POLITECNICO DI TORINO

Master's degree
in Mechatronic Engineering

Setup and configuration of a swarm of autonomous UAVs in an indoor environment for distributed target estimation and tracking

Supervisors
Prof. Giorgio Guglieri
Dr. Stefano Primatesta
Ing. Enrico Ferrera

Candidate
Davide Morazzo
301239

October 2023

Abstract

The main objective of this thesis project is to develop a working configuration for a swarm of drones in an indoor flight environment. The swarm is composed of 4 small quadcopters, each equipped with an onboard computer that is able to run programs to control the drone movements, and also to communicate with all the other agents in the network, like a ground station and the other drones. The flight environment consists of a safety cage where drones can fly inside, and a Vicon motion capture system to locate the position and the orientation of each drone in order to have an external position feedback. The work concerned both hardware and software aspects of the drones, starting from the physical wiring of the boards to the software setup of each component, establishing also a communication framework using ROS 2. Another important aspect of the work is the integration of the Vicon system in order to accurately estimate the drones' position. The secondary objective of this project focuses on the development of a task to be implemented on the former mentioned setup. First, an algorithm is implemented for a diffused estimation, with an information form Kalman filter, of the position of an ArUco marker on the ground. Then a flocking algorithm is used, in order to control the inter-drone distances and make the swarm follow the position of the ArUco marker. The aforementioned algorithms implement the work already proposed by the research group by Fausto Francesco Lizzio et al. in "Design and SITL Performance of an online Distributed Target Estimation for UAV Swarm". The aforementioned work has been tested by the authors only in simulation, therefore the implementation of the proposed algorithms on the drone environment in the laboratory has both the aim of proving the working setup, and also collecting the performance of the algorithm in a real-life scenario. This work has been accomplished in collaboration with Links Foundation which provided all the materials for the drones and the flight environment, which is a part of their Robotics Laboratory.

Contents

List of Figures

Introduction

Drones, also known as UAVs (Unmanned Aerial Vehicles), are unmanned aircraft that are controlled remotely or autonomously through computer algorithms. These devices have gained increasing popularity in recent years due to their versatile applications in a wide range of industries, from aerial photography, to agriculture, security and surveillance, and more. One important developments in drone technology is the formation of drone swarms. This involves the coordinated use of multiple drones working together to achieve specific objectives. Drones in a swarm can communicate with each other and cooperate to perform complex tasks, such as mapping large areas, material distribution, environmental monitoring, and search and rescue. Thanks to their ability to fly in inaccessible or dangerous locations for humans and their flexibility, drones and drone swarms are revolutionizing numerous industries, offering new opportunities and challenges in the field of technology and automation.

The main aim of this thesis work is to start from a set of assembled drones and obtain a working swarm of drones able to communicate between them and fly reliably in an indoor controlled environment, ready to have an application deployed and tested on them. Work has been done on the drones both on the hardware, for example the creation of physical connections between the various boards were needed, and on the software, establishing the communication between drones and also to a ground station. The main elements composing each drone are: a flight controller (FC), radio receiver, motor driver (ESC), companion computer, and a camera module. A more detailed description of the drones architecture and the process of troubleshooting all the faced problems is described in chapters 2 and 3.

A key component present in the flight environment is the Vicon Motion Capture system, that is used to give to each drone a position groundtruth so a precise flight is possible. Motion Capture is a technique that make use of multiple high resolution static cameras that are positioned in different places of the flight area, in such a way that markers can be placed in any position and they will be detected by multiple cameras. The markers are special infrared reflectors that can be easily be differentiated from the scene by the motion capture cameras. Each camera takes simultaneously a picture and all the frames are fused together to interpolate the position of all the markers. Each drone is equipped with a unique rigid constellation of markers, visible in Figure 1 that makes it recognisable to the system and also allows its orientation can be calculated. This system has very good

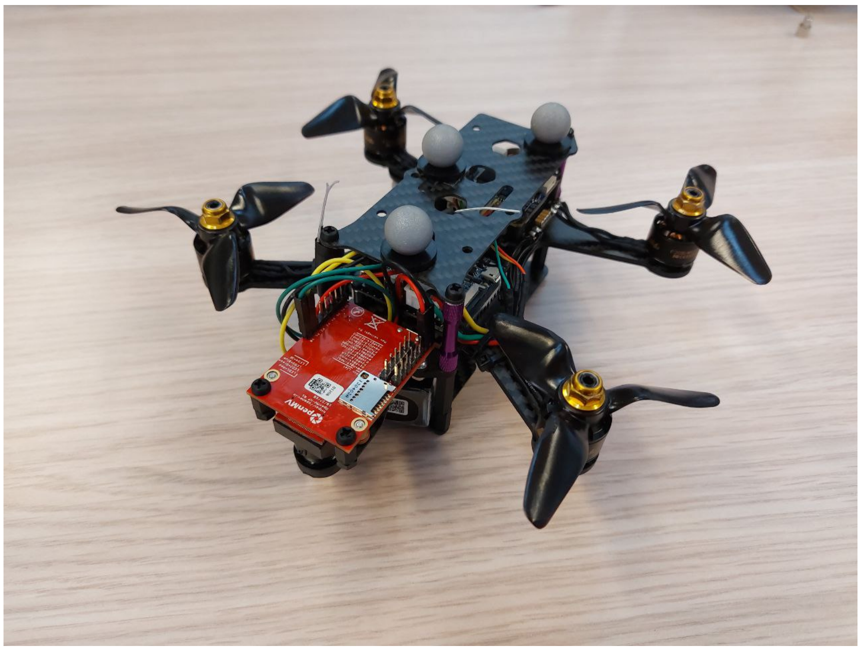


Figure 1: *Drone fully configured*

performance, allowing high precision on the position estimate, in the order of couple of millimeters and a high frame rate up to 300 Hz, allowing to obtain an extremely reliable position groundtruth for most applications. In our case the estimate produced by the motion capture system is used to feedback to the drone autopilot the current position measurements that is then fused with other measurements from onboard sensors in order to make the position estimate more precise and stable. This allows us to control the drone imposing simple setpoints like position, velocity or acceleration.

In chapter 4 of this thesis is presented the implementation of an algorithm for diffused target estimation and a flocking algorithm, used to command the drones to stay close together and track the target estimated position. The used target is an ArUco marker and its position is measured by each drone using the onboard camera. The implementation of this algorithm was developed and debugged in a simulation environment with *Gazebo*: the simulation was made to resemble as closely as possible the real conditions, emulating the same firmware and communication interfaces as the real drone, thus making the transition from simulation to the real implementation very easy and reliable.

The implementation of this application is aimed to further test the setup performed on the drones with a complex task. The developed algorithm for the Kalman filter and the flocking algorithm is heavily based on the work developed in [9], that was only tested in simulation using ROS 1 and MAVros, and has been implemented in the ROS 2 framework. The available sensor on the real drones for sensing the target is a down facing camera, and was simulated with Gazebo in order to correctly integrate this sensor with the used algorithm. In chapter 4 is presented the development of the algorithm, including the work relative to the computer vision necessary to locate the marker. Chapter 5 presents the simulation implementation and performances, and the deploy and testing on the real drones.

Chapter 1

State of the art

1.1 Multicopter Control

The UAV's used in this thesis work are categorized as multicopters, an aerial vehicle whose motion is controlled by the downward thrust produced by multiple propellers. A multicopter is composed of a main body, where the center of mass is and radially extended arms to hold the motors and propellers. The number of motors can vary depending on the platform, in this case each drone has 4 motors, hence it can be also called a quadcopter. Each drone has a reference mobile frame attached to it: the origin positioned in the center of mass and the axes direction follows a conventions called FRD. The convention FRD means X axis points forward, Y axis right and Z axis down. Another convention used in this thesis will be FLU, imposing X pointing forward, Y pointing left and Z pointing up.

The motors of a quadcopter are mounted in a coplanar (XY plane) and parallel way, and thus they can produce thrust only in the Z axis: this means that the drone cannot produce directly a force to move horizontally, in fact this type of platform can be viewed as underactuated. It is still possible to control the drone in all directions, by changing the direction of the thrust when the body tilts.

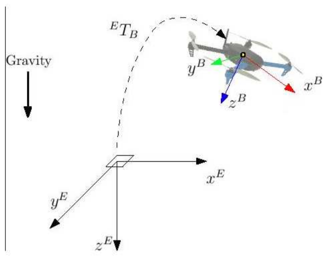


Figure 1.1: *FRD coordinate system applied to a drone.* [6]

A common way to control the horizontal position of these platforms is to control the attitude, meaning to control the three angles (roll, pitch, and yaw) that define the body orientation. If the thrust direction is only in the Z axis (the platform is leveled) then no resultant horizontal force is present. By tilting the body a resultant force is generated in the horizontal direction, that depends on the angle of inclination of the body, as shown in [1] and [20].

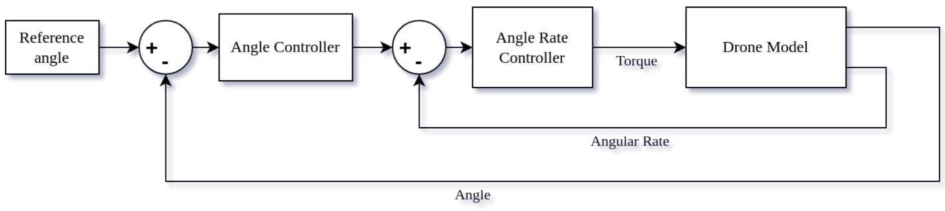


Figure 1.2: *One dimensional angle controller*

To control a multicopter the only variable that can be modified is the motor speed of each motor, that directly correlates to the produced thrust. The control algorithm is composed of a cascade of PI or PID controllers, in particular for pitch and roll axis that are responsible for the horizontal dynamics. Yaw control is much simpler and the angle error is directly used to compute a torque on the Z axis: this torque is generated leveraging the motors torque balance. In case of a quadrotor, the net torque from all the motors should be zero, and this is achieved having 2 clockwise spinning motors and 2 counterclockwise. Imposing the total thrust to remain constant, a difference in speed between the two pairs result in a net total torque different from zero.

To impose a horizontal acceleration the body tilt angles (roll and pitch) needs to be controlled to the desired value. There is a direct correlation between the tilt angle and the horizontal acceleration. The angle control is achieved using an angle rate controller that generates the reference torques to be applied to the drone body to achieve the desired angle. Torques on X and Y axes can be easily generated changing the thrust of each motor individually, and thus the motor speed, taking into consideration the physical parameters like arm length and motor characteristics. As seen in the diagram in Figure 1.2 the error terms for the controllers are calculated using the measured angles and angular rates. In the real drone the measurements are retrieved by the onboard Ineartial Measurement Unit (IMU) able to provide angular and linear accelerations measurements.

To ignore the non linear transients of the controllers each cascaded controller needs to have an update frequency much higher than the previous one. For example, in the implementation of the used autopilot stack the frequency for the angular rate controller is 1 kHz and for the the angle controller is 250 Hz.

