POLITECNICO DI TORINO

Master's Degree in Mechatronic Engineering

Master's Degree Thesis

Autonomous Precision Landing for UAVs

Supervisors
Prof. Alessandro RIZZO
Dr. Stefano PRIMATESTA
Dr. Orlando TOVAR ORDOÑEZ

Candidate
Claudio Roberto DE CEGLIA

December 2022

Abstract

In modern industry, automated technologies are particularly useful in simplifying several applications. This benefit is emphasized for performing dangerous and difficult-to-handle operations. Especially, Unmanned Aerial Vehicles (UAVs) are developed for operations without a pilot on board and can be used in contexts where human intervention is not directly possible. These types of vehicles have several features including safety, rapid data extraction, and accessibility to remote locations. These also enable new application opportunities in both indoor and outdoor environments (such as emergency situations following a catastrophic event, search and rescue missions, or even aerospace applications). All these case studies share the requirement of using fully autonomous aerial vehicles in such a way as to, partially or totally, eliminate human intervention. In addition, there are many benefits such as speed, safety, efficiency, low cost, quality, and accuracy as a result of the ever-increasing growth in market demand in recent decades.

The present master thesis work aims to realize an architecture of an octocopter drone and its autonomous behaviour in the FIXIT project. In particular, regarding the precision landing of the UAV on board the rover. The FIXIT project is being developed by the Competence Industry of Manufacturing 4.0, located in Turin, Italy. The drone is an unmanned system capable of flying in industrial environments, both outdoors and indoors by managing specific missions, collecting and processing data, performing a stable and autonomous flight in any scenario thanks to an integrated obstacle avoidance algorithm, and, at the end of the task, perform a precision landing on-board an autonomous ground robot. Among the many solutions currently available to perform a precision landing, some use optical guidance systems, predictive control model methods, computer vision-based recognition systems, and systems using GNSS/RTK or UWB positioning. The solutions analysed, have considerable errors in the landing phase that would cause a failed landing. In order to provide a precision landing with an adequate accuracy, a combined approach of the above solutions, considerably improves the accuracy.

At the preliminary stage, the adopted solution uses two localization systems having at most 10 cm of error, which is considered an optimal threshold for the project. In the outdoor environment, GNSS positioning with a Real Time Kinematics system was used, which allows a correction factor to be sent from the ground station to the moving aircraft, which will improve the data received from the satellite; in the indoor environment, on the other hand, UWB technology with a tag and a POZYX anchor system was chosen, which allows its localization in the absence of signal from GNSS satellites. Once the accurate location was obtained, a computer-vision based system was developed, which uses a camera to identify a unique marker placed on board the rover. Following the identification of the marker, the landing phase will be carried out by keeping the marker in the field of view of the on-board camera; this will allow it to remain on the landing site during the deployment even in the presence of external disturbances that could displace it from the final target.

Table of Contents

List of Figures

Chapter 1

Introduction and State of the Art

1.1 UAV applications

Nowadays, in the industrial environment, the automated technologies are very useful to simplify many applications part of modern society that may be dangerous or difficult to be managed manually. In particular, the Unmanned Aerial Vehicles (UAVs) are intended for operation without an onboard pilot and are used in those contexts where the human intervention is not possible directly. These types of vehicles have many capabilities such as secure, rapid extraction of data and accessibility to remote locations and these ones allow still new opportunities for drone applications both in indoor and outdoor solutions (like emergency situations after a catastrophic event, search and rescue missions and, also, aerospace applications). All these applications share the desire to use highly autonomous aircraft to eliminate partially or totally the human intervention. Moreover, there are a lot of benefits such as swiftness, safety, cost efficiency, quality and precision as the result of the rapid growth in market's requests in the last few decades.

The present master thesis is aimed to the autonomous behavior of the FIXIT's drone in figure (1.1), in particular, about the precision landing of the UAV on the rover. The FIXIT is a project developed by CIM 4.0, the Competence Industry of Manufacturing 4.0, which is based in Turin. The drone consists of an UAV that can fly in an industrial environment, both in indoor and outdoor situations: it can manage specific missions, collecting and elaborating data, providing a stable and automated flight in every scenario with an integrated obstacle avoidance algorithm and, when it completes the task, has to land on an autonomous mobile robot.

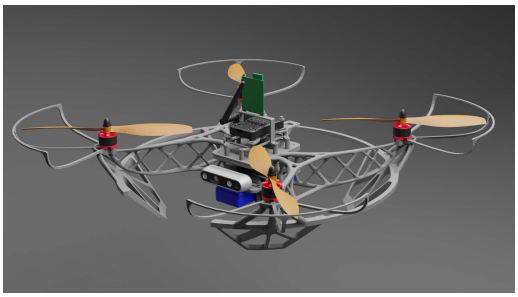


Figure 1.1: Prototype of FIXIT drone final design

1.2 DRONE Architecture

1.2.1 Main characteristics

The actual testing drone in figure (1.2) has a quadcopter configuration using only brushless motors. The brushless DC electric motors are synchronous motors that use a direct current (DC) power supply; they use an electronic controller to switch DC currents to the motor windings producing rotating magnetic fields which the permanent magnet rotor follows. Fundamental is the choose of coherent components in our application. The motors are chosen on the base of desired thrust/weight ratio: this value is influenced by the final payload, which is the overall weight at the take-off (by including also battery and sensors). Then, the correct ESC (Electronic Speed Controller) must be connected to each motor (or, eventually, is possible to use a single 4in1 ESC for all the motors): these are devices that allow drone flight controllers to control and adjust the speed of the aircraft's electric motors. A signal from the flight controller causes the ESC to raise or lower the voltage to the motor as required, thus changing the speed of the propeller. The LiPo battery is chosen in order to be able to provide the correct amount of current to all the connected devices: it is defined by two parameters, the discharge rating C and its capacity in mAh. Then, knowing the dimension of the frame we can choose the diameter for our propellers.

Figure 1.2: Testing drone

The UAV is composed by several sensors and systems that consents many features, like autonomous driving system, precise localization in both outdoor and indoor environments and obstacle avoidance functionality.

1.2.2 RTK GPS

The element 1 in the figure (1.2) is an RTK GPS receiver/antenna.

Real-time kinematic positioning (RTK) is the application of surveying to correct for common errors in current satellite navigation (GNSS) systems. It uses measurements of the phase of the signal's carrier wave in addition to the information content of the signal and relies on a single reference station or interpolated virtual station to provide real-time corrections, providing up to centimetre-level accuracy of the moving station. RTK systems use a single base-station receiver and a number of mobile units. The base station re-broadcasts the phase of the carrier that it observes, and the mobile units compare their own phase measurements with the one received from the base station; this allows the units to calculate their relative position to within millimeters. In our application, we decide to use a TAOGLAS HP5010A in figure (1.3 a).

(a) RTK GPS **(b)** UWB Module

Figure 1.3: Outdoor/Indoor Positioning Devices

1.2.3 UWB antenna

The element 2 in figure (1.2) is an Ultra Wide Band module.

UWB is a technology for transmitting information across a wide bandwidth (>500 MHz). This allows for the transmission of a large amount of signal energy without interfering with conventional narrowband and carrier wave transmission in the same frequency band. UWB is useful for real-time location systems, and its precision capabilities and low power make it well-suited for radio-frequency-sensitive environments. UWB is also useful for peer-to-peer fine ranging, which allows many applications based on relative distance between two entities. So, it is used in indoor environment where the GNSS navigation is not available due to the absence of signal from satellites and the presence of ferromagnetic disturbances. In our application, we decide to use a DECAWAVE DWM1001C module in figure (1.3 b) inside a white protection case. Ultra Wide Band represents a good solution, but it does not always return perfect results. In order to improve the measurements of the position, a sensors fusion is needed.

1.2.4 PIXHAWK Autopilot

The element 3 in figure (1.2) is a Pixhawk Board. Pixhawk autopilot is an open-source autopilot system oriented that allows a remotely piloted aircraft to be flown out of sight. The benefits of the Pixhawk system include integrated multithreading, a Unix/Linux-like programming environment, completely new autopilot functions such as sophisticated scripting of missions and flight behaviour, and a custom PX4 driver layer ensuring tight timing across all processes. These advanced capabilities ensure that there are no limitations to autonomous vehicle. Moreover, all hardware and software is open-source and freely available to anyone. In our application, we decide to use a PIXHAWK 2.4.8 in figure (1.4).

Figure 1.4: Pixhawk 2.4.8

For what the autopilot software is concerned, the choice is made between PX4 and ArduPilot. They are very similar, however the main two differences are that Arudpilot is older and, as a consequence, with a slightly more progressed and stable code. The second big difference is the license: ArduPilot has a GPL (General Public License) one, while PX4 has the BSD (Berkeley Source Distribution) one. Thus, every change made to the ArduPilot code must be push on Github and it becomes public, while the code cloned by PX4 can be maintained private. The latter is also one of the reasons for which the ArduPilot code is more thorough between the two. So that, the drone is controlled by the PixHawk 2.4.8 with Ardupilot. Moreover, inside the Pixhawk, there is also the Inertial Measurement Unit (IMU): it is based on the accelerometers and gyroscopes that are used to detect rotation and movements around the three main axes, in particular it is possible to define the angles for yaw, roll and pitch. The flight controller hardware has also a barometer.

However, since the 2.4.8 version has not a very powerful processor, an on-board companion computer is added, used to handle the Intel RealSense depth camera and the UWB module.

1.2.5 Companion Computer

The element 4 in figure (1.2) is the Companion Computer Jetson Nano.

The CC is an additional unit used to obtain more power in terms of processing data and instructions in real time. This is fundamental during flight, especially when autonomous behavior is needed. The used CC is connected and communicates, with the Pixhawk board, via MAVLink protocol over a serial connection. The CC software refers to the programs and tools that run on it: they can read, understand and run the telemetry data. Since the Robotic Operating System (ROS) is very common and flexible, it is chosen for this use's case.

In our application, we decide to use the Nvidia Jetson Nano in figure (1.5).

Figure 1.5: Nvidia Jetson Nano

1.2.6 Depth Camera

The element 5 in figure (1.2) is the Depth camera Intel RealSense.

The Intel RealSense is an RGB camera with a stereoscopic depth technology able to measure distances and thus it is considered as a reliable component for the obstacle avoidance. The camera has its own microprocessor and IMU, in this way it is able to understand its position and orientation. The combination of a wide field of view and global shutter sensor make it a good solution for applications such as robotic navigation and object recognition. The wider field of view allows a single camera to cover more area resulting in less "blind spots". The global shutter sensors provide great low-light sensitivity allowing robots to navigate spaces with the lights off.

In our application, we decide to use an Intel RealSense D435i in figure (1.6).

Figure 1.6: Intel RealSense D435i Depth Camera

1.2.7 Architecture and Hardware Updates

Frame Configuration

Since in our application, the total payload is bigger than the maximum offered by the thrust/weight ratio, in the past experiences, there are some stability problems and thrust loss. So, it is necessary to redefine some parameters that allow to improve the ratio and stability during the flight. To obtain this result, it is possible to use more powerful motors that guarantee a bigger thrust; this may leads to higher energy consumption and faster battery discharge, so less autonomy. Another solution is to implement an hexacopter or, even, octacopter frame configuration: the better distributed weight for each arm of the drone, allows a better stability during its utilization and more total thrust; as a disadvantage, the frame dimension should be increased to guarantee the minimum distance between propellers, so becomes difficult to flight in indoor environments. The chosen solution, is to maintain the actual quadcopter concept but in coaxial configuration: this means that, for each arm of the drone, there are two motors that rotate in counter-rotating way, as in figure (1.7). The proposed solution, guarantees almost the same efficiency, in terms of thrust, and stability obtained from the octacopter design by maintaining the same dimension of the drone; moreover, with this configuration, the octa-copter drone is capable of lifting about 4kg total, thus 1.3kg extra to the weight of the drone itself. Another advantage that can be achieved in the forthcoming improvements concerns battery sizing; in fact, the current batteries have been chosen as a result of the quadcopter configuration. By doubling the motors per arm, however, they will share the power required to lift the same weight and this leads into less discharge factor required by the battery and therefore less weight of the battery itself. In future upgrades to the drone, therefore, it will be possible to correctly size the battery used in such a way as to avoid carrying unused weight and oversizing the batteries, but above all to guarantee greater flight autonomy.

Figure 1.7: Example of coaxial configuration Quadcopter

Magnetometer

Another improvement proposed, is to use an external magnetometer (in addition to the Pixhawk'one). Despite the fact that, inertial sensors provide good quality for the collection of data, regarding the drone orientation and movement, the implementation of an additional magnetometer is essential for precise heading measurements. In this application, it is considered the RM3100 magnetometer, in figure (1.8).

Figure 1.8: RM3100 Magnetometer

It provides no drift, low noise, high sensitivity, no hysteresis, and it can be integrated with I2C or SPI interfaces. Although it represents a good element for outdoor missions, indoor it can show some problems due to ferromagnetic material. Furthermore, is shown the dependencies of height in the indoor positioning precision; such uncertainties must be taken into account during the analysis and validation of the sensor accuracy.

Lidar

To get better identification of the environment, it is useful to add LIDAR sensor: it makes the distance measurement from the objects: its functional principle is based on time of flight (ToF), which consists of sending a laser in a narrow beam and measuring the time taken by the pulse to be reflected by surrounding objects. The previous implementation foresees two of these sensors, one on top of the drone, and one on the bottom in order to obtain the height of uav in indoor, by measuring the distance from the ceiling and from the floor. Building up the accuracy, it is possible to use just one lidar, in particular the Benewake TFmini Lidar in figure (1.9), at the bottom to measure the distance from the ground. It has also a good integration with ArduPilot and is very thin in terms of weight (only 5g).

Figure 1.9: Benewake TFmini Lidar

Optional sensors

The previous solutions are very important to obtain the precise indoor/outdoor localization and stability during the flight. Besides, it can be possible to add other sensors to improve the localization and cover the blind spots. In fact, by adding ultrasonic sensors, on both sides of the drone and on rear, is possible to recognize the presence of obstacles that the camera cannot see, since they do not come from the front. For example, the cheap sensor HC SR-04, in figure (1.10) emits high frequency sound wave (40 kHz) via one of its piezoelectric transducers, detects the returning pulse (echo) and converts it to a proportional voltage variation. The accuracy depends on the light conditions, the absorption of reflecting material and, also, noise, temperature and humidity. Due to these drawbacks, the sensor is not use as main device for obstacle avoidance nor for height measure.

Figure 1.10: Ultrasonic Sensor HC SR-04

At the end of drone architecture, the following step is analyze how to achieve the precision landing and which technologies, hardware and software, can be useful to realize it correctly. So, in the next section there will be analyzed the solutions present in the state of art.

1.3 Literature Review

Several analyses are done to know and understand which are the different solutions in order to obtain the accurate landing of an UAV on the ground or on a landing station. These solutions present many methods to reach the aim, like Vision-based recognition systems, Global positioning systems, Mathematical approaches, Optical guidance systems, Composite or fiducial landmark methods, Model predictive control methods, RTK positioning approaches, etc. Most of the solutions, in literature, are developed by means of vision-based architecture since this is, in many uses case, the best solution which gives good results in terms of accuracy and precision landing. Some approaches prefer the faster solution, by using only GNSS positioning or, alternatively, a differential GNSS positioning to reduce measurement errors. Since, the past experiences describe the presence of many drawbacks due to lightning conditions, environment noises and disturbances, technology limitations (like the absence of signal in indoor for the applications which use the GNSS positioning), some of the cited solutions are implemented simultaneously to prevent and cover all the blind spots that a single technology may have. In this way, it is possible to reach the desired result in most of the cases.

1.3.1 Vision-based recognition systems

In the solution proposed by [1], a visible light camera integrated with a Digital Signal Processing (DSP) processor is installed on the UAV and an optical filter is fixed in front of the camera lens. In addition, four infrared light-emitting diode (LED) lamps are placed behind ideal landing site on the runway. In this way, the infrared lamps in the image are distinct even if the image background is complicated. The function of the camera is to capture images of the infrared lamps, and the function of the DSP processor is to process the images, calculate pose parameters of the UAV and export the calculation results. Finally, high precision space position of the UAV can be calculated according to the installation relationship between the camera and the UAV. This solution leads to a good precision landing but, as a disadvantage, requires the presence of infrared lamps at the landing site which, in our case, is a mobile rover; due to its size, it is difficult to add further devices that would increase both the computational cost and the hardware needed to add to the rover.

In the study presented by [2], an autonomous landing point retrieval method for UAV in unknown environment is proposed. Firstly, plane extraction is carried out on the point cloud of real-time three-dimensional reconstruction of binocular. The extracted results are mapped into two-dimensional images, and the landing point is identified by using random forest classifier. Finally, the UAV flight sampling at a certain height and the sampling results are discriminated. The results show that the UAV can achieve the perception of autonomous landing, but this solution requires a high-definition camera pointing at the floor and an on-board computer that processes the images taken in order to understand whether it is possible to initiate the landing phase.

The same hardware requirements are present in [3], where the application is the autonomous landing of UAV on a ship. The solution consists of two parts, a sensor

framework to estimate the state of quadrotor using vision based approach and a controller design which generates appropriate actuator commands. A computer vision approach is proposed which tracks the pre-specified oriented roundel object continuously while maintaining a fixed distance from the roundel and also simultaneously keeping it approximately in the center of the image plane. For stable position estimation system, an oriented roundel is used as an external reference. The position estimation is done on the image pattern coordinates, allowing hovering over a fixed position, and takeoff and landing on the oriented roundel. For this whole process, the coordinates of roundel are calculated and fed to the controller. Quadrotor's relative pose is estimated through dead reckoning and control approach is implemented which seeks for full autonomy of the robot, by considering only internal sensors and processing unit. The limitation of the present algorithm to track the moving object is limited to 1.5 meter altitude. This can be removed by changing the algorithms and a better camera.

1.3.2 Fiducial/composite landmark systems

In the paper proposed by [4], it is presented a quadrotor system capable of autonomously landing on a moving target using only onboard sensing and computing without a-priori knowledge about the location of the moving landing target. The system detects the landing target using an onboard camera that detects an already known marker on target. The correct result is reached using two on board computers: the first one for the downward-looking camera and the inclined camera and, the second one, for data recording and rapid prototyping.

In the study carried out by [5], a special design for the marker to be placed on the landing site was designed. This is because, in previous experiments, it was seen that the approach of the drone to the landing plane resulted in the complete marker leaving the field of view and, therefore, recognition was lost. The solution presents a concentric triangular marker that is recognizable at different distances allowing a correct landing phase.

The solution performed by [6], presents a combined system from a composite marker (a notched ring with an R2D landmark inside) and a prediction system for landing on a moving site. Thanks to the fusion of GPS and IMU sensors it is possible to estimate the pose of the MAV while, through an encoder and the R2D landmark, it is possible to estimate the pose of the landing platform. By merging these two estimates, it is possible to implement a landing strategy and plan a trajectory that will be monitored through a controller.

The paper by [7], proposes a vision-based target following and landing system for a quadrotor vehicle on a moving platform. The system is consisted with vision-based landing site detection and locating algorithm using an omnidirectional lens. Latest smartphone was attached on the UAV and served as an on-board image acquisition and process unit. In order to land on the specific visual landing pad in outdoors, fisheye lens and its calibration model helped the shrinking FOV problem while descending above the visual pad. Measurements from the omnidirectional camera are combined with a proper dynamic model in order to estimate position and velocity of the moving platform. An adaptive control scheme was implemented

on the flight computer to deal with unknown disturbances in outdoor environment.

In the study presented by [8], is used a multiple markers landing pad, which is a kind of simplified Apriltags. Small markers are overlaid on the large marker, make it possible to have a wide detection range in a limited pad area. In this way, marker recognition is possible from both distant and close-up positions.

In the solution by [9], for the autonomous landing task, a novel landing pad was designed for robust detection, ensuring the detectability from both high and low altitudes. A 3D points cluster algorithm for pose estimation was presented to solve the problem of mirror effect and occasional misidentification. A simplified dynamic model for quadrotors in landing phase was proposed and thus a PD controller was designed accordingly to ensure the landing on an either static or moving pad.

In the method of [10], QR code image with strong error correction ability is adopted as the cooperation target, Unreal Engine 4 (UE4) is used to build UAV landing simulation scene, and the UnrealCV plug-in is installed to enable UE4 to communicate with external programs. Finally, an object detection algorithm based on deep learning is implemented to detect the cooperative target position on the runway. So, in the process of autonomous landing, QR image is adopted as the cooperative target, combined with the object detection algorithm YOLOv3 to detect the cooperative object position placed on the runway, the UAV is then guided to an accurate and robust autonomous landing.

