. This type of landing is necessary for operational flexibility in areas without sufficient landing strip, and is quite difficult for a human pilot to execute appropriately. A non-linear model predictive control was used to accurately control the deep stall by first controlling the UAV altitude, then significantly lowering its speed, before finally controlling its path angle.

2.4 Research Problem Formulation

Based on the review of the literature above, the research problem for this thesis is sub-divided into two key components:

- **Creation of a generic Precision-landing Algorithm**: Timothy and Jesse [2] showed very impressive results for precision-landing in both day and night conditions, with tolerances of ca. 20 cm. The Author has observed that the problem with such algorithm as well as similar algorithms developed by many researchers is that those algorithms are highly context sensitive. This means that:

  - small changes in the infrastructure or model of the UAV will significantly affect the precision algorithm in the long-run.

In order to overcome this problem, the Author aims to develop a precision-algorithm which is less dependent on the type of Drone being used or a highly specialized model. This will allow the algorithm to be easily adopted on multiple Drone platforms.

- **Enabling the developed algorithm to easily replicated and scaled on Multiple Drone platforms by using FlytOS**: All the papers evaluated all developed their own custom solutions for the problem. However, for a pragmatic application of precision-landing, the Author believes that it is important that the developed algorithm should be able to scale on other drones, with few modifications.

FlytOS provides one such platform to develop custom applications for UAVs and it has the widest compatibility with commercial flight controllers.

These are two key issues that the Author seeks to address via this Master Thesis.

2.5 Justification of Chosen Approaches and summary

In this section, comparative analysis of some competitive methods related to this Study are made. First, the control methodology for the proposed precision-landing algorithm is discussed. Secondly, the use of Aruco markers for the development of the landing platform is discussed. Lastly, further reasons why the Author has chosen to develop this precision-landing application based on the API developed by FlytOS are discussed.

Majority of studies done with respect to Precision-landing as seen in section 2.3 showed the following key features:

- **Model-dependent Algorithms**: High-precision algorithms such as those in [14], [15], [16], [17], [18] and [19] used a specialized model of the multirotor UAV to accomplish their algorithms.

- **Drone-dependent Algorithms**: Most of the studies where either built on a custom drone with unique features such as IR and GNSS modules or they used the popular AR Drone which comes with a unique API for just that Drone.

These features while highly specialized and where able to achieve quite impressive results will be diffucult to use in a commercial setting because it will demand the use of exactly those Drones or the recreation of models for each specific Drone.
This kind of approach will create a huge maintainance overhead if the precision-algorithm is to be used in commercially.
In order to overcome the above context-based challenges, the Author proposes the use of the following simple guidance algorithm, briefly illustrated below;

- Locate landing marker; else remain hovering
- Move only in x-axis until error is minimum then,
- Move only in y-axis until error is minimum
- When x and y axes errors are minimum, Land.

The details of these steps are illustrated and elaborated upon in Chapter 5. The Author estimates that this algorithm while simple, will still bring the Drone to within 10 cm tolerances within 1 minute landing time.
The advantage will be the ease of application of the algorithm on the widest variety of platforms and thus it can be quickly developed upon.

The use of fiducial markers is a common practice for robot visual-servoing, the major fiducial markers used in Robotics are;

- Aruco Markers
- April Tags
- AR Tags

Aruco Markers were chosen for this study, the main reason being that it is readily supported on major Robotics development resources such as opencv and ROS by well developed libraries and many tutorials.
The Author aims to use just one Aruco Marker as a landing reference, this is sufficient to provide the pose and attitude of the UAV. This data can then be used as a reference to actuate the Drone. The Author choses to adopt this method, since it has been proven to work really well, and is easily applicable.

Finally, concerning the use of FlytOS, it must be noted that FlytOS is owned by Flytbase, which also produces FlytDock (their precision-landing solution) based on FlytOS. FlytDock however is quite expensive and thus inaccessible to the Author, thus the

Author decided to create an open-source application based on FlytOS that can be accessible to anyone who would want to investigate this problem in future.
The application of these key features should aid the Author to accomplish the afore-mentioned **objectives**;

- UAV landing-accuracies should be better than already implemented precision-landings (i.e. should be less than 20 cm tolerances)
- UAV landings should be consistently accurate within desired precision levels.
- The landing set-up should have low infrastructural cost.
- The developed control system should be easily scalable even on different Drone platforms.
- The precision-landing system should able to operate in a wide variety of environmental conditions, including day, night, rain, and smoke.
- UAV should land in at least as fast as already implemented precision-landings (i.e. should land in <= 2 minutes).

2.6 Overview of Software and Hardware Used

In this section, the Author elaborates on principle software and hardware used for this Study.

The following software was used;

- FlytOS

- ROS

The following are physical apparati used;

- Aruco-Marker based landing platform; This platform was also back-lit with LEDs.

- On-board flight controller; Raspberry pi 3 [29] computer

- Voltage-Regulator; DC-DC voltage converter to draw power from the on-board battery.

2.6.1 Overview of FlytOS

After going through the large amount of research work done as well as commercial deployments of Drone-deliveries and precision-landing, it is demonstratively obvious that precision-landing is not a new concept or mystery any more.

What is a challenge however, is deploying precision-landing algorithms in such a way that they can be applied on a wide variety of Drones without much hassles of fine-tuning technical parameters or recreating sophisticated mathematical models of the system for each Drone.

FlytOS provides an answer to this problem. FlytOS has a vision to become a standard language with which Drone Developers use to talk to each other. FlytOS provides APIs (FlytAPIs) through which communication can be made with the widest variety of Drones and Autopilots- i.e. ArduPilot, PX4, and DJI Drones and Autopilots

The FlytOS Architecture is illustrated below:

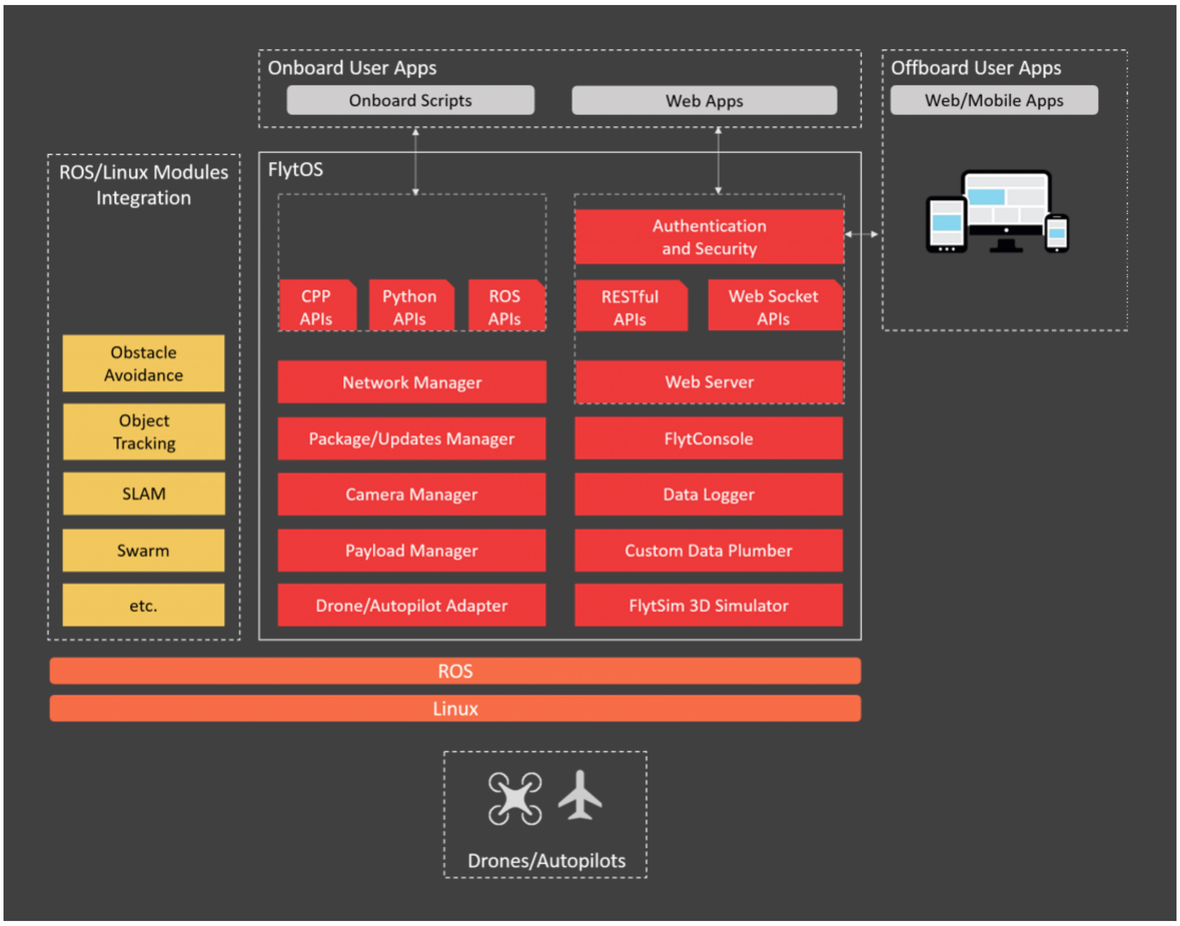


Figure 1: FlytOS Architecture Diagram [1]

Figure 1 captures the main features of FlytOS. At a glance we can observe key features which make it such a powerful development tool for Enterprise focused Drone-application developers, including the following;

- **Onboard APIs**- this provides an interface for Developers to create applications written in Python, C++, and with ROS which will run on an Onboard Computer carried along with the UAV in flight.

- **Web Application APIs**- FlytOS also provides RESTful and Web Socket APIs which are used in conjunction with Onboard applications as well as stand-alone applications (especially with smaller Drones).

  Most importantly, high-level functionality can be supervised from the FlytConsole, and useful troubleshooting data are stored in the Data Logger and can be downloaded when needed.

- **Peripheral Modules**- These include; Offboard User Apps and ROS/Linux Modules integration. These Modules enable Web/Mobile Apps to be created easily and expose advanced functionalities of ROS/Linux respectively.

Figure 1 also shows that FlytOS is built on ROS (Robot Operating System) and Linux [1]. ROS and Linux are both widely used for both Commercial and Research purposes, and thus form a proper foundation on which relevant applications can be built and maintained.

The Linux and ROS versions used together with FlytOS for this Study are:

- **ROS Version**: ROS-Kinetic 16.04, this is not the latest version of ROS but is still used widely in the robotics community because of its stability and support for many robotic tools libraries. ROS-Kinetic has long-term support until 2023 and thus is a robust development environment for future use.

- **Linux Version**: Ubuntu-MATE, this version is similar to the popular Desktop Ubuntu OS, but Ubuntu-MATE has a modified interface which makes it more suitable for operation on lower-capacity computers such as Raspberry Pi, (with less working-memory space) which are usually used as Onboard-computers to control a Drone.

In the remaining sections of this chapter, an overview will be given of all the hardware used for this Study.

2.6.2 Overview of Raspberry Pi

Raspberry Pi computers are widely popular because of their relatively large processing power, small size and affordability. More importantly, for this Study **Raspberry Pi 3,**

Model B was chosen because of its ready support for FlytOS and also a large online support community.

A detailed diagram showing major features of this controller can be seen in Figure 2, in the Appendices section.

Key features that make Raspberry Pi 3 Model B suitable for this Study are:

- **Easily accessible GPIO pinouts**- 6,8 and 10 (Ground, Transmitter pin and Receiver Pin, respectively) which are necessary for establishing UART (Universal Asynchronous Receiver-Transmitter) communication between the Drone and the Raspberry Pi.

- **CSI Camera Connector**- The Camera Serial Interface (CSI) provides a faster transmission rate for video data coming from a camera module attached to this interface, compared to the USB interfaces.

- **Ethernet Port**- This enables the board to be connected to LAN (Local Area Network) and achieve large data rates while FlytOS is being setup and updated on the Raspberry Pi.

- **Onboard BCM43438 wireless LAN and Bluetooth Low Energy (BLE)**- This enables the board to broadcast a wifi signal once it is powered on. A Computer can then connect to this Wifi signal and remotely connect to this Raspberry Pi conviently without an external module.

Peripheral Ports- These include the **HDMI** and **USB 2.0** ports which make it easy to physically connect a Monitor, Keyboard and Mouse to the board and access the GUI (Graphics User Interface) of Ubuntu-MATE, which makes program development easier and faster.

2.6.3 Overview of the Raspberry Pi-Camera (RPi Cam) Used

In order to enable the Drone to cover the largest expanse of space at a given time, a wide-angle Raspberry-Pi Camera was used- **RPi Camera Fisheye Lens**.

Diagram, illustrating how the RPi-cam was connected to the RPi is shown in Figure 3 in the Appendices section.

Key parameters that make this Camera suitable for this Study are:

- **160-degree angle of View**- This is more than double the size of regular RPi Cameras (about 72 degrees), thus giving the Drone a very wide field of view.

- **5 megapixel OV5647 sensor**- This makes the images appear sharper. Thus the image processing algorithm has better data to work with.

2.6.4 Overview of the Drone Used

For this Study, it was important to the Author to test out the precision-landing Algorithm in real-world conditions rather than in only simulation environments or indoor environments.

The **DJI Matrice 210** Drone was used for this Study. Key features of this Drone that were utilized in this Study are:

- **FlytOS support for this Model of Drone**: FlytOS officially supports DJI Matrice 200 series Drones from their 1.5-6 FlytOS release and upward. Documentation was provided by FlytOS on the procedures for setting up the Drone with FlytOS which will be explained in Chapter 5.

- **Easily accessible UART Port**: This easily accessible UART/Serial port makes data exchange between the Drone and the Onboard Computer (RPi) fast and easy to wire-up, as shown below:

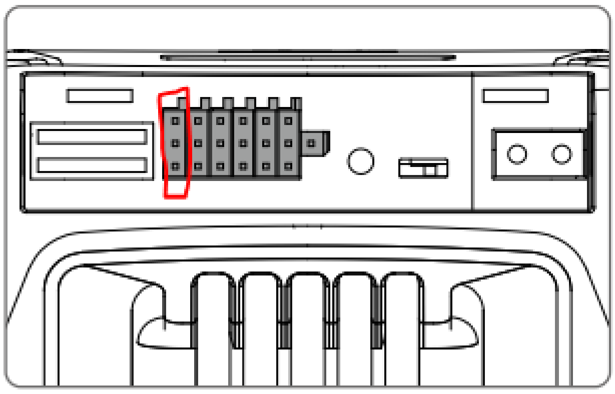


Figure 4: DJI M210 Expansion-Port, highlighting the OSDK (Onboard Software Development Kit) port. [30]

In Figure 4, the highlighted region, from top-to-bottom represents the Transmitter, Receiver and Ground for the UART connection.

- **Gimbal-Mounting Mechanism**: M210 has a quick release mechanism for Camera gimbals to be mounted and unmounted to the Drone, as shown below:

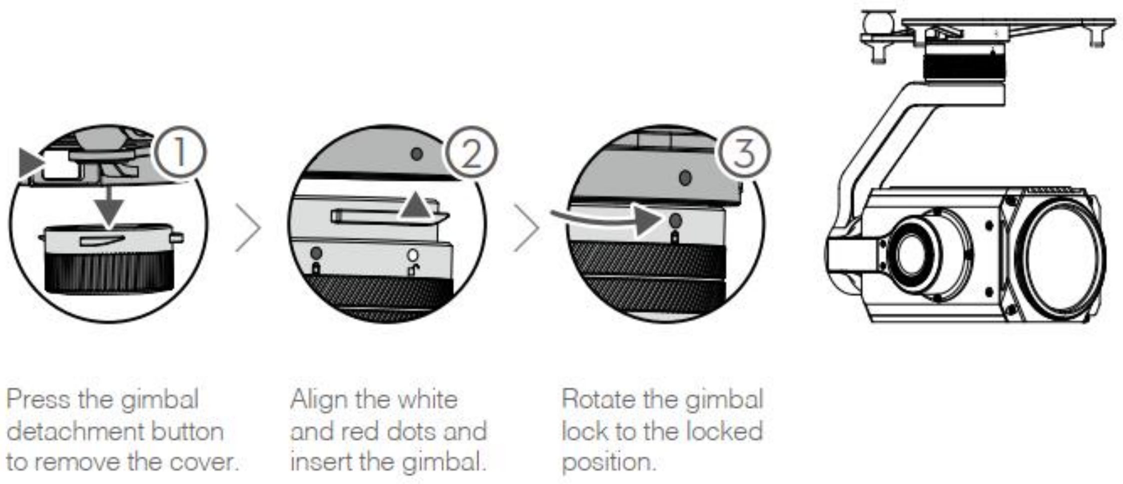

1
2
3
Press the gimbal detachment button to remove the cover.
Align the white and red dots and insert the gimbal.
Rotate the gimbal lock to the locked position.

Figure 5: M210 Gimbal-mounting mechanism [30]

This mounting mechanism was adopted by the Author to create a create a quick-release mechanism for the Onboard computer and Camera which enable the precision-landing algorithm, as will be illustrated in Chapter 4.

- **Extended Power Port**: This port is used by the Drone to supply power to external devices, it outputs voltage in the range of 18 V to 26 V with a current of 2 A [30].

2.6.5 Overview of the Voltage Regulator Used

In order to power-up the RPi from the Drone, it was necessary to step-down the Voltage coming from the Drone to values which are suitable for the RPi. The 18 V to 26 V signal coming from the Drone was stepped-down and regulated at 5 V with a current of 2.5 A using the **XL6009E1 voltage-regulator module**.

Pictures, which illustrate this voltage regulator are displayed in Figure 6, in the Appendices section.

From Figure 6, we can observe that this module simply receives an input signal and the regulator is used to adjust the voltage-level until the desired output voltage is achieved. The input wires are soldered on the input terminal and the output wires are soldered at the output terminals. Details of how this module was used are documented in Chapter 3.