To control other quantities relative to the multicopter like linear velocity and position the common solution is to use again a cascade of two PID or PI controllers as shown in Figure 1.3. To compute their relative error, an estimation of these quantities is needed since sensors usually can't provide a direct measurement. Estimations are done with sensor fusion techniques such as Kalman filters, which are covered in Section 1.2.

The most important part of a multicopter are the motors, that are what allows it to move and change its position. In case of the quadcopter they consist of 4 brushless DC motors, each directly connected to a propeller. Thrust is proportional to the square of the rotational speed $F_Z \propto n^2$, so a force can be imposed by simply controlling the rotational speed. Each motor is controlled with an Electronic Speed Controller (ESC) that takes

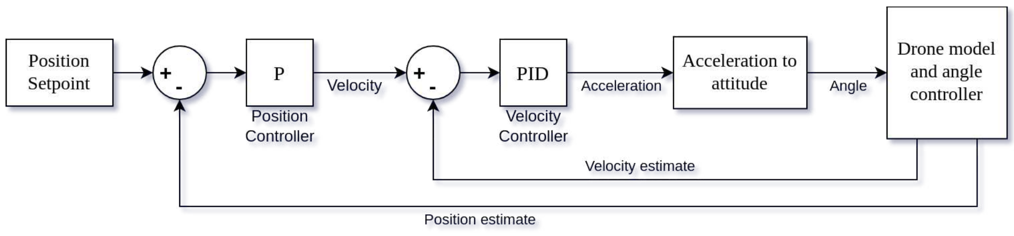


Figure 1.3: *PX4-Autopilot position controller. The angle reference is fed to the block "drone and position controller", represented in Figure 1.2*

care of generating the correct voltages for the motors in order to achieve the desired speed. In particular the ESC is only a driver that imposes the speeds requested by the Flight Controller to the motors.

A brushless DC motor is a three phase synchronous motor, meaning it needs three voltages to be generated with the correct timing and shape in order to work. A speed controller for a 3 phase brushless DC motor takes as input the speed setpoint and a DC power input. Then a switching cell stage is needed to impose different voltages on the motor windings in order to create a rotating magnetic field [7]. The common technique adopted for these type of motors is trapezoidal control, also known as six-step. A feedback on the rotor position is needed to keep the rotating magnetic field synchronized with the rotor magnetic field, in our case this is performed in a sensorless way, analyzing only the back EMF on the motor windings.

Sensored approaches for estimating the rotor position are also used, for example using Hall effect sensors or resolvers. They are common on bigger motors, and also the speed controller needs to support the rotor position sensor as a physical input, which is not the case with the used ESC. The used ESC already implements all of the techniques needed to control the motors and no setup is needed.

1.2 Kalman filter

Kalman filter in an algorithm that is able to produce an estimate of an unknown variable by taking as input multiple sensor measurements and the model of the system and fusing all the information together. If well designed, the filter estimate is usually more precise than directly estimating the quantities from the raw measurements of the single sensors.

This method uses the system dynamical model and its inputs to predict the system states, and also includes all the measurements from actual sensors, weighted proportionally to their uncertainty. This approach results in being very resilient to external disturbances and noisy measurements, extracting the most amount of information from the real sensor measurements when correctly designed and tuned. Each measurement needs a covariance value associated to it, needed by the filter in order to correctly trust the information provided from the new data. By increasing or decreasing the covariance associated to each source of data, the behaviour of the estimate can be changed to rely more or less on a certain measurements.

Instances of Kalman filters are used in the drone autopilot software stack in order to estimate the drone states: the role of the filter in fusing the external position estimate to obtain a precise and stable flight is analyzed in Section 3.3.4.

The standard formulation of the Kalman filter uses the system dynamical model with state update equation and output equation:

$$x_{n+1} = Ax_n + Bu_n + \omega_n \tag{1.1}$$

$$y_n = Hx_n + v_n \tag{1.2}$$

where ω_n and v_n are additive stochastic white noises. It is composed of two main steps: the prediction and measurement update.
Prediction step:

$$\hat{x}_{n+1,n} = A\hat{x}_n + Bu_n \tag{1.3}$$

$$P_{n+1,n} = FP_{n,n}F^T + Q \tag{1.4}$$

Measurement update step:

$$\hat{x}_{n,n} = \hat{x}_{n,n-1} + K_n(z_n - H\hat{x}_{n,n-1}) \tag{1.5}$$

$$K_n = P_{n,n-1}H^T(HP_{n,n-1}H^T + R_n)^{-1} \tag{1.6}$$

$$P_{n,n} = (I - K_nH)P_{n,n-1}(I - K_nH)^T + K_nR_nK_n^T \tag{1.7}$$

Where K is the Kalman gain, z is the measurement vector, H is the observation matrix, Q is the process noise covariance, P is the estimated covariance.

The two steps are executed at each discrete time instant: the quantity $\hat{x}_{n+1,n}$ represent the prediction of the state at the next instant without any information about the $n+1$ measurements since they are not yet available. The quantity $\hat{x}_{n,n}$ is the updated state estimate calculated from the previous state prediction and including the measurement of the current time instant. The quantity $(z_n - H\hat{x}_{n,n-1})$ is called innovation and it has a very important meaning: it contains the new information about the system state, and in particular when the Kalman filter is the optimal estimator it extract the maximum possible amount of information from the measurements.

There are multiple formulations of the Kalman filter, and one of them is called information form. The information form Kalman filter is an equivalent formulation of the standard filter, as dimonstrated in [2] and [13], where P^{-1} is propagated instead of P. The information matrix is defined as $I_n = P_{n,n}^{-1}$ and represent the certainty of the state estimate: large I value means high confidence in the estimate. The advantages of this formulation can be seen in terms of computational effort in case of distributed estimations with high number of sensors. The standard form needs the inversion of the R matrix whose dimension is equal to the number of sensors instead, in the information form, the largest matrix to invert has size $n \times n$ which is the number of the system's states. Also the initialization of $\hat{x}_0$ and I_0 is easier, since the information matrix can be set to zero or very close to zero to numerically allow inversion. In fact this means we have no information about the state estimate when starting the filter.

1.3 Flocking

Flocking is a behaviour of a large number of interacting agents with some common objectives. This concept can easily be applied to a swarm of drones, with the aim of group flight and coordination between close by agents. Multiple classifications can be done depending on the characteristics of the interaction between the agents, described by [14], for example the type of information sensed by each agent and centralization of the control. If agents can sense their position in a global coordinate system, they don't need to sense the relative position respect to other agents. If agents can sense their position in a global coordinate system, they don't need to sense the relative position with respect to other agents. This situation is rather complex, requiring specific sensors and elaborate infrastructure. A more common situation would have the agents knowing the relative distance from the close by neighbours.

More precise rules to define a flocking behaviour were proposed by Reynolds as described in [15]. Those can be considered as a starting point for developing a flocking algorithm. They can be summarized in 3 rules, known also as *cohesion*, *separation* and *alignment*:

- Flock Centering: attempt to stay close to nearby flockmates.
- Obstacle Avoidance: avoid collisions with nearby flockmates.
- Velocity Matching: attempt to match velocity with nearby flockmates.

A control algorithm can be developed with multiple terms where each one has a different aim, for example one term for each Reynolds rule. Each agent has dynamic $\ddot{q}_i = u_i$, meaning the control input fed to the agent can be interpreted as a force applied to it. In a simple algorithm proposed by Olfati-Saber [16] the control input is composed of three terms:

$$u_i = f_i^g + f_i^d + f_i^\gamma \tag{1.8}$$

In particular the gradient term $f^g = \nabla_{q_i} V(q)$, f^d is a damping term relative to velocity matching of the swarm and f^γ is a navigational feedback term relative to the group objective [8]. If the navigational term is absent there is no group objective situation and the algorithm is known as *protocol of flocking*. For example the navigational term can be given by

$$u_i^\gamma = -c_1(q_i - q_r) - c_2(p_i - p_r) \tag{1.9}$$

where we define a γ-agent that represent a target to track and has state (q_r, p_r), this is a secondary objective of an agent. This algorithm embodies all 3 Reynolds rules, but leads to a correct flocking behaviour only in a limited set of initial conditions, and fragmentation is an issue that can arise. Fragmentation happens when the agents form small and detached flocks instead of a single big one and more complex strategies have to be adopted to avoid this problem .

Chapter 2

Drone Hardware Setup

Each drone presents multiple components that needs a software and hardware configuration. The software configuration consists in choosing the firmware and software versions for all the components that are presented later in this thesis work, in order to ensure compatibility between them. A detailed description of the software setup is presented in chapter 3.The hardware configuration is covered in this chapter. This part of the project consisted into working on the physical connections between the boards and other details related to the mounting of the components on the frame. Each drone presents the same boards and they are all set up in the same way. This allowed to just copy the steps performed on the first drone onto the others since they are all equal. The main components present on each drone are:

- *Flight controller (FC)*: the main board responsible for the flight of the drone. Multiple sensors are present onto it and the control algorithm related to the flight is run by this board's microcontroller.

- *Electronic speed controller (ESC)*: the driver for the motors, taking as input the velocity setpoint for each motor from the FC. It delivers power directly from the battery to the motors.

- *Companion computer*: a general purpose single board computer connected to the FC. In this application is setup to run ROS 2 and to communicate via WiFi to the FC.

- *Onboard camera*: A camera module with an integrated microcontroller that can communicate with the companion computer to stream the captured images.

The interconnection of the components present on each drone is shown in Figure 2.1. In this chapter a detailed analysis of each component is presented, explaining their role and the process needed to make the setup, including all the troubleshooting of the faced problems.