The paper proposed by [11], presents an autonomous landing method for unmanned aerial vehicles, aiming to address those situations in which the landing pad is the deck of a ship. Fiducial markers are used to obtain the six-degrees of freedom (DOF) relative-pose of the UAV to the landing pad. In order to compensate interruptions of the video stream, an extended Kalman filter (EKF) is used to estimate the ship's current position with reference to its last known one, just using the odometry and the inertial data. The EKF performs well enough in providing accurate information to direct the UAV in proximity of the other vehicle such that the marker becomes visible again. Due to the use of inertial measurements only in the data fusion process, this solution can be adopted also in indoor navigation scenarios, when a global positioning system is not available.

In the work presented by [12], a fully autonomous vision-based system addresses the wind disturbances by tightly coupling the localization, planning, and control, thereby enabling fast and accurate landing on a moving platform. The platform's position, orientation, and velocity are estimated by an extended Kalman filter using simulated GPS measurements when the quadrotor-platform distance is large, and by a visual fiducial system when the platform is nearby. The landing trajectory is computed online using receding horizon control and is followed by a boundary layer sliding controller that provides tracking performance guarantees in the presence of unknown, but bounded, disturbances. To improve the performance, the characteristics of the turbulent conditions are accounted for in the controller. The landing trajectory is fast, direct, and does not require hovering over the platform, as is typical of most stateof-the-art approaches.

The paper by [13], presents an approach for precise UAV landing using visual sensory data. A new type of fiducial marker called embedded ArUco (e-ArUco) was developed specially for a task of a robust marker detection for a wide range

of distances. E-ArUco markers are based on original ArUco markers approach and require only ArUco detection algorithms.

1.3.3 Optical guidance systems

In the solution carried out by [14], it is used an optical guidance architecture to generate the guidance signal and then control the landing of the UAV along an ideal trajectory. Optical guidance is an automatic UAV landing system which can measure the position of the UAV continuously. These measurements are transmitted to flight control system and will be used as feedback in the speed or position control loop. This method is only functional in cases where the landing point is fixed in time, such as airport runways.

In the study of [15], it is developed an algorithm to estimate the target position with respect to the flying vehicle through an infrared camera and beacon. Next, by using Kalman filter theory, is it possible to estimate the velocity of the target from its position information.

1.3.4 Mathematical approaches

In the paper proposed by [16], it is made an intense observations of the data concerning the autonomous landing approach such as the intersection point between the two moving bodies, the position of the platform/UAV and the inclination angle required to land. Here, a mathematical approach to this problem is presented in the X-Y plane based on the inclination angle and state position of UAV during the landing procedure. So, to achieve an autonomous landing robust control algorithm on a moving platform, the inclination angle is simultaneously calculated according to the platform parameters. The actual and predicted intersection points between the UAV and platform and also the position of a UAV during this process including inclination angle are used in planning an accurate and precise landing trajectory.

In the work of [17], it is presented an autonomous landing method that can be implemented on micro UAVs that require high-bandwidth feedback control loops for safe landing under various uncertainties and wind disturbances. The system architecture, includes dynamic modeling of the UAV with a gimbaled camera, implementation of a Kalman filter for optimal localization of the mobile platform, and development of model predictive control (MPC), for guidance of UAVs.

The solution presented by [18], contains a vision based autonomous landing control approach for UAVs. The 3D position of an unmanned helicopter is based on the homographies estimated of a known landmark. The translation and altitude estimation of the helicopter against the helipad position are the only information that is used to control the longitudinal, lateral and descend speeds of the vehicle. The control system approach consists in three Fuzzy controllers to manage the speeds of each 3D axis of the aircraft's coordinate system. Also in this case, the solution is valid when the landing spot is fixed in time.

The scope of the study by [19], is a proof-of concept methodology which uses a signal prediction algorithm to facilitate safer autonomous UAV-ship landings. This study uses laser ranging and detecting devices (LIDAR) in conjunction with a signal

prediction algorithm (SPA) to forecast when the ship motion is within safe landing limits. The results show that with the use of the SPA, the number of UAV landing attempts was decreased by an average of 2 attempts, per test case, when compared to a system that did not use an SPA. Moreover, the results indicate that with revised tuning of the SPA, the likelihood of a safe landing can be further improved.

In the solution by [20], is implemented a PI regulator used for the tracking problem while descending is made by controlling relative vertical velocity. A finite state machine (FSM) approach is chosen to manage multiple robot states and recover from failures. A software framework is also developed in order to manage general flight missions and, in this case, the landing assignment. So, it is proposed a simple but effective method for horizontal tracking by closing the internal controller with an external position loop (with target as reference) including a velocity feed-forward term (with target velocity as reference). On the other hand, descending is performed by forcing the relative vertical velocity to a reference, which is linearly decremented with respect to the distance from the landing surface until touchdown. The entire procedure was modeled with a FSM dividing the problem in different states and, this method, appeared to be convenient for the tasks where some unpredictable event may occur.

The paper by [21], explain a development of the automatic landing of a Micro Air Vehicle (MAV) on a moving vehicle. A Proportional Navigation controller was used for the long range approach, which subsequently transitioned into a PID controller at close range. A Kalman filter was used to estimate the position of the MAV relative to the landing pad by fusing together measurements from the MAV's onboard INS, a visual fiducial marker and a mobile phone.

1.3.5 Global positioning systems

In the paper proposed by [22], is estimated the position and velocity of the UAV by RTK positioning, the quaternion that describes its attitude, the carrier phase integer ambiguities related to both the attitude and position, and the accelerometer bias with a Kalman filter. The raw measurements were obtained from the ANavS Multi-Sensor RTK module with its 3 Multi-frequency, Multi-GNSS receivers and a MEMS-based Inertial Measurement Unit (IMU).

In the analysis presented by [23], the combination of GPS and visual recognition through the camera is exploited. Since the information obtained through GPS has an unacceptable error in precision landing, ultra-wideband ranging measurement is used to locate and approach the landing station. Then, once the UAV locates the marker, the GPS is deactivated and the UWB is activated which, along with the visual information, will result in an accurate landing phase.

In the study of [24], the precise position of the unmanned aerial vehicle at landing is provided by an image analysis where the specially designed landing platform is detected. All calculations for precision landing guidance are performed directly on board. An infrared (IR) camera is used as the main sensor for monitoring the IR light beam; so, the core of the landing system consists in a visual camera scanning the modulated IR beam to facilitate smart navigation onto a precisely landing spot. This aircraft guidance technique does not necessitate data transfer to and from the

landing station. More complex but expensive systems is RTK GPS which can be suitable solution to replace visual sensor.

1.3.6 Neural networks approaches

The solution presented by [25], uses a deep convolutional network based vision pipeline to detect the landing mark and estimate system state in real-time. The landing sequence is controlled by vehicle trajectory points using a linear predictive algorithm. The trajectory points are fed to a low level position controller and velocity controller.

1.4 Possible solution of precision landing

At the end of a thorough analysis of the state of the art, it is possible to set up a procedure to carry out a precise landing phase. The first step is to understand if the rover is stationary or moving; based on this state, the landing process will be more or less complex in computational terms and accurate. In this thesis work, the rover will be considered stationary and then the solution will be improved to make it work also for the moving situation. The second step to be analyzed is the localization of the landing point with respect to the drone: this phase will have to be developed exploiting the technologies already present on board the drone; in particular, the RTK gps for outdoor environments and the UWB for indoor ones. Both localization methods mentioned, guarantee high precision (range of a few centimeters) and therefore, in the third and last step, a not too complex technology will be needed for the correct identification of the landing point. From the previous studies, it has been seen that the use of a medium resolution camera, is sufficient to support the previous localization mechanisms, for the identification of a unique marker present on the rover that will be recognized in the first place, and used as a reference point of the landing spot. Then, the drone will position itself and stay on the marker, reducing little by little the power to the engines. To further support the camera, there will be a LIDAR that will allow to understand the distance between the drone and the landing spot and, at a certain height considered safe, it will be possible to turn off the engines and complete the landing.

Chapter 2

Building the new Drone

2.1 Drone Frame

The first step to be addressed in the construction phase of the new drone is the assembly of the new frame, in the figure (2.1). Unlike the most common frame on the market, the chosen frame is made of carbon fiber, so as not to increase the total weight of the drone; this choice was made in order to obtain a very strong chassis and, at the same time, with a similar weight to the previous one while increasing its diameter and the number of motors and ESCs on board. In addition, the frame chosen, allows manual folding of the arms so as to reduce the space during transport. Shown is the assembled frame that will be used in our application: a Readytosky ZD550 Quadcopter/Octacopter Carbon Fiber Frame. It is possible to view videos of the assembly at the following [26].

During the frame assembly phase, problems were encountered, which are described below regarding the following parts of the drone:

- Frame Plates;
- Space for the Battery;
- Space for the PCB distribution board;
- Space for the Intel Realsense camera and Jetson Nano companion computer.

First, it was seen that the frame in question, and especially the plates where to attach the motors, did not have the holes in the correct positions for housing the motors positioned upside down. This is because, the supplier specifies that the same frame can be used for an octacopter by buying plates similar to those for upward positioned motors. In our application, the owned plates were adapted by drilling them in the correct positions. This made it possible to mount the motors on both plates and complete the coaxial quadcopter. A second issue, which arose during the frame assembly phase, is related to the spaces dedicated to the PCB and the battery. For the first one, it was not possible to fit it in the space dedicated to it by default, because it is narrower than the width of the board designed ad-hoc for the drone. The second, on the other hand, turns out to be bulkier than the intended slot. In addition, there is the lack of a dedicated space where we can attach the

high-level controller (Jetson Nano) and the camera used for obstacles avoidance and precision landing. To solve the aforementioned problems, carbon fiber sheets of certain dimensions were purchased, drilled in the necessary places and used as housing for the distribution board, the cameras and the Jetson Nano. While the battery was placed in the housing originally designed for the PCB and secured with cable ties. An important observation, to be verified after the chassis assembly is completed, is that the slots where the motors are housed, are perfectly horizontal (parallel to a horizontal plane) so that the thrust generated by the motors is totally vertical and not tilted, which would cause thrust losses.

Figure 2.1: Readytosky ZD550 Frame 550mm Folding Carbon Fiber

2.2 Electronic components

After completing the frame assembly, the electronic components (e.g. motors, ESCs, distribution board, Pixhawk, ...) were assembled. The procedure followed is given at the following [27]. During the electronics assembly phase, several problems occurred as described below:

- Motors Screws;
- Positions of Motors Housing;
- Cables from ESCs to Pixhawk;
- PCB Extensions;
- Integrated Circuit on the PCB in Protection mode;
- Plastic Screws and Bolts;
- On-board Wiring.

Particular attention should be paid to the assembly of the motors on the add-on housings: on the official manufacturer site of the Readytosky 2212 920KV motors it is indicated that, the same are attached to the frame by means of 9mm hexagonal M3 screws; this information is partially correct, since these indications refer to a generic rigid plastic frame for a quadcopter, which has housings about 6mm thick. By using 9mm screws, therefore, about 3mm will enter the stator of the motor, which will allow it to function properly. In our case, however, the motor housings are 2mm thick so, using the indicated screws, about 7mm of screw will enter the stator, touching and bending the brushless motor windings. Following an initial power-up, the bent windings crossed by current will be damaged and will not ensure proper operation and rotation of the brushless motor. Basically, great attention must be paid to the length of the screws that are used, in relation to the thickness of the frame housings, to fix the motors since, they must absolutely not touch the motor windings otherwise they will irreversibly damage it. In our application, therefore, 5mm hexagonal M3 screws have been used, which allow the correct rotation of the motors and do not damage them.

A further check to be made on the frame used, is related to the position of the motor housings: in fact, they should be positioned at the ends of the arms and in a perfectly horizontal position so as to guarantee purely vertical thrust, as already described in the previous section; in this way, there will be no imbalance or loss of thrust during flight. A spirit level was used to verify correct positioning.

Another problem occurred with regard to the connection of the ESCs with the Pixhawk autopilot; in particular, it was noted that the ESCs used had a connection cable to the autopilot that was too short. To solve this issue, the factory dupont connectors were cut, the ground and voltage cables that are not needed were isolated, separately (they already come from the PBC), extensions were soldered to the signal cable (white cable) at the ends of which new female pins were crimped, later inserted into new Dupont connectors; this operation was performed only for

the ESCs connected to the front motors, whose initial cables did not reach the Pixhawk. Having finished, then, the modification to make the extension, it was possible to correctly connect all the ESCs to the ports on the back of the autopilot; all the cables were, then, tied with a cable tie to prevent the air displacement during the flight phase from causing them to move and could become entangled with the propellers or be damaged. Another issue regarding the ESCs used, is related to their type: they are ESCs usually intended for fixed-wing aircraft and also have the 5V cable in the Dupont connector. This cable turns out to be problematic as it has unnecessary power consumption due to the linear voltage regulator: in particular, they plan to deliver 5V each but, if someone has a lower/higher voltage, since they are placed in parallel, they try to regulate themselves and this results in dissipation to each other. In our application there is no need for this self-regulation and, to avoid unnecessary waste of energy, the red 5V wires were removed keeping only the signal wires (or at most, the ground and signal wires).

Because the location of the PCB turns out to be lower than that provided on the frame, extension cables for powering the ESCs were soldered onto it with XT30 connectors at the ends.

Another major problem encountered during the electronics assembly phase concerns the integrated circuit that provides the switch between charging mode (drone at rest and battery charging) and drone mode (drone in use and battery use). It was observed that, at the time of power on and connection to the battery, the PCB would short circuit/protect: this was due to the presence of capacitors on the ESCs that start out discharged and, at the time of power on, require a high peak current to charge them. The IC seeing this current spike would go into protection and block the whole circuit to avoid burning components. Even adding a second integrated in parallel (thus higher current carrying) did not bring solution to the issue at hand. Therefore, two alternative solutions present each on each drone were thought of. On one drone, the integrated that was sending the circuit into protection was short-circuited leaving the ESCs with capacitors while, on the second drone, the capacitors of the ESCs were removed and the integrated circuit allowing the two modes of operation at the PCB was left. Only in the first case, the solution worked and the PCB no longer shorted but maintained the correct voltage at the ends of the different outputs (about 12V for the ESCs and about 5V for the Pixhawk and the Jetson Nano). In the second one, instead, the absence of capacitors on the ESCs proved critical since they contain some of the charge needed to start the motors; Without the capacitors, in fact, the motors had difficulty starting and hiccupped during movement due to the voltage not being too stable/linear. An alternative solution was devised by placing a parallel of NTC resistors between the PCB and the battery; NTCs are resistors that change their operation according to their temperature: when cold, they behave as simple resistors thus reducing the current reaching the PCB while, when hot, they behave as a short circuit presenting no resistance and consequent no current reduction. This solution turned out to be invalid because the peak current was required by the capacitors of the ESCs and not by the battery, so it was necessary to place these resistors downstream of the integrated (between the integrated and the ESCs) and not upstream of it (between the integrated and the battery); for this reason, in future versions of the PCB Fixit, a soft start system

will be thought of that involves the presence of NTC resistors that once the peak current that sends the integrated into protection at power-on has been attenuated, will be shorted automatically. This solution allows us to maintain both the ability to switch modes of operation of the PCB and the presence of capacitors on the ESCs, which ensure a constant voltage in case there is an instant of voltage that is not too constant.

For the attachment of the PCB to the carbon fiber plate, it was initially thought to use metal spacers that would ensure rigidity of the structure once assembled. However, this turned out to be problematic in terms of conduction: in fact, the PCB being made of copper, conducts anywhere on the PCB, even in the holes for attachment to the frame. So by using metal spacers, it was seen that they conducted and overheated, burning the carbon fiber plate. To avoid this inconvenience, plastic screws and bolts were used between the PCB and the metal spacers. With this solution, a rigid structure and proper insulation was ensured to cause damage to the frame structure.

A final special focus, concerning the electronics on board the drone, is on the connecting cables between the sensors and the autopilot. In particular, with the exception of the cables already supplied by the manufacturers and working properly (buzzer cable, safety switch cable, GPS RTK cable, and radio control cable), many cables were configured and modified ad-hoc, following the indications on the sites of Ardupilot or the vendors of the sensor under analysis.

The first cable under analysis is the power cable to be connected between the Pixhawk's Power port and the 5V connector on the PCB (or possible Power Module): it has the configuration as shown in the link [28] and depicted in the image (2.2).

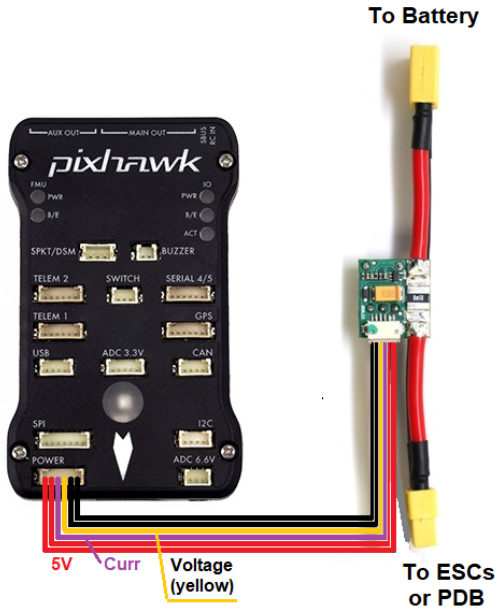
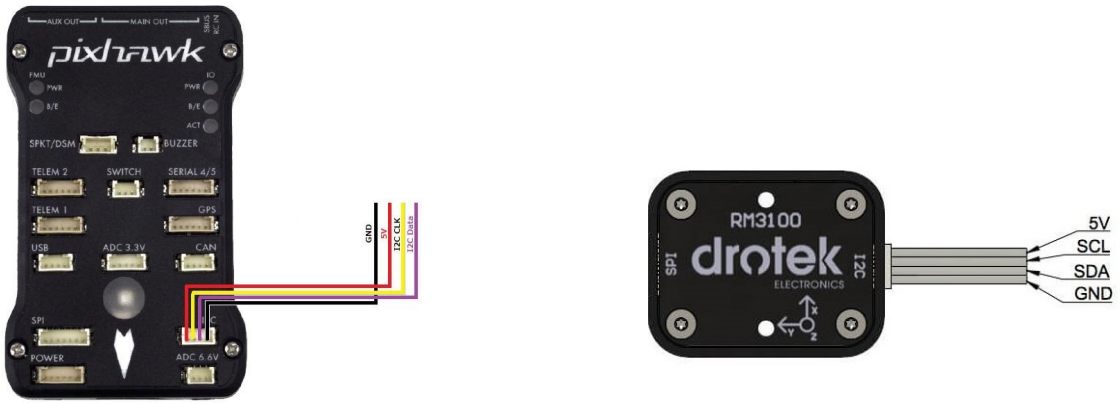


Figure 2.2: Configuration Cable between Pixhawk and Power Module

As can be seen in the figure, the cable configuration appears to be mirrored, and following the description, the 6-pin connector cable was modified as depicted.

Another connection cable that needs attention is the one that connects the

Drotek magnetometer to the Pixhawk's I2C port. It comes with an I2C connector that is not compatible with the one on the Pixhawk and therefore needs to be modified; however, you must pay attention to the position of the wires in the connector by following the diagrams in the figure (2.3) and making them match.



(a) Pixhawk I2C port **(b)** Drotek Pinout

Figure 2.3: I2C configuration between Pixhawk and Drotek Magnetometer

Attention must also be paid to the connection of the Air Telemetry Module, which will be connected to the Telem1 port of Pixhawk. It has 4 pins while the Telemetry 1 connector has 6 pins: two therefore will not be used and will remain empty (CTS and RTS ports). For proper cable configuration, the pin legend in the figure (2.4) will be followed:

(a) Pixhawk TELEM1 Port **(b)** Air Telemetry Module Pinout

Figure 2.4: TELEM1 configuration between Pixhawk and Air Telemetry Module

A final configuration concerns the SERIAL 4/5 port to which the TFmini Lidar will be connected according to the configuration described at the Ardupilot link [29]. As in the case of the Telem1 port, in this connector, two pins will not be used and the configuration will be as in the figure (2.5).

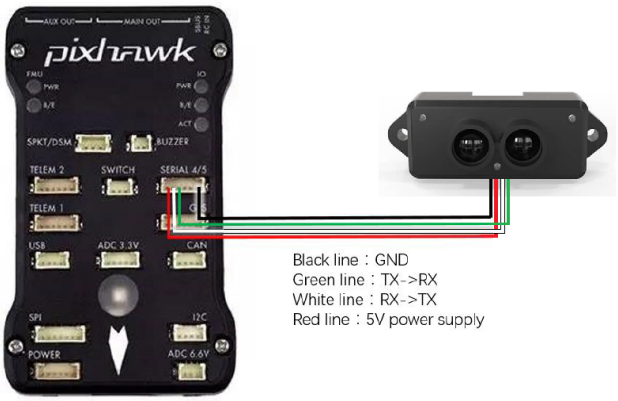


Figure 2.5: Configuration Cable between Pixhawk and Lidar TFmini

All remaining ports and configurations related to Pixhawk are shown on the official Ardupilot website at the following link [30].