Key features of this Module include:

- Input Voltage range: 3.5 V – 32 V DC,

- Output Voltage range: 5 V – 35 V DC,
- Maximum Current Output: 2.5 A [31]

3 HARDWARE SET-UP

In this Chapter, the hardware setup for this Study are discussed in detail. As discussed in Chapter 3, the hardware used are:

- One RaspberryPi Computer,
- One Fisheye lens RaspberryPi Camera,
- One Voltage regulator, and
- One DJI M210 Drone

The hardware components were setup according to the following flowchart:

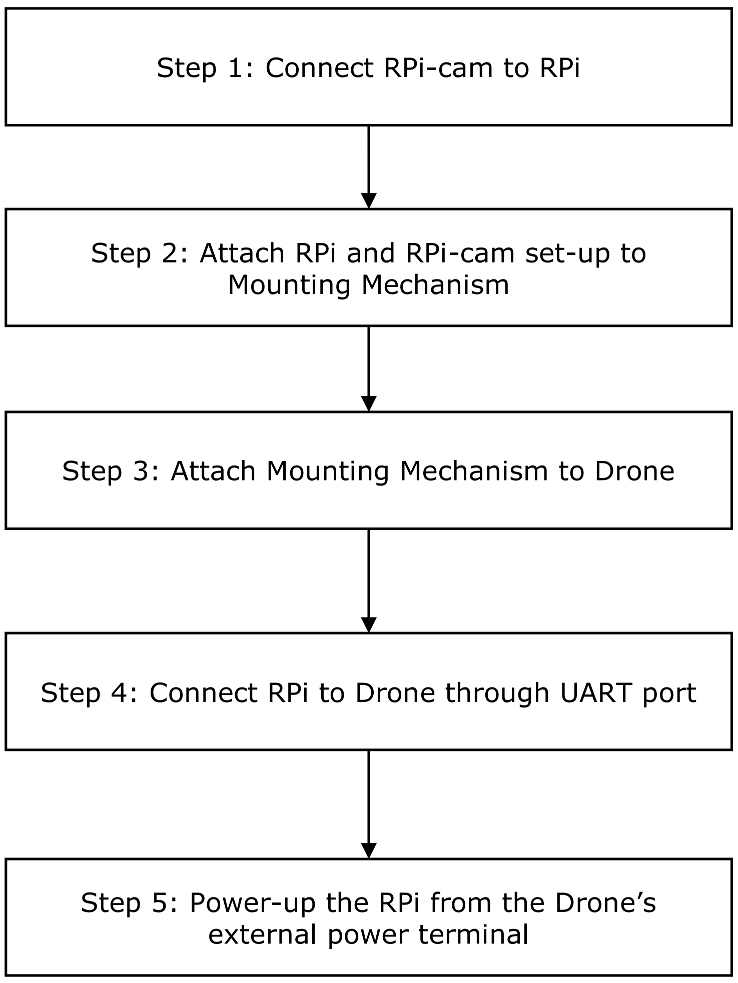

Step 1: Connect RPi-cam to RPi
Step 2: Attach RPi and RPi-cam set-up to Mounting Mechanism
Step 3: Attach Mounting Mechanism to Drone
Step 4: Connect RPi to Drone through UART port
Step 5: Power-up the RPi from the Drone's external power terminal

Figure 7: Flowchart showing the Hardware set-up sequence

From Figure 7, the hardware set-up sequence was actualized in the following way:

- **Step 1 and Step 2**:

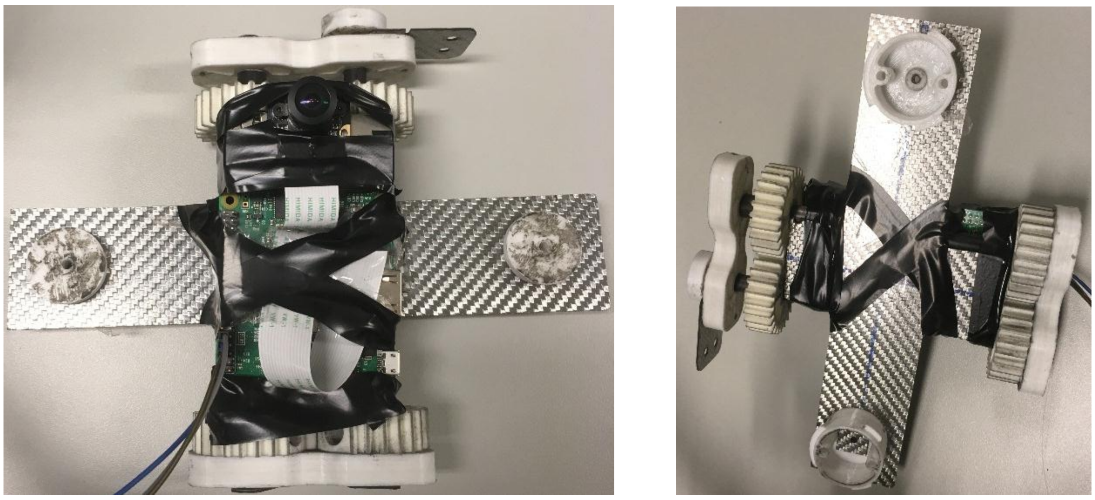


Figure 8: RPi and Camera set-up attached to Mounting Mechanism (Top and Bottom-View)

Figure 8 shows how the RPi and Camera were connected together (check Figure 3) and then attached (taped) to a Mounting mechanism made of carbon-fiber sheets.

- **Step 3**: Here the Mounting mechanism was attached to the Drone as shown below:

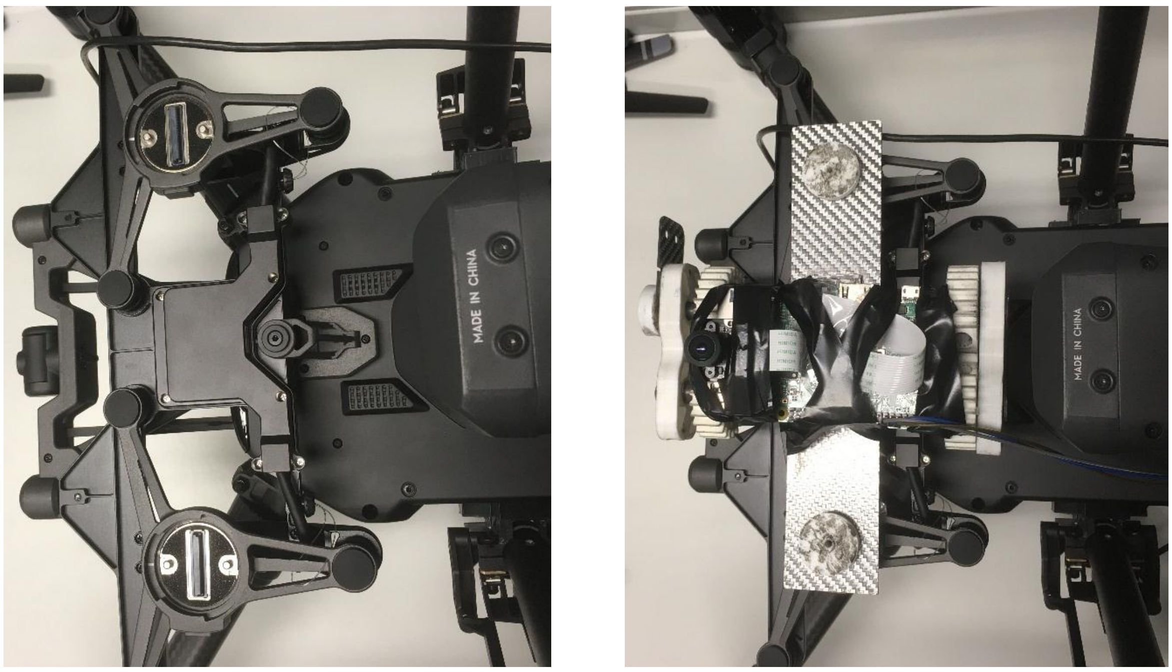
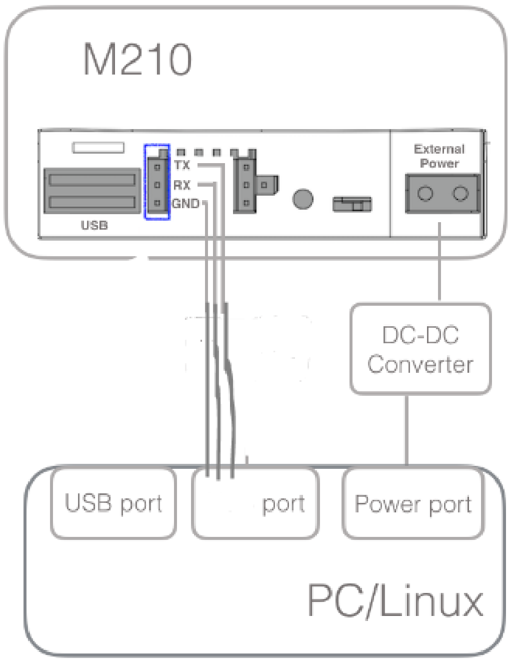


Figure 9: Attachment of Mounting Mechanism to Drone. (Before and After)

Figure 9 shows how the Mounting mechanism to attached to the Drone using the 2 slots which were meant to hold camera gimbals (see Figure 5).

- **Step 4**: Here the RPi is connected to the OSDK/UART port that enables the Onboard computer (RPi) to communicate with the Drone. The wiring for this is illustrated below:



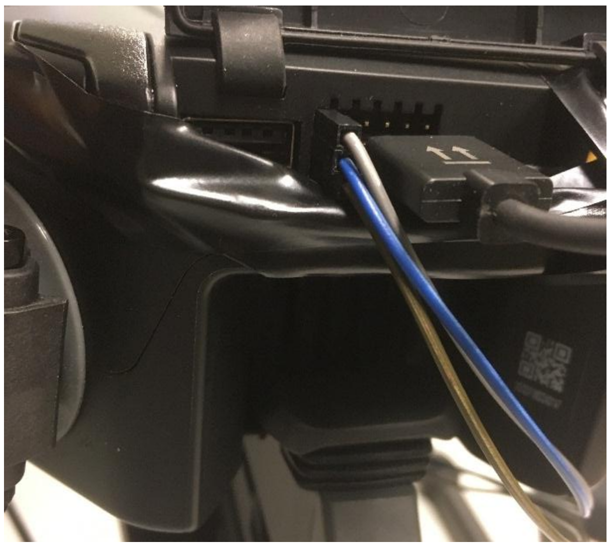


Figure 10: Connecting the RPi to the M210 Drone (Schematic [32] and Actual)

Figure 10 shows how the Onboard computer was connected to the OSDK port of the Drone. The following UART connections were made:

- The ground (GND) of the RPi was connected to the GND of the M210
- The transmitter (Tx) [GPIO14] of the RPi was connected to the Rx of the M210.
- The receiver (Rx) [GPIO15] of the RPi was connected to the Tx of the M210.

This is also illustrated above.

- **Step 5**: Finally, the RPi was powered-up from the Drone's external power-terminal through a DC-to-DC converter (see the schematic in Figure 10), as shown below:

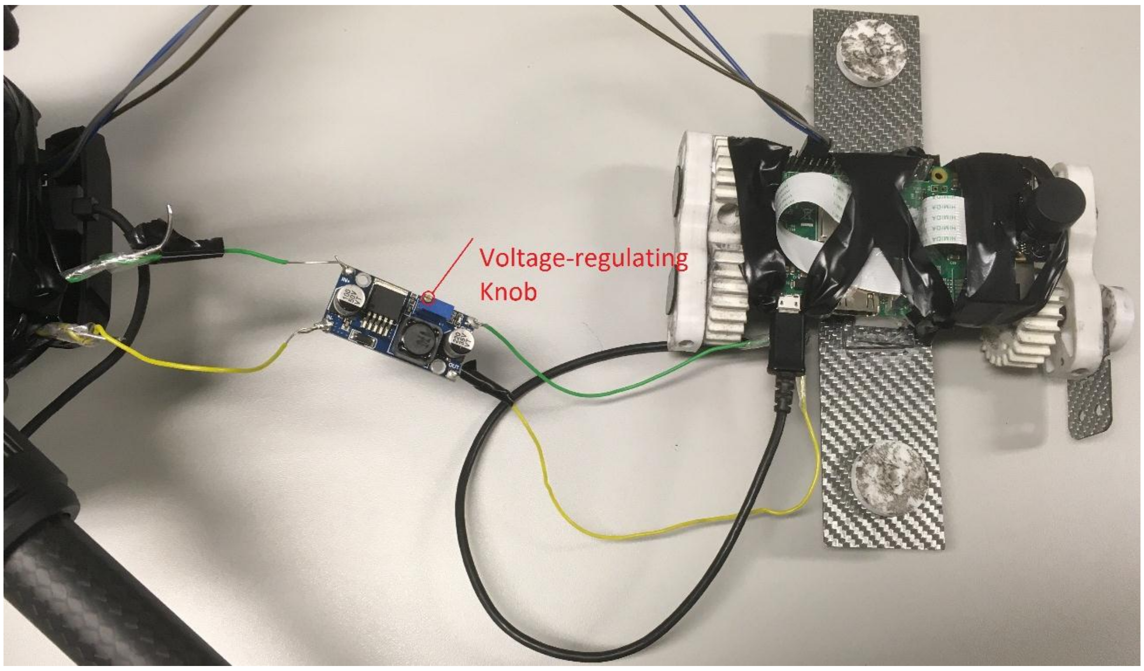


Figure 11: RPi Power Connections

In Figure 12, we observe how the RPi was connected to the Drone power outlet through a DC-to-DC converter. In order to achieve this set-up successfully, the following was done to ensure that the proper electric signal-values were attained:

- The polarity of the Drones' external power-port were confirmed using a voltmeter.
- Wires were soldered to both terminals of the voltage regulator and it was connected to the Drone.
- The input to the voltage-regulator was measured and confirmed to be ca. 26 V.
- The output of the voltage-regulator was measured with the voltmeter and adjusted with the voltage-regulating knob highlighted above.
- The previous step was repeated continuously until the desired output voltage of ca 5.2 V was attained. This setting also enabled the RPi to draw it rated current 2.5 A properly.

In the next chapter, details will be given on how the respective software modules were set-up in order to successfully carry-out this study.

4 SOFTWARE SET-UP

In this Chapter, the Author goes into details to explain how FlytOS was set-up for this Study and also explains how ROS was used to organize the modules of this Precision-landing Application.

The following key-points will be elaborated upon in this Chapter:

- Setting-up FlytOS with DJI M210
- Setting-up ROS for this Study

4.1 Setting up FlytOS with DJI M210

As mentioned previously, FlytOS started officially supporting DJI M200 series Drones from their 1.5-6 release and upward. In this section, the Author explain all the details necessary to set-up FlytOS with the Drone.

There is an official documentation on how to do this, but from experience, the Author has discovered that there are some extra steps that may need to be executed in order to achieve a successful set-up. Hence a holistic set-up procedure for installing FlytOS on a Raspberry Pi will be elaborated upon, highlighting the discrepancies in the procedure which are not covered in the official documentation. This is procedure is illustrated in the flow-chart below:

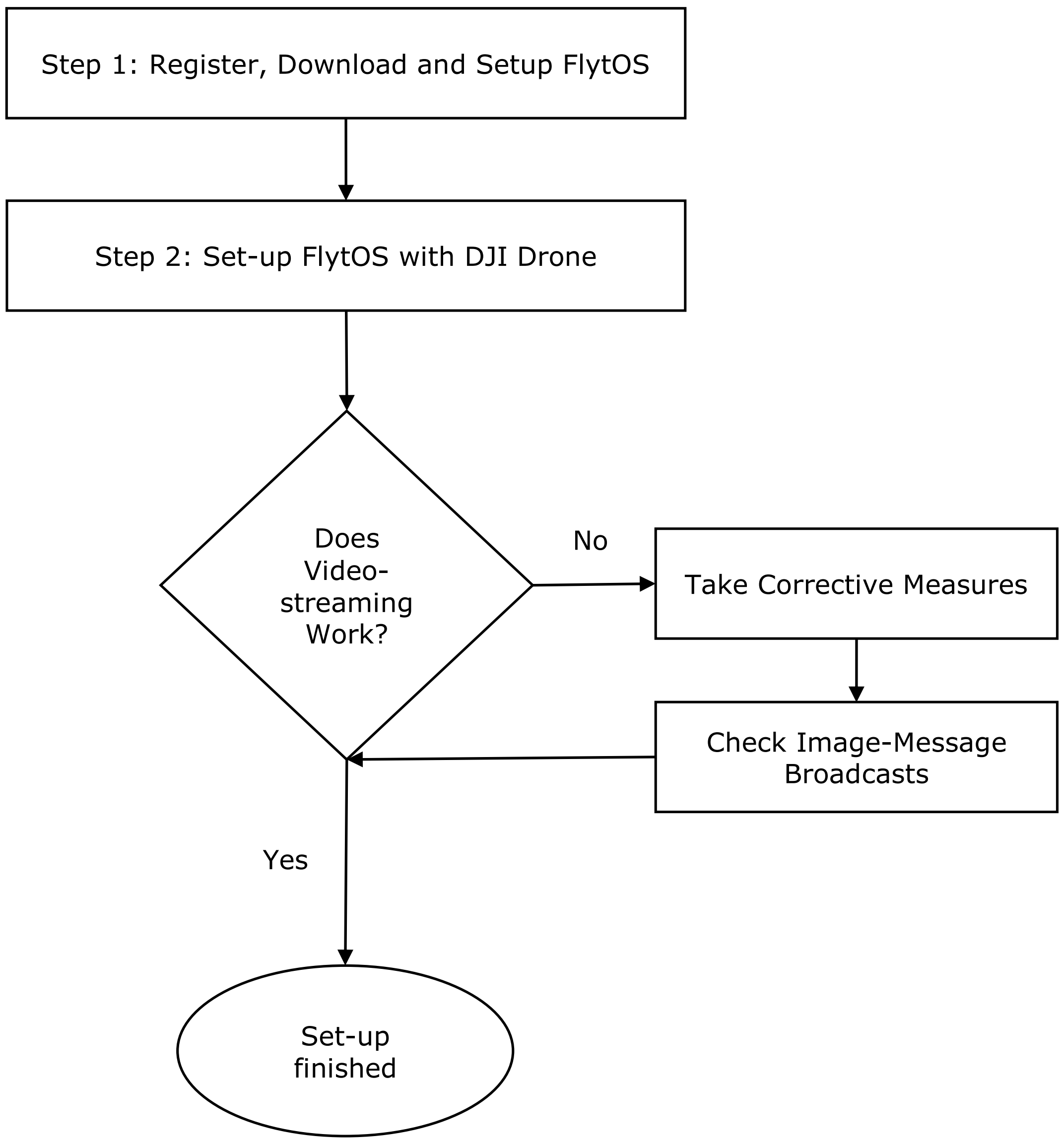

Step 1: Register, Download and Setup FlytOS
Step 2: Set-up FlytOS with DJI Drone
Does Video-streaming Work?
No
Take Corrective Measures
Check Image-Message Broadcasts
Yes
Set-up finished

Figure 12: Flowchart showing holistic FlytOS and Drone set-up procedures.