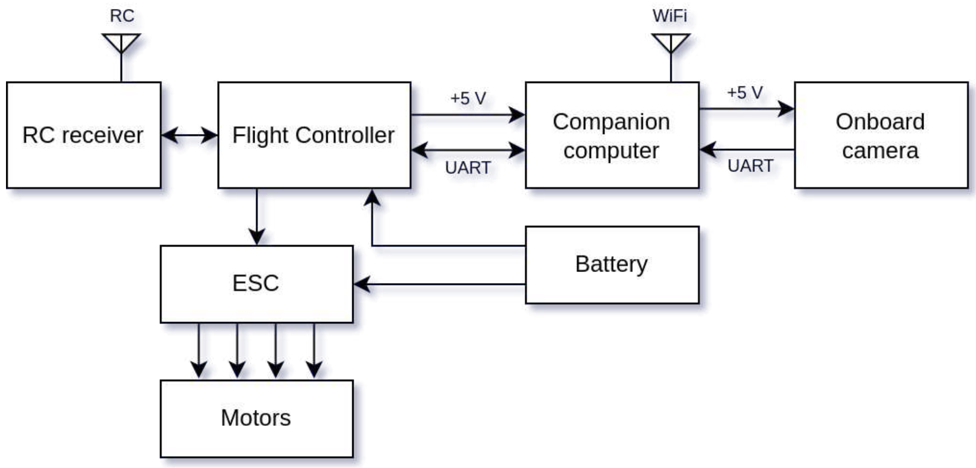


Figure 2.1: *Layout of the boards and components present on each drone.*

2.1 Flight Controller

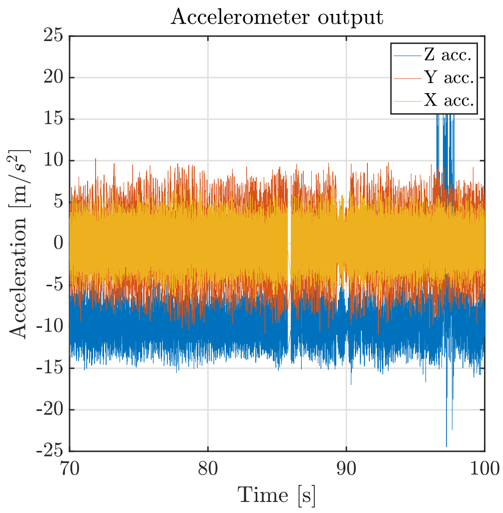
The Flight Controller (FC) is the main board that allows the flight. Its main task is to control the motors speed in order to fly. It is composed by a microcontroller and multiple sensors directly integrated onto the board, like a barometer, accelerometer and gyroscope. In particular the board model is an *Omnibus F4SD* that is suited for small drones and for manual control, offering very limited sensors capability. This model of FC is in fact targeted for racing drones. The setup managed to work around its hardware limitations and make it usable also in an autonomous application. The role of the FC is to impose the motors speed in order to achieve a stable flight, by controlling the body pitch, roll and yaw angles to the correct setpoints. This is done by reading the onboard sensors and running control algorithms implemented in the firmware (in this case PX4-Autopilot). The control algorithms needs a feedback on the controlled quantities and this is achieved using the sensors implemented onto the FC. The quality of the measurements directly translates to the stability of the flight, and an extreme case of bad sensor readings can result also in loss of control of the vehicle. The main disturbances that can affect the FC sensors during flight are electromagnetic noise and mechanical vibrations. An effort was made to reduce these effects on the FC, in order to improve the flight quality and reliability: high mechanical vibrations are one of the causes of wrong estimations of the Kalman filter used by the autopilot. Electromagnetic interference is generated by the motor control and the high currents flowing into them. Motors are in fact controlled using a voltage switching technique and this creates a lot of electromagnetic noises due to the high current flowing into the motors windings. Since these quadrotors are quite small, these effects can be significant, so for example the choice of a robust protocol to communicate with the electronic speed controller (ESC) was aimed to avoid possible electrical interference.

The FC directly interfaces with the ESC to control the motor speeds. The ESC takes as input a command signal for each motor and performs the correct motor control algorithm to adjust the motors speed. The most commonly used communication protocol is a PWM signal for each motor where the duty cycle is proportional to the requested motor speed. Another commonly used protocol is DShot, that has been used in this application instead of PWM. This choice was made because the DShot protocol is supported by the ESC and has higher performance in terms of resolution and robustness. DShot is a digital protocol, so it's much more resilient to electrical noise and is also immune to oscillator drift. The PWM protocol relies on time measurement to determine the duty cycle, so if the flight controller and the ESC oscillators frequencies are not very close wrong readings can occur. DShot is a digital protocol so the requested value is encoded in binary and it also has an higher resolution than normal PWM.

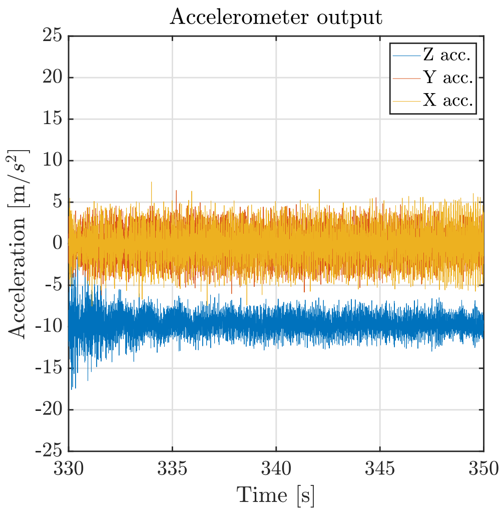
2.1.1 Mechanical vibrations

Mechanical vibrations are a very important aspect to consider in such a small platform: propellers speeds are very high, so any small imbalance is highly amplified and can affect the flight performance a lot, until complete lost of control. The main component affected by mechanical vibration is the Inertial Measurement Unit (IMU) sensor. It's role is to measure both accelerations and rotational orientation utilizing MEMS (Micro Electro-Mechanical-Systems) technology. A mass-spring system is built into the silicon chip and its movements are measured (for example capacitance changes are directly correlated to the displacement of the mass), so any excessive vibration is directly affecting the quality of the measurement. The IMU is the most important sensor used to control the drone in a stable position: it's the main source of feedback for estimating the vehicle attitude, needed by the angle controller to keep the drone stable. The common way to reduce vibrations transmitted to the IMU, and thus improve the measurement quality, is to use a compliant joint to secure the flight controller to the vehicle frame. In particular, a rubber mount is used. Also great care is needed in making sure that all the parts on the drone are well secured and nothing is moving because that can also be an important source of vibrations and may be very hard to identify.

Issues regarding vibrations were encountered: during flight high frequency vibrations were present and the IMU measurement was affected as showed in Figure 2.2(a). In addition of a quite high noise baseline, some spikes are also visible in particular in Z axis probably due to unsecured components which sometimes also started to vibrate. This problem was important to be resolved, since the noise spikes affected greatly the position and velocity estimates, making the drone drift in a wrong direction uncontrollably. Multiple solution were tested in order to reduce the effect of the noisy IMU measurements. The first approach was to perform a tuning of the filter that estimates the measured quantities, making it "trust" less these measurements, but it didn't have any effect. There are also the possibility to set software filters on the raw sensor data, for example low pass type that remove the high frequency dynamics. Unfortunately these solution didn't had any effect: performing a frequency analysis on the logged data showed that the noise component was constant in the entire frequency range. This implied that the problem was not a particular resonance frequency that could be removed. Then the focus was put on the mounting of the FC to the frame and, in an effort to isolate better the sensors from the vibrations, the rubber mounting studs that secured the flight controller to the frame were



(a) *High noise*



(b) *Low noise*

Figure 2.2: *Comparison between IMU measurements, Figure (a) shows the high noise before the improvement, Figure (b) shows the output of the IMU after additional damping mounts were used*

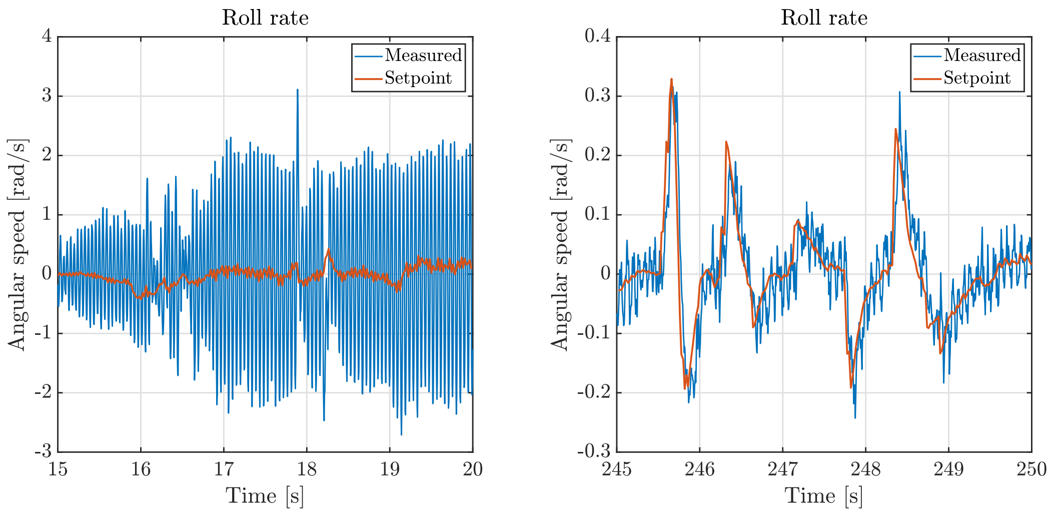
doubled. This solution improved significantly the IMU performance as shown in Figure 2.2(b). Consequently the position and velocity estimations were a lot more stable, which was the goal of this troubleshooting since the autonomous control of the drone relies on these estimates.

Further attempts to improve the mechanical vibrations may be improving the balancing of the propellers, but better performance may be achieved on a bigger platform. With the used drones, the frame is quite small (around 20 cm) and is difficult to well decouple the motor vibrations from the other component since they are very close. The aim of this troubleshooting however is to improve the quality of the estimations and there is no need to work further on this aspect, since the resulting position and velocity estimates improved a lot and the Kalman filter is now able to give a good result when fusing also the Vicon measurements. The reliability and quality of the position estimation of the drones is discussed more in depth in later chapters, in particular in section 3.4 when discussing the integration of the Vicon system.

2.1.2 PID tuning

The control of the drone is done with a cascade of multiple controllers. The main algorithm used are PID type, this require a tuning phase to choose the correct proportional, derivative and integral weights in order to achieve a stable flight. The process of choosing a correct set of weights for each controller is trial and error, requiring a test after each change of the parameters to evaluate the performance changes. Furthermore, in case of cascaded controllers the tuning process starts from the fastest controller and then moving onto the slower ones. In this case the tuning should start from the angle rate controller and end with the position controller. Each parameter has a different effect of the behaviour of the control. The proportional gain is related to the responsiveness, and if increased too much generates oscillations and vibrations. The derivative term dampens the oscillations, but when increased it reacts more to the noise. Finally the integral term reduces the steady state error, but again if increased too much oscillations are generated and instability can occur. The wrong choice of these parameters can cause flight instability and wrong behaviours (for example vibrations). The tuning phase is a particularly important aspect if high performance and fast response time are required. Since these characteristics are crucial for racing drones but not for the aim of this thesis work, that tuning aim was to only achieve a stable flight. The PX4-Autopilot software after being deployed for the first time on a new flight controller does not need a full PID tuning: all the controllers have already the gains set to values that works in the majority of the cases.

An issue was encountered in the angle rate controller (shown in Figure 1.2) that produced very strong vibrations in the pitch and roll axes, imposing very high changes in motor speeds as control input. To achieve a correct behaviour without high oscillations and overshoots, represented in Figure 2.3b, the proportional gain of the rate controller was decreased, from 1 to 0.3: this gain was far too high, producing big overshoots of the controlled quantity.