2.3 First Calibration and Configuration

The last step, after building the chassis and assembling all the on-board electronics, concerns the first calibration and configuration of the drone by using Mission Planner software. The procedure followed is given at the following [31]. This last step is also divided into steps to be followed carefully to achieve proper and stable operation during flight:

- Install Firmware;
- Connect Mission Planner to the AutoPilot;
- Select Frame Type;
- Set the Initial Parameters;
- Calibrate Accelerometer;
- Calibrate Compass;
- Calibrate Radio Control System;
- Calibrate ESCs;
- Set the Flight Modes;
- Set FailSafe situation;
- Set Battery Monitor;
- Motor Test.

2.3.1 Firmware Installation

Even before connecting the autopilot to the pc using the mission planner, it is necessary to install the correct firmware on board the Pixhawk. So, in the Setup tab of the mission planner, we need to select "Copter V4.2.1 OFFICIAL" (or the newest one) by clicking on the frame we are using, i.e. that of the coaxial quadcopter, following the instructions, and completing the installation.

2.3.2 Connection to the Pixhawk AutoPilot

After installing the firmware, we proceed to connect the Mission Planner with the autopilot, either via the USB cable or via the ground radio telemetry module. Once the USB or Telemetry Radio is attached to the pc, Windows will automatically assign to the autopilot a COM port number; The appropriate data rate for the connection is also set (typically the USB connection data rate is 115200 and the radio connection rate is 57600). After selecting the port and data rate, clicking on the connect button, top right, we will connect to Pixhawk via Mavlink.

2.3.3 Frame Type and Initial Parameters

Once the connection is made, the Setup tab will update, showing the Mandatory Hardware section; the latter will contain much of the necessary configurations to be made. The first item indicates the Frame Type and, within it, we will go to verify that the frame type selected, matches the one in use. Next, we will check the information in the Initial Parameter Setup tab, checking or entering the propeller size (in inches), the battery type, the number of battery cells, and the voltage value per cell relative to the full charge/discharge of the battery.

2.3.4 Accelerometer and Compass Calibration

Moving on, the first actual calibration is carried out in the Accel Calibration tab. In this section, we will level the drone, setting offsets on the default accelerometer levels, placing the drone on a flat surface, and following the different positions of the drone as indicated (horizontal, on the left side, on the right side, etc.). On the next Compass tab, however, we will perform compass/magnetometer calibration. Before doing this calibration, we need to verify that the compass with the highest priority in the list, is the one we want to use and perhaps disable the others we do not want to use. After that, we start with the calibration, by making random rotational movements of the drone (possibly 360°), on all rotational axes (yaw, pitch and roll), so that the system will self-calibrate correctly.

2.3.5 Radio and ESCs Calibration

The calibrations conclude with these last two. First, to calibrate the Radio Control System, we will need to turn it on and connect it with the receiver on board the drone; if this procedure fails, we will need to either re-bind the remote control with the receiver or verify that the same remote control is connected in PPM (not PWM), as the receiver is connected to the Pixhawk via the PPM port. If the connection is successful, start the calibration into Radio Calibration tab and we will have to move all the remote control sticks, reaching the most extreme positions, to calibrate the limit points of the same sticks. Second, the calibration of ESCs is done in the ESC Calibration tab, where we will need to start it, disconnect the USB connection (if there is one) and the power supply via a battery, reattach the battery, and press the safety switch on board the drone; an audible confirmation will assure us that the calibration was successful.

2.3.6 Flight Modes, Failsafe and Battery Monitor Setup

These latter setups are to be carried out to keep the drone and flight missions safe, both for the drone and the surrounding environment. The first item, in the Flight Modes tab, is used to set the safe flight modes accessible via the three positions (high, middle, low) of Switch C on the radio control; in our application, three flight modes were set: Stabilize, PosHold and AltHold (in case of missing GPS signal) in Flight Mode 1, 4 and 6 respectively. Also, from the same radio control, the other switches have been enabled: with the receiver off on the drone, hold ok on the radio

control and you will enter the menu, go to System Setup and scroll down to Aux Switches; here the switches you want to use can be enabled and, increase the active channels, if not enough for the active switches (Roll, Pitch, Yaw and Throttle are the first 4 occupied channels, if it is desired to enable other switches you have to take into account that each "active" position corresponds to an occupied channel), once the change is made you hold down the cancel button to save and return to the previous menu. After that it is possible to go from the main menu to Functions Setup, Aux Channels and select the radio channel to which we want to associate the switch: in our application we have associated switch C to radio channel rc5, switch A to rc6, switch D to rc7 and switch B to rc8; holding Cancel on the radio control, we save the changes just made. Finally, the configuration in Mission Planner's full parameter list is completed by entering the values of the functions we want to achieve in that particular switch: in rc6 we put the command 31 which corresponds to Motor Emergency Stop that will stop the engines immediately (activate it only in case of emergency because it can seriously damage the aircraft), in rc5 we left the value 0 because the modes were associated in the Flight Modes window, in rc7 we entered the value 18 which corresponds to Land and in rc8 we can enter any value for the desired function (RTL, Smart RTL, Circle, etc.). With these configurations, the Safety Pilot can safely manage the flight phase and switch to manual mode should an emergency situation arise in autonomous flight. The second setting is used to set failsafe cases and, in our application, is set to land in case of low battery or critical voltage level. To properly configure the former situation, we set in the Optional Hardware->Battery Monitor tab the voltage and current value that the battery is delivering, using the multimeter and current clamp; With this configuration, voltage and current values can be monitored directly in the Data window of the Mission Planner, during the flight phase. An important observation should be made about the battery monitor. It was found, in fact, that the voltage reading with the motors on and the throttle at a value above 50%, was found to be wrong on Mission Planner. In the same, the voltage was found to have a drop of about 3/4V and this, could be attributable to some issue on the PCB, drone electronics, power module or even the autopilot voltage port. After numerous tests with an oscilloscope, with different batteries, with other autopilots and with, even, buffers on the voltage divider port of the Pixhawk it was verified that there was no problem at the circuit level. The misreading issue was caused at the Dupont connections on the back of the autopilot, coming from the ESCs; The ESCs used, in fact, turn out to be usable for both drones with brushless motors and fixed-wing aircraft having, in the Dupont connector, the white cable for the signal, the red cable of the 5V and the black cable of the ground. The red 5V cable was, however, removed from all connectors as it could cause extra power consumption on the battery; the voltage to the speed controllers comes from the PCB so this additional power supply can be removed. For the ESCs closest to the autopilot, those related to the motors on the back of the drone, both the white signal and ground cables were left. For ESCs further away from the autopilot (those of the motors on the front of the drone), an extension cable was made for the signal cable only; the ground turns out to be virtual and, therefore, no extension cable is needed for it. This was precisely the cause of the problem in the voltage reading on the Mission Planner battery monitor; the presence of the grounds in the rear ESCs

connectors only, caused an imbalance in the voltage when the circuit was under load. The solution to this problem was then to remove the ground from the Dupont connectors coming from the rear ESCs, and the reading was found to be correct. As a result of this, it was preferred to reduce the unused wires by isolating them separately for cleaner and clearer onboard wiring; also, at the rear of the autopilot, it was preferred to plug in the signals with a single 8-pin Dupont connector so that there were not 8 individual connectors. With correct on-screen voltage and current values, a failsafe can be set in case of low battery capacity. In addition, failsafes are also set in case of signal loss with the radio control or ground telemetry module (the latter solution is not recommended, since in the vast majority of cases, the ground station will be out of range of the UAV and, therefore, its activation is not recommended so as not to incur unanticipated failsafes). In the future, a geofence failsafe could also be added, i.e., if the drone were to leave a predefined geographical area, it will go into failsafe and perhaps land immediately or return to home position. All these additional configurations are described in the guide at the following [32].

2.3.7 Motor Test

The last step to be performed before conducting a first autonomous/manual flight mission is related to engine verification. Specifically, before flying, you will need to verify that the signal connections to the ESCs, on the back of the Pixhawk, match the correct numeration found on the Ardupilot website at the link [33]. On this page, referring to the diagram of the OctoQuad X8, you will also need to check the directions of rotation of the motors and their propellers. They will necessarily have to follow the direction of rotation in the figure (2.6). To see the current direction of rotation, we need to go to the Motor Test tab in Optional Hardware and set the throttle to, at least, 20% and click on "Test motor X" to start the rotation; if the motors do not turn in the correct direction, it will suffice to randomly reverse two of the three phases of the brushless motor so as to obtain the opposite direction of rotation. The ideal direction of motor rotation is also indicated by the color of the cap above the propeller; this color indicates the direction of reverse threading. In other words, the silver cap is present on motors manufactured for Counterclockwise rotation while the black cap is present on motors intended for Clockwise rotation: in both, the direction of the threading is opposite to the direction of rotation; this ensures that, during flight, the screws do not unscrew risking the propeller coming out of the housing on the motor.

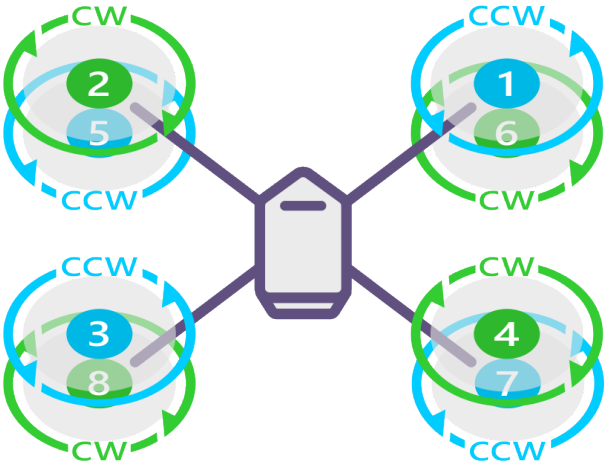


Figure 2.6: Motors Configuration for Octacopter

For the propellers, on the other hand, they will need to be observed laterally: if the propeller cutout has the top on the right, the propeller is set up for clockwise rotation, otherwise if it has the top on the left, the propeller is set up for counterclockwise rotation.

2.4 Planning flight missions with Mission Planner

After completing all safety calibrations and configurations, initial autonomous flight tests can be conducted using the Plan tab of Mission Planner. With this feature, it is possible to plan a mission that can be simple takeoff and land, but also by adding waypoints via GPS that the drone will follow, along the planned route. For planning using this tool, it is possible to see the following links [34].

2.5 Improvements and Updates on the Hardware

After numerous configurations and initial flight tests, it was necessary to go for hardware improvements; this ensures us a better performing device and devices that will continue to be supported in the future by developers. The changes made are as follows:

- Cube Orange Autopilot;
- Motors T-Motor AT2312 1150KV;
- Bashing Gens-Ace Battery 4S 5000mAh;
- Holybro SiK Telemetry Radio V3;
- Carbon Fiber Multistar Propeller 9x5.

2.5.1 Cube Orange Autopilot

The Cube autopilot, in the figure (2.7), is a further evolution of the Pixhawk autopilot. It is designed for commercial systems and manufacturers who wish to fully integrate a autopilot into their system. On top of the existing features of Pixhawk, it has the following enhancements:

- 3 sets of IMU sensors for extra redundancy;
- 2 sets of IMU are vibration-isolated mechanically, reducing the effect of frame vibration to state estimation;
- IMUs are temperature-controlled by onboard heating resistors, allowing optimum working temperature of IMUs;
- The entire flight management unit (FMU) and inertial management unit (IMU) are housed in a relatively small form factor (a cube). All inputs and outputs go through a 80-pin DF17 connector, allowing a plug-in solution for manufacturers of commercial systems. Manufacturers can design their own carrier boards to suite their specific needs.

An additional difference with Pixhawk 2.4.8 is the presence of ports with GH1.25 connectors; this made it necessary to change all wiring from the previous configuration (MX1.25) to these new connectors. In addition, the different peripherals have been placed in different ports from the previous configuration: to the POWER port is connected the power module that provides power to the autopilot, to the GPS1 port is connected the GPS module and the safety switch, to the GPS2 port is connected the lidar, to the I2C port is connected the magnetometer, to the TELEM2 port is connected the telemetry module for the ground station, and finally to the USB port is connected the buzzer and a microUSB connector for connection via Mavlink. More information on the pinouts of all ports and Ardupilot support can be found at the following link [35].

Figure 2.7: Cube Orange Hex Autopilot

2.5.2 Motors T-Motor AT2312 1150KV Long Shaft

A necessary improvement concerned the engines used; it was observed, after a test phase, that the previous ReadyToSky 2212 920KV engines, did not provide sufficient thrust to our aircraft, which, during mixed or hovering flight phases, lost power-to-weight ratio and struggled to maintain altitude. Therefore, it was concluded that, for the current weight of the drone, more powerful motors were needed in terms of thrust and RPM. The motors chosen, shown in the figure (2.8), are the 1150KV Long Shaft T-Motor AT2312, which were found to be very high performing in terms of thrust. To these motors, however, a modification to the long shaft was mandatory, which did not allow them to be mounted in a counter-rotating configuration on our chassis; therefore, the shafts were cut with a Dremel taking great care to cover the motor cases with scotch paper so that residue and steel dust, caused by the cut, would not enter the copper windings of the motor, permanently damaging it.

Figure 2.8: T-Motor AT2312 1150KV Long Shaft

2.5.3 Bashing Gens-Ace Battery 4S 5000mAh

Another hardware improvement, related to the choice of motors, concerns the choice of batteries; in particular, the batteries in the previous configuration (3S) did not provide sufficient power to the motors. Therefore, four-cell (4S) 5000mAh batteries were chosen in the figure (2.9). These provide up to 8 minutes of mixed flight mission time with the current configuration; if the same weight was to be maintained, it would be necessary to consider using six-cell battery packs with at least 10000mAh capacity in order to increase flight time and lengthen mission duration.

Figure 2.9: Bashing Gens-Ace Battery 4S 5000mAh

2.5.4 Holybro SiK Telemetry Radio V3

As a result of signal loss between ground station and drone, consideration was given to replacing the on-board telemetry module with a higher-performance, better-quality one. The Holybro module in the figure (2.9) was chosen, which is lightweight, small in size, and allows very high ranges (unobstructed) that can be extended with the use of additional antennas. This module also turns out to be plug-n-play and uses open source firmware designed to work with MAVLink packages and be integrated with Mission Planner, Ardupilot, QGroundControl, and PX4 Autopilot.

Figure 2.10: Holybro SiK Telemetry Radio V3 433MHz

2.5.5 Carbon Fiber Multistar Propeller 9x5

The last hardware modification, concerns the choice of propellers of the right size and pitch. For our configuration, 9x5 carbon fiber propellers were chosen, shown in the figure (2.11). This choice was made based on the size and weight of the drone but also based on the rotational speed of the motors and the power delivered by the batteries. If the previous propellers were used with the new motors, it would result in a lower thrust than the maximum thrust achievable with properly sized propellers. Similarly, if the new batteries had been used with the previous motors and propellers, the motors could have overheated to the point of burning out. Thus, the choice of battery, motors and propellers is to be made based on the final configuration of the drone and are closely related to each other.

Figure 2.11: Carbon Fiber Multistar Propeller 9x5

After all hardware modifications and upgrades are completed, and all wiring of all devices is completed, the completed drone is shown in the figure (2.12):

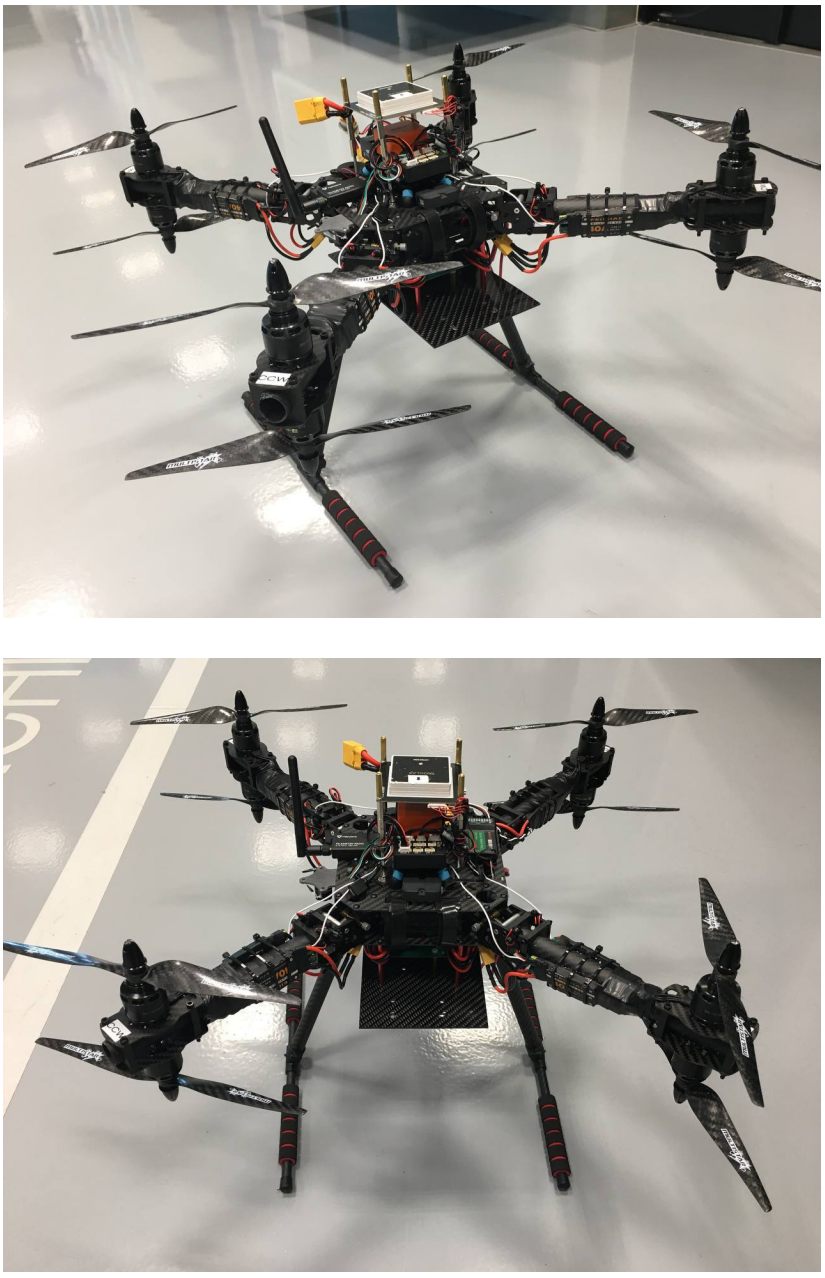


Figure 2.12: Final Configuration of the Octacopter Drone

2.6 Advanced Tuning

Upon completion of the hardware design of the new drone and its basic configuration/calibration, an advanced tuning procedure was carried out. This process is necessary if to optimize the operation of the drone and avoid possible in-flight issues, due to a lack of adaptation of parameters; and it is essential to adapt the ArduPilot firmware and its parameters to one's application since, the firmware itself, is designed generically and may not work properly on one's hardware configuration (just think of the different sizes of the drones, their weight, the power of the motors chosen and the batteries used: different UAVs have, for example, different throttle parameters in hovering phase due to the above-mentioned reasons. The following parameters should be set correctly based on the specifications of our aircraft. Each one impacts the quality of the tuning process.

2.6.1 Battery Setting

It is very important to ensure that the thrust curve of your VTOL motors is as linear as possible. A linear thrust curve means that changes in the actual thrust produced by a motor is directly proportional to the thrust being demanded by ArduPilot. If the thrust curve is badly non-linear then it is not possible to produce a good tune, and in some cases may end up with such a bad tune that the vehicle can become completely unstable and crash. It typically starts by setting the voltage range to cope with a possible voltage drop. Valuing these parameters linearizes the motor thrust curve.

- MOT_BAT_VOLT_MAX: 4.2V×No.Cells
- MOT_BAT_VOLT_MIN: 3.3V×No.Cells

The next step is to set the thrust expo. In a professional application, you need to accurately measure the true thrust for the motor/ESC/propeller combination as the throttle varies. For a scientific or hobbyist application, you can use the graph in the figure (2.13) to estimate the correct MOT_THST_EXPO value for our aircraft configuration (a value of 0.63 was chosen for our propeller size).

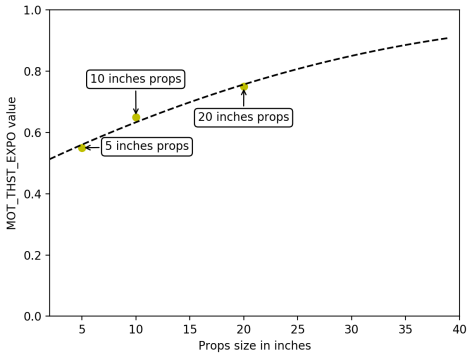


Figure 2.13: Approximate relationship between MOT_THST_EXPO value and props size in inches

2.6.2 Motors setup

The motor parameters define the PWM output range sent to the ESCs. This is critical to ensure that the entire range of throttle values used in flight is within the linear range of the propulsion system.