This procedure is exactly the same for both the Personal Edition (PE) or the Commercial Edition (CE).

4.1.1 Register, Download and Setup FlytOS with Raspberry Pi 3

There are a few methods for accomplishing this step, but from experience, the Author recommends the following steps (also the official recommendation).

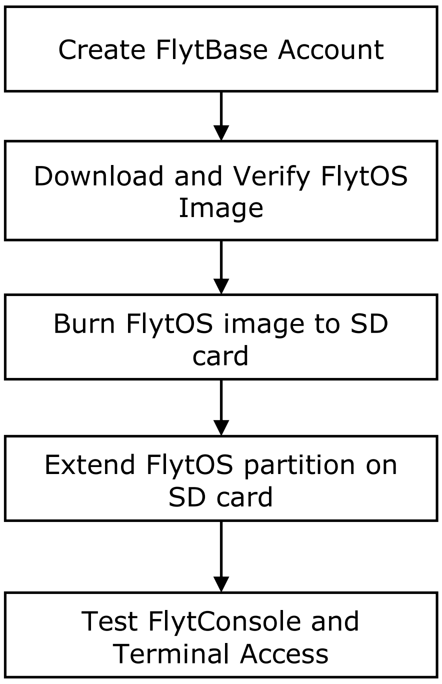

Create FlytBase Account
Download and Verify FlytOS Image
Burn FlytOS image to SD card
Extend FlytOS partition on SD card
Test FlytConsole and Terminal Access

Figure 13: Flowchart for setting up FlytOS with Raspberry Pi 3

In this stage, the user accomplishes the following:

- Creates a FlytBase account in order to access FlytOS.
- Downloads the FlytOS image from his/her account.
- Verifies the integrity of the image using a checksum provided from his/her account.
- Writes the verified FlytOS image to an SD card: The author recommends using a high quality SD card, with high data transfer rates, because this will significantly affect the speed of the setup procedure, as well as the overall performance of FlytOS.

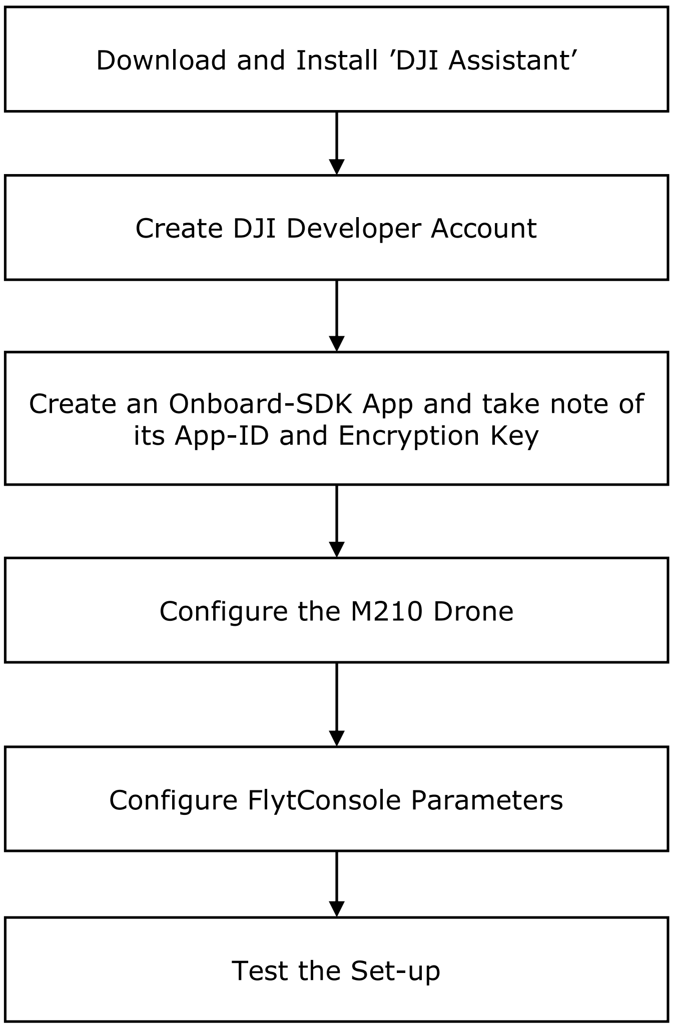
  The Author used ‘SanDisk 32GB Extreme UHS-I microSDHC’ Memory Card, for this Study.
- Extends the partitioning of FlytOS to cover the entire SD card.
- Acquires login-access to FlytConsole and the FlytOS terminal.

The fine-details of these processes can be seen in the official documentation [28].

4.1.2 Set-up FlytOS for a DJI-M200 Series Drone

As mentioned in the introductory chapter, the Vision of FlytOS is to be a common language upon which Drone-application developers execute their code. To this effect, FlytOS is compatible with the most popular Drones and Auto-Pilots- DJI Drones, ArduPilot and PX4.

For this Study, the Author used a DJI-M210 Drone provided by Tallinn University of Technology. The set-up procedure for this Drone are summarized in the flowchart below:



Download and Install 'DJI Assistant'
Create DJI Developer Account
Create an Onboard-SDK App and take note of its App-ID and Encryption Key
Configure the M210 Drone
Configure FlytConsole Parameters
Test the Set-up

Figure 14: Flowchart for setting-up FlytOS for a DJI M210 Drone

The user executes the following at this phase:

- **Installation of 'DJI Assistant'**: DJI Assistant (or DJI Assistant 2) is the primary software for configuring, troubleshooting and simulating DJI drones.

- **Creation of DJI Developer Account**: This is needed in order to have access to DJI's SDKs
- **Creation of an Onboard-SDK App**: Here, the user registers an Application on DJI's website which is intended to make use of their Onboard SDK. Special note must be taken of the App-ID and its Encryption Key, as it will be used later.
- **DJI M210 Configuration**- The following are done here:
    - key parameters of M210 are set in the DJI Assistant App to control the behaviour of the Drone while been used with FlytOS.
    - Drone firmware is updated to the latest and then restarted.
    - UART Connection is made between the RPi and the OSDK port of the Drone.
- **Configuring FlytConsole Parameters**: FlytConsole serves as a supervisory and control centre for FlytOS-controlled Drones/Autopilots, the following are done here:
    - FlytConsole is opened in a browser. (details in [34])
    - The App ID and Encryption key are populated into FlytOS settings field.
- **Testing the Set-up**: To confirm that everything has been done properly, the following should be done:
    - Start and Stop FlytOS from the terminal using the given commands, (details in [34]).
    - Make sure to remove the propellers from the Drone.
    - Ensure that there GPS signals available.
    - Start the simulator in DJI Assistant.
    - Set M210 to P-mode using the flip-switch on the controller.
    - Use the Joystick App in FlytConsole control the Drone in the simulator.

The official documentation for this procedure are available [34], but some points have not been updated by FlytBase, hence from experience the Author has modified the sequence so that it is easier for a novice to follow.

After the above steps, it is necessary to confirm if the images coming from a camera connected to the RPi are visible from FlytConsole, (the Author encountered a challenge with the video-streaming). If it is not, troubleshooting steps are elaborated in the next section.

4.1.3 Troubleshooting the Video-streaming Functionality

As this is an image-based precision-landing algorithm, it was necessary to ensure that video data from RPi were visible on to FlytOS and could be used for developing the algorithm. The following should be done if Video Streaming data is not visible on FlytConsole:

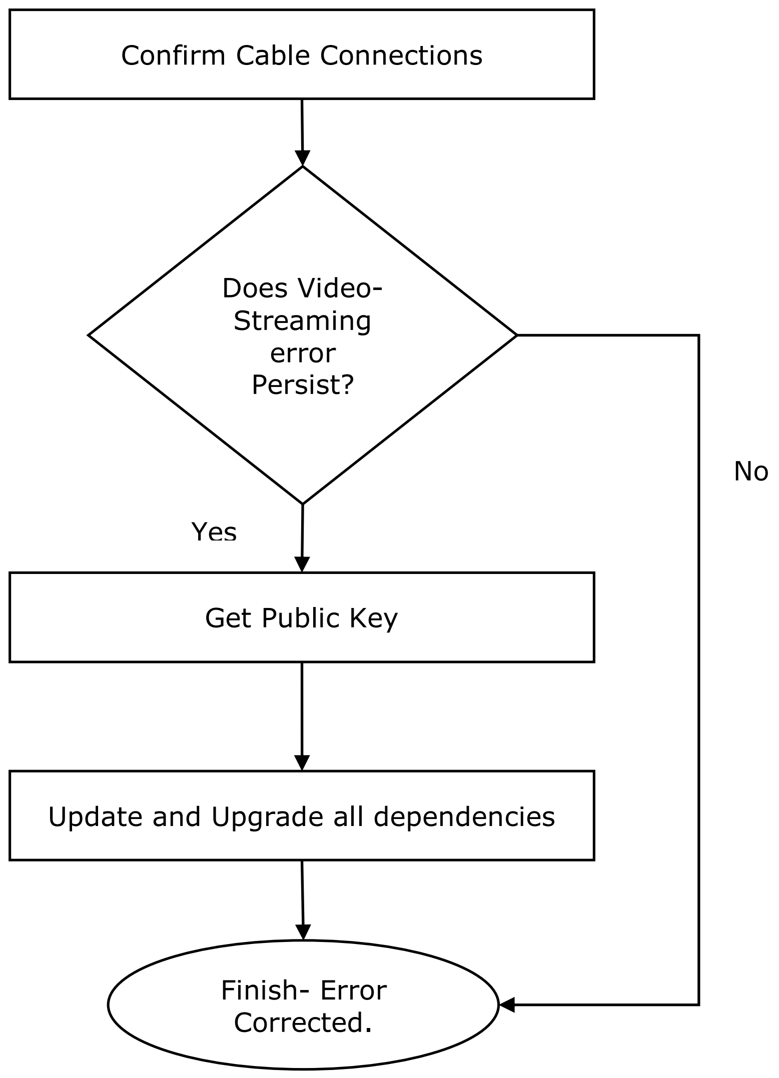

Confirm Cable Connections
Does Video-Streaming error Persist?
No
Yes
Get Public Key
Update and Upgrade all dependencies
Finish- Error Corrected.

Figure 15: Flowchart depicting how to troubleshoot Video-Streaming error in FlytOS

The details of this sequence are explained below:

- **Confirm if the RPi-cam is well inserted into the CSI slot**: Turn-off the RPi, remove and reconnect the RPi-cam.
- If the problem still persists then the following should be done in the FlytOS terminal:
    - **Get Public-Key**: Get a public key by executing this command in the terminal-

      ```
      sudo apt-key adv –keyserver 'hkp://keyserver.ubuntu.com:80' –recv-key C1CF6E31E6BADE8868B172B4F42ED6FBAB17C654
      ```

      This is necessary in order to have the authority to execute the next command.
    - **Update and Upgrade all dependencies**: run the following command to update and upgrade all FlytOS dependencies (including opencv which is usually the problem)-

      ```
      sudo apt update && sudo apt upgrade –y
      ```

After these steps, the video-streaming error will be eliminated. In the next section, the development of the precision-landing algorithm is discussed.

4.2 Setting up ROS for this Study

As mentioned in the Software overview chapter. FlytOS is based on ROS and Linux, hence using FlytOS for this Study will demand utilizing basic ROS functionalities and Linux terminal commands. The following were done, in order to set up ROS for this Study:

- Creating a ROS Package
- Updating ROS environmental variables

4.2.1 ROS-Package Creation

A ROS package is the main unit that is used to keep software in ROS organized. The ROS package may contain various program units including nodes, libraries, configuration files and so on. In general, all dependencies needed for a program to run are indicated in its ROS package. A ROS-package named **marker_detector** was created for this Study in the following way:

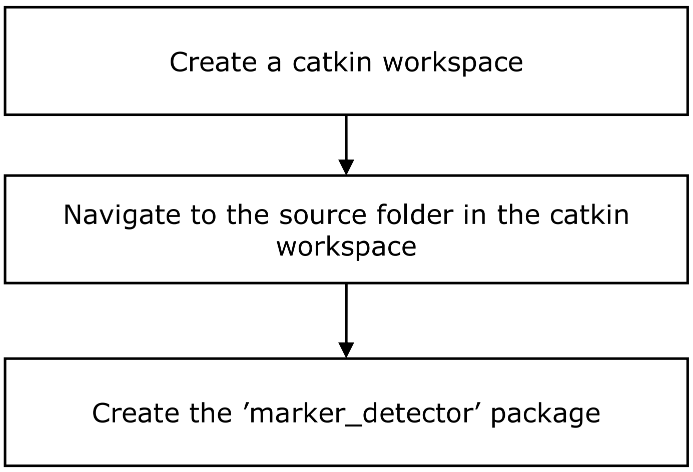

Create a catkin workspace
Navigate to the source folder in the catkin workspace
Create the 'marker_detector' package

Figure 16: Flowchart showing the creation of the 'marker_detector' package

Details for the sequence above are explained thus:

- **Create a catkin workspace**: A catkin workspace (catkin_ws) can be understood as the highest directory in a ROS user workspace. A catkin workspace can contain several packages which are each dedicated to unique applications.

  A catkin_ws is fundamental to most ROS projects and details on how to create one in a Linux terminal are available [35]. A catkin_ws consists of 3 basic folders, namely;

  - Source folder (src)
  - Development folder (devel)
  - Install folder (install)

  For this Study our focus is on the src folder.

- **Navigate to the src folder**: It is in the source folder of the catkin workspace that a package is created. We navigate to this folder using the following Linux command:

  ```
  cd catkin_ws/src
  ```

- **Create "marker_detector" Package**: Once in the catkin_ws/src folder we can now begin creating packages. The process of creating packages have been automated in the ROS ecosystem. It has the following syntax:

```
catkin_create_pkg <name_of_package_to_be_created> <dependency1> <dependency2> <dependency3>
```

For this Study, the marker_detector package was created with a dependency on only 'rospy'. 'rospy' is the main python library built for ROS, hence the package is created as follows:

```
catkin_create_pkg marker_detector rospy
```

Then finally, the path to ROS package must be indicated in the ROS_PACKAGE_PATH environmental variable, in order for it to be used. Thus marker_detector package path has to be updated into this variable as will be explained in the next sub-section.

4.2.2 Updating ROS environmental variables

ROS environmental variables can be thought of as parameters that affect how a ROS installation performs. There are many environmental variables, but only the ROS_PACKAGE_PATH variable needs to be updated. For this Study, it is required to update the aforementioned variable in every new Linux terminal that is opened, otherwise the marker_detector package will not be found by ROS. The process for doing so is summarized thus:

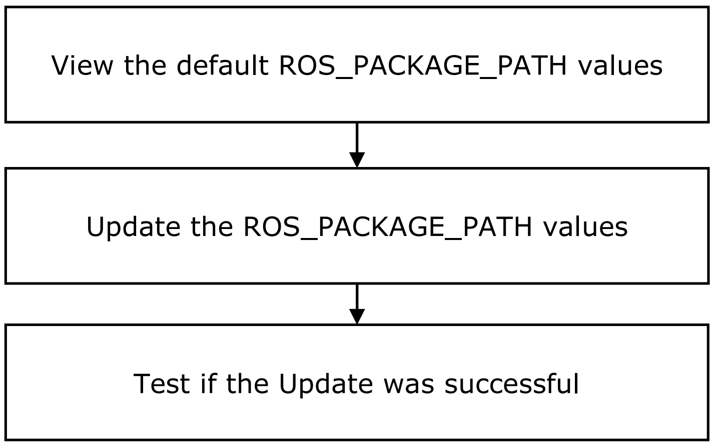

View the default ROS_PACKAGE_PATH values
Update the ROS_PACKAGE_PATH values
Test if the Update was successful

Figure 17: Flowchart describing how to update the ROS_PACKAGE_PATH

Details for this sequence are explained below:

- **View the default ROS_PACKAGE_PATH values**: The default values of ROS_PACKAGE_PATH can be viewed on any Linux terminal by using the following command:

```
echo $ROS_PACKAGE_PATH
```

After the values of this variable have been displayed, it should be highlighted and copied (ctrl+shift+c)

- **Update the ROS_PACKAGE_PATH variable**: In order to update a ROS environmental variable, the ’export’ command is used. The ROS_PACKAGE_PATH variable should updated with the file path for the ’marker_detector’ package created in the previous sub-section using the following command:

  ```
  export ROS_PACKAGE_PATH=<the values copied from the previous
  step>:~/catkin_ws/src/marker_detector
  ```

  It must be confirmed if the values actually changed.

- **Testing the Update**: to test if the value of the ROS_PACKAGE_PATH has actually been updated and that there are no error, it is necessary to view the value of the variable again according to first step above. It should then be observed that the value of this variable has been updated according to the previous step.