(a) *Before PID tuning* (b) *After PID tuning*

Figure 2.3: *Roll rate control performance, Figure (a) shows the high vibrations induced by the wrong PID tuning. Figure (b) shows the behaviour after the tuning.*

Another consequence of this wrong behaviour was that the controller, in order to change quickly the angular rates, it imposed high and fast changing motor speeds to the motors. From Figure 2.4 is possible to see that the motor control signal goes from 0%

to as high as 90% multiple times per second, requiring a big torque to be produced by the motors. This torque is directly proportional to the current flowing into the windings. The resulting high currents flowing into the motors generated very high temperatures into the windings even after just a couple of seconds of flight. Maintaining the flight for a longer period had the potential to ruin the motor or the ESC if the maximum operating temperatures were exceeded. All of the drones during their setup required this step of tuning of the angle rate controller since it presented always the same issue. It was not an issue related to a defect of a single drone but the base tuning offered by PX4 is not fully suited for these particular quadcopters. After performing the PID tuning the flight was very stable, without vibrations and the motor temperatures stopped being a problem anymore.

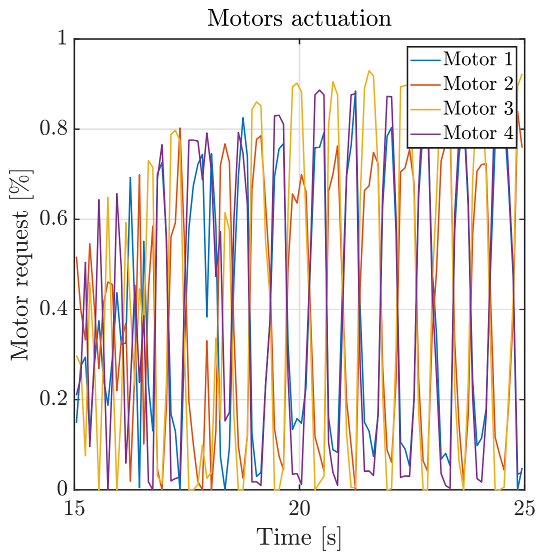


Figure 2.4: *Motors actuation command during oscillations created by the angle rate controller improper tuning.*

2.1.3 RC receiver

The main intended way to control the drone is using the remote radio controller (RC), that communicates to the FC using a dedicated radio receiver connected directly to the FC. In this case its setup supports only data sent towards the FC: information about the remote radio controller buttons and analog sticks positions are sent to the drone's radio receiver. The buttons on the radio controller are directly mapped to functionality like arming, changing flight mode and, of course, imposing the target throttle, roll, pitch and yaw angles. The one way stream of data allows to use a particular connection called SBUS that uses only one data pin and has a reserved breakout on the FC. More complex configurations can be implemented for example sending telemetry data directly to the radio controller to read useful data directly onto it and perform simple automation tasks like return to home or failsafe tasks. This possible configuration requires another dedicated UART port that is not available on the *Omnibus F4SD* since the one used to connect to the companion computer is usually reserved for GPS (not used in this configuration) and no other free ports are present.

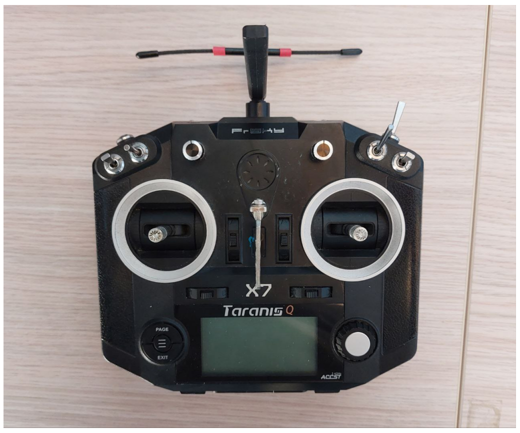


Figure 2.5: *The remote radio controller*

The frequency used for the radio controller is 2.4GHz and there are multiple protocols of transmission usually developed directly by each manufacture. The radio transmitter and receiver must use the same transmission protocol and frequency to correctly communicate. Note that the used frequency is the same as 2.4GHz WiFi and no direct compatibility between the two protocols is present. The transmission power is very high, in order to achieve long range communication between the radio controller and the FC. The close distance between the onboard radio receiver and the companion computer WiFi antenna, suggest a possible electromagnetic interference between the two systems. This topic is analyzed more in depth in Chapter 2.2.

After the complete setup of the FC the drone is fully capable of flying controlled manually with the radio controller.

2.2 Companion computer

Each drone is equipped with an onboard general purpose computer, called companion computer, that connects with the flight controller and other sensors. Its role is to perform computations and it communicates directly with the FC. Since it's mounted inside the drone, it is particularly suited for latency critical computations that benefit from the direct communication instead of relying on a wireless communication to a ground station. In this case the main role of the companion computer is to enable WiFi communication, not available directly from the FC. The WiFi communication is used to exchange information between the drones and also receive data from the Vicon system. It also has the capability to execute any program in order to be independent from a ground station, in particular ROS 2 Humble was deployed onto it so there is the possibility to execute any ROS 2 package, benefiting in terms of latency due to the direct communication. The important hardware limitations of this board have to be taken into account if computationally intensive tasks need to be performed.

The board model is a *NanoPi Neo Air*, a single board computer equipped with 512 MB of RAM and a quad-core 32bit ARM processor. Other than WiFi connectivity also multiple serial ports are present, that are necessary to establish communication to other boards. The setup of this board starts by flashing on an SD card the operating system image: the OS images are directly provided by the board manufacture, so they can be deployed directly using and SD card without any additional work. Then using a TTL-232R serial to USB adapter, the UART0 debug serial port is connected to a computer and a console of the system can be obtained. Once logged in the operating system the WiFi connection can be configured and from this point the network protocol *SSH* was always used to connect to the board and interact with it.

2.2.1 WiFi connectivity

Establishing the WiFi connection was not trivial at first: even following precisely the manual [4], the board did not detect any access point. It only managed to connect if the board and the WiFi router were very close, but the received signal strength both from the NanoPi and the router were very weak. For example, from a distance of just a couple of meters the received signal power measured from the router was -90 dBm. That is very low and is only expected if distance to the router is much bigger. The board presents a connector for an external antenna, that is mandatory for WiFi connectivity, but is never mentioned explicitly into the manual and it was assumed that the antenna was integrated inside the WiFi chip, like in similar boards (e.g. Raspberry Pi). After connecting a suited antenna the connection to an access point was then a straightforward process.

WiFi connectivity still presented some issues regarding the stability of the connection, sometimes stopping the transmission or losing packets, without a clear cause. This problem affected a lot the flight performance since the position estimation relies on the Vicon measurements that are sent to the drone via WiFi with a precise frequency. The loss of reception of these data results in a significant drift in position estimation caused by the unreliable inertial measurements. This resulted in the drone loss of control when performing an autonomous task that relies on a correct position estimate. A more detailed description of problems regarding states estimation is discussed in section 3.3.4.

The root cause of this problem is the electromagnetic interference between the 2.4GHz WiFi band and the 2.4GHz radio link with the radio hand controller. The radio controller is a *FrSKY Taranis Q X7* using the 2.4GHz band transmit to the drone receiver. This radio link does not comply with the WiFi standard and, in addition with the high transmission power, caused the random disruption of the communication. The solution to this problem is to change the radio receiver with one that uses the 900 MHz band instead if the presence of a radio controller is required. The radio controller in this case is only used as a backup to regain control of the drone in case of unexpected behaviour, or to perform a flight termination (immediate shutdown of the drone). The other possible solution is to not use the radio controller and also remove the radio receiver connected to the flight controller. In this case the only way to communicate with the drone is via ROS 2. Since the flight termination action can also be performed via ROS 2, no features are missing if the radio controller is not present. After identifying this issue the radio receiver was removed and all the issues regarding loss of communication and unreliable packet reception were resolved. Resolving this issue, and thus greatly improving the reliability of the WiFi connection, was a crucial step in obtaining a correct setup suitable for offboard control since without this improvement the behaviour was too unstable and, using the drones in offboard configuration was not feasible.

In order to identify this problem the main indicator was the loss of ROS 2 messages that were expected with a regular frequency, both towards and from the companion computer. A fundamental tool was the deploy of ROS 2 on the companion computer, which was not trivial given the 32-bit architecture. Its implementation is described more in depth in Section 3.2. In order to characterize the frequency of a ROS 2 topic, a builtin introspection tool for topics was leveraged [19]: this tool shows in real time the average frequency, and the maximum and minimum time elapsed between messages of a certain topic. If messages were lost, a higher than expected elapsed time between two messages was shown by the tool. The messages generated from the ground station (e.g. the Vicon measurements) when measured directly from it showed good frequency metrics, but when measured with the same tool executed on the companion computer they showed big elapsed time between messages, indicating loss of communication. The same situation happened for the messages generated by the companion computer (e.g. vehicle real-time odometry), showing good metrics when measured directly from the companion computer but losses were encountered when measuring on the ground station. These tests allowed to identify the wireless communication as the main source of the issue, focusing then the efforts only on this part allowed the discovery the interference between RC and WiFi.

2.2.2 Serial port configuration

The companion computer is connected to the FC and to the onboard camera with a UART serial port for each one. UART (Universal Asynchronous Receiver-Transmitter) is a serial asynchronous protocol using 3 wires: one to receive (RX) one to transmit (TX) and a reference ground (GND). An additional 4-th wire was used in all the connections to connect also the 5V supply, so with 4 wires communication and power are established between the flight controller, companion computer and camera module. The described connections were performed creating custom adapters with wires and connectors, and studying the FC and NanoPi datasheets in order to locate the correct connection points to use.

Since the transmission is asynchronous, both terminals have to agree on the same transmission speed. Choosing a suitable baud rate for the UART connections was not trivial: a fast enough transmission speed was needed since the publication rate of data from the FC is very high, but not any value is available. A valid baud rate should be derived from the UART controller clock frequency divided by an integer number: this is very important because the transmission is asynchronous so both boards must know the exact bit time. The encountered issue was that a lot of values that should have been supported by NanoPi did not allow a proper transmission because the NanoPi clock frequency could not be divided to obtain accurately the requested baud rate. A lot of values

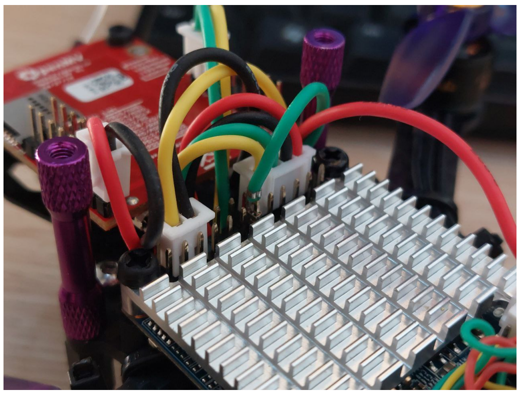


Figure 2.6: *Custom connectors created to connect the flight controller and the camera module to the companion computer. The red and black wires are for +5V and GND. The yellow and green wires are for UART TX and RX.*

were tested and a working one was found at 921600 bit/s, much higher than the minimum value specified by the PX4 manual, but able to establish the communication correctly.