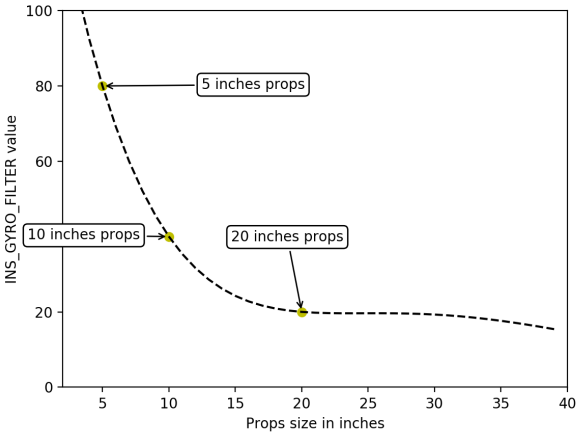
- MOT_PWM_MAX: Check ESC manual for fixed range or 2000µs;
- MOT_PWM_MIN: Check ESC manual for fixed range or 1000µs;
- MOT_SPIN_ARM: Use the motor test feature in Mission Planner to determine a value which will reliably start the motors spinning at a low rpm as an indication of the armed state (0.08 in our application);
- MOT_SPIN_MAX: 0.95;
- MOT_SPIN_MIN: Use the motor test feature in Mission Planner to set the lower range of linear thrust; the default value is usually adequate for hobby uses (0.17 in our application);
- MOT_THST_HOVER: 0.25 or below the expected actual hover thrust percentage (0.1902064 in our application);
- ATC_THR_MIX_MAN: 0.1.

2.6.3 PID Controller Initial Setup

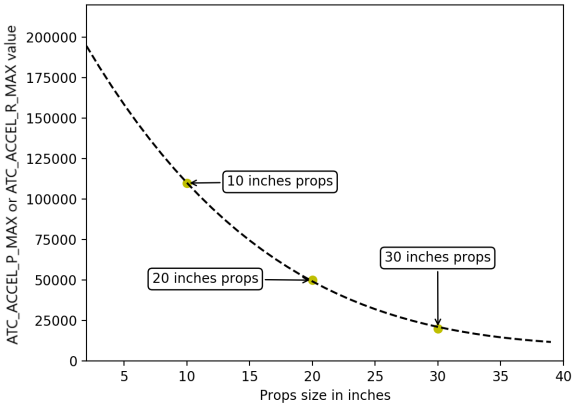
The settings below are meant to get the PID controller acceleration and filter settings into the right approximate range for the vehicle; these parameters are critical to the tuning process. The PID controller default values for axis P/D/I values are usually safe for first test hovers of most vehicles.

- INS_ACCEL_FILTER: 20Hz;
- INS_GYRO_FILTER: 46Hz following the figure (2.14a);
- ATC_ACCEL_P_MAX: 125900 following the figure (2.14b);
- ATC_ACCEL_R_MAX: 125900 following the figure (2.14b);
- ATC_ACCEL_Y_MAX: 27900 following the figure (2.14c);
- ATC_ANG_YAW_P (For Copter-4.2): 0.5×ATC_ACCEL_Y_MAX÷4500.

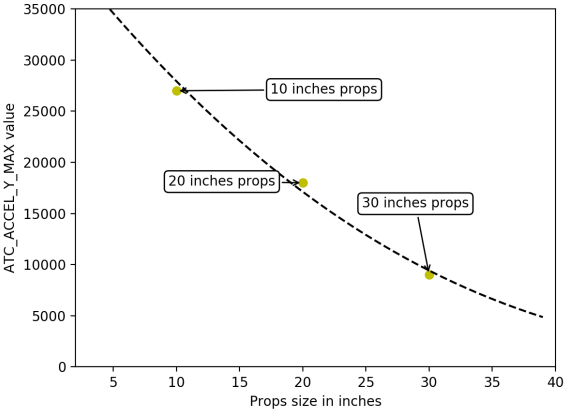
After that setup, it is possible to check the above parameters in the SETUP/Mandatory Hardware/Initial Parameter Setup tab in Mission Planner, by putting the airscrew size in inch, the battery cellcount, the battery cell fully charged voltage and the battery cell discharged voltage, and by clicking on Calculate Initial Parameters.



(a) Approximate relationship between INS_GYRO_FILTER value and props size in inches



(b) Approximate relationship between ATC_ACCEL_P_MAX or ATC_ACCEL_R_MAX value and props size in inches



(c) Approximate relationship between ATC_ACCEL_Y_MAX value and props size in inches

Figure 2.14: Approximate relationship for tuning phase

2.6.4 First Flight and Initial Tune

1. Ensure that the aircraft is in STABILIZE mode;
2. Arm the aircraft;
3. Increase the throttle slowly until the aircraft leaves the ground;
4. If the aircraft starts to oscillate, immediately abort the takeoff and/or land;
5. Reduce all the following parameters by 50%;
    - ATC_RAT_PIT_P;
    - ATC_RAT_PIT_I;
    - ATC_RAT_PIT_D;
    - ATC_RAT_RLL_P;
    - ATC_RAT_RLL_I;
    - ATC_RAT_RLL_D;

This process is repeated until the aircraft can hover without oscillations being detectable visually or audibly.

2.6.5 Test AltHold

This test will allow to test the altitude controller and ensure the stability of the aircraft.

1. Check MOT_HOVER_LEARN is set to 2. This will allow the controller to learn by itself the correct hover value when flying;
2. Take off in STABILIZE and increase altitude to 5m, then switch to AltHold. Ensure that the aircraft has spent at least 30 seconds in hover to let the hover throttle parameter converge to the correct value;
3. Land and disarm the aircraft;
4. Set these parameters on ground and preferably disarmed:
    - PSC_ACCZ_I equal to 2×MOT_THST_HOVER;
    - PSC_ACCZ_P equal to MOT_THST_HOVER.

If AltHold starts to oscillate up and down, the position and velocity controllers may need to be reduced by 50%. These values are: PSC_POSZ_P and PSC_VELZ_P. After there is an hover without oscillations, the next step is to get a good notch filter setup to reduce noise to the PID controllers.

2.6.6 Measuring Vibration with IMU Batch Sampler

The IMU BatchSampler can be used to record high-frequency data from the IMU sensors to the dataflash log on the autopilot. This data can be analysed post-flight to diagnose vibration issues using graphs created from Fast Fourier Transforms (FFT) of the data. FFT transforms data from the time domain into the frequency domain. Put another way, accelerometer data recorded over time (i.e. a flight) can be converted into a graph showing the frequencies of the vibration. A frequent feature of these graphs is a spike at the propeller's "blade passage frequency" (the frequency at which the blade crosses over the arms) which causes an acceleration in the aircraft body.

Pre-Flight Setup

- Set INS_LOG_BAT_OPT=0 to do pre-filter 1kHz sampling;
- Set INS_LOG_BAT_MASK=1 to collect data from the first IMU.

Flight and Post-Flight Analysis

- Perform a regular flight (not just a gentle hover) of at least 30s and download the dataflash logs;
- Open Mission Planner, press Ctrl-F, press the FFT button, press "IMU Batch Sample" and select the .bin log file downloaded above.

On the graph it should be possible to identify a significant peak in noise that corresponds to the motor rotational frequency. On a larger Copter this is likely to be around 100Hz. There will usually be harmonics of the motor rotational frequency (2×, 3× that frequency) also. In our application, we obtain the following graph (2.15):

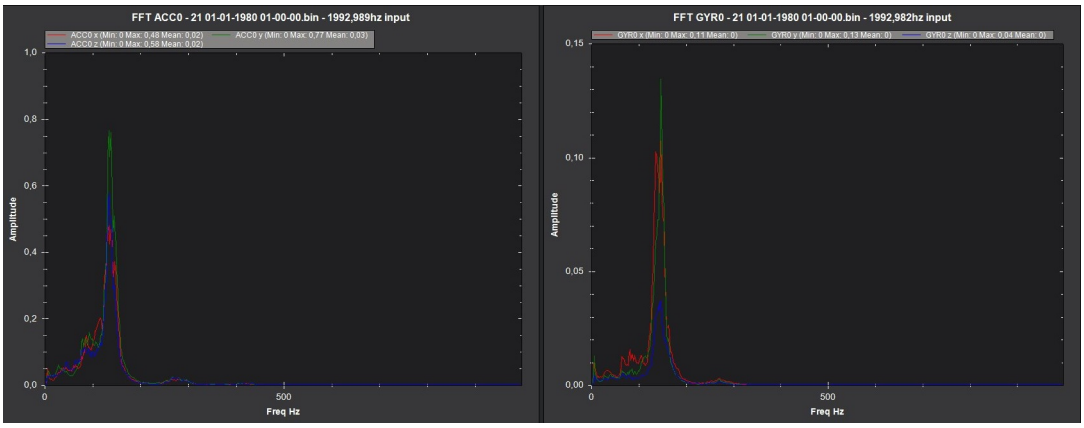


Figure 2.15: IMU Batch Sampler Before Filtering

With the same log, open it in the regular way in mission planner and graph the throttle value (CTUN/ThH). From this identify an average hover throttle value and compare it with the MOT_THST_HOVER (it should be the same).

Throttle Based Dynamic Notch Filter Setup

In order to configure the throttle-based dynamic harmonic notch filter it is important to identify the motor noise at the hover throttle level. To do this, we need to use the batch sampler to obtain logs for analysis (previous step). Once this is done, the center frequency of the notch(s) can be set and other parameters configured:

- Set INS_HNTCH_MODE and INS_HNTC2_MODE=1;
- Set INS_HNTCH_ENABLE and INS_HNTC2_ENABLE=1 to enable the harmonic notch;
- Set INS_HNTCH_REF and INS_HNTC2_REF=*hover_thrust* to set the harmonic notch reference value;
- Set INS_HNTCH_FREQ and INS_HNTC2_FREQ=*hover_freq* to set the harmonic notch reference frequency;
- Set INS_HNTCH_BW and INS_HNTC2_BW=*hover_freq÷2* to set the harmonic notch bandwidth.

Check the performance of the filter(s) after setup by doing another post filter configuration test flight and analyzing the logs.

Post Configuration Confirmation Flight and Post-Flight Analysis

- With INS_LOG_BAT_MASK still set to =1 to collect data from the first IMU;
- Set INS_LOG_BAT_OPT=2 to capture post-filter gyro data.

Perform a similar 30s flight (not only hover), plus at least 10s hover with no pilot stick input. Then analyze the dataflash logs in the same way. This time, there should be a significant less noise and, more significantly, attenuation of the motor noise peak. If the peak does not seem well attenuated, then it is possible to experiment by increasing the bandwidth and attenuation of the notch.

After the filter tuning, we obtain the following graph (2.16):

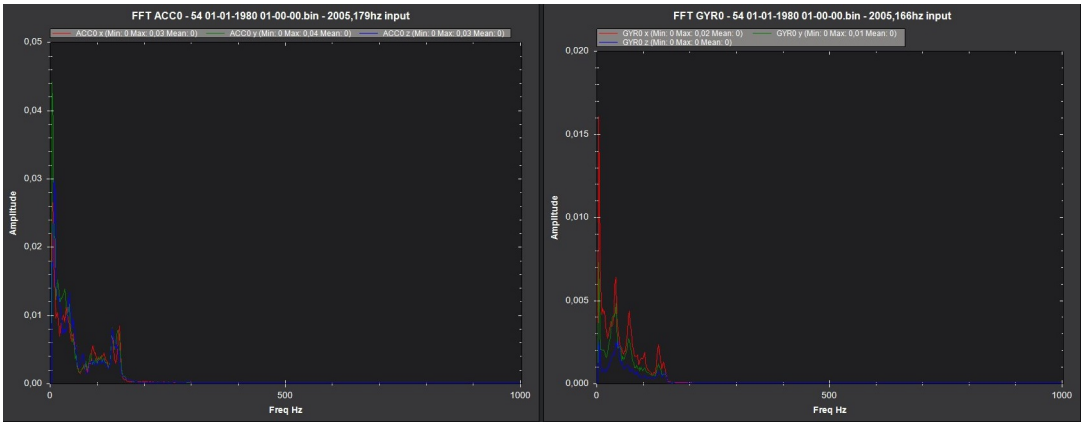


Figure 2.16: IMU Batch Sampler After Filtering

Be sure to reset the INS_LOG_BAT_MASK and INS_LOG_BAT_OPT to "0" when finished with analysis flight to free uo the RAM consumed by this feature.

2.7 Advanced Compass Setup

Accurately setting up the compass is critical because it is the primary source of heading information. Without an accurate heading the vehicle will not move in the correct direction in autopilot modes (i.e. AUTO, LOITER, PosHold, RTL, etc). This can lead to circling (aka “toiletbowling”) or fly-aways.

CompassMot - Compensation for interference from the power wires, ESCs and motors

This is recommended for vehicles that have only an internal compass and on vehicles where there is significant interference on the compass from the motors, power wires, etc. CompassMot only works well if you have a battery current monitor because the magnetic interference is linear with current drawn. It is technically possible to set-up CompassMot using throttle but this is not recommended. To make a correct CompassMot calibration, the steps are:

- Enable the current monitor (in Initial Setup tab/Optional Hardware/Battery Monitory or by modifying the parameter BATT_MONITOR in the full parameter list);
- Disconnect propellers, flip them over and rotate them one position around the frame. In this configuration, they should push the copter down into the ground when the throttle is raised;
- Secure the copter (perhaps with tape) so that it does not move;
- Turn on the transmitter and keep throttle at zero;
- Connect the vehicle’s LiPo battery;

- Connect the autopilot to the computer with usb cable;
- Open the **Initial Setup | Optional Hardware | Compass/Motor Calib** screen;
- Press the **Start** button;
- It should be heard the ESCs arming beep;
- Raise the throttle slowly to between 50%∼75% for 5∼10s;
- Quickly bring the throttle back down to zero;
- Press the **Finish** button to complete the calibration.

Check the % of interference displayed. If it is less than 30% then the compass interference is acceptable and it should have good Loiter, RTL and AUTO performance. If it is 31%∼60% then the interference is in the "grey zone" where it may be ok (some users are fine, some are not). If it is higher than 60%, should be tried to move the autopilot further up and away from the sources of interference or consider purchasing an external compass (or GPS+compass module).

In the figure (2.17) below, a compass motor calibration performed on the current configuration can be observed, and from the graphical results, it can be seen that the % of interference (red line) is in the optimal value range for clean and disturbance-free flight at the heading.

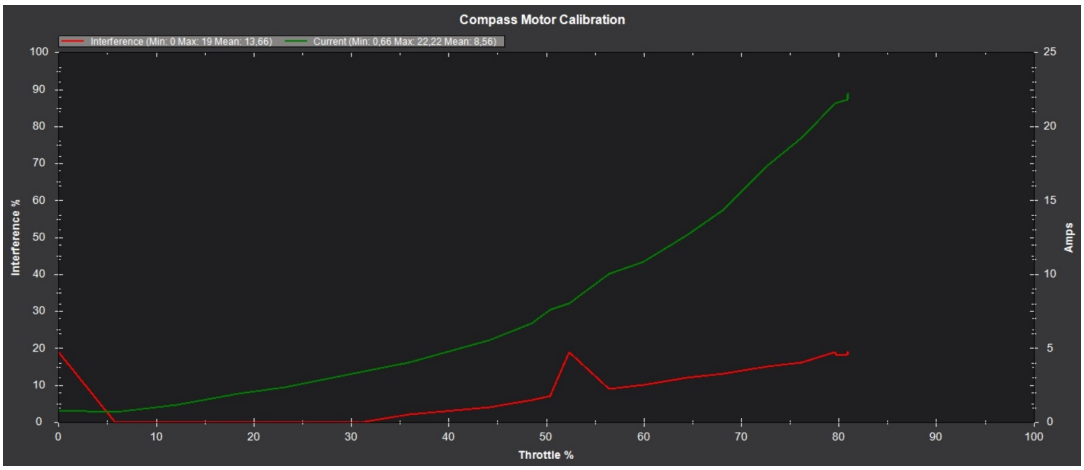


Figure 2.17: Compass Motor Calibration

2.8 RTK GPS Correction

It is possible to increase the normal position reporting accuracy of the GPS using RTK (Real Time Kinematics). Normal GPS accuracy is 3-5 meters using the Ublox M8N generation of GPS. If the region in which our application works, has SBAS (Satelite Based Augmentation Service) with geo-synchronous satellites reporting the general propagation conditions and corrections to the GPS, then accuracy can improve to the 1 meter range. But centimeter accuracy ranges can be obtained by using Real Time Kinematics correctional data with the newer F9P generation of GPSes. Since our application has two GPS modules compatible with RTK correction, this method was applied so as to increase location accuracy and help the precision landing phase. Propagation and timing corrections (RTCM data) for each satellite can be fed to the vehicle's RTK GPS in several ways:

- From a local RTK base station (fixed or moving base) connected to the GCS, which is also connected to the vehicle (via MAVLink);
- By forwarding RTCM correction data from an NTRIP server via the internet connected GCS, which is also connected to the vehicle (via MAVLink);
- Wirelessly from a local RTK base station directly to the vehicle GPS's secondary UART port.

The solution chosen is the first in the list: Mission Planner allows the connection of an external RTK Base GPS via USB. The correction data is passed from the Base unit through the Mission Planner to the vehicle's MAVLink connection, providing the vehicle's RTK GPS with correction data to enable its position reporting to become more accurate. Key to the success of this method requires that the RTK Base station's position be precisely known or determined. Usually, this requires that the GCS command the RTK Base GPS to "survey-in" itself. The GPS takes many measurements determining its mean location, applying interim corrections, and continues to refine its location deviations until the measurements are within a set threshold of error (usually a few meters) for a given time period (usually 60 seconds). The GPS then uses that as its location and start outputting the correction data to the GCS for forwarding to the vehicle's GPS. It is possible to program the RTK Base's exact location directly into it if its is known using the UBlox programming tool or Mission Planner if that location has been previously "surveyed-in". In order to set up this RTK correction, we need to perform the following steps:

1. Set GPS_TYPE=1 in the parameter list;
2. Connect the GPS RTK that will act as a fixed base via USB to the pc and start Mission Planner (be careful to place the module in a location that will remain unchanged for all future missions);
3. Go to the Setup window, then Optional Hardware | RTK/GPS Inject section;
4. At the top left, it will be necessary to select the COM port to which the module is connected via USB (leaving 460800 bandwidth) and click Connect.

The "Link Status" should start showing data being input and the satellite constellations locked will be shown with green indicators;

5. Do the "survey-in" of the GPS module, in mission planner, choosing the "Survey-in Accuracy" in meters and pressing the "Restart" button (Ublox recommends a SurveyIn ACC of 5m or less, 2m is commonly used). Now the GPS will download the almanac and query all the satellites at its disposal to receive its position by correcting it thanks to the many queries; in the "Survey-in" window on the right, it is possible to monitor the status progress of the survey until getting "Position is Valid" with a green indicator;

6. Once the position is obtained, it is possible to save it in a list for future use, if the Base is placed at that exact location in the future, to avoid another survey procedure. To use an existing position, first assure that the GPS attached to the Mission Planner PC is indeed in that exact location, then press the “USE” button for that location in the list. RTK updates to the vehicle will begin immediately.

Once RTK data is being processed by the vehicle’s GPS, its status will change to “RTK float” and then to “RTK fixed” in the GCS HUD indicating that its ready for use. “RTK float” means that it is using correction data, but has not moved to the highest precision mode yet. Thanks to this localization mode, a very high accuracy (about 10 cm) and consequent high performance during flight missions can be maintained; it can be seen in the figure (2.18) how, during a PosHold, there is a deviation of a few centimeters on the map instead of a few meters.

Figure 2.18: Accurate Localization on Mission Planner

2.9 Quadcopter Drone Setup

After finishing the hardware configuration and calibration of the octocopter, it was decided to fix the quadcopter drone from the previous work; this was because, in the preliminary testing stages, a high probability of failure and crashing of the craft was anticipated, and using this "battle drone" avoids damage to the octocopter drone that mounts expensive hardware on board. Therefore, the previous drone was completely deassembled and started from the bare chassis again.
The first reconstruction step is to redo all the soldering on the distribution board, removing the previous ones, cleaning out the excess tin, and making new soldering of the ESCs and Power Module with the PDB. Then proceeded with the assembly of:

- The same ESCs as the previous configuration (ReadyToSky 40A);
- Motors (ReadyToSky 2212 920KV) that are sufficient for a low weight drone;
- 9×5 carbon fiber Multistar propellers (pay attention to the fact that the shaft of the engine where to attach the propellers has a larger diameter than the hole of the propellers themselves and, therefore, the hole was enlarged with the use of a drill);
- 4S 5000mAh battery pack;
- Pixhawk 2.4.8 autopilot;
- Lidar TFmini (since this autopilot, has a less performing barometer than that of the CubePilot, so it is preferable to support it with a ToF to ground for correct height measurement);
- All the peripheral hardware like safety switch, buzzer, telemetry module, rc receiver.