5 PROGRAM DEVELOPMENT AND CODING

In this chapter, the Author describes the main algorithm and programming techniques underlying this precision-landing application. The main aspects of this program are:

- Receiving and Processing Video Streams
- Actuating the Drone based on image-data

5.1 Receiving and Processing Video Streams

As a vision-based precision-landing algorithm, the first-step of the program development is to acquire the video data at a reasonable rate. In this section, the Author will discuss how an Image stream was acquired and processed. The general sequence for this is illustrated below:

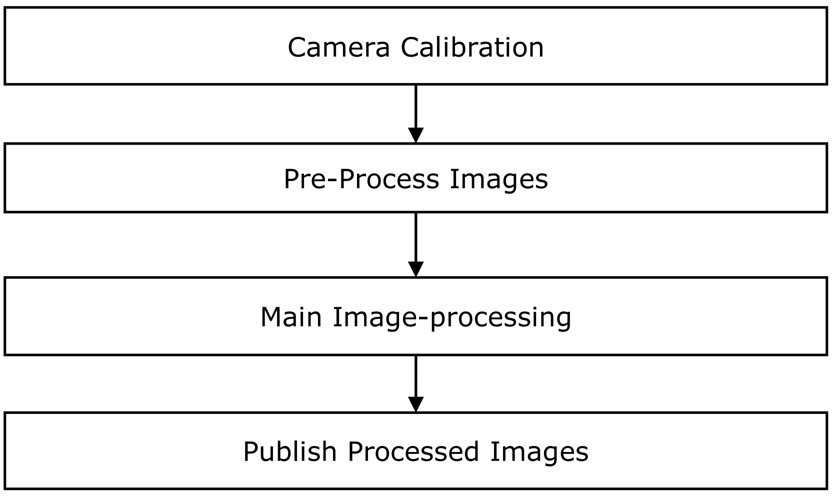


Figure 18: Flowchart showing the general process of reading and processing image data

This module of the precision-landing algorithm is the primary module for this study, and it is based on Python scripts developed by Tiziano Fiorenzani [36]. The Author adapted this code in order for it to work with FlytOS. The details of the sequence in Figure 17 will be explained in the following sub-sections.

5.1.1 Camera Calibration

Camera calibration is the process of estimating the parameters that define a camera's focal length, optical sensor and lens-distortion. Camera calibration is important because, just like finger prints, every single camera is different, and thus the mathematical model representing a camera must capture these values, in order to improve performance of the precision-landing algorithm.

Opencv is the most popular library used for robotics research, and as briefly mentioned in section 5.1.3, is one of the dependencies in FlytOS. Opencv uses a pin-hole camera model in order to estimate a camera matrix:

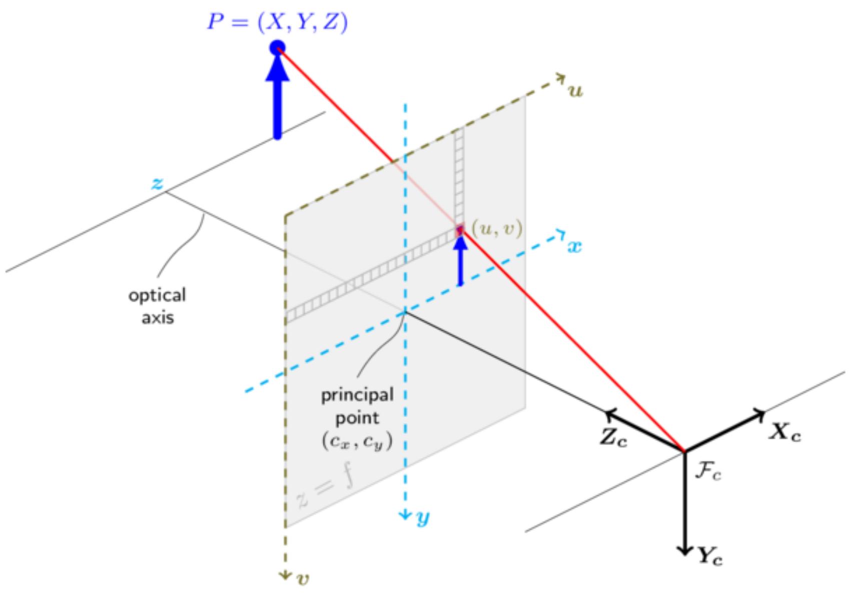


Figure 19: Opencv pinhole camera model [38]

Figure 19 shows that when an object in the real-world (point P) is projected unto a 2D space, it loses its depth perception. In order to compensate for this change, the Opencv pinhole model estimates the intrinsic camera parameters from the following formula [33]:

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

(1.0)

Where:

(X,Y,Z) are the coordinates of a 3D point in the world coordinate space, [38]

(u,v) are the coordinates of the projection point in pixels, [38]

(c_x,c_y) is a principal point that is usually at the image center (the optical sensor), [38]

(f_x,f_y) are the focal lengths expressed in pixel units. [38]

In addition to these, lens distortion of the camera must also be accounted for. Opencv, models this distortion by the following formula [37]:

$$Distortion_c = [k_1, k_2, p_1, p_2, k_3]$$

(2.0)

Where:

k_1,k_2,and k_3 are radial distortion coefficients, and [37]

p_1, and p_2 are tangential distortion coefficients. [37]

In order to estimate all these 9 parameters, the Author used the camera-calibration method described in [37].

For the fisheye camera (see section 3.3) used for this Study, the camera matrix and Distortion coefficients are shown below:

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} 3.04079e+02 & 0.00000e+00 & 3.13905e+02 \\ 0.00000e+00 & 3.03399e+02 & 2.26608e+02 \\ 0.00000e+00 & 0.00000e+00 & 1.00000e+00 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

(3.0)

$$Distortion_c = [-3.52291e-01, 1.74888e-01, -2.49997e-03, -9.60937e-05, -5.16956e-02]$$

(4.0)

These calibration parameters were stored in individual text files and were called in the program as follows:

```
# Get camera calibration parameters

calib_path = “”

camera_matrix  = np.loadtxt(calib_path+’cameraMatric_webcam.txt’,delimiter=’,’)  #get camera matrix file.

camera_distortion = np.loadtxt(calib_path+’cameraDistortion_webcam.txt’,delimiter=’,’)#get distortion coeff
```

5.1.2 Pre-Processing Images

In order to ensure that the data received from the camera were properly processed, some settings needed to be made and the reference frame of the camera needed to be adjusted in the code. The following were done in this sub-section:

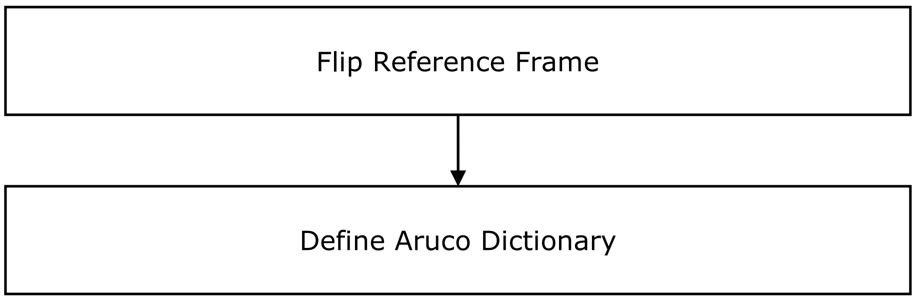


Figure 20: Flowchart showing pre-processing stages

The details of these stages are elaborated below:

- **Flipping the Reference frames**: It is important to flip the reference frame of the camera with respect to the Aruco marker. The reason for this illustrated below:

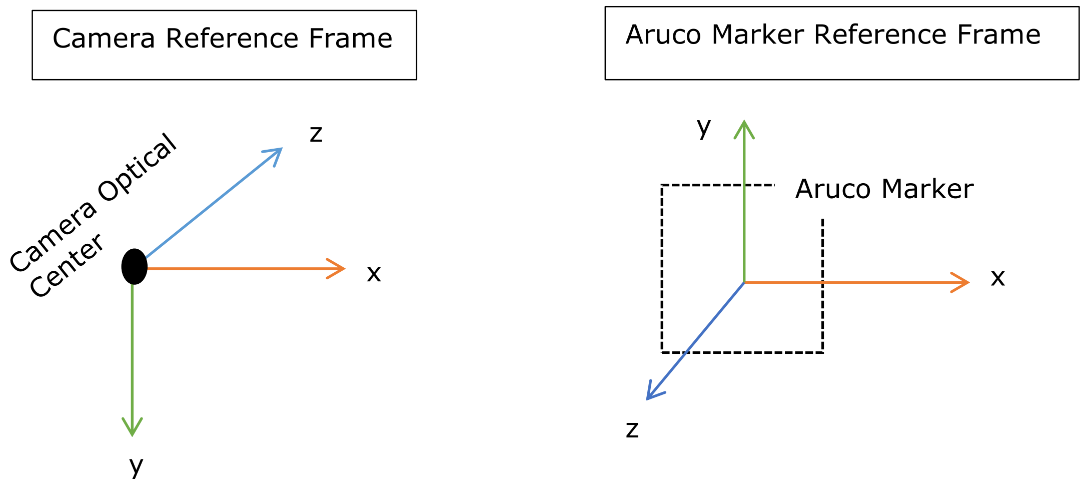


Figure 21: Comparison of the camera and marker reference frames

Figure 20 shows that the camera reference frame has to be flipped by 180 degrees for the 2 reference frames to be equitable. This was executed in code as follows:

```
# Here we create a 180-degree rotation matrix around the x-axis
```

```
R_flip = np.zeros((3,3), dtype=np.float32) #  Create a 3x3 zero-matrix

R_flip[0,0] = 1.0 # Leave the x-axis constant

R_flip[1,1] = -1.0 # Invert the y-axis

R_flip[2,2] = -1.0 # Invert the z-axis
```

This flipped rotation matrix will be used in the next sub-section.

- **Defining an Aruco Dictionary**: As mentioned in section 2.5, Aruco markers are very prominent in robotics research, especially due to the fact that there are ready-libraries already implemented in Opencv that make it easy to use. The Aruco library chosen for this study is the original Aruco dictionary- **DICT_ARUCO_ORIGINAL**, developed by the university of Maryland. The main reason for this choice was the authors experience with this library and also this library has an easy-to-use online marker generator [39].

  The definition of the Aruco dictionary was executed in code as shown below:

```
# Define the Aruco Dictionary to be used

aruco_dict = aruco.getPredefinedDictionary(aruco.DICT_ARUCO_ORIGINAL) #get original dictionary

parameters = aruco.DetectorParameters_create() # define handler for Aruco parameters.
```

5.1.3 Main Image-processing

In this sub-section, the main procedure that converts the Aruco marker image into parameters that can be used by the drone will be explained. The main procedure is as follows:

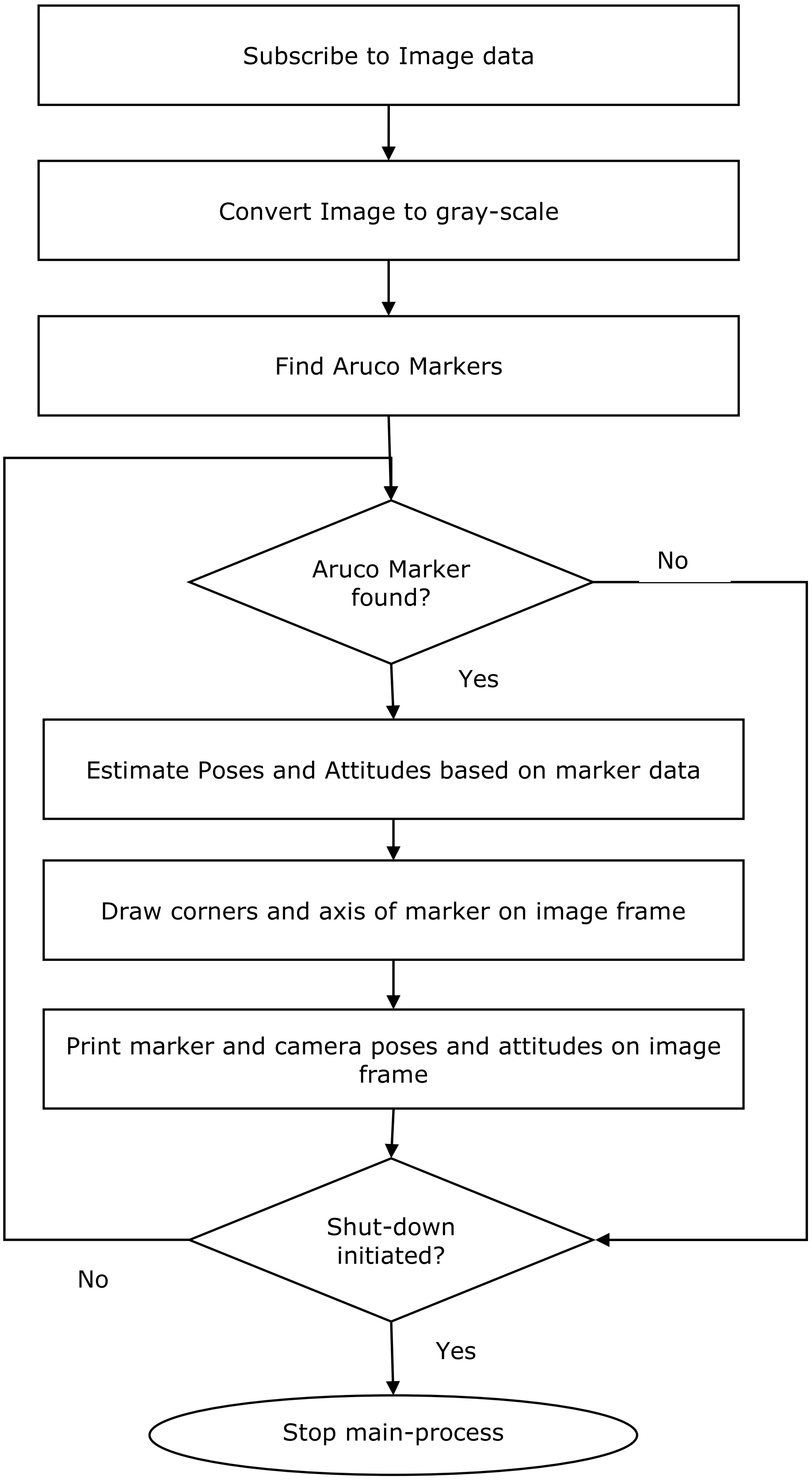

Subscribe to Image data
Convert Image to gray-scale
Find Aruco Markers
Aruco Marker found?
No
Yes
Estimate Poses and Attitudes based on marker data
Draw corners and axis of marker on image frame
Print marker and camera poses and attitudes on image frame
Shut-down initiated?
No
Yes
Stop main-process

Figure 22: Flowchart depicting the main image-processing algorithm

The details of the procedure in Figure 21 are elaborated below:

- **Subscribing to Image data**: As elaborated in section 5.2, FlytOS is built upon ROS. ROS allows nodes (data-processing units) to communicate among each other primarily by allowing nodes to publish to each other and also allowing nodes to subscribe to messages being published.

  FlytOS automatically publishes the images from a RPi-cam connected to the CSI port of the RPi to a ROS-message topic named- **/image_capture**. Thus, this /image_capture topic must be subscribed to in order to get the video stream coming from the RPi-cam, so that it can be processed by Opencv.

  Unfortunately, ROS and Opencv have different message formats and hence ROS-image messages cannot be processed by Opencv by default, thus an extra library was utilized to solve this problem.

  Cv_bridge() is a library that is used to interface ROS messages with Opencv messages. It allows messages coming from ROS to be formatted in a way that Opencv can interpret them and vice-versa. Thus, in order to subscribe to the /image_capture topic, the key points were executed in code:

  ```
  # Subcribe to /image_capture topic

  …

  From cv_bridge import CvBridge, CvBridgeError

  …

  # subcribe to /image_capture topic and send data to the camera_callback function within the class.

  Self.image_sub = rospy.Subscriber("/flytos/flytcam/image_capture", Image, self.camera_callback)

  Self.bridge_object = CvBridge() # create a handler for the cv_bridge function.

  …
  ```

  The above code snippet, derives the image-data and sends it to the camera_callback function within the Python class created for this study. It is in this camera_callback function that the main image-processing is carried out, as is explained next.

- **Converting Images to grayscale**: It is helpful to convert images to grayscale, in order to aid the processing efficiency of this algorithm. As the target of this algorithm is to process Aruco markers, which are black and white in colour, then it is helpful if we ignore every other colour to reduce the data-overhead of the images that we receive from the RPi-cam.

  Opencv provides a method for converting color-images to grayscale, called '**cvtColor**'. This was executed in code in the following way:

```
# Here we convert the received images to grayscale within the camera_callback function of the class

Def camera_callack(self,data):

…

# convert colored-images to OpenCv format and store them in a variable called 'cv_image'

cv_image = self.bridge_object.imgmsg_to_cv2(data, desired_encoding="bgr8")

# change the colored OpenCv image to grayscale, and store in a variable named 'gray'.

gray = cv2.cvtColor(cv_image, cv2.COLOR_BGR2GRAY)

…
```

  After this, the next step of the algorithm is to find Aruco markers in the video-stream. This will be explained next.

- **Finding Aruco Markers**: Opencv is able to detect Aruco markers in an image using the '**detectMarkers**' method within the Aruco class which was imported into the program. This function was executed in code by the following excerpt:

```
# Here we find Aruco Markers in an image and output 3 parameters.

corners, ids, rejected = aruco.detectMarkers(image = gray, dictionary=aruco_dict,
parameters=parameters, cameraMatrix=camera_matrix, distCoeff=camera_distortion)

…
```

  In the code snippet above, we see that the 'detectMarkers' function takes the grayscale image, developed from the previous step, as well as the Aruco dictionary and camera calibration parameters as inputs and returns 3 variables:

  - Corners: the identified corners of the Aruco marker.

  - Ids: The identified id of the marker (Each Aruco marker has a unique id which is chosen by the application developer)

  - Rejected: This is used to indicate image points that could not be recognized by the Aruco dictionary in use.

The next step is a conditional statement to check if a marker was found.

- **Checking if an Aruco Marker was found**: Here, a simple conditional statement was used to check if any Aruco markers have been identified in the image frame. Several Aruco markers can be identified, but for this study, the Author is only interested in finding one Aruco marker which was indicated at the beginning of the program. This process was indicated in code by the following snippet:

```
# Here we check if the Aruco Marker of interest was found.

…

Id_to_find  = 25 # The Author is interested in finding marker-25.

Marker_size = 10 # This is the size in which the marker was printed (units in cm).

…

        If ids is not None and ids [0] == id_to_find:

        …

…
```

Once a marker is found, we continue the image processing as explained next.