The 5V power is distributed by the flight controller: the FC is the only board directly connected to the battery, transforming with a voltage regulator the 11.1 V of the LiPo battery to 5V. The power is then distributed to other boards by connecting directly to 5V pins of the FC. This solution is not adequate if the boards needs a lot of power since the FC is not designed to directly supply high currents to other components. In this case the companion computer is not very powerful and the power provided by the FC was enough. If this wasn't the case, then a power distribution board is needed to correctly supply each component: this kind of board connects directly to the battery and then can supply all the boards.

2.3 OpenMV Onboard Camera

In the front of the drone, a down facing camera is present useful to many applications, for example, the identification of ArUco Markers. The identification and the pose estimate of the ArUco marker is a fundamental part of the estimation task as better described in chapter 4. The camera module is a *OpenMV Cam H7*, it features a camera sensor integrated with an STM32H7 microcontroller. The microcontroller allows to directly execute computer vision algorithm and to control the IO pins, without the need to stream the image to the companion computer. This platform uses as programming framework a custom version of micro-Python [17]: it is an implementation of Python that is able to run on microcontrollers and control with a high level API (respect to the usual C/C++ programming) all the IO and camera functionalities. This implementation is useful for fast prototyping and implementation of machine learning algorithms, where Python is a very diffused language.

The camera is connected to the companion computer with a UART connection, that also provides the 5V power. A computer vision algorithm is run on the camera module microcontroller and only the result of the computation are transmitted over the serial link. In case of the identification of an ArUco marker, the microcontroller reads the image from the sensor and locate the marker in pixel coordinates, then only those resulting coordinates are sent over serial communication. This solution is simple and offloads some computational resources to the camera microcontroller. It relies, however, on a quite limited device with very constrained memory. The limited memory offered by the microcontroller mandates an important down-scaling of the image, that affects the effective field of view of the camera. The original resolution is 640x480 pixels and, after the down-scaling needed to make the computer vision algorithm work, the resolution is set to 160x120 pixels. The low resolution is also affecting the capability to correctly recognise the checkerboard pattern, that is used for the camera calibration process. More details on the camera calibration are presented in section 4.2.2. In order to avoid to reduce so much the resolution, it's possible to serialize the image at full resolution (640x480) and send it over the UART port, to then make the companion computer perform the computations. The companion computer offers a much greater computational power and can easily perform the computer vision algorithm. No libraries were available to correctly handle the serial transmission of the images from the camera module, so the first solution was used for simplicity. The low resolution was a problem only when increasing too much the distance from the marker to the sensor, otherwise satisfying performance were obtained.

Chapter 3

Firmware and Software

After the setup described in the previous chapter, all the boards present on the drone are provided with power and are also connected to each other, in order to exchange information and perform the intended task. Each board needs a software setup: the main steps are to correctly choose the firmware for the FC, and setup the companion computer to allow the integration of PX4 with ROS 2. The fundamental choice that guided most of this part of setup is in fact the use of PX4-Autopilot on the FC: the flight controller is the most important board allowing the drone to fly, so all the other software is chosen ensuring compatibility with PX4. After PX4-Autopilot was established as the software flight stack to use, then the communication framework was selected: ROS 2 [10] can be directly integrated with the autopilot firmware and there is currently an active development on this direction. This communication framework does not use the MAVLink protocol but it directly interfaces PX4 with the DDS protocol. Finally after setting the ROS 2 version to Humble the companion computer operating system was set to Ubuntu 20.04 as it offers the most compatibility with this version of ROS 2 and most importantly is directly available from the manufacturer to deploy on the NanoPi.

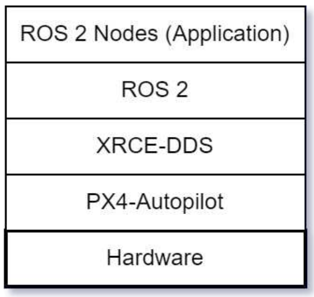


Figure 3.1: *Software stack used for each drone.*

In this chapter the software setup performed for each major component will be analyzed, describing also all the troubleshooting needed to establish a reliable ROS 2 communication and to allow execution of nodes on the companion computer.

3.1 Ground station software

A ground station is a general purpose computer that is wirelessly connected to the aerial vehicles ans it's able to communicate with them. The main tasks of a ground station are monitoring the vehicle telemetry and sending commands in order to perform certain actions like landing and hovering. Another very important task is the configuration of the vehicle both on the ground and also during flight. In particular, with PX4, this represents the main way of doing the setup and various calibrations. The software used in this project that offers all the previously mentioned functionalities is QGroundControl. It runs on a laptop connected to the same network as the drones and it's able to interface directly with PX4 with the MAVlink protocol. In this project it was used as the main tool to configure the autopilot parameters. It is very useful to monitor the drone while in the air, but a wireless connection transmitting the MAVlink messages is needed. Since the FC does not have directly the wireless capability, another serial port was needed in order to route the MAVlink messages via the companion computer (then able to then retransmit the messages via WiFi). On the used FC isn't equipped with another available UART port (the only one was used to enable ROS 2 communication) so QGroundControl was not used during flights. Connecting, however, the drone to the computer via USB, enabled the communication from PX4 to QGroundControl that was used for the parameters setup: it allowed to perform the majority of actions for configuring the autopilot, from flashing a new PX4 firmware version to calibrating the sensors of the flight controller.

Another task that can be performed by the ground station is the execution of the algorithms instead of executing them on the companion computer. Using ROS 2, it allows to execute the nodes on any platform, provided that is connected to the same local network, since the topics and messages are visible to every computer. This is very useful during development especially if the companion computer intended to run the nodes is not very fast at compiling the source code. In this project the ROS 2 nodes responsible for interacting with the Vicon server to retrieve the position measurements are executed on the ground station.This is done because the Vicon SDK library only supports 64-bit architectures so it can't be deployed onto the 32-bit companion computer installed on the drones. This configuration introduces a small amount of latency in the samples provided by the Vicon, since they have to be transmitted through the ground station instead of directly to the companion computer.

Finally the ground station is used to log into a file all the messages sent over the ROS 2 framework using *rosbag* [18], so it will be possible to analyze all the data in a later moment. The recorded information includes all the messages sent over the ROS 2 framework, for example the telemetry data of all the drones, and also all the commands sent to and from each node.

3.2 ROS 2 configuration

ROS 2 is a powerful tool aimed at the development of robotic applications. This version is the second generation of the robotic operating system, offering an important redesign respect to ROS 1, focusing on improving aspects related to network transport, architecture, platform support and others. It offers many features and the most important is inter-process communication with a publisher subscriber paradigm, or with client and server requests. Each process is represented by a node and, in order to communicate with other nodes, it has the ability to publish messages to specific topics or to read messages sent from other nodes. The main difference from the first generation is the lack of a central server (formerly known as *roscore*): now each node is independent and perform a peer to peer discovery. Additionally the peer-to-peer discovery implemented in ROS 2 is an advantage in the case of a swarm of drones: each drone runs an independent instance of the ROS 2 nodes so, in case of a crash of some nodes, the communication between others is not affected. This unfortunate scenario would happen in case of the crash of the *roscore* server in ROS 1.

In this project ROS 2 had to be integrated with the autopilot stack: topics related to odometry, vehicle status, etc. are published from PX4. Topics related to external commands to be executed by the autopilot are published from other nodes, for example arming, takeoff or setting a position or velocity setpoint. The bridge from PX4-Autopilot software to the DDS world is done with *eProsima Micro-XRCE-DDS Agent*, a middleware used to convert a message from DDS to XRCE serialized messages (and vice versa), to be streamed to the FC via the UART serial connection. A more detailed description of the communication between FC and companion computer is explained in Section 3.3 as the XRCE software is deeply integrated in PX4. The XRCE agent is executed by the companion computer and thanks to the DDS protocol the messages are already correctly broadcasted to the entire local network. At this point a ROS 2 node can be executed on any computer located in the LAN (e.g. ground station) and it is already able to communicate correctly with the autopilot. A better solution is to deploy ROS 2 directly on the companion computer and run on it the nodes that control the respective drone, in order to decrease latency and prevent issues in case of loss of wireless communication, making each drone completely independent. The layout diagram of the pieces of software involved in the ROS 2 to PX4 communication is shown in figure 3.2.

3.2.1 Deploy on the companion computer

Multiple challenges were faced in order to deploy ROS 2 on the companion computer: the system architecture is *armhf* 32-bit, hence a direct installation with binaries is not supported by the developers. Other solutions were tested, namely: using a Docker container, native compilation and finally cross-compiling the source code on a workstation for the NanoPi architecture.

Docker was the first solution to be tested, since ROS 2 Docker images are available to be downloaded and executed. A Docker image contains all the code and dependencies to

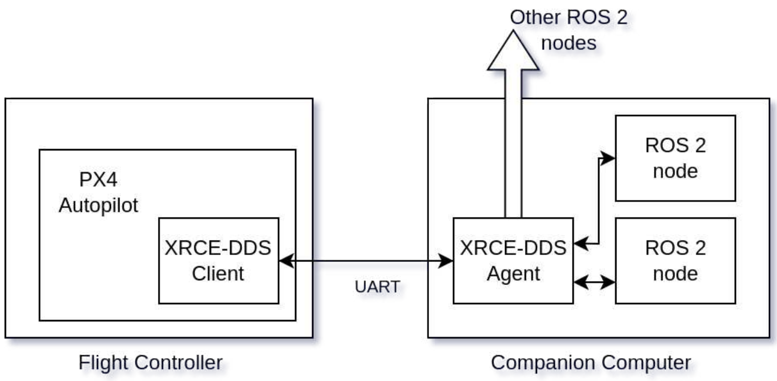


Figure 3.2: *XRCE Agent and Client that allows communication from PX4 to ROS 2 nodes.*

run the application, so there is no need to install additional software (other than Docker) and it abstracts the content of the image from the actual environment where it is executed. An important aspect to check is the system architecture, that has to be compatible with the one offered by the image: no images of ROS 2 Humble are available for ARM 32-bit architecture so this solution had to be abandoned.

Another way to install ROS 2 for non-directly supported platform and operating systems is to compile the source code locally and generating the binaries to execute. It's very important that the compilation is performed by the intended target machine (i.e. NanoPi board), since the code depends on many other packages installed in the operating system, but most importantly, the compiled binaries must match the target CPU architecture: a binary compiled with a 64-bit with x86 architecture machine can't be executed by a 32-bit machine with *armhf* architecture, since the CPU instructions used by the two of them are completely different and incompatible.