In this configuration, a high-performance M8N GPS module from Holybro, shown in figure (2.19), was chosen because the TAOGLAS module was mounted on the octocopter drone.

Figure 2.19: GPS M8N Holybro

With this set of elements, the drone is ready for manual flight; to work in autonomous mode, it will need an on-board computer, and a Raspberry Pi 4 Model B was chosen, in figure (2.20). This companion computer has less computational capabilities than the Jetson Nano but is sufficient for this configuration because it will only have to process the terrain images that it will use to recognize the landing site (in the final configuration, a more computationally intensive computer is needed because, in this one, there is also an Intel Realsense Depth Camera used for obstacle avoidance).

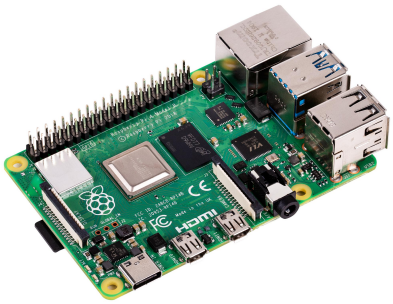


Figure 2.20: Raspberry Pi 4 Model B

For marker recognition at the landing point, a Raspberry Pi Camera NoIR V2 was chosen, shown in figure (2.21), positioned at a point further forward than the base of the chassis so as to have a wide field of view of the terrain.

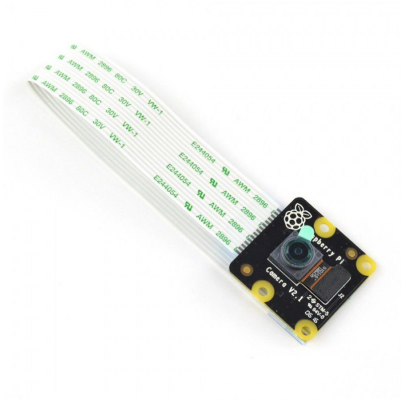


Figure 2.21: Raspberry Pi NoIR Camera V2

After the assembly was completed, an initial configuration phase (firmware, gyro, compass, motor test, etc.) was carried out. This was followed by battery monitor calibration, compass motor calibration, and finished with the advanced tuning phase to calibrate the PID controllers, hovering parameters, and configuration of a notch filter.
After tuning, however, a bouncing landing phenomenon was observed; the barometer failed to realize that it was on the ground in the automatic landing phase and, after touching the floor, would rise a few centimeters and then re-land. In fact, by analyzing the logs, it was noticed that there were several "LAND_COMPLETE_MAYBE" messages at the end of the landing phase, but in the absence of error, there should

be only one of them. This problem is caused by the presence of the raspberry and its protective case on the top of the autopilot which, in the vicinity of the floor, caused a buildup of air reflections (generated by the spinning propellers close to the floor) for the barometer reading, sending it into confusion; to solve this phenomenon, a ToF toward the ground was added and the vertical descent velocity (LAND_SPEED parameter), in the autonomous landing phase, was increased to 60cm/s so as to reduce the transient that could generate errors during landing.
The complete configuration of the quadcopter to be used for preliminary testing is as follows in the figure (2.22):

Figure 2.22: Final Configuration of the testing Quadcopter Drone

Chapter 3

On-Board Companion Computer

3.1 Companion Computer First Configuration

After completing the hardware configuration of the craft, the next step was to choose the on-board computer having the task of managing the autopilot as a high-level controller. Specifically, this computer will be in charge of receiving the camera images, processing them, and exploiting the results in conjunction with the information received from the autopilot (GPS positions, drone attitude, drone height, etc.). For the application to be developed, either the Raspberry Pi 4 Model B or the NVIDIA Jetson Nano can be used, and in our case, it was chosen to use the former on the quadcopter and the latter on the octacopter and then compare their computational performance and efficiency in terms of accurate and repeated results over time. The first step that needs to be taken in order to use these two computers as on-board computers is to do their initial configuration correctly. We then proceeded in installing the operating system, specifically Raspberry Pi OS (from the link [36]) and Jetson Nano Developer Kit SD (from the link [37]), both Linux-based operating systems. In addition, Secure SHell (SSH) has been enabled so that work can be continued remotely without needing a monitor, keyboard and mouse connected to the computer to be configured; to do this, it is recommended that a static IP address be assigned to the device so that it can always be reached through it. Then, the packet nano is needed to manage and modify files into the pc; it is possible to install it with the command:

```
$ sudo apt-get install nano
```

Only on the Raspberry, it is also necessary to enable the port where to connect the camera and the serial port that will be used for communication with the autopilot. Once connected to the above via SSH, the command:

```
$ sudo raspi-config
```

will be entered, and item number 5 "Interfacing Options" will be selected. Here the port 1 Camera will need to be enabled, while at port 6 it is necessary to disable the Serial Login Shell and enable the Serial Port Hardware. In addition, it will be necessary to increase the swap memory with the command:

```
$ sudo nano /etc/dphys-swapfile
```

and setting **CONF_SWAPSIZE=1024**. Then, the service will then be stopped and reinitiated through the commands:

```
$ sudo /etc/init.d/dphys-swapfile stop
$ sudo /etc/init.d/dphys-swapfile start
```

At this point, for both Raspberry and Jetson, it will upgrade the newly installed system with "sudo apt-get update" and "sudo apt-get upgrade" (the first time will take a while). After that X11 tunneling will need to be enabled via the command:

```
$ sudo /etc/ssh/sshd_config
```

and set up **X11Forwarding yes**, **X11DisplayOffset 10** and **X11UseLocalhost yes**. This will help to take advantage of the companion computer's graphics applications that are based on X11 graphics server, and display them on a separate machine via SSH using the two tools PuTTY and Xming (i.e. Marker Recognition). To use the Python libraries, it will use pip, which will need to be installed with the command:

```
$ sudo apt-get install python-pip python3-pip
```

In this way, it will be possible to install all the Python libraries, necessary for communication between the computer and autopilot, through the following:

```
$ sudo apt-get install python-dev python3-dev
$ sudo pip install future
$ sudo pip3 install future
```

For the compilation and execution of the Dronekit package, used for serial communication, the following will also need to be installed:

```
$ sudo apt-get install screen python-wxgtk3.0
$ sudo apt-get install python-matplotlib python3-matplotlib
$ sudo apt-get install python-opencv python3-opencv
$ sudo apt-get install python-numpy python3-numpy
$ sudo apt-get install libxml2-dev
$ sudo apt-get install libxslt1-dev
$ sudo pip install pyserial
$ sudo pip3 install pyserial
$ sudo apt-get install git
$ mkdir git
$ cd git
$ git clone https://github.com/dronekit/dronekit-python.git
$ cd dronekit-python
$ sudo python setup.py build
$ sudo python setup.py install
$ sudo python3 setup.py build
$ sudo python3 setup.py install
```

3.2 Autopilot and Computer Communication

The following step is critical to change the read and write permissions of the serial/USB ports (otherwise, it is not possible to request or set settings on the autopilot via the companion computer):

```
$ sudo adduser $USER $(stat --format="%G" /dev/ttyACM0 )
$ sudo adduser $USER $(stat --format="%G" /dev/ttyAMA0 )
```

At this time, all the packages and libraries needed for the communication between our high-level and low-level controllers are installed and we can proceed with the hardware connection. Specifically, for the quadcopter, a telemetry port was chosen to allow the serial communication between Pixhawk 2.4.8 and Raspberry Pi 4, by following the pinouts depicted in the figure (3.1):

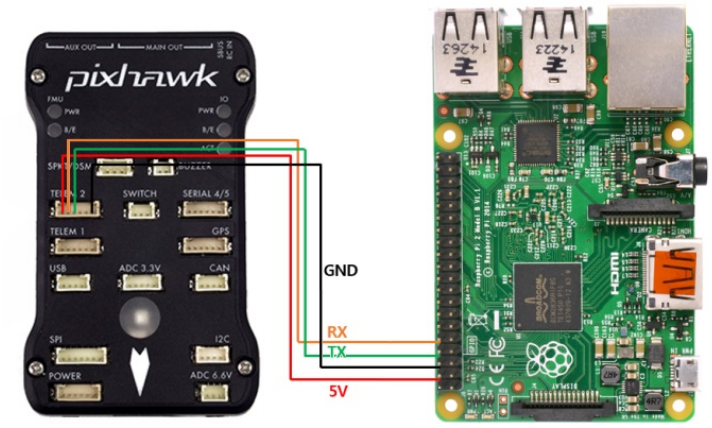


Figure 3.1: Connection Pinout between Pixhawk 2.4.8 and Raspberry Pi 4

Instead, for the connection between Cubepilot Orange and Jetson Nano onto the Octacopter, the USB-MicroUSB connection was chosen since, the autopilot itself, has an extra usb port unlike Pixhawk which has only one (to be kept free for communication with the ground station in case telemetry is not available at the moment). Once the hardware connection is also finished, it is possible to make the companion computer talk to the autopilot, through the use of Python and the Dronekit developer tool. The script "01_connection_parameters_reading.py" then allows a connection to be made with the autopilot and ask it for some information such as the type of firmware installed, the position at that instant, the ground speed, whether it can be armed, etc.
With the following, it is possible to enable the connection and start the communication:

```
#here is set the used port for serial communication
#(ttyACM0 for USB connection, ttyAMA0 for serial/telemetry port)
connection_string = "/dev/ttyACM0"
baud_rate = 921600
```

```
print(">>>> Connecting with the UAV <<<")
vehicle = connect(connection_string, baud=baud_rate, wait_ready=True)
#wait_ready flag hold the program until all the parameters are been read
```

Now that the two devices are talking to each other, different information can be extracted:

```
#-- Read Information from the Autopilot:
#- Version and Attributes
vehicle.wait_ready('autopilot_version')
print('Autopilot version: %s'%vehicle.version)
#- Read the Actual Position
print('Position: %s'% vehicle.location.global_relative_frame)
#- Read the Actual Attitude Roll, Pitch, Yaw
print('Attitude: %s'% vehicle.attitude)
#- Read the Actual Velocity (m\s)
print('Velocity: %s'%vehicle.velocity) #- North, East, Down
#- When the Last Heartbeat is received
print('Last Heartbeat: %s'%vehicle.last_heartbeat)
#- Is the Vehicle good to Arm?
print('Is the vehicle armable: %s'%vehicle.is_armable)
#- Which is the Total Ground Speed?
print('Groundspeed: %s'% vehicle.groundspeed) #(%)
#- What is the Actual Flight Mode?
print('Mode: %s'% vehicle.mode.name)
#- Is the Vehicle Armed?
print('Armed: %s'%vehicle.armed)
#- Is the state estimation filter ok?
print('EKF Ok: %s'%vehicle.ekf_ok)
```

In addition, a callback listener function can be added to be able to read certain information for a specific instant of time, such as additude information for 5s, as follows:

```
#--- Adding a Listener
def attitude_callback(self, attr_name, value):
print(vehicle.attitude)
print("")
print("Adding an attitude listener")
#--- Now the attitude is printed from the callback for 5 seconds
vehicle.add_attribute_listener('attitude', attitude_callback)
time.sleep(5)
#--- Then the Callback is removed
vehicle.remove_attribute_listener('attitude', attitude_callback)
#--- At the end, the communication is closed
vehicle.close()
print("done")
```

The output generated by the previous script is as follows in the figure (3.2), (moving the drone manually to see its attitude during the listener function):

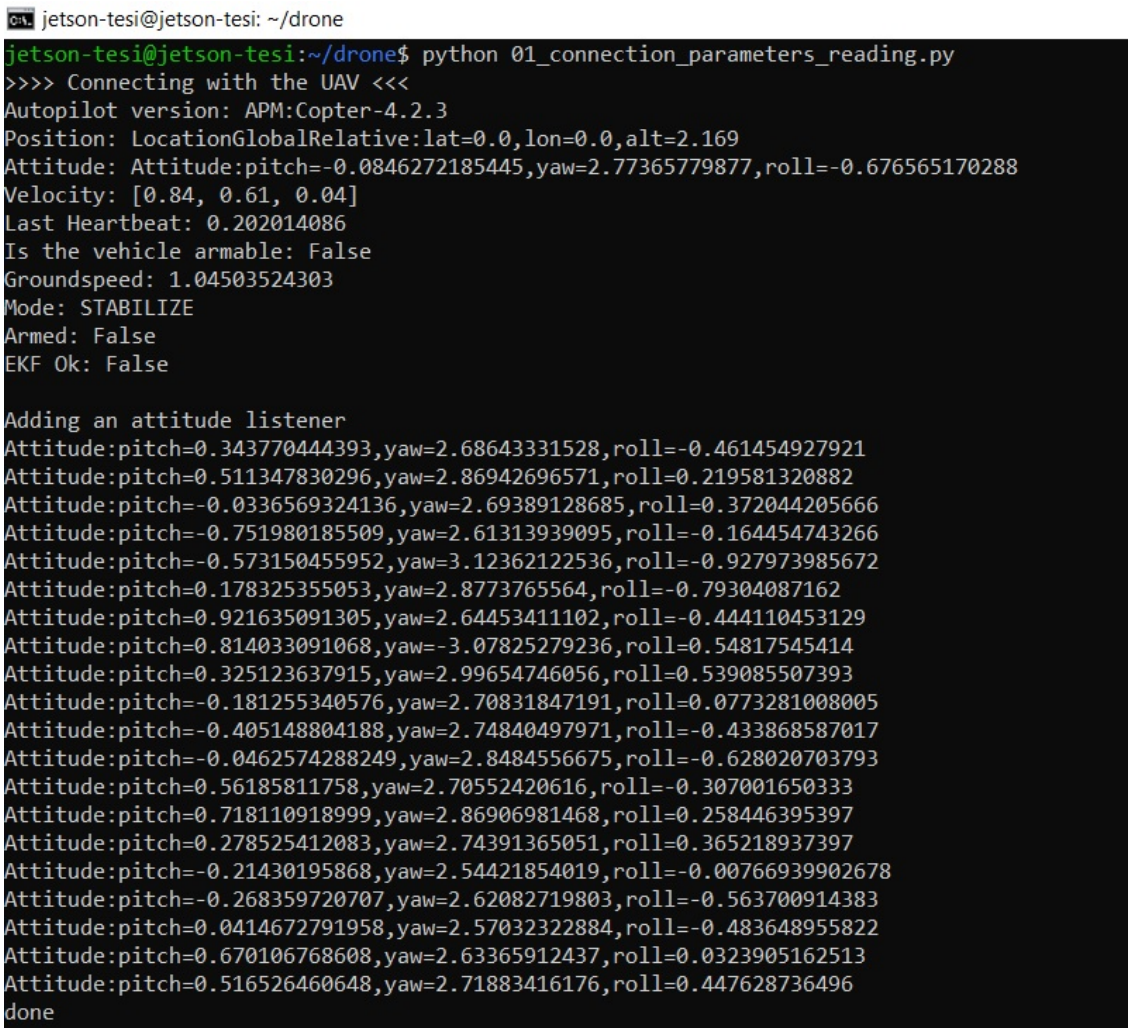


Figure 3.2: Output from 01_connection_parameters_reading.py

Chapter 4

Marker Recognition

4.1 Aruco Original Marker

Continuing the development of the project, two very powerful tools in the field of computer vision were made use of.
The first is called OpenCV (OPEN source Computer Vision library) and is an open source, cross-platform software library developed for real-time computer vision; it contains multiple functions that support the acquisition, analysis, and manipulation of visual information sent to a computer from a camera, video file, or other device. It is possible to use simple functions to draw a line or shape, but the most interesting and advanced parts of the library contain algorithms for detecting faces, tracking motion, and analyzing shapes as shown in figure (4.1):

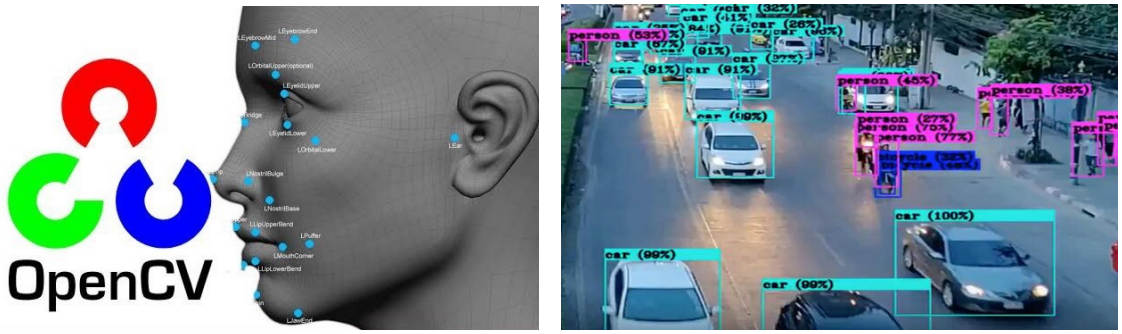


Figure 4.1: Object Detection by using OpenCV

The second tool is the fiducial marker: there are different types of them (ArUco, AprilTags, etc.) and they are fundamental in the world of computer vision. One of the most important functions, which we are going to use, is Pose Estimation: this process is based on finding correspondences between points in the real environment and their 2d image projection. One of the most popular approaches is the use of binary square fiducial markers. The main benefit of these markers is that a single marker provides enough correspondences (its four corners) to obtain the camera pose. Also, the inner binary codification makes them specially robust, allowing the possibility of applying error detection and correction techniques. In the application

under analysis, ArUco Markers were chosen to be used (examples in figure (4.2)) based on the ArUco library, a popular library for detection of square fiducial markers developed by [38].

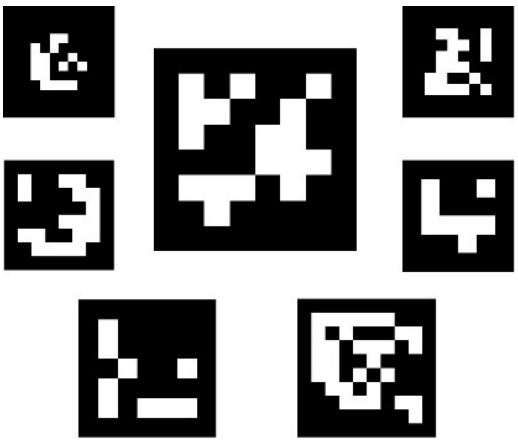


Figure 4.2: Example of Markers Images

4.2 OpenCV Configuration

In order to enable proper marker recognition, therefore, the on-board computer will need to be configured by entering OpenCV libraries and packages. Then, the installation of the previous is proceeded by the following commands:

```
$ sudo apt-get install build-essential cmake pkg-config
$ sudo add-apt-repository 'deb http://ports.ubuntu.com/ubuntu-ports
xenial-security main'
$ sudo apt-get install libjpeg-dev libtiff5-dev libjasper-dev libpng-dev
$ sudo apt-get install libavcodec-dev libavformat-dev
$ sudo apt-get install libswscale-dev libv4l-dev
$ sudo apt-get install libxvidcore-dev libx264-dev
$ sudo apt-get install libgtk2.0-dev
$ sudo apt-get install libatlas-base-dev gfortran
$ sudo pip install numpy
$ sudo pip3 install numpy
```

After installing all the libraries needed for marker recognition, it is needed to download and build the latest version of OpenCV and OpenCV-Contrib (the last version at the moment is used), through the following steps:

```
$ wget -O opencv.zip https://github.com/Itseez/opencv/archive/4.6.0.zip
$ unzip opencv.zip
$ wget -O opencv_contrib.zip https://github.com/Itseez/opencv_contrib/
archive/4.6.0.zip
$ unzip opencv_contrib.zip
```

After downloading both packages, the next step is to create the build folder, prepare the make file and complete the installation (the operation can take more than two hours):

```
$ cd opencv-4.6.0/
$ mkdir build
$ cd build/
$ cmake -D CMAKE_BUILD_TYPE=RELEASE \
> -D CMAKE_INSTALL_PREFIX=/usr/local \
> -D INSTALL_PYTHON_EXAMPLES=ON \
> -D OPENCV_EXTRA_MODULES_PATH=~/opencv_contrib-4.6.0/modules/ \
> -D BUILD_EXAMPLES=ON ..
$ make -j4
$ sudo make install
$ cd
$ sudo ldconfig
```

After these steps, the installation can be verified as successful by opening python from the command line and importing the cv2 module; if there is no error message then everything was configured correctly.

```
$ python
>>> import cv2
```

4.3 Camera Calibration

Now that everything is ready to take advantage of OpenCV functions in the computer vision application, the next step is to calibrate the camera. Calibration is critical because all cameras are different from each other, even cameras of the same brand or model differ slightly in focus and lens alignment. Specifically, OpenCV treats cameras as a pinhole model that consists of a mathematical representation of a 3D object projected into a 2D one. By projecting something that is in three dimensions into a 2D plane, depth perception is lost. The pinhole model is described by an intrinsic camera parameter called the Camera Matrix 3×3 (shown below) containing f_x and f_y being the focal lengths and c_x and c_y indicating the optical centers (all these dimensions are in pixels).