- **Estimating Poses and Attitudes**: At this stage, the most important data is acquired from the Aruco marker. One of the advantages of Aruco markers is that, even with just a single marker, the pose and attitude of a vehicle can be derived.

  - **Pose**: is basically the x, y and z coordinate of an object with respect to another. This is useful for making translational motions. While,

  - **Attitude**: is basically the roll, pitch and yaw values of an object with respect to another. This is useful for making rotational motions.

The details of executing this process are summarized in the following flowchart:

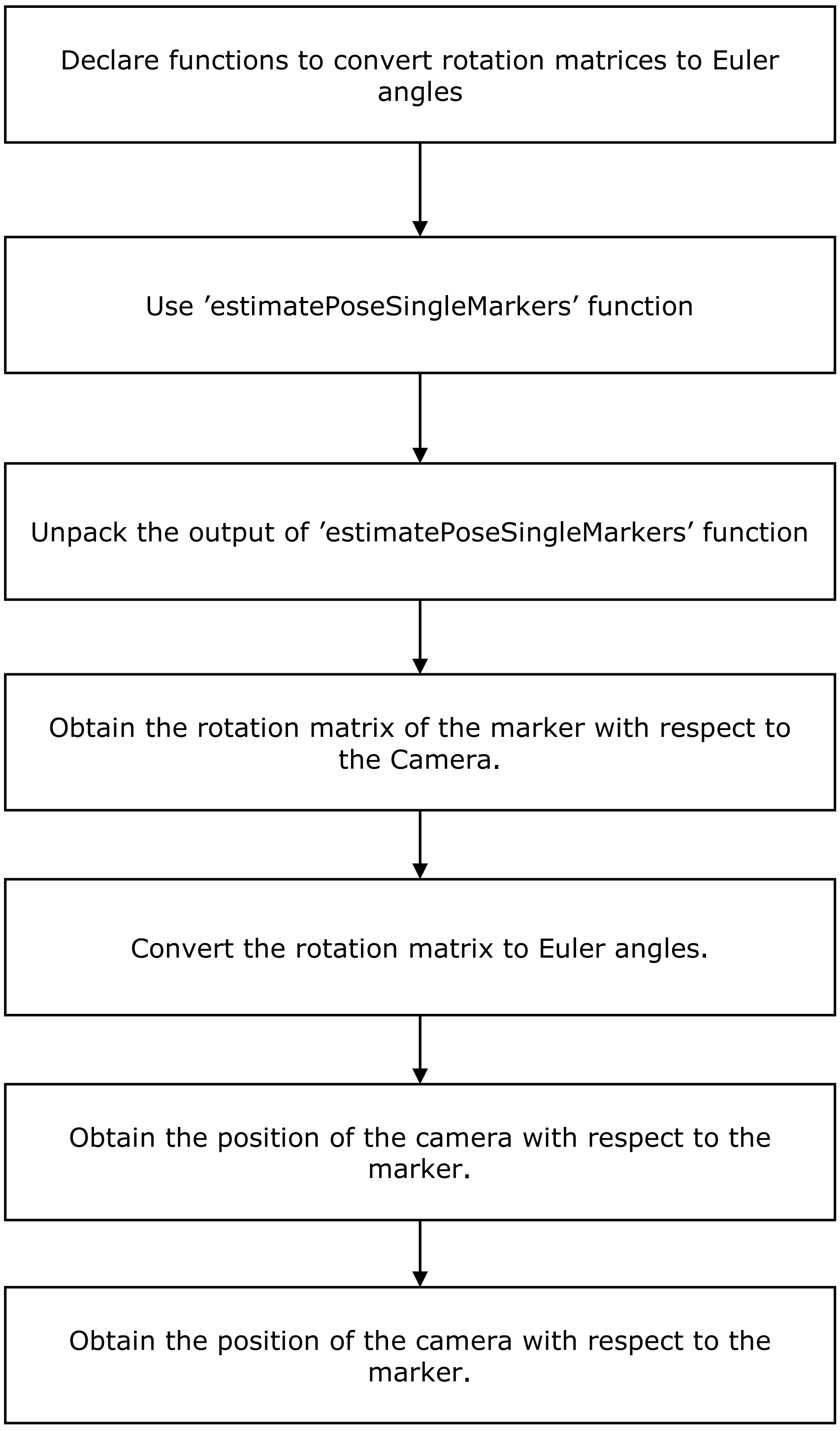

Declare functions to convert rotation matrices to Euler angles
Use ’estimatePoseSingleMarkers’ function
Unpack the output of ’estimatePoseSingleMarkers’ function
Obtain the rotation matrix of the marker with respect to the Camera.
Convert the rotation matrix to Euler angles.
Obtain the position of the camera with respect to the marker.
Obtain the position of the camera with respect to the marker.

Figure 23: Flowchart showing the process of deriving poses and attitudes.

The procedures for actualizing this process in code are detailed in the following code snippet:

```
…

# Here, the functions that handle rotation matrices are declared. They were copied from here [40].

def isRotationMatrix(R): # This function checks if a Matrix is a valid rotation matrix. [40]

        Rt = np.transpose(R)        [40]

        shouldBeIdentity = np.dot(Rt, R)      [40]

        I = np.identity(3, dtype=R.dtype)      [40]

        n = np.linalg.norm(I - shouldBeIdentity)        [40]

        return n < 1e-6     [40]

def rotationMatrixToEulerAngles(R): # This function converts rotation matrices to Euler angles. [40]

        assert (isRotationMatrix(R))            [40]

        sy = math.sqrt(R[0, 0] * R[0, 0] + R[1, 0] * R[1, 0])        [40]

        singular = sy < 1e-6            [40]

        if not singular:      [40]

                x = math.atan2(R[2, 1], R[2, 2])        [40]

                y = math.atan2(-R[2, 0], sy)            [40]

                z = math.atan2(R[1, 0], R[0, 0])        [40]

        else:       [40]

                x = math.atan2(-R[1, 2], R[1, 1])       [40]

                y = math.atan2(-R[2, 0], sy)            [40]

                z = 0       [40]

        return np.array([x, y, z])      [40]

…
```

```
# Here, a function is used to estimate a single marker pose. This function takes the ‘corners’,
#marker_size, camera_matrix and camera_distortion parameters derived in previous stages to
#create a new variable ‘ret’.

ret = aruco.estimatePoseSingleMarkers(corners, marker_size, camera_matrix, camera_distortion)

# Next, the ‘ret’ variable is unpacked to acquire the pose and attitude of the marker with respect to
#the RPi-cam.

rvec, tvec = ret[0][0,0,:], ret[1][0,0,:]  # get the variables for only one marker.

# Next, the rotation matrix of the marker with respect to (wrt) the camera is obtained.

R_ct = np.matrix(cv2.Rodrigues(rvec)[0]) # rotation matrix of camera wrt marker.

R_tc = R_ct.T # rotation matrix of marker wrt camera.

# Here, the rotation matrix is converted to Euler angles, by using a function declared above.

roll_marker, pitch_marker, yaw_marker = rotationMatrixToEulerAngles(R_flip*R_tc)

# Next, the pose of the camera wrt the marker is obtained.

Pos_camera = -R_tc*np.matrix(tvec).T

# Then the attitude of the camera wrt the marker is also obtained.

roll_camera, pitch_camera, yaw_camera = rotationMatrixToEulerAngles(R_flip*R_tc)

…
```

The next stage of this image-processing procedure is to draw visual markers on our image. This is useful, especially for the purpose of troubleshooting errors in the program.

- **Drawing on the image of the marker**: It is helpful to draw the corners and the axis representation of the marker on its image frame for troubleshooting purposes. Once again, the Opencv library provides methods for accomplishing these. This process is executed in code by the following code excerpt:

```
aruco.drawDetectedMarkers(cv_image, corners) # draw the corners on the detected image.

aruco.drawAxis(cv_image, camera_matrix, camera_distortion, rvec, tvec, 10) # draw an axis on the
#detected marker.
```

In the next stage, the poses and attitudes derived a few stages above will be printed on the marker camera frame as well for troubleshooting purposes.

- **Printing Poses and Attitudes on Marker-Image**: In this stage, a string is created to hold the desired values, then the '**putText**' method provided by Opencv is used to print the the generated string on the image. The following code excerpt details this process:

```
…

Font = cv2.FONT_HERSHEY_PLAIN # font to be used for printing texts.

…

# Here, the marker position wrt the camera is printed on an image.

str_position = “MARKER Position x=%4.0f y=%4.0f z=%4.0f”%(tvec[0], tvec[1], tvec[2])

cv2.putText(cv_image, str_position, (0, 100), font, 1, (0, 255, 0), 2, cv2.LINE_AA) # print green text.

# Here, the marker attitude wrt the camera is printed on an image.

str_attitude = “MARKER Attitude r=%4.0f p=%4.0f
y=%4.0f”%(math.degrees(roll_marker),math.degrees(pitch_marker),math.degrees(yaw_marker))

cv2.putText(cv_image, str_attitude, (0, 150), font, 1, (0, 255, 0), 2, cv2.LINE_AA) # print green text.


# Here, the camera position wrt the marker is printed on an image.

str_position = “CAMERA Position x=%4.0f y=%4.0f z=%4.0f”%(pos_camera[0], pos_camera[1],
pos_camera[2])

cv2.putText(cv_image, str_position, (0, 200), font, 1, (0, 255, 0), 2, cv2.LINE_AA) # print green text.

# Here, the camera attitude wrt the marker is printed on an image.

str_attitude = “CAMERA Attitude r=%4.0f p=%4.0f
y=%4.0f”%(math.degrees(roll_camera),math.degrees(roll_camera),math.degrees(roll_camera))

cv2.putText(cv_image, str_attitude, (0, 250), font, 1, (0, 255, 0), 2, cv2.LINE_AA) # print green text.

…
```

The processes described in the previous stages will keep on running until the main process is stopped as described in the next stage.

- **Stopping the main-process**: The above stages in which, the poses and attitudes of the marker and camera are derived will keep on running in an infinite loop until the process is stopped when there is a keyboard interrupt. In ROS and the Linux terminal in general, a keyboard interrupt is initiated from the keyboard using 'Ctrl + c' command. This exception is captured within the class created for this image-processor in the following way:

```
# Here the 'MarkerDetector()' class created for this study is called and looped continuously except
#there is a keyboard interrupt.

…

def main(): # create main function.

        rospy.init_node('marker_detecting_node', anonymous=True) # Initialize ROS node.

        marker_detector_object = MarkerDetector()

        try:

                rospy.spin() # loop infinitely

        except KeyboardInterrupt:

                print("Shutting down")

if __name == '__main__':

        main() # call the main function
```

In the next section, the process of publishing the processed image and required poses and attitudes will be explained.

5.1.4 Publishing Processed Images and data

The outputs of the previous sub-section are processed images and various data representing the relative poses and attitudes of the marker and the camera. In order to make all these data available to other modules of programs, as well as in order to display the processed images via FlytOS, it is necessary to publish them.

ROS has standard procedures for publishing topics in python environment [41], by using the 'rospy' class. For this Study, different message-types are involved, thus their publication procedures are slightly different, the topic published are:

- **/processed_image**: This topic has an 'Image' message format.
- **/pose_data**: This topic has a 'Float32MultiArray' message format.

Publishing the /processed_image topic: A topic named '/processed_image' was created and published by the following sequence summarized thus:

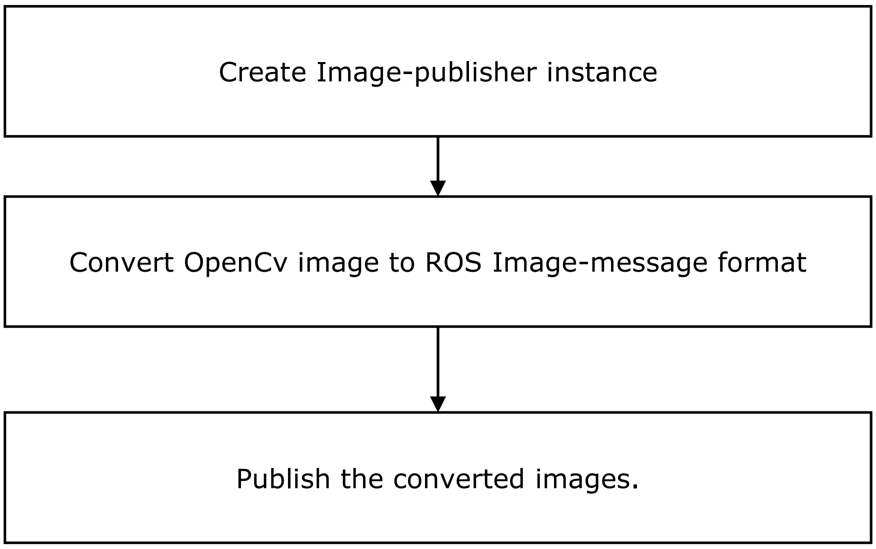

Create Image-publisher instance
Convert OpenCv image to ROS Image-message format
Publish the converted images.

Figure 22: Flowchart showing the process of publishing '/processed_image'

The process was executed in code by the following excerpt:

```
…

# Here, we create the Image-Publisher Instance inside the MarkerDetector class

self.image_pub = rospy.Publisher("/processed_image", Image, queue_size=10)

…

# Here, the OpenCv image is converted back to a ROS Image-message format

msg = self.bridge_object.cv2_to_imgmsg(cv_image, "bgr8")

# Finally, we publish the topic.

self.image_pub.publish(msg)
```

…

In the code snippet above, it is import to emphasize that the image-message format (in this case "bgr8") must be specified for the proper functioning of the program. Next the /pose_data publication process is explained.

Publishing the /pose_data topic: The contents of /pose_data is an array of data values that represent the various poses and attitudes that are required to actuate the Drone for this precision-landing algorithm. The use of the 'Float32MultiArray' message type for this topic requires, the following processes illustrated below:

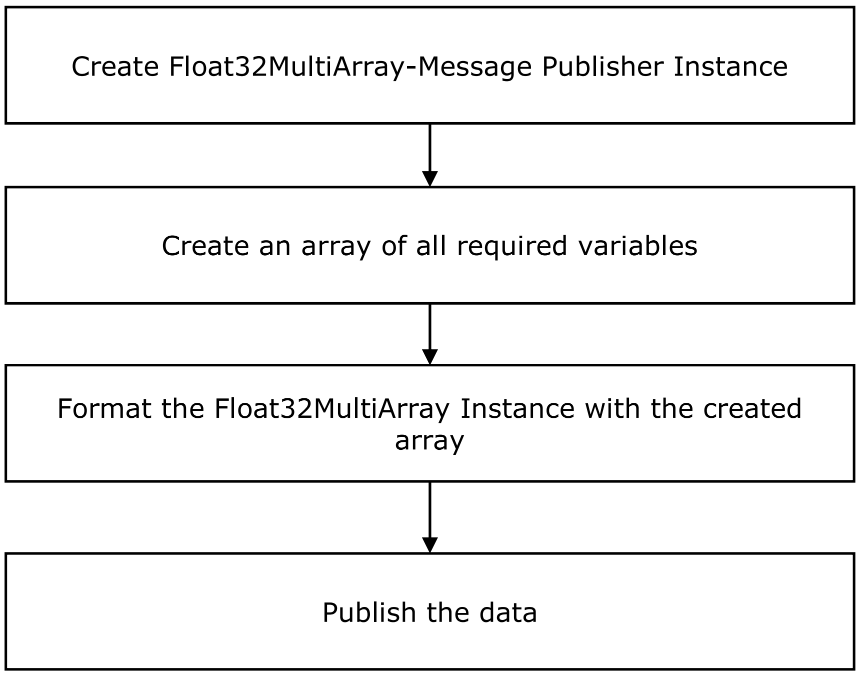

Create Float32MultiArray-Message Publisher Instance
Create an array of all required variables
Format the Float32MultiArray Instance with the created array
Publish the data

Figure 23: Flowchart showing the procedure for publishing the /pose_data topic.

The execution of these procedures in code are described in the following code snippet:

```
…

# Here the Float32MultiArray-Message Publisher instance is created

Self.pose_pub = rospy.Publisher("/pose_data", Float32MultiArray, queue_size=10)

# Next, an array is created to hold the variables that are important for this study.

array = [pos_camera[0], pos_camera[1], pos_camera[2],tvec[0], tvec[1], tvec[2]]

# Here the Float32MultiArray instance is formatted
```

```
Position = Float32MultiArray(data=array)

# Finally, the data are published.

self.pose_pub.publish(position)

…
```

In the next section, a different Python script will be developed, which will actually actuate the Drone based on data received from this topic.

5.2 Actuating the Drone based on image-data

In this section, the following procedures will be followed in order to actuate the drone.