In order to compile the source code, first the repository containing all the source code of ROS 2 Humble needs to be downloaded onto the machine and all the packages required by ROS 2 need to be installed. Then the compilation can start using the dedicated build tool *colcon*. Due to the quite limited resources of the NanoPi, the procedure required hours to progress but it never managed to finish correctly. Analyzing the RAM usage during compilation, it showed the high demand of memory by the build tool and very little space was left for the operating system. From the entire 500 MB of RAM available, more the 400 MB were used to compile the source code, causing the operating system to crash or to abort the procedure when no free memory was left. A swap partition was added in order to increase available memory. This particular partition is placed in the mass storage and is used by the operating system in a similar way to the RAM, having of course a very low speed. This attempt also resulted in a failed compilation or a crash of the operating system due to lack of free memory, even when the number of parallel operations was reduced to the minimum in order to save as much RAM as possible.

3.2.2 Cross-compilation for *armhf* architecture

The last attempt to make ROS 2 available to use on the companion computer was to perform the compilation procedure on a machine with the capabilities to complete it successfully, and ensuring that the resulting binaries are fully compatible and able to be executed by the NanoPi. This technique is called cross-compilation and many details have to be considered in order to make a system create binaries to be executed on another platform. The main tool to allow this is the correct toolchain, in particular the C and C++ compilers and linkers. They are installed on the workstation directly using the packet manager since *armhf* is a common architecture and they are directly available without the need of further work. The main idea is to use *arm-linux-gnueabihf-gcc* instead of the already installed *gcc* compiler to create the binaries. This package can be executed on the 64-bit workstation but the output is a binary that can be executed by the target architecture. A further complication is that these steps needs to be executed by the building tool designed for ROS 2 that is *colcon*, so a setup of its parameters is necessary. This tool handles all the dependencies of the code and it's designed in particular to build ROS 2 and its packages. Another important aspect to consider is to correctly manage all the code external dependencies, that have to be installed on the target board and placed in the correct folders, otherwise the code will not work correctly when executed on the target board.

The environment used for the cross-compilation is a Docker container with Ubuntu 20.04, that matches the operating system present on the *NanoPi*: using the same version of the OS makes it easier to match the libraries versions from build environment to the target operating system (e.g. GLIBC version is very important it's strictly related to the Ubuntu version). Another reason to use a Docker container is to have a clean environment without any other installation or configuration, and also to avoid breaking the installation of other software present on the workstation. All the operations executed on the workstation are all performed inside the described container. Note that the container architecture is the same as the workstation, so it is no different from the perspective of the compilation tools from running outside of it.

The source code is then downloaded onto the container and the build tool needs to be configured to perform a correct cross-compilation. To configure the build tool, a file containing CMake instructions and variables to set is passed to it at the start of the procedure. The main aspects to consider in the CMake configuration file are the toolchain executables to use, the target root file system path and the strategy to find libraries. The two latter requirements are needed to correctly solve dependencies during compilation. The compiler needs to know the location and version of the libraries installed on the target board. This is accomplished by installing first all the dependencies on the target board and then by copying the root file system to the container on the workstation (just the folders */lib, /usr, /etc* are needed). The path of these folders is used by the build tool (variable *CMAKE_SYSROOT*) in order to search for the necessary library files. It's very important that the libraries are searched in the target root file system and not on the workstation because they may be found on both systems with the same names, but any 64-bit version of the libraries will make the compilation fail. A very important workaround used in situations where the tool manages to find only the workstation system's library, even if present in the target file system, is to create a symbolic link from the desired 32-bit library file, overwriting the found 64-bit library file. This way, if the library is found on the 64-bit machine is anyway redirected to the correct file inside the target root folder. Note that this method can break the installation of software, but since this process is performed into an isolated container, this issue can be ignored as the container can be discarded or reset without problems. Another way to fix errors due to libraries not found is to append to the environment variable *LD_LIBRARY_PATH* the folder path containing the missing library, in case the folder for some reason is not searched.

CMake variable names	**Value**
CMAKE_SYSROOT	/opt/rootfs
CMAKE_C_COMPILER	/bin/arm-linux-gnueabihf-gcc
CMAKE_CXX_COMPILER	/bin/arm-linux-gnueabihf-g++
CMAKE_FIND_ROOT_PATH_ MODE_PROGRAM	NEVER
CMAKE_FIND_ROOT_PATH_ MODE_LIBRARY	ONLY
CMAKE_FIND_ROOT_PATH_ MODE_PACKAGE	ONLY
CMAKE_FIND_ROOT_PATH_ MODE_INCLUDE	ONLY
PYTHON_SOABI	cpython-38-arm-linux-gnueabihf
PYTHON_EXECUTABLE	/opt/rootfs/usr/bin/python3.8
CMAKE_CROSSCOMPILING_ EMULATOR	/usr/bin/qemu-arm-static

Table 3.1: *Some of the most important CMake variables set to perform cross compilation and find libraries in the target file system.*

Another important aspect to consider during the cross-compilation of ROS 2 is the correct integration of Python in the resulting binaries. Python is an important part of ROS 2 and is directly integrated with the C++ source code. During the compilation, the Python installation present on the target file system needs to be used and it also needs to be emulated because its binaries are 32-bit. This step is mandatory since using the Python installation present on the workstation will result in a failed procedure. The correct Python executable to use is set again in the CMake file configured for the cross-compilation.

Multiple trials were needed to correctly set all the variables that allow to perform a successful cross-compilation, since no guide was found listing a complete and correct list of them. The complete list of the used CMake variables and installed packages is present in the project repository [12].

When the procedure is finished, a folder containing all the ROS 2 binaries will be created. To install it on the target board this folder needs to be copied into the actual NanoPi file system (the binaries produced by the build tool are present in the *install/* folder). Note also that the resulting binaries will work on any machine with the same architecture and the same libraries installed. Once the folder is copied onto the companion computer ROS 2 is fully functional and can be used in the same way as a normal installation.

A very similar procedure was also developed to cross-compile packages created by the user but, depending on the size and complexity of the package, it may be not as necessary. With packages composed of a small source code, in particular if using the Python library, the NanoPi is fully capable of performing a native compilation of the package in a completely reasonable time.

3.3 PX4-Autopilot

The chosen firmware for the flight controller is PX4-Autopilot. It is an open source project that supports a great variety of flight controllers, and is able to control multiple kinds of aerial vehicles, for example multicopters and fixed wing. This software can also be easily customized to fit any particular application, from the customization of its features to the tuning of the flight performance. PX4 is, in fact, a software stack that covers different parts of the control of a vehicle, from the controllers to the communication protocols like MAVLink or ROS 2.

This piece of software is the firmware executed by the flight controller and includes the most important aspects regarding the flight of the drone: it manages every aspect of the flight controller, from reading the sensors to executing the main control algorithms, imposing the motor speeds. It's also the main software to interface with to command the drone: it reads inputs from the remote radio controller and also supports other communication protocols to receive commands sent by other applications. This second feature is particularly important, since it allows the integration with ROS 2 in order to perform an offboard control. Regarding the control aspect, PX4 offers different levels of control of a multicopter: from a low level control, for example imposing directly motor speeds or angle rates, to position control. Moreover other simple high level tasks are directly supported, like automatic takeoff and landing or performing a waypoint mission. Depending on the desired level of control, different parts of the flight stack are leveraged, enabling or disabling different control algorithms and injecting the desired setpoints in the control loop. For example, in order to achieve an acceleration control, the acceleration setpoint is sent to PX4 (for example from a ROS 2 node) and, at the same time, the position and velocity controllers are disabled. The controllers layout is shown in figure 1.2 and 1.3.

Another important task accomplished by PX4 is the estimation of multiple quantities relative to the drone flight. Direct measurements are very noisy and the sensors present on the FC are limited, so the estimation is mandatory in order to derive quantities like position or velocity. The algorithm used is an extended Kalman filter, referred to as EKF2. This approach is very useful in particular when integrating to the other measurements the external position measurements (i.e. Vicon), in order to increase the accuracy of the estimate.

Flight modes let the user choose different behaviours of the system, offering a direct control or a higher level of autonomy. Some examples of commonly used modes are:

- *Manual*: the sticks on the radio controller directly impose the setpoint of the angle controller: to make the drone move forward the pitch stick is moved forward to make it lean forward. It does not require any external position information and it was heavily used for testing, since does not suffer from position estimate errors.

- *Hold*: when entering this mode the vehicle current position is set as setpoint of the position controller, so it remains still in the same position. Using the radio sticks the position setpoints can be slowly changed so it's very easy to control.
- *Offboard*: this mode is for controlling the drone with external commands (e.g. from the companion computer). It's the mode used to control the drone from a ROS 2 node running an algorithm. For example, there is the possibility to impose directly angle rates or to perform a higher level control by imposing a position setpoint, leveraging the position controller already built into the autopilot stack.

3.3.1 Build and parameters setup

In order to deploy PX4 onto the flight controller the main steps to follow are to download the full source code, compile it for the intended board, finally flash it onto the flight controller memory. Already built binaries are available directly from the PX4 repository [5], but some needed features are not included by default like the Kalman filter module. For the compilation an already available target was available, supporting the *Omnibus F4SD* board, but again this build configuration did not include some important modules. The autopilot stack is modular so before compilation it is possible to choose the wanted functionalities. This mainly affects the final size of the executable, that needs to fit onto the constrained memory of the FC. An notable limitation of the used flight controller is the limited amount of flash storage, as the memory size is 1 MB, so the PX4 executable needed to be smaller in order to be loaded onto it. The main feature missing from the default configuration was the Kalman filter estimator (EKF2), essential to be able to fuse external vision measurements. That module when added made the executable too big to fit into the FC memory. A process of trial and error was used, in order to find a configuration that produced an executable smaller than 1 MB and included the wanted features. Unused modules like GPS drivers were removed in an attempt to free enough space to include the estimator module. This phase was harder than expected due to the lack of clear documentation regarding each module tasks, and due to dependencies between modules which prevented a successful compilation if some required modules were not included.

The PX4 stack offers a large set of parameters that can be customized in order to enable or disable some features and also tune the control algorithms. After the deploy of the autopilot firmware onto the FC, some parameters have to be set regarding mostly the estimator setup and the vehicle geometry. These parameters can be set using QGroundControl or using the SD card: a file can be written onto it that is executed at each startup, so that the specified parameters are set. This solution was used to control more accurately the entire set of parameters used, and also to execute the command to start the XRCE-DDS Client, a crucial component to allow communication with ROS 2.

3.3.2 Motor ordering

Since PX4 can be customized to fit any aerial vehicle, it needs a description of the geometry of the drone, in particular the position of the actuators. This is important to correctly calculate the speed to impose to each motor, in order to achieve the desired attitude. The attitude control problem relies on imposing a torque on the drone body by changing the thrust produced by each motor. The relation of the motor speed (and thus thrust) to the torque produced on the body, depends on the position and orientation of the motor respect to the center of mass. In case of a quadcopter then the X and Y positions of the 4 motors are specified with the parameters *CA_ROTOR0_PX*, *CA_ROTOR0_PY*,*CA_ROTOR1_PX* ... A mismatch in the PX4 default motor layout was found due to incorrect wiring of the ESC: the motor number 1 was expected from the software in the front-right position but it was wired to another motor. This resulted in a complete instability of the drone, since the required torque needed to control the body angle rate was generated using the wrong motors, and resulted in a complete inability of the drone to fly. After configuring each motor to the actual configuration the problem was solved. An earlier attempt to fix this problem was to swap the wires connecting the FC to the ESC. This resulted in a correct flight of the drone, however the software solution was much more simple to implement that modifying the wiring of each drone.