$$CameraMatrix = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

In addition to this, one must also consider the lens distortion that each lens has (as in the following figure (4.3)), and then rectify the image in order to make sense of what is seen in the mathematical model obtained: this distortion is modeled by five parameters contained in the Distortion Matrix:

$$DistortionMatrix = \begin{bmatrix} k_1 & k_2 & p_1 & p_2 & k_3 \end{bmatrix}$$

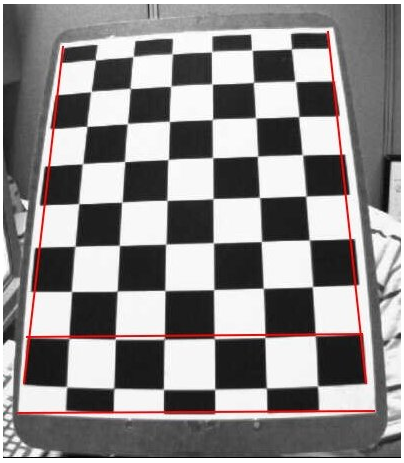


Figure 4.3: Example of Lens Distortion

Together with the previous four parameters, there is a need to estimate nine total parameters. The process of estimating these parameters is called calibration [39]. To perform proper calibration, a script in Python will be used to take snapshots at a particular chessboard 9×6 in figure (4.4) in different distances, orientations and inclinations (as depicted in the following figures: (4.5))

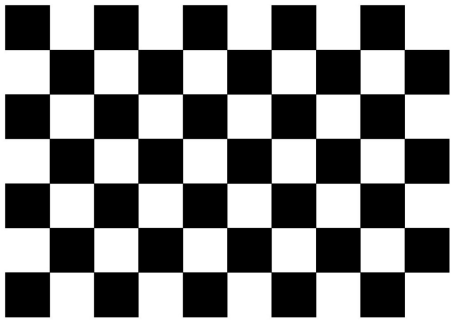


Figure 4.4: Original ChessBoard

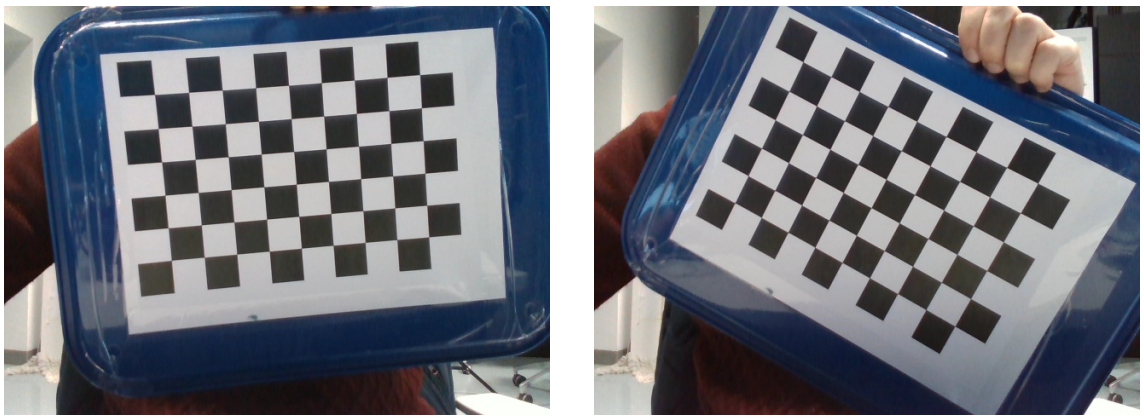


Figure 4.5: Snapshot Examples for Calibration

In order to display the camera graphics window, on a Windows machine, it will be necessary to use Xming and PuTTY and execute, on the latter, the scripts that need to display the graphical part of the on-board computer. The script "02_save_snapshots.py" used for saving snapshots is as follows:

```
import cv2
import time
import sys
import argparse
import os

def save_snaps(width=640, height=480, name="snapshot",
folder="./camera_01"):

cap = cv2.VideoCapture(2)
if width > 0 and height > 0:
print("Setting the custom Width and Height")
cap.set(cv2.CAP_PROP_FRAME_WIDTH, width)
cap.set(cv2.CAP_PROP_FRAME_HEIGHT, height)
try:
if not os.path.exists(folder):
os.makedirs(folder)

# ----------- CREATE THE FOLDER -----------------
folder = os.path.dirname(folder)
try:
os.stat(folder)
except:
os.mkdir(folder)
except:
pass

nSnap  = 0
w = cap.get(cv2.CAP_PROP_FRAME_WIDTH)
h = cap.get(cv2.CAP_PROP_FRAME_HEIGHT)

fileName = "./%s/%s_%d_%d_" %(folder, name, w, h)
while True:
ret, frame = cap.read()

cv2.imshow('camera', frame)

key = cv2.waitKey(1) & 0xFF
if key == ord('q'):
break
if key == ord(' '):
print("Saving image ", nSnap)
```

```
cv2.imwrite("%s%d.jpg"%(fileName, nSnap), frame)
nSnap += 1

cap.release()
cv2.destroyAllWindows()


def main():
# ---- DEFAULT VALUES ---
SAVE_FOLDER = "./camera_01/"
FILE_NAME = "snapshot"
FRAME_WIDTH = 640
FRAME_HEIGHT = 480

# ----------- PARSE THE INPUTS -----------------
parser = argparse.ArgumentParser(
description="Saves snapshot from the camera"
\n q to quit \n spacebar to save the snapshot")
parser.add_argument("--folder", default=SAVE_FOLDER)
parser.add_argument("--name", default=FILE_NAME)
parser.add_argument("--dwidth", default=FRAME_WIDTH, type=int)
parser.add_argument("--dheight", default=FRAME_HEIGHT, type=int)
args = parser.parse_args()

SAVE_FOLDER = args.folder
FILE_NAME = args.name
FRAME_WIDTH = args.dwidth
FRAME_HEIGHT = args.dheight

save_snaps(width=args.dwidth, height=args.dheight,
name=args.name, folder=args.folder)
print("Files saved")

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

In order to save the photo, it will be necessary to press the space bar while, to stop the execution of the script, it will be necessary to press the q button on the keyboard.

After taking a considerable number (at least forty) of photos of the chessboard, an additional calibration script will be used that will recognize a certain path in each photo as shown in the following figures (4.6); photos with the correct path will be accepted (on left image below) while those with an incorrect path will be discarded (on right image below).

Figure 4.6: Correct/Wrong Calibration Path

The script "03_camera_calibration.py" used for calibrating the camera and extrapolating its intrinsic parameters is as follows:

```
import numpy as np
import cv2
import glob
import sys
import argparse

#----SET THE PARAMETERS
nRows = 9
nCols = 6
dimension = 25 #- mm side lenght of a single square
workingFolder   = "./camera_01"
imageType       = 'jpg'
#----
# termination criteria
criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER,
 dimension, 0.001)

objp = np.zeros((nRows*nCols,3), np.float32)
objp[:,:2] = np.mgrid[0:nCols,0:nRows].T.reshape(-1,2)

# Arrays to store object points and image points from all the images.
objpoints = [] # 3d points in real world space
imgpoints = [] # 2d points in image plane.
```

```
# Find the images files
filename = workingFolder + "/*." + imageType
images = glob.glob(filename)

print(len(images))
if len(images) < 9:
print("Not enough images were found: at least 9 shall be provided!")
sys.exit()

else:
nPatternFound = 0
imgNotGood = images[1]

for fname in images:
if 'calibresult' in fname: continue
#-- Read the file and convert in greyscale
img = cv2.imread(fname)
gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)

print("Reading image ", fname)

# Find the chess board corners
ret, corners = cv2.findChessboardCorners(gray, (nCols,nRows),None)

# If found, add object points, image points (after refining them)
if ret == True:
print("Pattern found! Press ESC to skip or ENTER to accept")
#--- Sometimes, Harris cornes fails with crappy pictures, so
corners2 = cv2.cornerSubPix(gray,corners,(11,11),(-1,-1),criteria)

# Draw and display the corners
cv2.drawChessboardCorners(img, (nCols,nRows), corners2,ret)
cv2.imshow('img',img)
k = cv2.waitKey(0) & 0xFF
if k == 27: #-- ESC Button
print("Image Skipped")
imgNotGood = fname
continue

print("Image accepted")
nPatternFound += 1
objpoints.append(objp)
imgpoints.append(corners2)

else:
imgNotGood = fname
```

```
cv2.destroyAllWindows()

if (nPatternFound > 1):
print("Found %d good images" % (nPatternFound))
ret, mtx, dist, rvecs, tvecs = cv2.calibrateCamera(objpoints,
imgpoints, gray.shape[::-1],None,None)

# Undistort an image
img = cv2.imread(imgNotGood)
h,  w = img.shape[:2]
print("Image to undistort: ", imgNotGood)
newcameramtx, roi=cv2.getOptimalNewCameraMatrix(mtx,dist,(w,h),1,(w,h))
# undistort
mapx,mapy = cv2.initUndistortRectifyMap(mtx,dist,None,
   newcameramtx,(w,h),5)
dst = cv2.remap(img,mapx,mapy,cv2.INTER_LINEAR)

# crop the image
x,y,w,h = roi
dst = dst[y:y+h, x:x+w]
print("ROI: ", x, y, w, h)

cv2.imwrite(workingFolder + "/calibresult.png",dst)
print("Calibrated picture saved as calibresult.png")
print("Calibration Matrix: ")
print(mtx)
print("Disortion: ", dist)

#--------- Save result
filename = workingFolder + "/cameraMatrix.txt"
np.savetxt(filename, mtx, delimiter=',')
filename = workingFolder + "/cameraDistortion.txt"
np.savetxt(filename, dist, delimiter=',')

mean_error = 0
for i in range(len(objpoints)):
imgpoints2, _ = cv2.projectPoints(objpoints[i], rvecs[i],
   tvecs[i], mtx, dist)
error = cv2.norm(imgpoints[i],imgpoints2, cv2.NORM_L2)/len(imgpoints2)
mean_error += error
print "total error: ", mean_error/len(objpoints)

else:
print("In order to calibrate you need at least 9 good pictures,
   try again!")
```

At the end of the script, the Camera Matrix and Distortion Matrix will be obtained, which will be used in the future for estimating the marker pose during precision landing. In particular, the following relate to the Intel Depth camera D435i present on the quadcopter:

$$CameraMatrixQuad = \begin{bmatrix} 302.54676 & 0.0000000 & 210.93231 \\ 0.0000000 & 301.49758 & 120.30433 \\ 0.0000000 & 0.0000000 & 1.0000000 \end{bmatrix}$$

$$DistortionMatrixQuad = \begin{bmatrix} -0.01975 & 1.26843 & -0.00236 & 0.00024 & -4.56254 \end{bmatrix}$$

While the following relate to the same camera model present on the octocopter:

$$CameraMatrixOcta = \begin{bmatrix} 606.10796 & 0.0000000 & 325.58227 \\ 0.0000000 & 604.47564 & 238.56731 \\ 0.0000000 & 0.0000000 & 1.0000000 \end{bmatrix}$$

$$DistortionMatrixOcta = \begin{bmatrix} 0.01821 & 0.96342 & 0.00056 & 0.00629 & -3.32459 \end{bmatrix}$$

As can be seen, although they are cameras from the same manufacturer and identical in model, they differ in the intrinsic parameters that influence image perception for visual recognition.

4.4 Aruco Pose Estimation

Now that the camera's intrinsic parameters have been extracted, the recognition of the chosen unique marker can proceed. Before developing the script for marker recognition and pose estimation, it is important to note that, as stated in the OpenCV documentation:

- One marker is sufficient to estimate the pose and rotation of the camera;
- A marker is identified by the dictionary to which it belongs;
- There are multiple Aruco dictionaries (in this case, it will be used the Original one);
- The marker will be created by the University of Maryland's online generator [40].

For this type of recognition, some reference should also be made to reference frames represented in figure (4.7):

1. The camera reference frame is centered in the camera optical center with the Z axis is sticking out, the X axis going right and the Y axis going down;

2. The marker reference frame has the Z axis going toward the observer, the X axis going right and the Y axis going up.

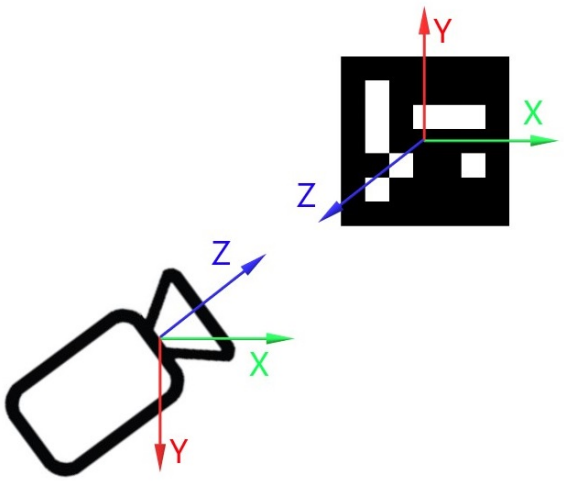


Figure 4.7: Camera and Marker Reference Frames

Thus, when the camera and the marker are directly observed, the two reference frames are the opposite of each other; this is obviously complex to handle, when the rotation of one frame with respect to the other needs to be analyzed. To simplify this issue, a flipped reference frame, in figure (4.8), will be created around which the rotations will be defined.

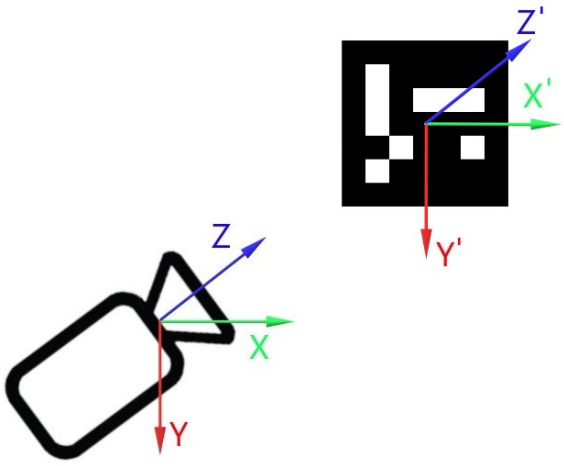


Figure 4.8: Camera and Marker Flipped Reference Frames

Once this detail of the reference frames has been analyzed, marker recognition can proceed (in this application, the original marker number 72 was randomly chosen) in the following script "04_aruco_pose_estimation"; the first step is to import all the necessary libraries and define the marker ID chosen and its size in cm:

```
import numpy as np
import cv2
import cv2.aruco as aruco
import sys, time, math

id_to_find = 72
marker_size = 9.6
```

Then, the results of camera calibration is loaded, such as camera matrix and camera distortion:

```
calib_path = ""
camera_matrix = np.loadtxt(calib_path+'cameraMatrix.txt', delimiter=',')
camera_distortion = np.loadtxt(calib_path+'cameraDistortion.txt',
   delimiter=',')
```

Then, it is necessary to define the 3×3 rotation matrix around the X axis and the Aruco Original Dictionary:

```
#--- 180 deg rotation matrix around the x axis
R_flip  = np.zeros((3,3), dtype=np.float32)
R_flip[0,0] = 1.0 #X Axis
R_flip[1,1] =-1.0 #Y Axis
R_flip[2,2] =-1.0 #Z Axis

#--- Define the aruco dictionary
aruco_dict  = aruco.getPredefinedDictionary(aruco.DICT_ARUCO_ORIGINAL)
parameters  = aruco.DetectorParameters_create()
```

After that, it is possible to capture the camera (this may also be a video or picture), making sure to use the same width and height used during the calibration procedure and the correct index referred to the actually camera:

```
cap = cv2.VideoCapture(2)
cap.set(cv2.CAP_PROP_FRAME_WIDTH, 640)
cap.set(cv2.CAP_PROP_FRAME_HEIGHT, 480)
#-- Font for the text in the video
font = cv2.FONT_HERSHEY_PLAIN
```

In the main loop, the camera frame is read first, after which, the image will be converted to gray-scale to compare with the dictionary through the cvtColor command and the BGR2GRAY option (this is because, in OpenCV, images are stored in blue, green, and red). Then, the function detectMarkers from the Aruco library can be used to which the dictionary, gray-scale image, parameters, camera matrix, and

distortion matrix will be passed. As a result, the marker corners, IDs and rejected will be obtained. If the tag is found in the vector IDs, then the estimatePoseSingleMarker function can be used, passing it the corners, the size of the marker, and the two matrices obtained from the calibration. The output of variable ret, will be a vector of vectors (tvec and rvec) containing the rotation and position of each marker within the camera frame: specifically, the rvec vector represents the attitude of the marker with respect to the camera frame while the tvec vector represents the position of the marker in the camera frame. Since in the application, only one marker will be had, just the first entry of these vectors will be checked (it is possible to modify this recognition script, adding more markers and reading more elements of the above cited vectors). After this, borders will be drawn to the detected marker and its reference frame, in the showed camera frame.

```
while True:
ret, frame = cap.read()
gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
#-- Find all the aruco markers in the image
corners, ids, rejected = aruco.detectMarkers(image=gray,
   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(frame, corners)
aruco.drawAxis(frame, camera_matrix, camera_distortion, rvec, tvec, 10)

cv2.imshow('CameraFrame', frame)
```

If it is desired to estimate Euler angles, it is possible to use a function already into opencv libraries [41], as follows:

```
# Checks if a matrix is a valid rotation matrix.
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])
```

The previous one, must be put before main loop; then, in the main again, it is possible to add informative text on the on-screen view, such as the position of marker relative to the camera frame, the marker's attitude with respect to flipped camera reference frame or, instead, the camera position and attitude with respect to the marker reference frame (thanks to the last two, it is possible to estimate the drone pose with respect from the marker) as if the marker is steady and the camera is moving (actual case of camera on board the drone and landing base with marker). Moreover, to calculate the rotation matrix, it will be used the Rodrigues method given in cv2 (this method is an efficient algorithm to transform all three basis vectors to compute a rotation matrix [42]): R_ct indicates the rotation of the tag with respect to the camera.

```
str_position = "MARKER Position x=%4.0f  y=%4.0f  z=%4.0f"%(tvec[0],
   tvec[1], tvec[2])
cv2.putText(frame, str_position, (0, 100), font, 1, (0, 255, 0), 2,
   cv2.LINE_AA)

#-- Obtain the rotation matrix tag->camera
R_ct = np.matrix(cv2.Rodrigues(rvec)[0])
R_tc = R_ct.T

#-- Get the attitude in terms of euler 321 (Needs to be flipped first)
roll_marker, pitch_marker, yaw_marker =
   rotationMatrixToEulerAngles(R_flip*R_tc)

#-- Print the marker's attitude respect to camera frame
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(frame, str_attitude, (0, 150), font, 1, (0, 255, 0), 2,
   cv2.LINE_AA)

#-- Get Position and attitude of the camera respect to the marker
```

```
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(frame, str_position, (0, 200), font, 1, (0, 255, 0),
   2, cv2.LINE_AA)

#-- Get the attitude of the camera respect to the frame
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(frame, str_attitude, (0, 250), font, 1, (0, 255, 0), 2,
    cv2.LINE_AA)
```

To terminate the recognition script, a keyboard button (in this case q for quit) is set that will interrupt the main loop.