- Setup
- Actuate
- Shutdown

The **Setup** processes include the following:

- Import necessary libraries and classes
- Initialize the FlytOS API
- Subscribe to /pose_data

The **Actuate** processes include the following:

- Take-off
- Acquire translation parameters
- Translate Drone translate Drone until desired tolerance is achieved

Finally, the **Shutdown** sequence is:

- Hold position,
- Delay, and
- Land

In addition to these, it was also important to understand by how much the Drone deviated from its desired position, in order know the range in which to set the Drone's tolerance values to minimize oscillations. To this effect, the Author applied a proportional control to the z-axis and allowed the Drone to hover, and then the values where plotted in order to see how much the Drone deflected by. These were the results:

For the z-axis:

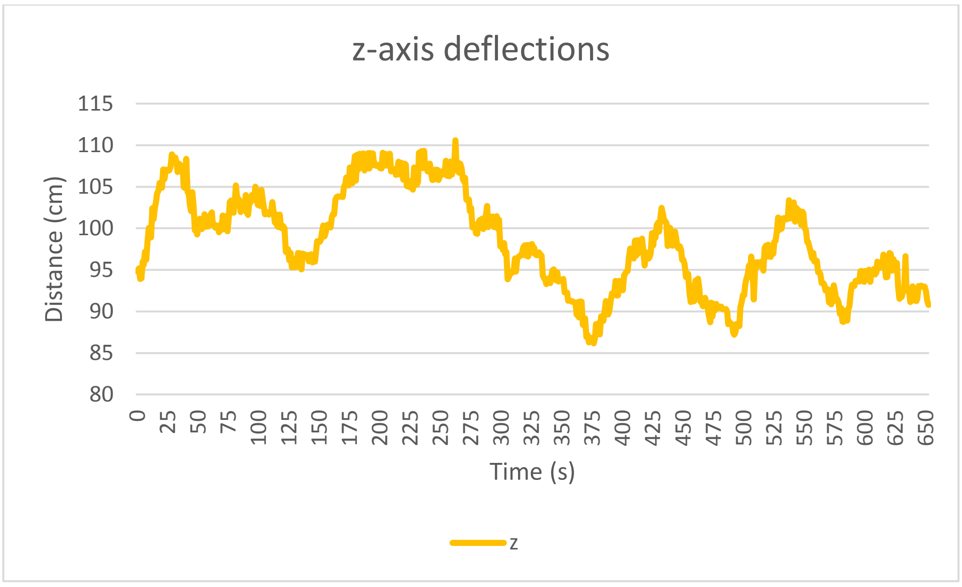


Figure 24: z-axis deflection while hovering

Figure 25 above, shows that the Drone tries to maintain a set-point at 100cm but it deflects upward by about 10 cm and downward by about 15 cm. The average value of its oscillations was about 98cm, thus 98 was used as the reference point in the algorithm.

For the x-axis:

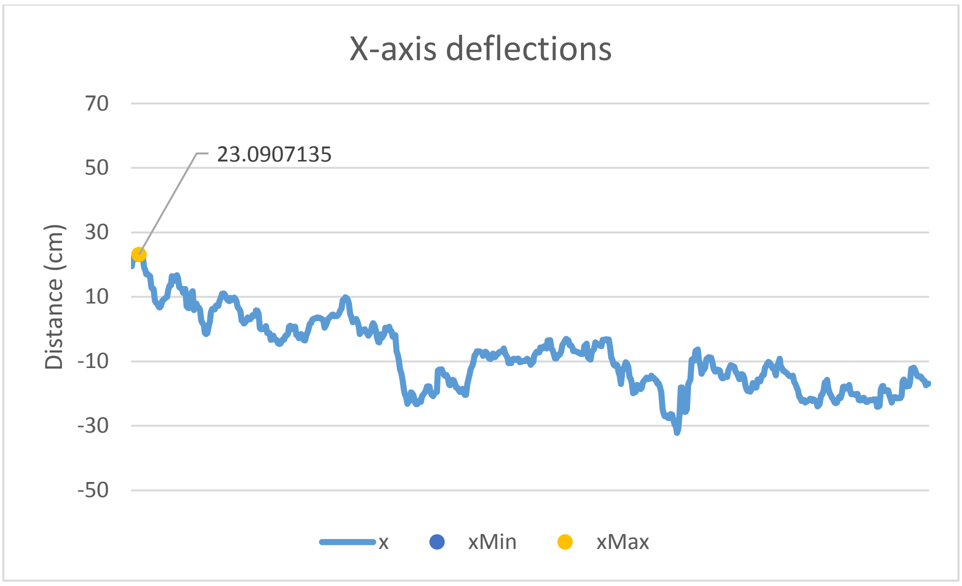


Figure 25: x-axis deflection while hovering

Figure 26, above shows that while the Drone tried to hold its position under the influence of various gusts of wind, it held an average value of -8 during the period of this oscillation and deflected by about 20 cm in either direction.

For the y-axis:

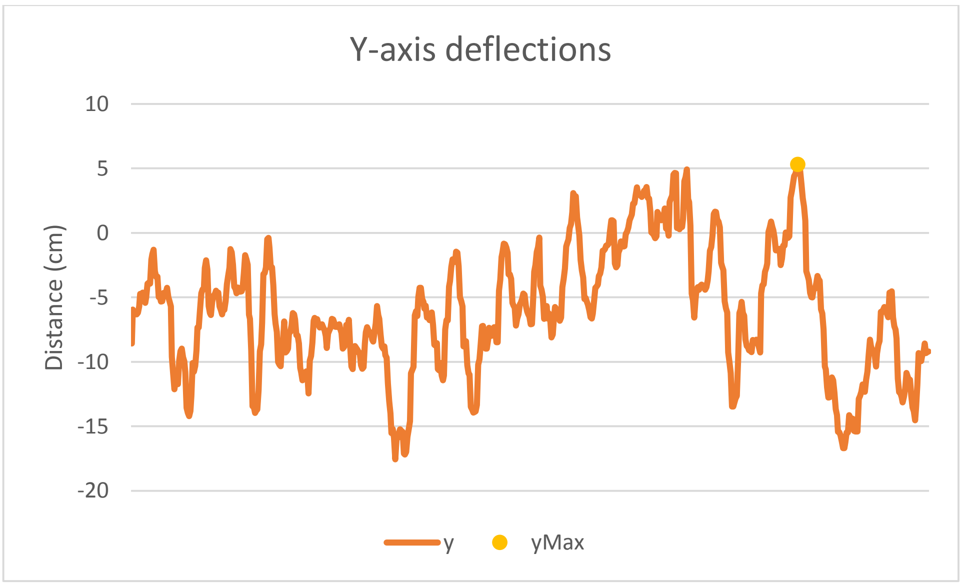
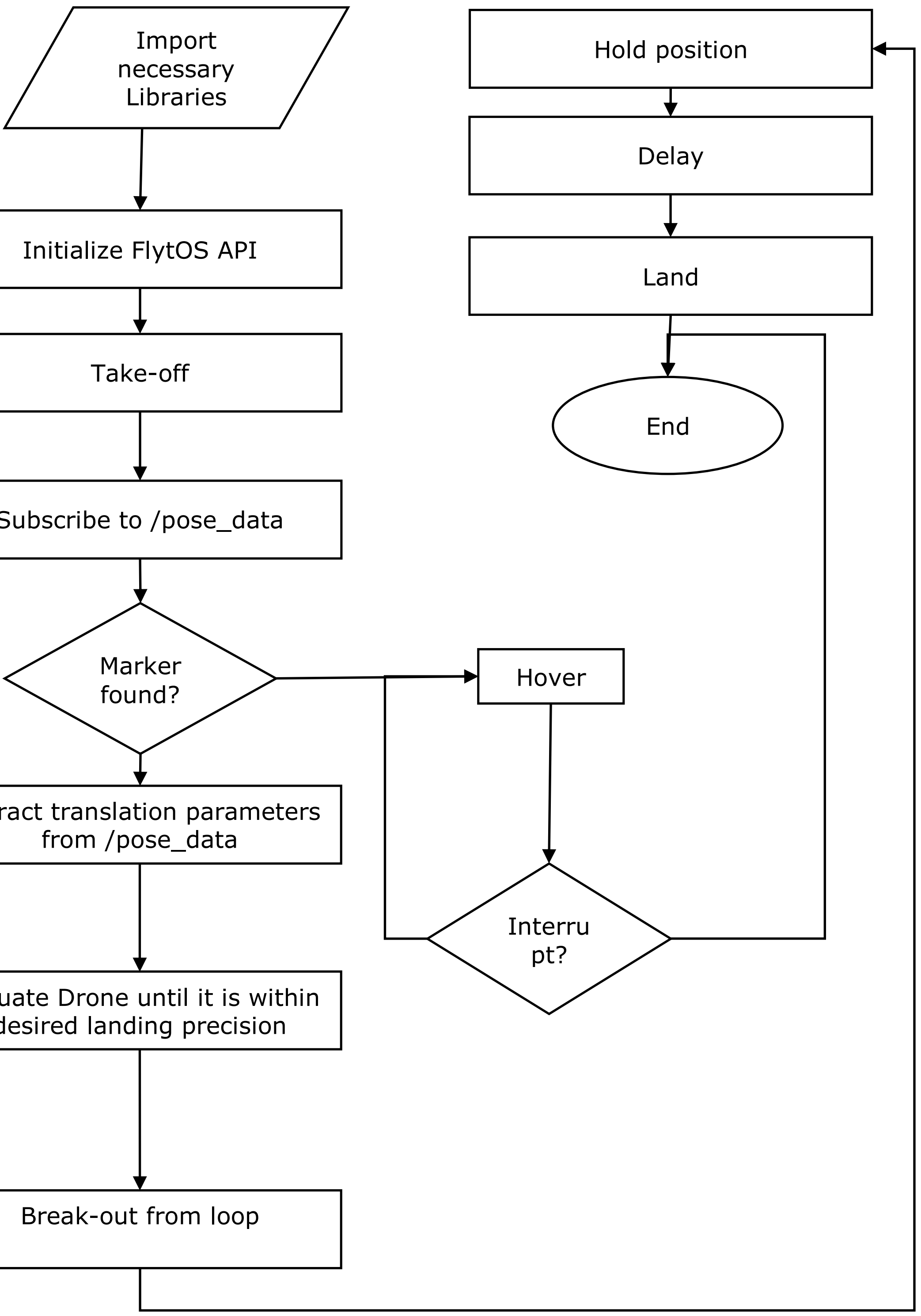


Figure 26: y-axis deflection while hovering

Figure 27 above shows how the Drone deflected on its y-axis while trying to hold its position. It held an average postion of -6 and deflected by about 9 cm in either direction.

These sequence is elaborated in Figure 28, below:



Import necessary Libraries
Initialize FlytOS API
Take-off
Subscribe to /pose_data
Marker found?
Extract translation parameters from /pose_data
Actuate Drone until it is within desired landing precision
Break-out from loop
Hover
Interrupt?
Hold position
Delay
Land
End

Figure 27: Flowchart showing the Actuation procedures

The coding of the processes in Figure 28 is detailed in ’PrecisionLandingTest.py’ in the Appendices section.

If the Drone does not find the marker, it continues hovering in its takeoff position until the keyboard interrupt (ctrl + c) is initiated. When the keyboard interrupt is initiated, the shutdown sequence commences and the drone lands as required.

6 EXECUTION OF PRECISION-LANDING ALGORITHM

In this chapter, the Author discusses how the precision-landing algorithm was tested in real-life conditions. The execution procedure will be divided into 3 sections:

- The Test-setup
- Remote Connection to RPi
- Monitoring and Troubleshooting

6.1 The Test-setup

The test setup is shown below:

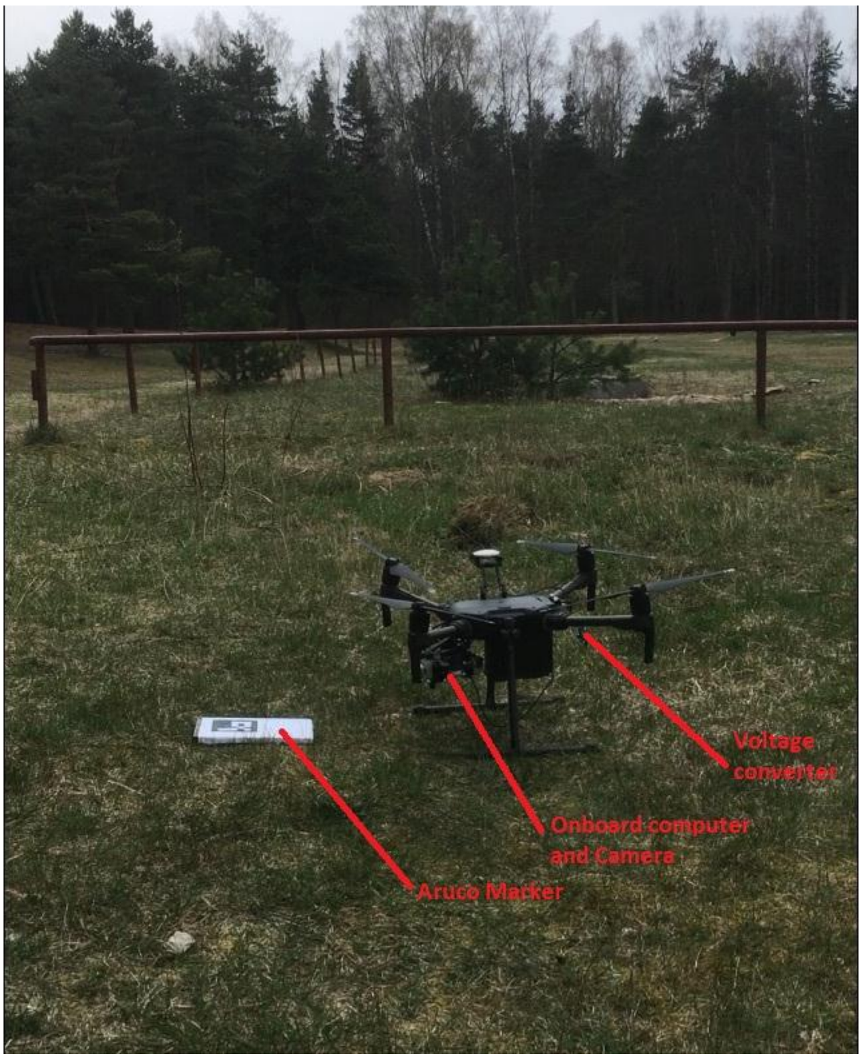


Figure 28: Test Setup

Figure 26 shows that the Drone (coupled with the Onboard computer and camera) was placed a few centimeters away from the Aruco marker. After this setup, the Author kept a considerable distance away from the setup (for safety) and then initiated the precision landing remotely using a computer wirelessly connected to the Rpi, as explained in the next chapter.

6.2 Remote connection to Rpi

FlytOS comes enabled with wifi, which broadcasts a signal whenever the Rpi is turned on. This wifi was connected-to on different computer (Windows OS), through which a remote connection (ssh) was made to the Rpi. The process for doing so through the ’Putty’ Windows application is summarised below:

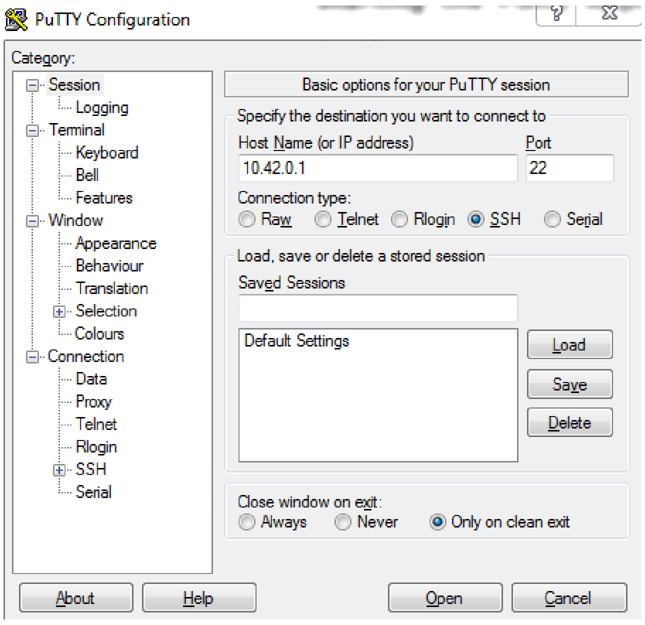
- Start the Putty Application
- Fill-in the IP address of Rpi
- Ensure that the ’connection-type’ is SSH
- Login to FlytOS

Starting the Putty Appication: This is a trivial matter of doubl-clicking the Putty desktop icon.

Inputing the Rpi IP address: By default it is **10.42.0.1**, but it is helpful to confirm this by initially doing the following:

- Connect a monitor and keyboard the Rpi
- Start the Linux terminal (ctrl+alt+t)
- In the Linux terminal, enter the ’ifconfig’ command.
- Then take note of the ’**inet addr**’ in the ’**wlan0**’ section.

This value can then be input into the **Host Name (or IP address)** field of Putty, as shown below:



Host Name (or IP address): 10.42.0.1
Port: 22
Connection type: Raw [✓ SSH] Telnet Rlogin Serial
Close window on exit: Always Never [✓ Only on clean exit]

Figure 29: Remote connection through Putty.

From Figure 27, it is also shown the **SSH option is chosen**.

After the 'open' option is clicked, a Linux terminal opens-up in Windows and askes for the login details. The FlytOS login details are:

- Username: flytos
- Password: flytos

After this process, the setup proceedures discussed in section 5.2.2 are performed. Then, the precision algorithm is initiated with the following commands, **performed in different Putty windows**:

- **Starting the Aruco marker detector program** is done with the following command:

  ```
  Rosrun markerdetector arucoTest.py
  ```

- **Starting the Precision-landing program** is done with the following command:

  ```
  Rosrun markerdetector PrecisionlandingTest.py
  ```

After running those commands the Drone should take-off and land over the Aruco marker. In the next section, the Author explains monitoring and troubleshooting steps.