3.3.3 Communication using ROS 2

The main way of communication of PX4 flight stack is using the MAVlink protocol: it is used both to communicate with QGroundControl software and is also used in the *mavros* packages available for ROS 1. Moreover the most diffused software stack to control aerial vehicles was ROS 1 with *mavros* package. Recently the PX4 project started supporting also ROS 2, previously with a bridge called RTPS and later changed to XRCE. This transition is still in progress during the development of this thesis and the PX4 version used is the beta version of v1.14 (the latest available). Using the beta version of the software created a lot of challenges, since the released code sometimes presents bugs and the documentation was not available or updated to the new changes. The new XRCE bridge was anyway the best solution in terms of compatibility with ROS 2 and the chosen operating system version. Attempts were made for using the more established RTPS bridge, but it was not supported and not working on Ubuntu 22.04. The supported OS is Ubuntu 20.04, but this mandates a downgrade to ROS 2 Foxy. In order to avoid downgrading all the pieces of software (especially considering ROS 2 Foxy is reached the end of life status), the beta version of the autopilot was used.

The new communication stack does not rely on MAVlink or *mavros*, but it directly exposes the inner workings of PX4. Inside the autopilot stack, to allow inter-process communication, a messaging system is used, the uORB messaging where different kinds of information and commands are exchanged. Using the new communication stack the XRCE bridge directly maps some of the uORB topics directly to ROS 2 topics. When a new uORB message is sent from the autopilot stack, for example a message relative to the vehicle status, it is also sent to the relative ROS 2 topic. A small and configurable subset of uORB messages are currently supported for ROS 2 communication. The opposite situation is also possible: an example is the arming command generated by a ROS 2 node. When received by the XRCE Agent, is directly translated and published to the associated uORB message inside the autopilot stack where the correct process will elaborate the message and perform the action. Interacting directly with the inner workings of the autopilot stack allows to access high frequency and low latency estimations and sensors measurements, that can be used to more accurately execute an attitude control algorithm in an offboard ROS 2 node. This communication is in fact aimed to achieve higher performance in terms of latency and responsiveness. Note that only a small subset of the uORB messages are interfaced with the XRCE bridge, as only the messages that are used to command the drone or read status information are available. The cause of this limit depends on the fact that interacting with the inner working of the stack has a high probability of introducing errors and crashing the system.

The layout of the communication software can be seen in Figure 3.2. The client side of XRCE-DDS is directly built inside the PX4 firmware and is run by the flight controller so it has a direct access to the uORB topics and messages. The client then serializes the messages and transmits them to the agent of XRCE-DDS: here is deserialized and published on a topic visible to all ROS 2 nodes. The communication between the agent and the client side of the bridge supports multiple types of transport protocols, like *UDP*, *TCP*, serial, etc. In this application, since the FC and the companion computer are connected via a UART port, the serial transport is chosen.

uORB topic	**Frequency [Hz]**
VehicleStatus	2
VehicleAttitude	120
VehicleLocalPosition	125
VehicleOdometry	125
FailsafeFlags	2

Table 3.2: *Example of uORB topics and the relative publishing frequency*

In Table 3.2 some examples of topics with the relative publication frequency are shown. The needed bandwidth is directly proportional to the number of topics and more importantly to their frequency of publication: the serial transmission from the flight controller to the companion computer needs a baud rate fast enough to handle all the published messages. The baud rate refers to the time a single bit takes to be transmitted, and is closely related to the bandwidth. If a too low baud rate is used, then the behaviour of the published topic is unpredictable: the publication frequency is very low and some topics are completely missing. In case the serial communication does not support a fast rate of communication, some of the most bandwidth demanding topics can be disabled. By disabling the topics with a high publication rate and enabling only the ones relative to status information like *VehicleStatus* and *FailsafeFlags*, the needed bandwidth can be as low as 7 kbps, requiring then also a very low baud rate.

3.3.4 Position and velocity estimation

A crucial aspect of the autopilot stack is the implementation of an extended Kalman filter [3], in the PX4 module called *EKF2*. The total number of states estimated in the PX4 implementation is 23, which include the position, velocity of the vehicle, attitude, IMU biases and others. The input of the filter are the measurements taken from the IMU and other sensors like the barometer and Vicon. Multiple sensors have to be used because providing an accurate estimation of position and velocity simply using the IMU measurement is not possible. These quantities, in fact, have to be calculated by means of numerical integration since the IMU can only provide acceleration measurements. A downside of numerical integration is the accumulation of measurement errors, which can result in an unusable position estimate that drifts over time. In particular, the vibrations affecting the IMU increase the uncertainty of the measurement and this effect is accentuated. In Figure 3.3 is shown the effect of accumulation of errors due to numerical integration, when using acceleration only data from the IMU: due to the high drift the estimate is not usable for precise positioning during flight.

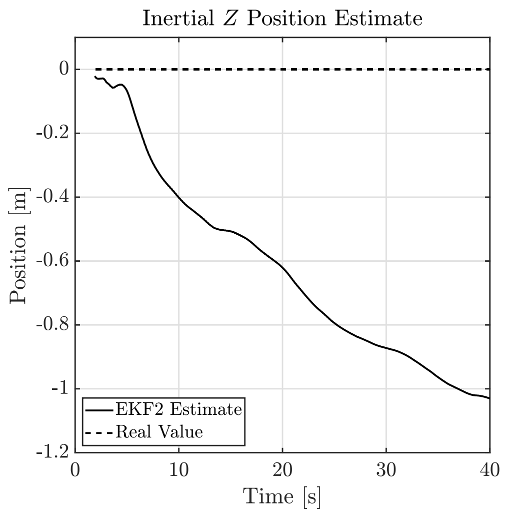


Figure 3.3: *Estimate of Z position in stationary condition, relying only on IMU measurements*

To allow a precise position control, an accurate estimate of this quantity is required. This is done using more measurements that directly measure the position, that is then fused by the Kalman filter into the final estimate. There are multiple options to directly measure position, and a common one is using GPS localization. In this project the position measurement was retrieved using the the Vicon system, since GPS was not available. Additionally the Vicon system performance is much better both in terms of accuracy and latency.

In order to fuse the Vicon measurements into the final estimate, a setup of the filter module is required. This module present a lot of parameters to work with, giving the possibility to enable different source of data and tune the performance relative to the accuracy of each data source. The fusion of the external vision measurements is enabled and a tuning of PX4 parameters is performed in order to let the estimate rely more on the Vicon measurements rather than the IMU. This is achieved by working on the covariance assigned to each measurement source: if the covariance is small then the measurement will have a greater effect on the final estimate, and vice versa. This had to be done because of the vibration problem affecting IMU (Section 2.1), that managed to make the estimate drift from the Vicon direct position measurement even if it was correctly fused. An example of this situation is represented in Figure 3.4 where data recorded from a test flight shows a huge difference between the real height (Vicon measurement) and the estimation, caused by the visible high noise spikes in the acceleration measurement. In order to reduce this issue, the accelerometer and gyroscope covariances were increased and the covariance relative to the Vicon measurement was set to a very small value of 1 mm. After reducing the vibration that affected the FC (section 2.1.1), the estimate quality improved again.

Another important issue encountered in this phase was the difficulty to establish a robust stream of measurements from the Vicon server to the flight controller: if the direct position measurements are not correctly received, the estimation will continue using only IMU data and the result will be uncontrolled behaviour of the drone. Particular effort was put in ensuring that the lateness of the samples was kept low and the frequency remained at acceptable levels (above 30 Hz), by minimizing losses of packets over the network. If the previous requirements are not met, then the estimator only relies on IMU measurements and the position estimate would start to drift from the real value and in case for example of position control the drone starts to move in a direction dictated by the wrong position estimate, often resulting in a crash. Using correctly the Vicon SDK to retrieve low latency measurements, needed for this real-time application, was an important part of improving the robustness of the position control of the drones.

The main cause of the wrong reception of the Vicon data relies in networking issues relative to the wireless part, already discussed in Section 2.2.

3.4 Vicon system

The flight environment is composed of a safety cage inside of which the drones can fly, needed in case of loss of control of a vehicle: drones will only crash inside the cage, preventing damage to anything outside. During the flight of the drones nobody is inside the cage and the vehicles are only commanded remotely using the radio controller or running an offboard control algorithm. Inside the cage a Vicon system is mounted: it's composed of 6 cameras that are able to measure the drone's position and orientation inside the entire flight area.

Drones, the ground station and the Vicon server are all connected to a wireless router isolated from the internet and from the company network. In a first attempt, a router connected with the rest of the network was used, but resulted in an increase of latency and sporadic loss of messages. In a further attempt to improve the network reliability when

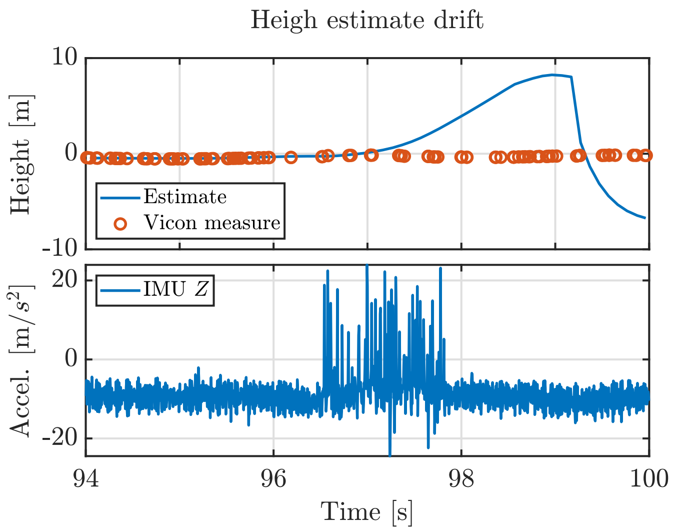


Figure 3.4: *Drift of the EFK2 height estimate from the Vicon measurement directly related to IMU excessive vibrations.*

facing important connectivity issues to the drones, multiple router models were tested. The most reliable solution found was to have a completely isolated router from other networks. The ground station and the Vicon server are also connected using Ethernet cables, so only the drones use a wireless connection. The wired connection of the ground station to the router also seemed to increase the reliability and lowered the latency of communication respect to using a wireless one. The node that interacts with the Vicon server and sends the position information to the drones is executed on the ground station, so wired connection directly improve the overall latency of the Vicon samples.

The Vicon system allows to measure very accurately the position of the drones inside the cage, offering a position groundtruth to be used by the autopilot estimator. Special infrared reflectors are positioned on each drone and the system can recognise each vehicle accordingly to the layout of the reflectors onto it. An example of reflectors is visible in figure 1, representing the actual configuration used. The system falls under the motion capture category: all the cameras simultaneously take a picture of the flight area and the infrared markers can be easily identified from the scene. Then all the information is then fused together (directly by the system) and a 3-dimensional position and orientation of each vehicle is provided.