```
key = cv2.waitKey(1) & 0xFF
if key == ord('q'):
cap.release()
cv2.destroyAllWindows()
break
```

With the previous script, it is therefore possible to perform recognition of the chosen marker, estimate its position and attitude relative to that of the camera and vice versa. An example of the result is as follows in the figure (4.9).

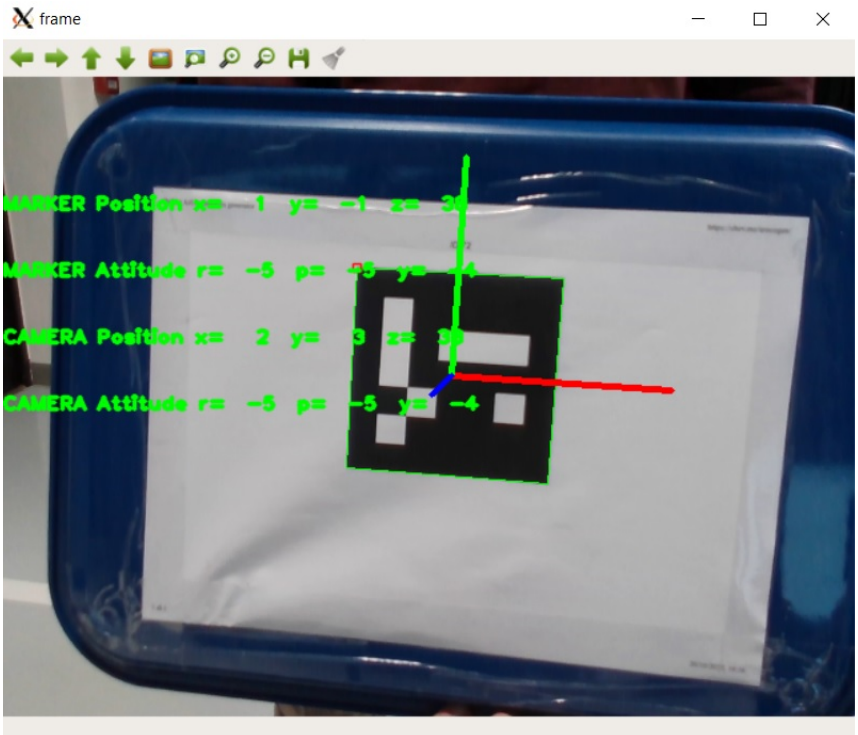


Figure 4.9: Aruco Marker Recognition, Attitude and Position Estimation

Chapter 5

Precision Landing Algorithm

After recognizing the marker correctly in the previous script, it will be converted to a python library that will then be called by the precision landing script. This script will connect to the vehicle, call the marker recognition library, fly the craft over the marker and land on it.

5.1 Precision Landing Logic

The algorithm designed follows seven simple steps to perform the precision landing:

- The marker is recognized and its location in the camera frame is obtained;
- This location will be converted from camera frame to body frame;
- The body frame location will be converted into North-East frame centered on the UAV center of gravity;
- The marker location latitude and longitude will be obtained by adding the north-east to the current vehicle's location;
- The vehicle is commanded to go to the marker location;
- The vehicle is commanded to fly over the marker and, if the error is low enough, it is commanded to descend;
- When the altitude is low enough, the vehicle is commanded to go to Land.

In the block diagram in the figure (5.1) below, the logic that the algorithm will follow during the precision landing phase is depicted.

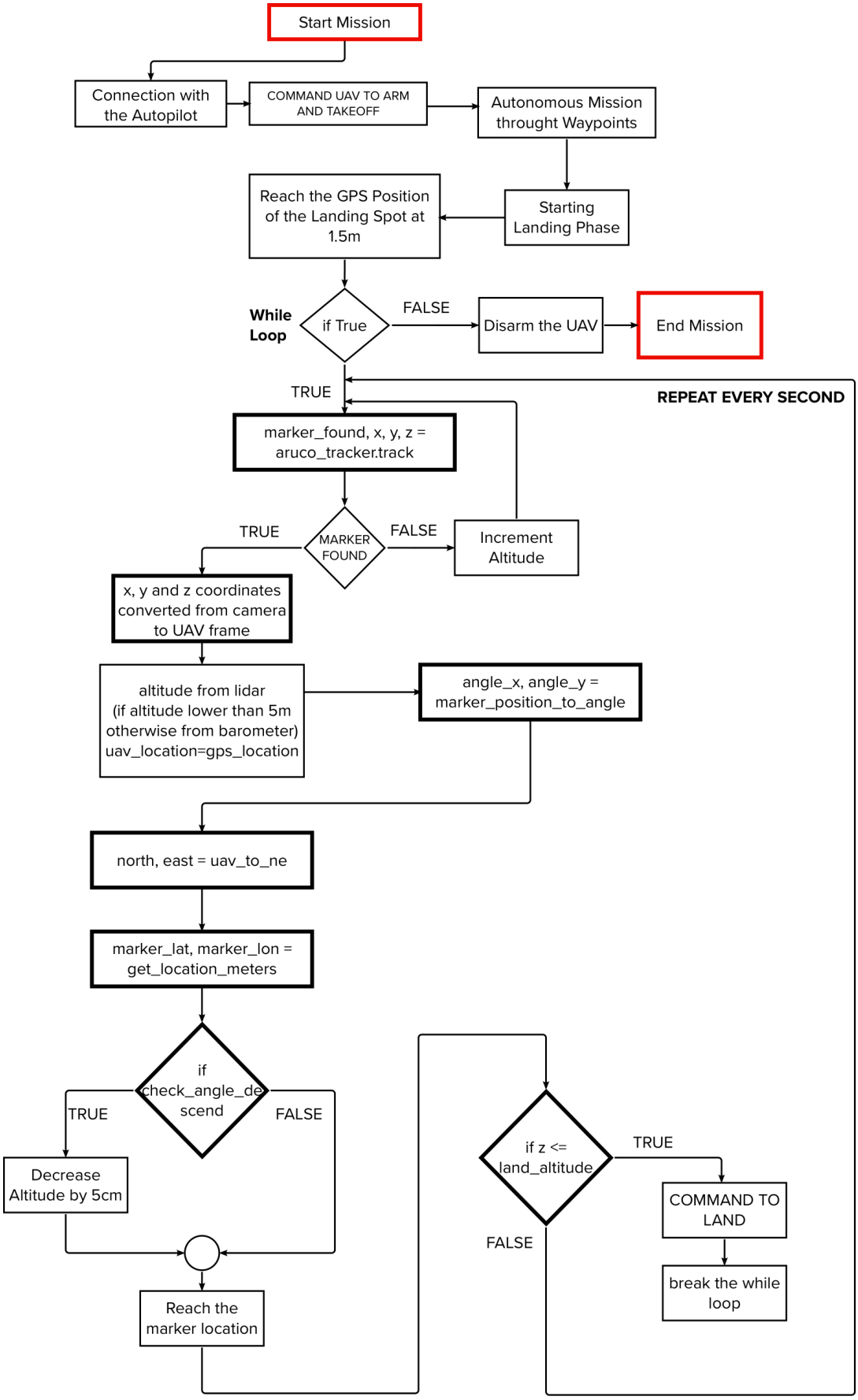


Figure 5.1: Precision Landing Algorithm

5.2 Precision Landing Script

5.2.1 Aruco Recognition into Library

As mentioned above, the first step is to convert the script presented in the previous chapter, to library so that it can be called in the precision landing script. Then, a class called ArucoSingleTracker will be created to which the id of the marker to be recognized, its size in centimeters, the camera and distortion matrix, the resolution of the camera, and the choice to show on screen the view of the camera on board the drone (this class is saved into *lib_ aruco_ recognition.py* file). With this library is possible to accomplish the first step so the marker is recognized and its location in the camera frame is obtained.

5.2.2 Camera frame to UAV frame Conversion

The second step is to convert the reference frame from camera to UAV. The camera has the X axis point right and the Y axis point down so, by assuming that the camera is down looking, the conversion will be:

```
def camera_to_uav(x_cam, y_cam):
x_uav =-y_cam
y_uav = x_cam
return(x_uav, y_uav)
```

The previous conversion can be easily understood by looking at the following two pictures: in the first one in figure (5.2), there is the upper view of drone and camera both facing up while, in the second one in figure (5.3) there is the rear view of the drone facing up and the rear view of camera facing down (as supposed when mounted on board).

Figure 5.2: Upper View of Reference Frames

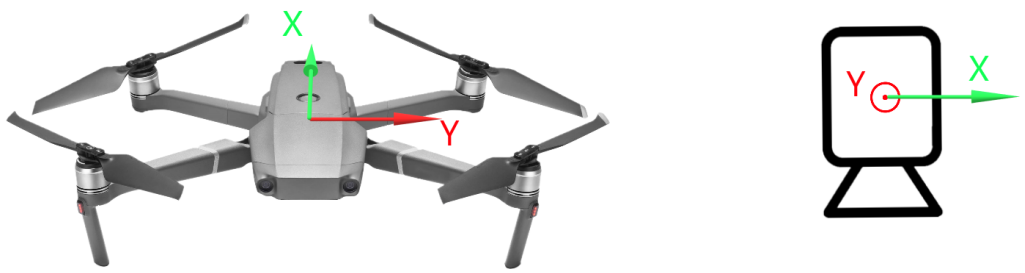


Figure 5.3: Rear View of Reference Frames

5.2.3 UAV frame to North-East frame conversion

The subsequent third step is to convert from the UAV body frame to north-east reference frame, rotating by the Yaw in radians:

```
def uav_to_ne(x_uav, y_uav, yaw_rad):
c = math.cos(yaw_rad)
s = math.sin(yaw_rad)

north = x_uav*c - y_uav*s
east = x_uav*s + y_uav*c
return(north, east)
```

5.2.4 Obtaining marker location Latitude and Longitude

In the following function, the actual location of the vehicle is increased by a deltaNorth and deltaEast in meters: in that way, it is possible to compute the latitude and longitude location of the marker, by starting from the actual location of the drone and by adding the converted distance from the center of the camera and the center of the marker. So, in the defined function below, it is possible to compute a Delta Latitude and Delta Longitude respect to the current position by assuming the ellipsoid model of the Earth; then, these delta quantities will be added to the current location obtained from the GPS and the location of the marker is the final result in terms of latitude and longitude as explained in the fourth step:

```
def get_location_metres(original_location, dNorth, dEast):

earth_radius=6378137.0 #Radius of "spherical" earth
dLat = dNorth/earth_radius
dLon = dEast/(earth_radius*math.cos(math.pi*original_location.lat/180))

print ('dlat, dlon', dLat, dLon)

#New position in decimal degrees
```

```
newlat = original_location.lat + (dLat * 180/math.pi)
newlon = original_location.lon + (dLon * 180/math.pi)
return(newlat, newlon)
```

5.2.5 Marker Position to Angle

The following function is really useful to estimate the line of sight angles of the current marker with respect to the center of the drone; this will be used to check whether the error is within a certain value and the descend is allowed or not:

```
def marker_position_to_angle(x, y, z):

angle_x = math.atan2(x,z)
angle_y = math.atan2(y,z)

return (angle_x, angle_y)
```

5.2.6 Main Script

After creating all the necessary functions to be used in the landing phase, the main script and loop can be created: First a connection to our autopilot will be made, then the marker that we want to recognize, its size and the update frequency to send commands to the autopilot will be indicated. In addition, it is necessary to specify the altitude under which the vehicle will be set in landing mode, the angle of descent (when the angle between the marker and the vehicle is within this value, the vehicle can descend) and the descending speed (in cm/s).

```
from os import sys, path
sys.path.append(path.dirname(path.dirname(path.abspath(__file__))))

import time
import math
import argparse
import socket

from dronekit import *
from pymavlink import mavutil

#--- Import our Aurco Library
from lib_aruco_recognition import *

# PARAMETERS FOR CONNECTION TO THE VEHICLE
connection_string = "/dev/ttyAMA0" #or ttyACM0 if connected with USB
baud_rate = 921600

print(">>>> Connecting with the UAV <<<")
vehicle = connect(connection_string, baud=baud_rate, wait_ready=True)
```

```
vehicle.wait_ready('autopilot_version')
print('Autopilot version: %s'%vehicle.version)

print("Landing Start.")
vehicle.mode = VehicleMode("GUIDED")

print ("Starting Landing Marker Recognition.")
vehicle.airspeed = 0.20 #-- set the default speed
vehicle.groundspeed = 0.20 #-- set ground speed

#-----------------------------------------------------
#-------------- FUNCTIONS
Here there will be the functions presented previously
#-----------------------------------------------------

rad_2_deg = 180.0/math.pi
deg_2_rad = 1.0/rad_2_deg

#-------------------------------------------------
#-------------- LANDING MARKER
#-------------------------------------------------
#--- Define Tag
id_to_find = 72
marker_size = 9.6 #[cm] - marker 100mm
freq_send = 1 #- Hz

land_alt_cm = 25.0 #for a safety land without damages
angle_descend = 20*deg_2_rad
land_speed_cms = 5.0 #[cm/s]
```

After the initial parameters, the distortion and camera matrix will be loaded into the script by getting the full path of current directory (it is mandatory to have the two file inside the same folder of the script or it is necessary to change the calibration path). Then, the *aruco_tracker* object will be defined by calling the class ArucoSingleTracker with the corresponding parameters:

```
calib_path = path.dirname(path.abspath(__file__))
camera_matrix = np.loadtxt(calib_path+'cameraMatrix.txt',
   delimiter=',')
camera_distortion = np.loadtxt(calib_path+'cameraDistortion.txt',
   delimiter=',')
aruco_tracker = ArucoSingleTracker(id_to_find=id_to_find,
   marker_size=marker_size, show_video=False,
   camera_matrix=camera_matrix, camera_distortion=camera_distortion)
```

Now, the main loop can start: here the track parameter is extracted as output from the aruco_tracker object; it will return whether the Aruco has been found or not and its position. And, then, if the marker is found, its coordinates will be converted into UAV reference frame; if the altitude of the drone relative to the marker is lower than 5m, the measurement of the Z-axis is taken from the lidar measurement, which is the most accurate height sensor (the barometer does not have very accurate measurements when the vehicle is close to the ground while the information taken from the camera is estimated and therefore not as accurate as that of the lidar). Moreover, the UAV location will be saved into a temporary object, the marker location will be converted into angles and, if the marker is found, there will be a message printed every time instant (with the frequency set at the beginning). After that, the function to convert the coordinates into north-east frame will be used and, so, it is possible to estimate the marker latitude and longitude (these two come from the UAV location plus the deltaNorth and deltaEast):

```
time_0 = time.time()

while True:

marker_found, x_cm, y_cm, z_cm = aruco_tracker.track(loop=False)

if marker_found:
x_cm, y_cm = camera_to_uav(x_cm, y_cm)
z_cm = vehicle.rangefinder.distance*100.0
uav_location = vehicle.location.global_relative_frame

#-- If high altitude, use baro rather than lidar
if vehicle.rangefinder.distance >= 5.0:
print
z_cm = uav_location.alt*100.0

angle_x, angle_y = marker_position_to_angle(x_cm, y_cm, z_cm)

if time.time() >= time_0 + 1.0/freq_send:
time_0 = time.time()

print (" ")
print ("Altitude = %.0fcm"%z_cm)
print ("Marker found x = %5.0f cm  y = %5.0f cm -> angle_x = %5f
   angle_y = %5f"%(x_cm, y_cm, angle_x*rad_2_deg, angle_y*rad_2_deg))

north, east = uav_to_ne(x_cm, y_cm, vehicle.attitude.yaw)
print ("Marker N = %5.0f cm   E = %5.0f cm   Yaw = %.0f deg"%(north,
   east, vehicle.attitude.yaw*rad_2_deg))

marker_lat, marker_lon = get_location_metres(uav_location, north*0.01,
   east*0.01)
```

The main loop is completed with the addition of two controls: the first (made with the set frequency as above) in which the descent angle is checked and, if less than the set value, commands the UAV to descend of 5cm in 1s as the set frequency (as long as the error on the angle does not exceed the threshold value or the marker is no longer seen and, in this case, the command is to reach the last marker position); the second control, is to check whether the altitude is below a set safety value and, when it is, the command to land the drone is given and the main loop is exited; then, it is possible to disarm the vehicle, close the connection with the autopilot and the mission is completed:

```
 #-- If angle is good, descend
if check_angle_descend(angle_x, angle_y, angle_descend):
print ("Low error: descending")
location_marker = LocationGlobalRelative(marker_lat, marker_lon,
   uav_location.alt-(land_speed_cms*0.01/freq_send))
else:
#-- Mantain the same altitude and save the marker location
location_marker = LocationGlobalRelative(marker_lat, marker_lon,
   uav_location.alt)

vehicle.simple_goto(location_marker)
print ("UAV Location Lat = %.7f  Lon = %.7f"%(uav_location.lat,
   uav_location.lon))
print ("Commanding to Lat = %.7f  Lon = %.7f"%(location_marker.lat,
   location_marker.lon))

#--- Command to land
if z_cm <= land_alt_cm:
print (" -->>COMMANDING TO LAND<<")
vehicle.mode = "LAND"
break

vehicle.armed = False #-- disarming the motors
# CLOSING CONNECTION WITH THE VEHICLE
vehicle.close()
print("Mission Complete.")
```

5.2.7 Adding Details

Through this just-concluded script, it is also possible to perform an autonomous mission by simply creating an arm_and_takeoff function and setting GPS waypoints to be reached within it or, alternatively, set the AUTO flight mode after takeoff, and have the autopilot follow the waypoints saved in its memory by setting them via Mission Planner, before switching to GUIDED mode and performing the final landing phase. Arm and Takeoff Function:

```
def arm_and_takeoff(tgt_altitude):
print("Arming motors")

vehicle.mode = VehicleMode("GUIDED")
vehicle.armed = True

while not vehicle.armed:
time.sleep(1)

print("Takeoff")
vehicle.simple_takeoff(tgt_altitude)

while True:
altitude = vehicle.rangefinder.distance

if altitude >= tgt_altitude - 0.1:
print("Altitude Reached!")
break
time.sleep(1)
```

Then, to create an autonomous mission through Python, after the connection with the autopilot, will be inserted:

```
print("Mission Start.")
arm_and_takeoff(2) #2 is the altitude in meters
time.sleep(1) #-- relax for a moment
vehicle.mode = VehicleMode("GUIDED")
vehicle.airspeed = 2 #-- set the default speed
vehicle.groundspeed = 2 #-- set ground speed

print ("Going to waypoint 1")
wp1 = LocationGlobalRelative(45.0314728, 7.6191611, 2)
vehicle.simple_goto(wp1)
time.sleep(1)

print ("Going to waypoint 2")
wp2 = LocationGlobalRelative(45.0314540, 7.6191780, 2)
vehicle.simple_goto(wp2)
time.sleep(1)
```

```
#print ("Going to waypoint 3")
wp3 = LocationGlobalRelative(45.0314713, 7.6191869, 2)
vehicle.simple_goto(wp3)
time.sleep(1)

#-- go to landing point
print ("Going to landing point")
lnd_wp = LocationGlobalRelative(45.0314470, 7.6191202, 1.5)
vehicle.simple_goto(lnd_wp)
time.sleep(1)

print ("Starting Landing Marker Recognition.")
vehicle.airspeed = 0.15 #-- set the default speed
vehicle.groundspeed = 0.15 #-- set ground speed

#-- starting landing marker recognition
#-- continue as above
```

Or, to create autonomous mission through Mission Planner, it is possible to set takeoff and different waypoints (or just waypoints) and save them into the memory of the autopilot while, into the script:

```
print("Mission Start.")
#arm_and_takeoff(2)
vehicle.mode = VehicleMode("AUTO")
#-- Here the saved mission in Mission Planner will be done
time.sleep(1)

#-- go to landing point
print ("Going to landing point")
lnd_wp = LocationGlobalRelative(45.0314470, 7.6191202, 1.5)
vehicle.simple_goto(lnd_wp)
time.sleep(1)

print ("Starting Landing Marker Recognition.")
vehicle.airspeed = 0.15 #-- set the default speed
vehicle.groundspeed = 0.15 #-- set ground speed