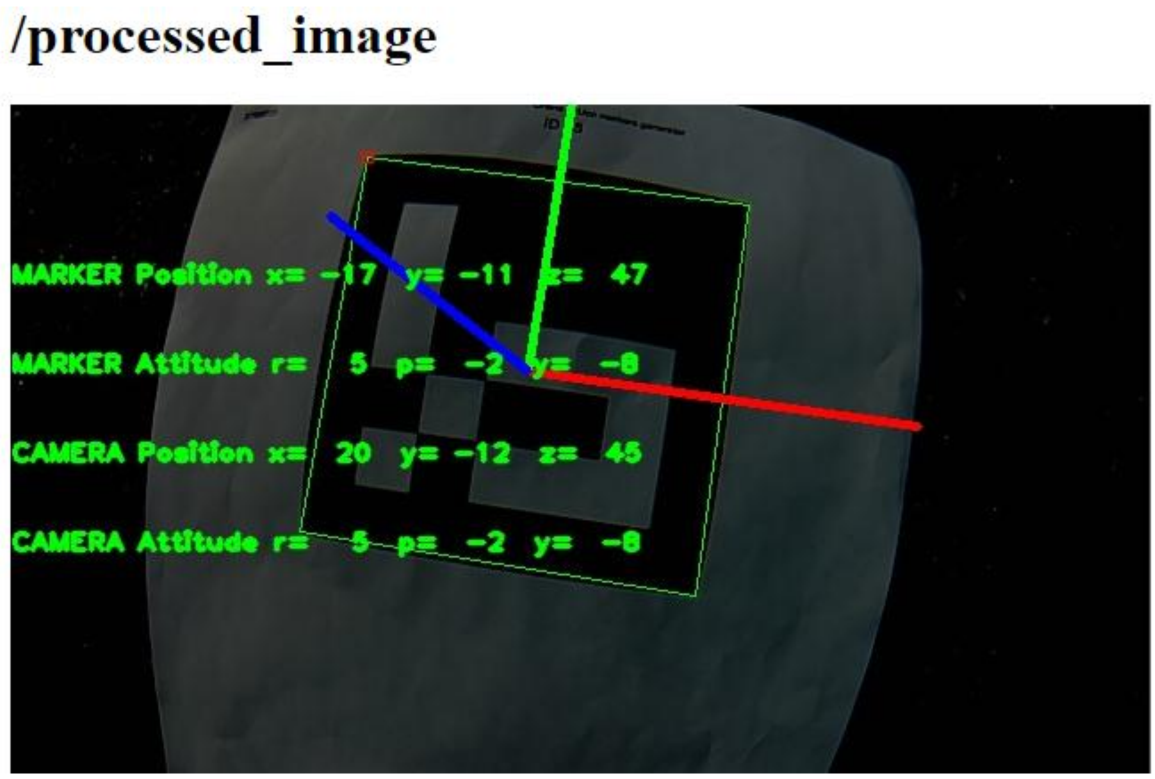
6.3 Monitoring and Troubleshooting

In order to observe what the algorithm is actually doing, it is helpful to be able to see the video stream that contains the processed images. This is enabled through FlytOS and can be accessed by doing the following:

- **View the list of FlytOS image topics**: This can be done in any web-browser on the windows computer, using the following URI (Uniform Resource Identifier):

  10.42.0.1:8080

- **Open the /processed_image topic**: From the list of 'Available ROS Image Topics:' click on the /processed_image topic. After doing so, a user will be able to see the marker and its parameters if the Drone is hovering over it as shown below:



/processed_image

```
MARKER Position x= -17  y= -11  z=  47
MARKER Attitude r=   5  p=  -2  y=  -8
CAMERA Position x=  20  y= -12  z=  45
CAMERA Attitude r=   5  p=  -2  y=  -8
```

Figure 30: Monitoring the processed images.

Finally, the Author discusses troubleshooting steps that were taken to fine-tune the algorithm, in order to get suitable landing accuracies. These steps are summarized as follows:

- **Ensure proper camera settings are used**: The Rpi-cam used for this Study can be adjusted via FlytOS. The parameters to be adjusted are:

- Sharpness – sets the image sharpness
- Contrast – sets the contrast (the more contrast, the better)
- Brightness – sets the brightness of the images
- Saturation – sets the image colour saturation
- Vstab- turns Video stabilisation on or off

The settings of the these parameters can significantly affect how the camera sees the Aruco-Marker in different ambient lighting conditions.

- **Ensure camera is well calibrated**: Camera calibration is perhaps the most important factor to be considered for this study. This is because the relative distances which are calculated by the rotation matrices are dependent on the pin-hole camera model used by Opencv. Thus, if the camera-matrix and distortion-coefficients for the camera are inaccurate, then the relative distances calculated will also be innaccurate.

- **Adjust actuation algorithm**: Finally, after each landing, the distance away from the desired position was measured and used to offset the translation command in the algorithm. This was necessary because the camera is not mounted at the center of mass of the Drone.

7 ANALYSIS OF RESULTS

In this Chapter, the Author considers the following points:

- Accuracy and Speed of Landings
- How the Algorithm compares with another one.

This analyses are detailed in the following sections.

7.1 Accuracy and Speed of Landings

Here, the Author summarizes the results of the precision-landing test. Key parameters for this study were:

- Accuracy of the Landings.
- Time taken to Land.

The results are summarized in the table below:

Test Number (#)	**Landing Error (cm)**	**Time Taken to Land (s)**
1	25	65
2	7	60
3	10	120
4	15	35
5	20	37
6	11	48
7	23	70
8	16	105
9	20	45
10	15	112

Table 1: Landing Accuracies and time-taken

In the next section, comparism will be made with how well these results compare with those of Timothy and Jesse [2].

7.2 Results Comparism

Timothy and Jesse [2] achieved very good results using a drone that they developed themselves which included an off-the-shelf auto-pilot. Then they used this drone to implement their 'varying-degree of freedom, image-based visual servoing' (VDOF IBVS) [2] algorithm, thus it will help put this study in context by comparing the results of this relatively simple algorithm with those achieved with a much more complex algorithm.

Test(#)	This Study		Timothy and Jesse [2]	
	Landing Error (cm)	**Landing Time (s)**	**Landing Error (cm)**	**Landing Time (s)**
1	25	65	6	26.1
2	7	60	5	26.3
3	10	120	9	25.5
4	15	35	6	24.8
5	20	37	10	28.1
6	11	48	8	23.4
7	23	70	8	22.5
8	16	105	9	25.6
9	20	45	4	26.1
10	15	112	5	35.4

Table 2: Comparison of Results

A study of Table 2 above shows that Timothy and Jesse's algorithm cleary performed better in real-world situations, achieving greater landing precision in a shorter time. They accomplished this by using a more complex algorithm, and nested Aruco markers (One small Aruco Marker, inside one large one). This study however used only translation commands to acuate the Drone and one 10 cm-sized Aruco marker, hence it was not as accurate.

It is important to emphasize once again, that the primary purpose of this Study is to develop a multi-platform precision-landing system for quadrotors using FlytOS, to enhance the reusablilty of Drone algorithms. The accuracy and speed of the landings were secondary objectives. The main focus of the study was to consolidate a system which can be easily reused for the widest variety of Drones.

8 CONCLUSIONS

The purpose of this study is to develop a precision-landing system which can be used on the widest variety of Drone platforms by using FlytOS. The Author intended to solve the problem of the reusability of code among different Drone platforms.
Based on the bulky amount of research articles related to precision-land it was clear that precision-landing has already been solved. What was difficult however was the process of testing-out the algorithms developed by the said publications. If the hardware used by a study were not available to an individual, then it would be difficult to reuse the codes.

In this Study the Author achieved the following:

- Developed a precision-landing system based on FlytOS that achieved reasonable landing tolerances: The main advantage is that this algorithm is that it, can be reused on PX4, ArduPilot and DJI drones. These Drone platforms are the most popular at the time of this writing.
- Developed a plug-and-play hardware to carry the RPi and RPi-cam to attach to DJI M210 drone.

Strategies for further development

The Author considers the following paths for development of this Study:

- **Use of Cascaded Aruco Markers**; the use of various markers embedded within each other, rather than just one marker will allow the drone to spot the marker from greater altitudes. This, will increase landing accuracy and speed.
- **Addition of a self-setup and calibration module to this algorithm**; A significant amount of time was spent finding a proper landing-pose and calibrating the camera used for this Study. These procedures can be programmed into a module that will initially run automatically whenever it is being used on a Drone for the first time or when the camera has been changed. This will make the user experience much more satisfying.
- **Rewriting the program in C**; from this study, it was observed that there was significant delay in the rate at which the camera image streams were updated. This is probably because the program and all the libraries used for the study were developed in Python. Python, is an interpreted language and thus in general is not as fast as compiled languages such as C. A faster processing time will improve the consistency of the results of the precision-landing algorithm.

9 JÄRELDUSED

Selle uuringu eesmärk on välja töötada täpsusmaandumissüsteem, mida saab FlytOSi abil kasutada võimalikult erinevatel drooniplatvormidel. Autor kavatses lahendada koodi korduvkasutatavuse probleemi erinevate drooniplatvormide vahel. Täppismaad käsitlevate mahukate teadusartiklite põhjal oli selge, et täpsusmaandumine on juba lahendatud. Raske oli aga nimetatud väljaannetes välja töötatud algoritmide testimine. Kui uuringus kasutatud riistvara pole üksikisikule kättesaadav, on koodide taaskasutamine keeruline.

Selles uuringus saavutas autor järgmise:

- Töötas välja FlytOSil põhineva täppismaandumissüsteemi, mis saavutas mõistlikud maandumiste tolerantsid: Peamine eelis on see, et see algoritm seisneb selles, et seda saab PX4, ArduPiloti ja DJI droonides uuesti kasutada. Need drooniplatvormid on selle kirjutamise ajal kõige populaarsemad.
- Töötas välja plug-and-play riistvara RPi ja RPi-cam kandmiseks, et kinnituda DJI M210 droonile.

Edasise arengu strateegiad

Autor kaalub uuringu arendamiseks järgmisi teid:

- **Kaskaaditud Aruco markerite kasutamine**; erinevate markerite kasutamine, mis on põimitud üksteise sisse, mitte ainult ühe markeriga, võimaldab droonil märgata markerit suuremal kõrgusel. See suurendab maandumise täpsust ja kiirust.
- **Sellele algoritmile on lisatud isehäälestamise ja kalibreerimise moodul**; Märkimisväärne aeg kulus õige maandumisposti leidmiseks ja selles uuringus kasutatud kaamera kalibreerimiseks. Neid protseduure saab programmeerida mooduliks, mis käivitub automaatselt automaatselt, kui seda esimest korda droonil kasutatakse või kui kaamerat vahetatakse. See muudab kasutajakogemuse palju rahuldavamaks.
- **Programmi ümberkirjutamine C-vormingus**; selle uuringu põhjal täheldati, et kaamera pildivoogude värskendamise kiirus oli märkimisväärselt hilinenud. Tõenäoliselt on see sellepärast, et programm ja kõik uuringus kasutatud raamatukogud töötati välja Pythonis. Python on tõlgendatav keel ja seega ei ole see üldiselt nii kiire kui koostatud keeltes nagu C.
  Kiirem töötlemisaeg parandab täpsusmaandumise algoritmi tulemuste järjepidevust.

REFERENCES

[1] "*FlytOS Architecture Diagram*". Accessed on: May 18, 2020. [Online]. Available: http://docs.flytbase.com/docs/FlytOS/Developers/Introduction.html

[2] J. S. Wynn and T. W. McLain, "Visual servoing for multirotor precision landing in daylight and after-dark conditions," *2019 Int. Conf. Unmanned Aircr. Syst. ICUAS 2019*, pp. 1242–1248, 2019.

[3] "*ROS/Introduction*". Accessed on: May 6, 2020. [Online]. Available: http://wiki.ros.org/ROS/Introduction

[4] M. Rieke, T. Foerster, J. Geipel, and T. Prinz, "High-Precision Positioning and Real-Time Data Processing of Uav-Systems," *ISPRS - Int. Arch. Photogramm. Remote Sens. Spat. Inf. Sci.*, vol. XXXVIII-1/, no. September, pp. 119–124, 2012.

[5] R. A. Oliveira, E. Khoramshahi, J. Suomalainen, T. Hakala, N. Viljanen, and E. Honkavaara, "Real-time and post-processed georeferencing for hyperpspectral drone remote sensing," *Int. Arch. Photogramm. Remote Sens. Spat. Inf. Sci. - ISPRS Arch.*, vol. 42, no. 2, pp. 789–795, 2018.

[6] J. N. Gross, R. M. Watson, S. D'urso, and Y. Gu, "Flight-Test Evaluation of Kinematic Precise Point Positioning of Small UAVs," *Int. J. Aerosp. Eng.*, vol. 2016, 2016.

[7] K. N. Tahar, "An evaluation on different number of ground control points in unmanned aerial vehicle photogrammetric block," *Int. Arch. Photogramm. Remote Sens. Spat. Inf. Sci. - ISPRS Arch.*, vol. XL-2/W2, no. November, pp. 93–98, 2013.

[8] "*Detection of ArUco Markers*". Accessed on: May 6, 2020. [Online]. Available: https://docs.opencv.org/trunk/d5/dae/tutorial_aruco_detection.html

[9] A. Gautam, P. B. Sujit, and S. Saripalli, "A survey of autonomous landing techniques for UAVs," *2014 Int. Conf. Unmanned Aircr. Syst. ICUAS 2014 - Conf. Proc.*, pp. 1210–1218, 2014.

[10] A. Cesetti, E. Frontoni, A. Mancini, P. Zingaretti, and S. Longhi, "A Vision-based guidance system for UAV navigation and safe landing using natural landmarks," *J. Intell. Robot. Syst. Theory Appl.*, vol. 57, no. 1–4, pp. 233–257, 2010.

[11] B.-M. Min, M.-J. Tahk, H.-C. D. Shim, and H.-C. Bang, "Guidance Law for Vision-Based Automatic Landing of UAV," *International Journal of Aeronautical and Space Sciences*, vol. 8, no. 1. pp. 46–53, 2007.

[12] J. Janousek and P. Marcon, "Precision landing options in unmanned aerial vehicles," *2018 Int. Interdiscip. PhD Work. IIPhDW 2018*, pp. 58–60, 2018.

[13] E. Nowak, K. Gupta, and H. Najjaran, "Development of a Plug-and-Play Infrared Landing System for Multirotor Unmanned Aerial Vehicles," *Proc. - 2017 14th Conf. Comput. Robot Vision, CRV 2017*, vol. 2018-Janua, pp. 256–260, 2018.

[14] M. F. Sani, M. Shoaran, and G. Karimian, "Automatic landing of a low-cost quadrotor using monocular vision and Kalman filter in GPS-denied environments," *Turkish J. Electr. Eng. Comput. Sci.*, vol. 27, no. 3, pp. 1821–1838, 2019.

[15] P. H. Nguyen, K. W. Kim, Y. W. Lee, and K. R. Park, "Remote marker-based tracking for uav landing using visible-light camera sensor," *Sensors (Switzerland)*, vol. 17, no. 9, pp. 1–38, 2017.

[16] M. F. Sani and G. Karimian, "Automatic navigation and landing of an indoor AR. Drone quadrotor using ArUco marker and inertial sensors," *1st Int. Conf. Comput. Drone Appl. Ethical Integr. Comput. Drone Technol. Humanit. Sustain. IConDA 2017*, vol. 2018-Janua, pp. 102–107, 2017.

[17] H. C. Kam, Y. K. Yu, and K. H. Wong, "An improvement on ArUco marker for pose tracking using kalman filter," *Proc. - 2018 IEEE/ACIS 19th Int. Conf. Softw. Eng. Artif. Intell. Netw. Parallel/Distributed Comput. SNPD 2018*, pp. 65–69, 2018.

[18] A. Marut, K. Wojtowicz, and K. Falkowski, "ArUco markers pose estimation in UAV landing aid system," *2019 IEEE Int. Work. Metrol. AeroSpace, Metroaerosp. 2019 - Proc.*, pp. 261–266, 2019.

[19] N. P. Santos, V. Lobo, and A. Bernardino, "A ground-based vision system for UAV tracking," *MTS/IEEE Ocean. 2015 - Genova Discov. Sustain. Ocean Energy a New World*, pp. 1–9, 2015.

[20] H. Choi, M. Geeves, B. Alsalam, and F. Gonzalez, "Open source computer-vision based guidance system for UAVs on-board decision making," *IEEE Aerosp. Conf. Proc.*, vol. 2016-June, pp. 1–5, 2016.

[21] M. Sereewattana, M. Ruchanurucks, P. Rakprayoon, S. Siddhichai, and S. Hasegawa, "Automatic landing for fixed-wing UAV using stereo vision with a single camera and an orientation sensor: A concept," *IEEE/ASME Int. Conf. Adv. Intell. Mechatronics, AIM*, vol. 2015-Augus, pp. 29–34, 2015.

[22] V. F. Vidal, L. M. Honório, M. F. Santos, M. F. Silva, A. S. Cerqueira, and E. J. Oliveira, "UAV vision aided positioning system for location and landing," *2017 18th Int. Carpathian Control Conf. ICCC 2017*, pp. 228–233, 2017.

[23] M. F. Silva, A. C. Ribeiro, M. F. Santos, M. J. Carmo, and L. M. Honório, "Design of Angular PID Controllers for Quadcopters Built with Low Cost Equipment," *2016 20th Int. Conf. Syst. Theory, Control Comput.*, pp. 216–221, 2016.

[24] E. I. Shirokova, A. A. Azarov, N. G. Wilson, and I. B. Shirokov, "Precision positioning of unmanned aerial vehicle at automatic landing," *Proc. 2019 IEEE Conf. Russ. Young Res. Electr. Electron. Eng. ElConRus 2019*, pp. 1065–1069, 2019.

[25] T. H. Nguyen, M. Cao, T. M. Nguyen, and L. Xie, "Post-Mission Autonomous Return and Precision Landing of UAV," *2018 15th Int. Conf. Control. Autom. Robot. Vision, ICARCV 2018*, pp. 1747–1752, 2018.

[26] I. Kalinov, E. Safronov, R. Agishev, M. Kurenkov, and D. Tsetserukou, "High-precision UAV localization system for landing on a mobile collaborative robot based on an ir marker pattern recognition," *IEEE Veh. Technol. Conf.*, vol. 2019-April, pp. 1–6, 2019.

[27] L. Shungui, T. Bo, Z. Weicai, Z. Xin, and H. Hongtai, "An UAV precision landing method based on virtual control point on the high voltage transmission line," *Proc. 2017 9th Int. Conf. Model. Identif. Control. ICMIC 2017*, vol. 2018-March, no. Icmic, pp. 693–698, 2018.

[28] S. H. Mathisen, T. I. Fossen, and T. A. Johansen, "Non-linear model predictive control for guidance of a fixed-wing UAV in precision deep stall landing," *2015 Int. Conf. Unmanned Aircr. Syst. ICUAS 2015*, pp. 356–365, 2015.

[29] "*Raspberry Pi Pinout Diagram | Circuit Notes*". Accessed on: May 18, 2020. [Online]. Available: https://www.jameco.com/Jameco/workshop/circuitnotes/raspberry-pi-circuit-note.html

[30] Matrice 200 Series M210/M210 RTK User Manual, V1.0, DJI, 2017.08. Accessed on May. 4, 2020. [Online]. Available: https://dl.djicdn.com/downloads/M200/20170821/Matrice_210_210RTK_User_Manual_EN_V1.01.pdf

[31] 400KHz 60V 4A Switching Current Boost/Buck-Boost/Inverting DC/DC Converter, XLSEMI. Accessed on May. 4, 2020. [Online]. Available: https://pdf1.alldatasheet.com/datasheet-pdf/view/1132229/XLSEMI/XL6009E1.html

[32] "*M210 + PC/Linux machine with Advanced Sensing*". Accessed on: May 18, 2020. [Online]. Available: https://developer.dji.com/onboard-sdk/documentation/

33] "*Flashing FlytOS Linux Image (RPi3)*". Accessed on: May 5, 2020. [Online]. Available: http://docs.flytbase.com/docs/FlytOS/GettingStarted/FlashingImgRpi.html#flashing-img-rpi

[34] "*Guide for A3/N3/M100/M600/M210 integration with FlytOS*". Accessed on: May 5, 2020. [Online]. Available: https://docs.google.com/document/d/1Q6vTM6LQ1jh-kpcCmUbdifHaNtjmWGDCPsxUa10ay3o/edit

[35] "*Creating a workspace for catkin*". Accessed on: May 5, 2020. [Online]. Available: http://wiki.ros.org/catkin/Tutorials/create_a_workspace

[36] Tiziano Fiorenzani, *How do Drones Work*, Github. Accessed on: May 5, 2020. [Online]. Available: https://github.com/tizianofiorenzani/how_do_drones_work

[37] Tiziano Fiorenzani, *OpenCv and Camera Calibration on a Raspberry Pi 3*, Jan. 20, 2018. Accessed on: Jan. 25, 2017. [Video file]. Available: https://www.youtube.com/watch?v=QV1a1G4lL3U

[38] "*Camera Calibration and 3D Reconstruction*". Accessed on: May 6, 2020. [Online]. Available: https://docs.opencv.org/2.4/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html

[39] "*ArUco markers generator!*". Accessed on: May 6, 2020. [Online]. Available: http://chev.me/arucogen/

[40] Satya Mallick "*Rotation Matrix To Euler Angles*". Accessed on: May 7, 2020. [Online]. Available: https://www.learnopencv.com/rotation-matrix-to-euler-angles/

[41] "*Writing a Simple Publisher and Subscriber (Python)*". Accessed on: May 7, 2020. [Online]. Available: http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29

[42] "*Waveshare Raspberry Pi Camera Module Kit 1080p Fisheye Lens Wider Field of View for Any Version of Raspberry-pi*". Accessed on: May 18, 2020. [Online]. Available: https://www.amazon.co.uk/Waveshare-Raspberry-Camera-Fisheye-Raspberry-pi/dp/B00RMV53Z2?th=1

[43] "*XL6009E1 DC-DC Adjustable Step-up Boost Power Converter Module Replace*". Accessed on: May 18, 2020. [Online]. Available: https://www.amazon.co.uk/XL6009E1-Adjustable-Step-up-Converter-Replace/dp/B0194QK3F2

APPENDICES

Below are the two main program modules developed for this study:

- The image-processing module – '**arucoTest.py**'
- The Drone-actuating module – '**PrecisionLandingTest.py**'

'**arucoTest.py**':

```
#!/usr/bin/env python


import rospy
from sensor_msgs.msg import Image
from cv_bridge import CvBridge, CvBridgeError
import cv2
import numpy as np
import cv2.aruco as aruco
import sys, time, math
import os


id_to_find  = 25
marker_size  = 10 #- [cm]


def isRotationMatrix(R):
    Rt = np.transpose(R)
    shouldBeIdentity = np.dot(Rt, R)
    I = np.identity(3, dtype=R.dtype)
    n = np.linalg.norm(I - shouldBeIdentity)

    return n < 1e-6




# Calculates rotation matrix to euler angles

# The result is the same as MATLAB except the order

# of the euler angles ( x and z are swapped ).

def rotationMatrixToEulerAngles(R):

    assert (isRotationMatrix(R))



    sy = math.sqrt(R[0, 0] * R[0, 0] + R[1, 0] * R[1, 0])



    singular = sy < 1e-6



    if not singular:

        x = math.atan2(R[2, 1], R[2, 2])

        y = math.atan2(-R[2, 0], sy)

        z = math.atan2(R[1, 0], R[0, 0])

    else:

        x = math.atan2(-R[1, 2], R[1, 1])

        y = math.atan2(-R[2, 0], sy)

        z = 0
    return np.array([x, y, z])



#--- Get the camera calibration path

calib_path  = ""

camera_matrix   = np.loadtxt(calib_path+'cameraMatrix_webcam.txt', delimiter=',')

camera_distortion             =     np.loadtxt(calib_path+'cameraDistortion_webcam.txt',
delimiter=',')


#--- 180 deg rotation matrix around the x axis

R_flip  = np.zeros((3,3), dtype=np.float32)

R_flip[0,0] = 1.0

R_flip[1,1] =-1.0

R_flip[2,2] =-1.0


#--- Define the aruco dictionary

aruco_dict  = aruco.getPredefinedDictionary(aruco.DICT_ARUCO_ORIGINAL)

parameters  = aruco.DetectorParameters_create()


font = cv2.FONT_HERSHEY_PLAIN


class MarkerDetector(object):

#        while(True):

         def __init__(self):
```

```
        self.image_sub = rospy.Subscriber("/flytos/flytcam/image_capture", Image, self.camera_callback)

        self.bridge_object = CvBridge()

        self.image_pub = rospy.Publisher("/processed_image", Image, queue_size=1)


    def camera_callback(self,data):

        try:

#            cv_image = self.bridge_object.imgmsg_to_cv2(data, desired_encoding="bgr8")

#            cap = cv_image


#            while(True):

            # We select bgr8 because its the OpenCV encoding by default

            cv_image = self.bridge_object.imgmsg_to_cv2(data, desired_encoding="bgr8")


            corners, ids, rejected = aruco.detectMarkers(image=cv_image, dictionary=aruco_dict, parameters=parameters,

                           cameraMatrix=camera_matrix, distCoeff=camera_distortion)


            if ids is not None and ids[0] == id_to_find:


                ret = aruco.estimatePoseSingleMarkers(corners, marker_size, camera_matrix, camera_distortion)
```

```
                              rvec, tvec = ret[0][0,0,:], ret[1][0,0,:]

                              aruco.drawDetectedMarkers(cv_image, corners)

                              aruco.drawAxis(cv_image, camera_matrix,
camera_distortion, rvec, tvec, 10)

                              str_position = "MARKER Position x=%4.0f  y=%4.0f
z=%4.0f"%(tvec[0], tvec[1], tvec[2])

                              cv2.putText(cv_image, str_position, (0, 100), font, 1, (0,
255, 0), 2, cv2.LINE_AA)

                              R_ct=np.matrix(cv2.Rodrigues(rvec)[0])
                              R_tc=R_ct.T

        roll_marker,pitch_marker,yaw_marker=rotationMatrixToEulerAngles(R_flip*R_t
c)

                              str_attitude = "MARKER Attitude r=%4.0f  p=%4.0f
y=%4.0f"%(math.degrees(roll_marker),math.degrees(pitch_marker),math.degrees(y
aw_marker))
```

```
                                cv2.putText(cv_image, str_attitude, (0, 150), font, 1, (0,
255, 0), 2, cv2.LINE_AA)

                                pos_camera = -R_tc*np.matrix(tvec).T

                                str_position = "CAMERA Position x=%4.0f  y=%4.0f
z=%4.0f"%(pos_camera[0], pos_camera[1], pos_camera[2])

                                cv2.putText(cv_image, str_position, (0, 200), font, 1, (0,
255, 0), 2, cv2.LINE_AA)

                                roll_camera, pitch_camera, yaw_camera =
rotationMatrixToEulerAngles(R_flip*R_tc)

                                str_attitude = "CAMERA Attitude r=%4.0f  p=%4.0f
y=%4.0f"%(math.degrees(roll_camera),math.degrees(pitch_camera),
                      math.degrees(yaw_camera))

                                cv2.putText(cv_image, str_attitude, (0, 250), font, 1, (0,
255, 0), 2, cv2.LINE_AA)

                        #cv2.imshow('Frame_1',cv_image)

                        #-- Create Image to Publish #
                        msg = self.bridge_object.cv2_to_imgmsg(cv_image, "bgr8")
                        self.image_pub.publish(msg)

                        #rate.sleep()


                        if cv2.waitKey(1) & 0xFF == ord('q'):

#                               break

                                cv2.destroyAllWindows()


                except CvBridgeError as e:

                                print(e)


#-- Here we crop the received image

#               height, width, channels = cv_image.shape

#               descentre = 160

#               rows_to_watch = 60

#               crop_img                                                        =
cv_image[(height)/2+descentre:(height)/2+(descentre+rows_to_watch)][1:width]

#               hsv = cv2.cvtColor(crop_img, cv2.COLOR_BGR2HSV)


def main():


        rospy.init_node('marker_detecting_node', anonymous=True)

        marker_detector_object = MarkerDetector()


        try:
                rospy.spin()
        except KeyboardInterrupt:
                print("Shutting down")


if __name__ == '__main__':
        main()
```

‘**PrecisionLandingTest.py**’:

```
#!/usr/bin/env python
import time
from flyt_python import api
import rospy
from sensor_msgs.msg import Image
from std_msgs.msg import Float32,Float32MultiArray
from rospy.numpy_msg import numpy_msg
from cv_bridge import CvBridge, CvBridgeError
import cv2
import numpy as np
import cv2.aruco as aruco
import sys, time, math
import os
import arucoTest
from simple_pid import PID

drone = api.navigation(timeout=120000)  # instance of flyt droneigation class

# at least 3sec sleep time for the drone interface to initialize properly
time.sleep(3)

drone.take_off(5.0)
#print data
drone.position_hold()
time.sleep(10)



class Controller(object):
        def __init__(self):

                self.pose_data_sub = rospy.Subscriber("/pose_data", Float32MultiArray,
self.control_callback)

        def control_callback(self,data):

#               for i in range (10):
#               pass
                x_translation = data.data[3]
                print "x_translation = ", x_translation
                y_translation = data.data[4]
                print "y_translation = ", y_translation
                z_translation = data.data[5]
                print "z_translation = ", z_translation

                #-- PID parameters
                p = 0.0020              #0.0005, 0.0010
                i = 0.000003            #0.0, 0.000001
                d = 0.0                         #0.0



                #-- Z control parameters
                z_setpoint = 100
                z_range = 9
                z_upperpoint = z_setpoint + z_range
                z_lowerpoint = z_setpoint - z_range

                pid1 = PID(-p, i, d, setpoint=z_setpoint)

                z_error = z_setpoint - z_translation
#               z_actuation    = -z_error * p
                z_actuation     = pid1(z_translation)

                #-- X control parameters
                x_setpoint = -8
                x_range = 9
                x_upperpoint = x_setpoint + x_range
                x_lowerpoint = x_setpoint - x_range

                pid2 = PID(-p, i, d, setpoint=x_setpoint)

                x_error = x_setpoint - x_translation
#               x_actuation    = -x_error * p
                x_actuation     = pid2(x_translation)

                #-- Y control parameter
                y_setpoint = -6
                y_range = 9
                y_upperpoint = y_setpoint + y_range
                y_lowerpoint = y_setpoint - y_range

                pid3 = PID(p, i, d, setpoint=y_setpoint)

                y_error = y_setpoint - y_translation
#               y_actuation    = y_error * p
                y_actuation     = pid3(y_translation)


                if z_translation > z_lowerpoint and z_translation < z_upperpoint:
#                       drone.position_hold()
                        if x_translation > x_lowerpoint and x_translation <
x_upperpoint:
#                               drone.position_hold()
                                if y_translation > y_lowerpoint and y_translation <
y_upperpoint:
                                        print "y is within range."
#                                       drone.position_hold()
                                        rospy.signal_shutdown("Finished!")
                                else:
                                        drone.velocity_set(y_actuation, 0, 0,
body_frame=True)
                        else:
                                drone.velocity_set(0, x_actuation, 0, body_frame=True)
                else:
                        drone.velocity_set(0, 0, z_actuation, body_frame=True)



def stop_function():
        print "\n\nThis happens when I shut the program down.\n"
        drone.position_hold()
        time.sleep(3)

        print 'Landing...'

        drone.land(async=False)
        drone.disconnect()    # shutdown the instance

def check(x_error):
        print "Checking..."
        print x_error



def main():

        rospy.init_node('controller_node', anonymous=True)
        controller_object = Controller()


        try:
                rospy.spin()
#               time.sleep(1)
                stop_function()

        except KeyboardInterrupt:
                print("Shutting down")

if __name__ == '__main__':
        main()
```

Figures:

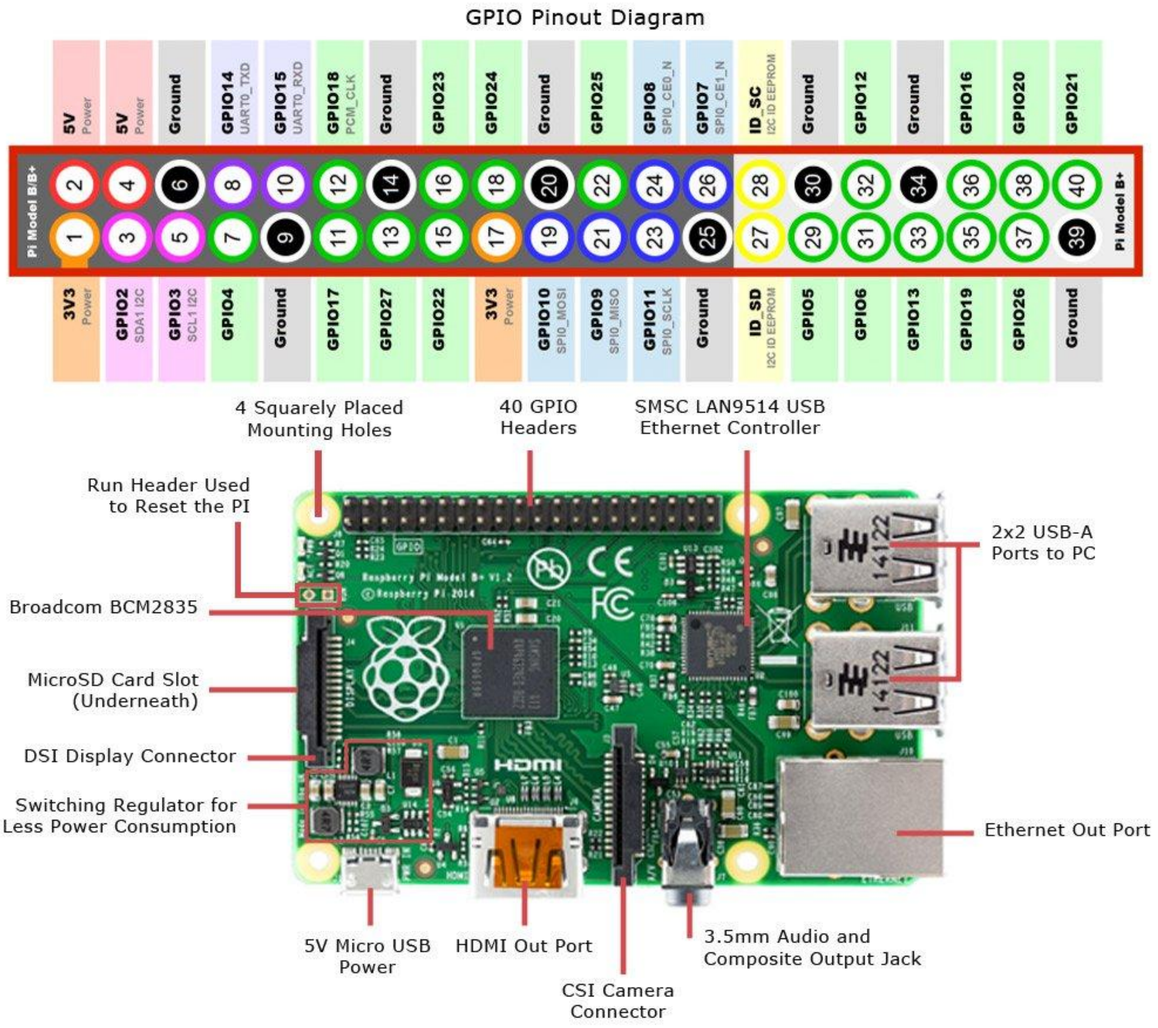


Figure 2: Labelled Raspberry Pi Model 3B board with GPIO pinouts [37]

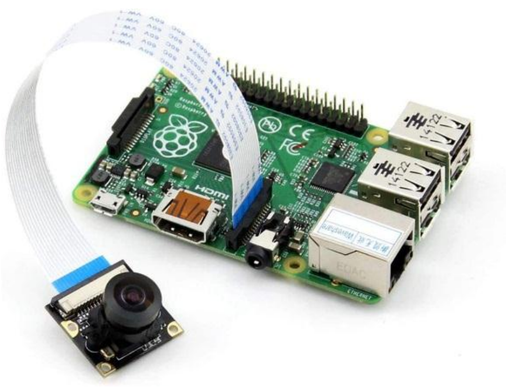


Figure 3: RPi Fisheye Lens Camera attached to a RPi CSI-interface [38]

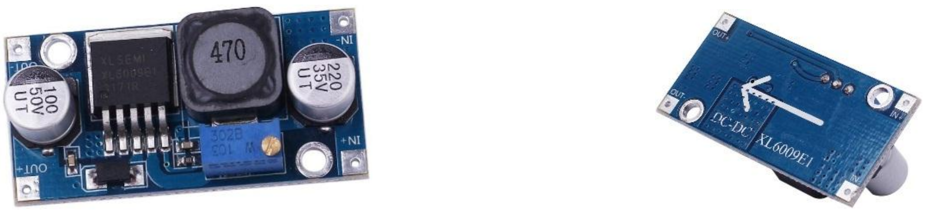


Figure 6: Voltage-regulator Module (top-view and bottom-view) [39]