#-- starting landing marker recognition
#-- continue as above
```

Chapter 6

Experimental Results

Upon completion of the implementation of the precision landing algorithm presented in the previous chapter, the experimental testing phase was carried out in an outdoor environment and with a static landing point on the ground. The phase of testing consists of manually or automatically bringing the drone to the hypothetical landing point and waiting for the on-board computer and autopilot to communicate with each other: as soon as this happens, the drone's flight mode will change from *STABILIZE* to *GUIDED* and the algorithm will continue as in the scheme shown in the figure (5.1). Numerous tests were carried out with this configuration, which presented several problems as below:

1. Descending speed;
2. Air/ground speed;
3. GPS tracking error;
4. Incorrect altitude measurement;
5. Light conditions;
6. Marker size;
7. Type of camera.

6.1 Descending Speed

A first issue is related to the descent speed, when the algorithm detects an error below the indicated threshold and tells the autopilot to descend: in fact, as explained in the 5.2.6 section, if the descent angle is below a certain value, the UAV height is reduced by the value (land_speed_cms×0.01)÷freq_send which returns a quantity in meters.

$$altitude\_reduction = uav\_location.alt - (\frac{land\_speed\_cms \cdot 0.01}{freq\_send})$$

$$altitude\_reduction = \mathrm{m} - (\frac{\mathrm{cm/s} \cdot 0.01}{\mathrm{Hz}}) = \mathrm{m} - (\mathrm{m/s} \cdot \frac{1}{\mathrm{Hz}}) = \mathrm{m} - (\mathrm{m/s} \cdot \mathrm{s}) = \mathrm{m} - \mathrm{m} = \mathrm{m}$$

Thus, following the previous logic, working with too high a descent speed was observed to result in greater loss of vision of the marker due to the companion computer's lack of proper timing of image processing. In addition, too high a descent speed can lead to displacement of the drone from the marker as a result of external disturbances. At the end of the various tests, the optimal solution that did not have the above was to set the descent speed to 5cm per second, and thus, each analyzed frame that meets the imposed characteristics leads to a descent of 0.05m per time.

6.2 Air/Ground Speed

An additional issue that has led to drift and loss of marker vision is caused by the drone's navigation speed. In particular, it was observed that when the drone had to reach the landing spot, if its movement speed was too high, it would overshoot the marker and fail to find it again. Or if the speed was too high during the positioning corrections in the landing phase, the drone would overshoot the spot where the marker was located as a result of the too-fast correction. The previous case histories ended in numerous time losses in the landing phase lengthening the time considerably and in temporary or permanent loss (also due to the 1-3m error of the normal GPS) of the maker and a subsequent failure of the mission. Hence, for the solution of the exposed problem, a reduction of in-flight speeds was chosen as follows, which ensured an almost complete zeroing of the above.

$$vehicle.airspeed = 0.15\mathrm{m/s} \text{ — } vehicle.groundspeed = 0.15\mathrm{m/s}$$

6.3 GPS tracking Error

A very influential experimental issue is related to the GPS error; in fact, in a preliminary stage, tests were carried out with a simple GPS without the RTK configuration thus maintaining the localization error of 1-3m. This greatly affected the tests since, when the marker went out of the field of view and the algorithm had to bring the craft to the last recorded position, it was affected by a considerable error and thus made it impossible to find the marker. This was also the case when during the flight, the number of connected satellites was reduced (e.g. due to cloudy weather) below ten increasing the localization error. Opting for an RTK configuration and

reducing the localization error to the order of tens of centimeters was essential to achieve optimal and repeated results over time (in the case of classical GPS-only, the landing phase was rarely successful and often the marker was lost because the drone was sent to a location affected by a large error).

6.4 Incorrect Altitude Measurement

Another observation derived from the practical tests is related to the reference sensors for the drone's altitude relative to the ground. Specifically, it is possible to obtain this information from the barometric reading or the estimate obtained by computer vision on the marker (the size in pixels of the marker in the camera frame is converted to meters and from this measurement it is possible to estimate how far the camera, and therefore the drone, is from the marker seen. Both readings are not effective, however: the barometric reading is greatly affected by the air currents that the drone generates to create lift and which are reflected, in large quantities, from the ground when the drone is at a height below about 3 meters; the height estimate obtained by video camera, on the other hand, is more accurate below 2 meters in height because the marker is clearly recognizable within the camera frame but, above the mentioned height, it is not clearly visible given its small size (10cm per side). In addition, the same estimation by video camera loses its effectiveness at low heights where the marker goes out of the field of view and, therefore, it is not possible to estimate the distance with an object that cannot be seen. To solve these height measurement issues, a lidar pointing downward was chosen to support the architecture used: for heights above 5m the barometer measurement was used which was very effective while, for heights below 5m the measurement given by the lidar was used which is accurate to the order of centimeters, thus usable even in close proximity with the terrain/marker.

6.5 Light Conditions

One issue due to the chosen solution is related to lighting conditions. In fact, the use of visual systems such as the simple camera is heavily characterized by the absence of light; moreover, during testing, a problem with excessive light also arose: using a rigid landing base with the marker fixed on it, it was noticed that during the hours when the sun directly hit the base, it reflected the sunlight. The presence of shiny tape also led to reflections. These light reflections proved problematic during recognition as they interfered with the clean visibility of the camera. To overcome this problem, an opaque landing base and tape was chosen so as to reduce light reflections altogether.

6.6 Marker Size

One of the most important observations to be made concerns the choice of marker size. The latter directly influences the height at which the marker is detected and recognized correctly. In fact, a marker of small size is not detected from high distances (more than 1.5m from the marker) but can be followed up to very small distances (about 20cm from the marker). On the other hand, however, an oversized marker, is detected from greater distances (up to 3.5m from the marker) but cannot be tracked when the vehicle will be in the vicinity of the marker (below 50cm from the marker). According to the algorithm, in fact, when the drone is below a set height (considered safe to land very close to the target), it is commanded to land and then switch to *LAND* mode; however, this results in the interruption of the tracking phase carried out by the script and the start of the final landing phase managed by the autopilot and, at this stage, there may be drift phenomena due to wind disturbances or GPS tracking errors. For an optimal solution, which agrees with the application, it must be imagined that the drone will also fly in indoor environments and will have to land aboard a rover about 90cm high: therefore, a marker size tradeoff was chosen, that would allow its recognition at heights indicative for indoor environments and that would be trackable as much as possible when drone and landing spot are close, so as to delay the landing command as much as possible and greatly reduce errors caused by possible drift. Therefore, the parameters chosen for a solution closer to the needs are as follows and ensured the completed landing always within a 10cm around the marker:

$$marker\_size = 9.6\text{cm} \text{ — } land\_alt\_cm = 25.0\text{cm}$$

6.7 Type of Camera

The first tests were performed using the Raspberry Pi Camera Noir: this choice was dictated by the already present Intel Realsense Depth camera used for obstacle avoidance. Therefore, to avoid overloading the on-board computer with a performant camera, a more basic camera was chosen. During testing, however, this choice proved to be ineffective since the said camera has low resolution and performance characteristics necessary for proper marker recognition while providing clean images for computer processing. Therefore, it was returned to using an Intel Realsense Depth camera that has much better hardware characteristics to support the application in progress. Thus, if in the future it is desired to use a vehicle that takes advantage of two Realsense cameras for obstacle avoidance and precision landing, a second on-board computer (each computer will receive and process images received from one camera) will have to be included.

Chapter 7

Conclusions and Future Implementations

7.1 Conclusion

At the end of the experimental tests, it was observed that after the modifications and improvements set in the previous chapter, the algorithm can correctly reach and detect the marker, keep within its range throughout the landing phase and land on it with a margin of error of about 10cm. Drift issues were reset almost to zero by reducing the airspeeds (of descent and lateral movement to correct the position of the UAV) and the altitude from which the land command was sent while those related to the error of positioning and finding the marker were reset to zero by using a gps with RTK system. The system was found to be robust and reliable during all tests, and no unexpected issues arose.

7.2 Future Implementations

The next steps for optimizing the algorithm involve adding safety systems for any unexpected situations, perceptual enhancement of the camera to increase the drone's field of view and removal of issues caused by unfavorable lighting conditions, and performance enhancement regarding marker tracking. With the aforementioned, the system will be more robust and perform better and can be used in the future as a base for landing on a mobile platform.

7.2.1 Robustness

To improve the system in terms of robustness, safety checks can be inserted for unexpected situations. For example, thinking about the situation where the marker is not seen, the drone is theoretically on its position, and is not at the altitude for landing, the algortim provides for increasing the altitude until the marker is found again; this could evolve into critical situations leading the drone to climb up to hit a ceiling when the mission is carried out in an indoor environment. An additional safety protocol would be to include virtual geographic fences that the drone would never have to leave, such as the industrial area in which it is used (the drone should

not fly in areas where flight is not allowed or expected due to the presence of people or other causes).

7.2.2 Computer Vision

In this application, two cameras were used, one of which, the Intel Realsense, is very high-performance. This, however, is a depth camera that sees its best use for obstacle avoidance algorithms; for precision landing, on the other hand, the most efficient cameras are stereo cameras assigned to tracking applications, such as Intel Realsense Tracking camera T265 or even cameras with fish-eye systems. Both of the above are excellent in extended field-of-view optics and useful, therefore, both in the preliminary phase of landing when looking for the marker at the GPS position reached, and in the final phase of landing when the drone is very close to the marker and misses it with ease: these systems would ensure the faster detection of the marker at the beginning of the landing phase and the achievement of even shorter distances, in the final phase, so as to lower the margin of error.
An additional improvement that can be made to the visual system concerns increased visual perception in low-light conditions. For this situation, it is possible to think of an infrared camera with night vision that, therefore, aids in the detection and tracking of the marker in low light situations or, alternatively, the camera can be supported with an IR-lock sensor positioned on the drone pointing toward the ground and an infrared beacon positioned at the landing site; combining the latter system with the algorithm presented in this thesis, adapting them to work symbiotically, can cover those situations in which the landing is carried out in unfavorable light conditions.

7.2.3 Tracking Performance

In the algorithm presented in this thesis work, the control system is carried out to keep the marker in the camera's field of view at all times: thus, the control loop, makes sure that if the marker partially leaves the field of view, its last stored GPS position is reached. It is possible to augment the control system by going to implement a PID that works on the distance between the center of the marker and the focal center of the camera: thus, the error to be set to zero, as an input to the control system, is exactly the distance between the two centers. By trying to keep this distance as small as possible, throughout the landing phase, the system will be significantly more accurate since it will no longer happen that the marker is lost during the descent by being at the limit of the field of view. Also, in the final landing phase, the marker will be able to be tracked more when the drone is at a reduced distance from it.

Bibliography

[1] Yang Gui, Pengyu Guo, Hongliang Zhang, Zhihui Lei, Xiang Zhou, Jing Du, and Qifeng Yu. Airborne vision-based navigation method for uav accuracy landing using infrared lamps. *Journal of Intelligent & Robotic Systems*, 72(2):197–218, Nov 2013.

[2] Zhanpeng Gan, Huarong Xu, Yuanrong He, Wei Cao, and Guanhua Chen. Autonomous landing point retrieval algorithm for uavs based on 3d environment perception. In *2021 IEEE 7th International Conference on Virtual Reality (ICVR)*, pages 104–108, 2021.

[3] Juhi Ajmera, Siddharthan PR, Ramaravind K.M., Gautham Vasan, Naresh Balaji, and V. Sankaranarayanan. Autonomous visual tracking and landing of a quadrotor on a moving platform. In *2015 Third International Conference on Image Information Processing (ICIIP)*, pages 342–347, 2015.

[4] Davide Falanga, Alessio Zanchettin, Alessandro Simovic, Jeffrey Delmerico, and Davide Scaramuzza. Vision-based autonomous quadrotor landing on a moving platform. In *2017 IEEE International Symposium on Safety, Security and Rescue Robotics (SSRR)*, pages 200–207, 2017.

[5] Rong Liu, Jianjun Yi, Yajun Zhang, Bo Zhou, Wenlong Zheng, Hailei Wu, Shuqing Cao, and Jinzhen Mu. Vision-guided autonomous landing of multirotor uav on fixed landing marker. In *2020 IEEE International Conference on Artificial Intelligence and Computer Applications (ICAICA)*, pages 455–458, 2020.

[6] Bo-Yang Xing, Feng Pan, Xiao-Xue Feng, Wei-Xing Li, and Qi Gao. Autonomous landing of a micro aerial vehicle on a moving platform using a composite landmark. *International Journal of Aerospace Engineering*, 2019:4723869, May 2019.

[7] JeongWoon Kim, Yeondeuk Jung, Dasol Lee, and David Hyunchul Shim. Outdoor autonomous landing on a moving platform for quadrotors using an omnidirectional camera. In *2014 International Conference on Unmanned Aircraft Systems (ICUAS)*, pages 1243–1252, 2014.

[8] Zhanglong Wang, Haoping She, and Weiyong Si. Autonomous landing of multirotors uav with monocular gimbaled camera on moving vehicle. In *2017 13th IEEE International Conference on Control Automation (ICCA)*, pages 408–412, 2017.

[9] Jiaqi Jiang, Yuhua Qi, Muhammad Ibrahim, Jianan Wang, Chunyan Wang, and Jiayuan Shan. Quadrotors' low-cost vision-based autonomous landing architecture on a moving platform. In *2018 15th International Conference on Control, Automation, Robotics and Vision (ICARCV)*, pages 448–453, 2018.

[10] Le Qi, Baoxi Yuan, Peng Ma, Yingxia Guo, Feng Wang, and Chen Mi. Scene simulation and cooperative target detection during uav autonomous landing. In *2020 International Conference on Robots Intelligent System (ICRIS)*, pages 40–43, 2020.

[11] Riccardo Polvara, Sanjay Sharma, Jian Wan, Andrew Manning, and Robert Sutton. Towards autonomous landing on a moving vessel through fiducial markers. In *2017 European Conference on Mobile Robots (ECMR)*, pages 1–6, 2017.

[12] Aleix Paris, Brett T. Lopez, and Jonathan P. How. Dynamic landing of an autonomous quadrotor on a moving platform in turbulent wind conditions. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 9577–9583, 2020.

[13] Artur Khazetdinov, Aufar Zakiev, Tatyana Tsoy, Mikhail Svinin, and Evgeni Magid. Embedded aruco: a novel approach for high precision uav landing. In *2021 International Siberian Conference on Control and Communications (SIBCON)*, pages 1–6, 2021.

[14] Zhou Jian, Wang Xin-Min, and Wang Xiao-Yan. Automatic landing control of uav based on optical guidance. In *2012 International Conference on Industrial Control and Electronics Engineering*, pages 152–155, 2012.

[15] Nguyen Xuan Mung, Jun Yong Lee, Seok Tae Lee, and Sung Kyung Hong. Target state estimation for uav, target tracking and precision landing control: Algorithm and verification system. In *2021 21st International Conference on Control, Automation and Systems (ICCAS)*, pages 173–177, 2021.

[16] Morteza Alijani and Anas Osman. Autonomous landing of uav on moving platform: A mathematical approach. In *2020 International Conference on Control, Automation and Diagnosis (ICCAD)*, pages 1–6, 2020.

[17] Yi Feng, Cong Zhang, Stanley Baek, Samir Rawashdeh, and Alireza Mohammadi. Autonomous landing of a uav on a moving platform using model predictive control. *Drones*, 2(4), 2018.

[18] Miguel A. Olivares-Mendez, Iván F. Mondragón, and Pascual Campoy. Autonomous landing of an unmanned aerial vehicle using image-based fuzzy control. *IFAC Proceedings Volumes*, 46(30):79–86, 2013. 2nd IFAC Workshop on Research, Education and Development of Unmanned Aerial Systems.

[19] Shadi Abujoub, Johanna McPhee, Cassidy Westin, and Rishad A. Irani. Unmanned aerial vehicle landing on maritime vessels using signal prediction of the ship motion. In *OCEANS 2018 MTS/IEEE Charleston*, pages 1–9, 2018.

[20] Andrea Nisticò, Marco Baglietto, Enrico Simetti, Giuseppe Casalino, and Alessandro Sperindè. Marea project: Uav landing procedure on a moving and floating platform. In *OCEANS 2017 - Anchorage*, pages 1–10, 2017.

[21] Alexandre Borowczyk, Duc-Tien Nguyen, André Phu-Van Nguyen, Dang Quang Nguyen, David Saussié, and Jerome Le Ny. Autonomous landing of a multirotor micro air vehicle on a high velocity ground vehicle**this work was partially supported by cfi jelf award 32848 and a hardware donation from dji. *IFAC-PapersOnLine*, 50(1):10488–10494, 2017. 20th IFAC World Congress.

[22] Patrick Henkel, Andreas Sperl, Ulrich Mittmann, Torsten Fritzel, Rüdiger Strauss, and Hans Steiner. Precise 6d rtk positioning system for uav-based near-field antenna measurements. In *2020 14th European Conference on Antennas and Propagation (EuCAP)*, pages 1–5, 2020.

[23] Thien Hoang Nguyen, Muqing Cao, Thien-Minh Nguyen, and Lihua Xie. Post-mission autonomous return and precision landing of uav. In *2018 15th International Conference on Control, Automation, Robotics and Vision (ICARCV)*, pages 1747–1752, 2018.

[24] P. Marcon, J. Janousek, and R. Kadlec. Vision-based and differential global positioning system to ensure precise autonomous landing of uavs. In *2018 Progress in Electromagnetics Research Symposium (PIERS-Toyama)*, pages 542–546, 2018.

[25] Ju Wang, Devon McKiver, Sagar Pandit, Ahmed F. Abdelzaher, Joel Washington, and Weibang Chen. Precision uav landing control based on visual detection. In *2020 IEEE Conference on Multimedia Information Processing and Retrieval (MIPR)*, pages 205–208, 2020.

[26] Claudio Roberto de Ceglia. Fixit assemblaggio telaio readytosky zd550 quadcopter frame. `https://www.youtube.com/watch?v=BsV0u_TfFQk`, `https://cim40.sharepoint.com/:v:/r/sites/repository/Documenti%20condivisi/PROGETTI/INTERNI/FLAGSHIP%20PROJECT/Documenti/Documenti%20tecnici/Lavoro%20di%20Tesi/Drone%20Navigation/Claudio%20Roberto%20De%20Ceglia/Video%20Documentazione/0.AssemblaggioTelaio.mp4?csf=1&web=1&e=zHpLAe`, 2022.

[27] Claudio Roberto de Ceglia. Fixit assemblaggio elettronica. `https://www.youtube.com/watch?v=MAHVaalIfJc`, `https://cim40.sharepoint.com/:v:/r/sites/repository/Documenti%20condivisi/PROGETTI/INTERNI/FLAGSHIP%20PROJECT/Documenti/Documenti%20tecnici/Lavoro%20di%20Tesi/Drone%20Navigation/Claudio%20Roberto%20De%20Ceglia/Video%20Documentazione/1.AssemblaggioComponentiDrone.mp4?csf=1&web=1&e=8y8aNx`, 2022.

[28] Randy Mackay, Henry Wurzburg, Hamish Willee, Graham James Addis. Connecting common power module to the autopilot. `https://ardupilot.org/copter/docs/common-3dr-power-module.html`, 2020.

[29] Randy Mackay, Henry Wurzburg, Tatsuya Yamaguchi, Geofrancis, Nicholas Kruzan, Norimboo. Connecting benewake tfmini/tfmini plus lidar to the autopilot. https://ardupilot.org/copter/docs/common-benewake-tfmini-lidar.html, 2022.

[30] Henry Wurzburg, Peter Barker, Hamish Willee, Randy Mackay, Stephen Dade, Lars Kellogg-Stedman, Graham James Addis, Charlie Johnson. Pixhawk overview. https://ardupilot.org/copter/docs/common-pixhawk-overview.html, 2022.

[31] Claudio Roberto de Ceglia. Fixit prima configurazione e calibrazione di un drone. https://www.youtube.com/watch?v=CzjUlMCvzWs, https://cim40.sharepoint.com/:v:/r/sites/repository/Documenti%20condivisi/PROGETTI/INTERNI/FLAGSHIP%20PROJECT/Documenti/Documenti%20tecnici/Lavoro%20di%20Tesi/Drone%20Navigation/Claudio%20Roberto%20De%20Ceglia/Video%20Documentazione/2.ConfigurazioneCalibrazioneIniziale.mp4?csf=1&web=1&e=AehlSh, 2022.

[32] Claudio Roberto de Ceglia. Fixit configurazione aggiuntiva di un drone. https://www.youtube.com/watch?v=DAov4YbCKzA, https://cim40.sharepoint.com/:v:/r/sites/repository/Documenti%20condivisi/PROGETTI/INTERNI/FLAGSHIP%20PROJECT/Documenti/Documenti%20tecnici/Lavoro%20di%20Tesi/Drone%20Navigation/Claudio%20Roberto%20De%20Ceglia/Video%20Documentazione/3.ConfigurazioneAggiuntiva.mp4?csf=1&web=1&e=XOmr8j, 2022.

[33] Randy Mackay, Henry Wurzburg, Hamish Willee, Peter Hall, Graham James Addis, Craig Elder, Andrew Tridgell. Connect escs and motors. https://ardupilot.org/copter/docs/connect-escs-and-motors.html, 2022.

[34] Claudio Roberto de Ceglia. Fixit pianificazione missioni di volo con mission planner. https://www.youtube.com/watch?v=S6om1NJ6QdE, https://cim40.sharepoint.com/:v:/r/sites/repository/Documenti%20condivisi/PROGETTI/INTERNI/FLAGSHIP%20PROJECT/Documenti/Documenti%20tecnici/Lavoro%20di%20Tesi/Drone%20Navigation/Claudio%20Roberto%20De%20Ceglia/Video%20Documentazione/4.PianificazioneMissioniDiVolo.mp4?csf=1&web=1&e=Or0Zrd, 2022.

[35] Henry Wurzburg, Matt, Pierre Kancir, Mirko Denecke, Ian, Madangler1, Hamish Willee. The cube overview. https://ardupilot.org/copter/docs/common-thecube-overview.html, 2022.

[36] Raspberry Pi Foundation. Raspberry pi operating system images. https://www.raspberrypi.com/software/operating-systems/.

[37] NVIDIA Developer. Jetson nano developer kit sd card image. https://developer.nvidia.com/embedded/l4t/r32_release_v7.1/jp_4.6.1_b110_sd_card/jeston_nano/jetson-nano-jp461-sd-card-image.zip.

[38] S. Garrido-Jurado, R. Muñoz-Salinas, F.J. Madrid-Cuevas, and M.J. Marín-Jiménez. Automatic generation and detection of highly reliable fiducial markers under occlusion. *Pattern Recognition*, 47(6):2280–2292, 2014.

[39] OpenCV-dev Documentation. Opencv camera calibration. `https://docs.opencv.org/3.0-beta/doc/py_tutorials/py_calib3d/py_calibration/py_calibration.html#calibration`.

[40] Aruco markers generator. `https://chev.me/arucogen/`.

[41] Satya Mallick. Rotation matrix to euler angles. `https://learnopencv.com/rotation-matrix-to-euler-angles/`, 2016.

[42] Wikipedia contributors. Rodrigues' rotation formula — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Rodrigues%27_rotation_formula&oldid=1122338410`, 2022. [Online; accessed 21-November-2022].