The aim of using the Vicon system is to retrieve real time measurements of the drones position, using the provided SDK integrated into a ROS 2 nodes. The SDK library offers a C++ API to retrieve the data connecting to the server that manages the Vicon cameras. Some simple implementation of the SDK ready to be used with ROS 2 are already availables: these libraries did not offer a satisfactory performance in terms of latency and frequency of the data. This issue was made worst by the wireless connectivity problems induced by the radio controller and wifi interference. A big effort was put in obtaining a good performance by writing a program to interface with Vicon from scratch, directly

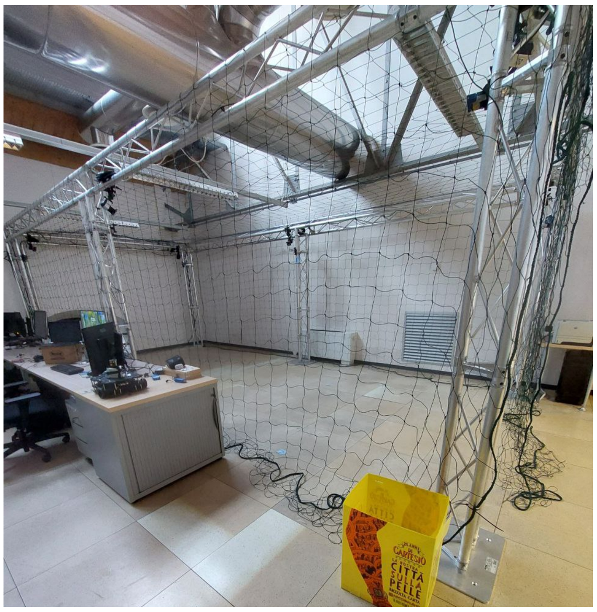


Figure 3.5: *The used safety cage for drone flight*

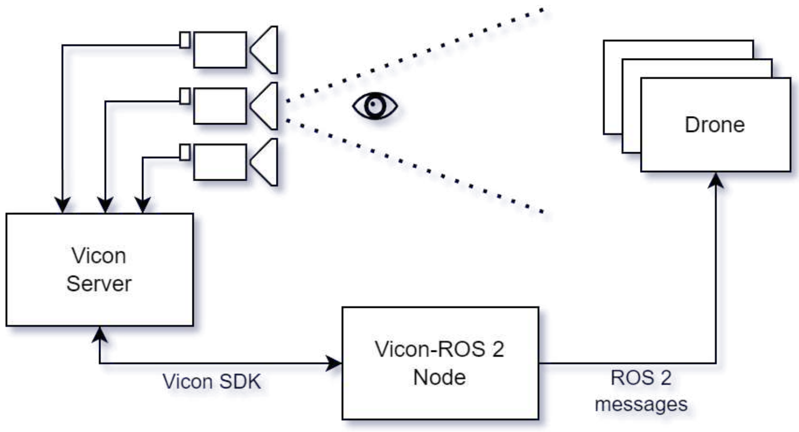
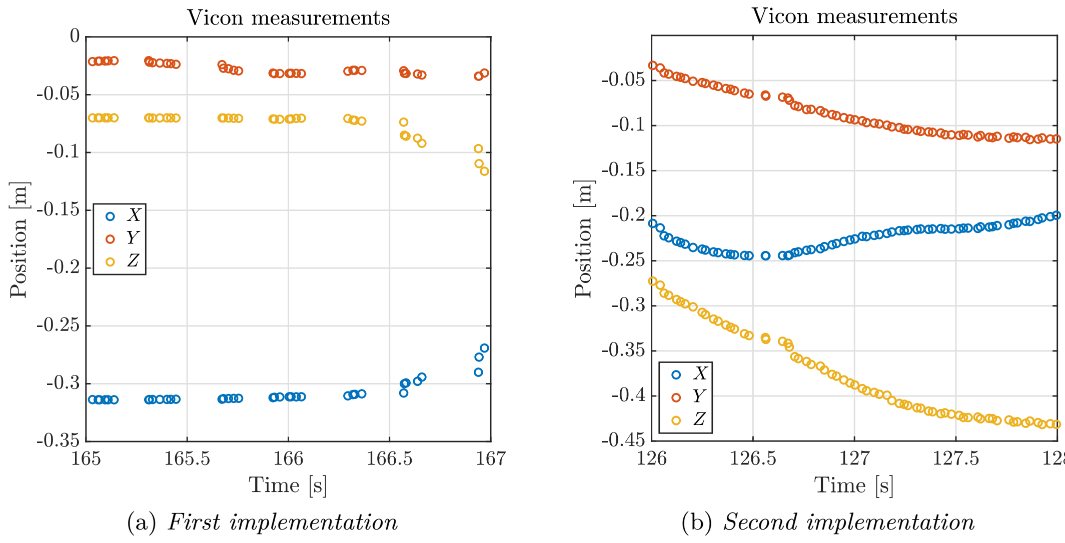


Figure 3.6: *Vicon data stream layout*

using the official SDK. The main issue to resolve was to provide a stable stream of low latency samples to send to the drones. If samples had high latency (200-300 ms) they were discarded by the *EKF2* estimator because they don't provide additional information for data fusion. Otherwise if the elapsed time between samples was too high, then the autopilot estimator recognises the external vision data stream as stopped and proceeded with only inertial measurements, resulting in a drift of the position estimate.

3.4.1 Vicon SDK Implementation

In the first iteration, basic functionality offered by the C++ SDK were implemented into a ROS 2 node: after getting the position of the drone from the Vicon server, it was directly translated into the appropriate ROS 2 message and sent to each drone. The topic used to transmit the position measurement is */fmu/in/vehicle_visual_odometry*. This solution relied on the base class that handles communication to the Vicon server called *ViconDataStreamSDK::CPP::Client*. This class returns all the samples recorded by Vicon in order, without skipping any: when calling the function to retrieve the sample, the following one from the last function call is returned, independently from the time elapsed between the calls. The function that retrieve the data is a blocking call, sometimes stopping the execution unexpectedly for 0.1 seconds, a very high time respect to the requested 50 Hz rate. This resulted in an accumulation of old samples, that were discarded due to the high latency. Various solutions were tried in order to improve the performance, in an attempt to meet the needed requirements: a multi-threading approach was used to avoid the blocking calls influencing other tasks, First In First Out buffers were used to store the Vicon samples and interface the various tasks. The old samples were directly discarded from the buffers, in order to avoid sending old data and speed up the algorithm.



(a) *First implementation* (b) *Second implementation*

Figure 3.7: *Comparison of the performances of the two implementation of the Vicon SDK. The plot shows the packets received by the flight controller in a 2 seconds interval. The expected rate is 50 Hz*

A lot of troubleshooting was done in this phase to understand the reason of the bad quality of the received data, but the results were still not good enough to work reliably with the autopilot. The stream of samples presented regular interruptions, seen in Figure 3.7, even after decoupling all the node's tasks with multi-threading and FIFO buffers.

The second implementation of the SDK was again done from scratch, to avoid introducing possible errors present in the first version. This time the class *ViconDataStreamSDK::CPP::RetimingClient* offered by the SDK was used. This class offers lower latency performance with respect to the previous one and also offers the possibility to linearly interpolate between two samples to achieve even higher temporal precision. A downside is that in order to reduce the latency, a little higher uncertainty on the position is present, but it is not a problem in this application since the data is then processed by the autopilot Kalman filter.

The main functionality used is *RetimingClient::WaitForFrame()* that blocks the execution until a new sample is available from Vicon. The key difference from the previously used class is that this function returns the newest sample, and all older samples are discarded. Again this function is also blocking, because it is waiting for a new sample, but since the samples are published with a frequency of 100 Hz, the need to use multiple threads to manage the functions didn't arise. Then the function *RetimingClient::UpdateFrame()* performs a linear interpolation on the last acquired sample, in order to improve further the temporal precision.

```
void vicon_rcv(){
    /* Wait for sample from Vicon */
    Output_WaitForFrame WaitOutput=vicon_client.WaitForFrame();
    if( WaitOutput.Result == Result::Success )
    {
      this->vicon_client.UpdateFrame();
      Frame new_frame = this->vicon_client.GetFrame();
      /* Send Vicon sample to PX4 */
      if(!new_frame.segments.empty())
      {
        this->publish_px4_msg(new_frame.segments[0]);
      }
    }
}
```

Listing 3.1: Function used to receive data from Vicon an publish it to PX4

The function in listing 3.1 is executed with a frequency of 50 Hz, generating a much better stream of samples, with very low temporal jitter. The downside is the presence in the stream of data of outliers: this is not an issue because the Kalman filter easily removes them, so no prior filtering is required. The performance of the second iteration of the Vicon SDK implementation are shown in Figure 3.7.

The performance obtained with the latter implementation of the Vicon SDK improved a lot the precision and stability of the estimations performed by the autopilot Kalman filter. At this point it was possible to safely use the position estimate for example to control the drone position without risking random drifts or loss of control.

Executing the *Vicon-ROS2* node on the ground station computer was mandatory since the SDK library provided by Vicon only supports 64-bit machines and the companion computer has a 32-bit architecture. The library provides a shared object file that is already compiled and ready to be linked with the rest of the project source code. In order to deploy this node onto the companion computer, an entire cross-compilation of the Vicon SDK source code was necessary. Executing the node on the ground station introduces some latency to the samples provided by the Vicon system, because data is not directly being trasmitted to the companion computer, however the performance loss is negligible.

3.5 Safety measures

A very important aspect to consider when flying drones are procedures to regain control of the vehicles in case of unexpected behaviours or to perform a complete shutdown if needed. In particular, these actions have to be performed remotely because the drones can't be handled by hand when flying due to the spinning propellers that are very dangerous. In this project, all the interaction with the drones can happen both with the remote radio controller (each radio is linked with only one drone) or using commands generated from an offboard algorithm.

There are multiple situation where a loss of control can happen, but the main source of unexpected behaviour in this case is the offboard control. The algorithm that is responsible for controlling the drones may impose wrong commands that result in a crash into another drone or into the cage walls. An example of possible scenario is the complete loss of connectivity to the drone when controlling it in offboard mode: if the companion computer can't impose waypoints or acceleration commands then a crash can happen. Note that a message is sent to the autopilot on the topic */fmu/in/offboard_control_mode* with a frequency greater than 2 Hz, representing the liveliness of the computer executing the offboard algorithm. In case the autopilot stops receiving this message, an autonomous landing is performed immediately handling situations of complete loss of connectivity.

The two ways to communicate to the drone are using the remote radio controller and the ROS 2 framework and both have the possibility to switch flight mode or terminate the flight if needed. The first action performed to regain control of the drone if the offboard control is not performing as expected, is using the remote radio controller connected to the drone to override the external commands and control the drone manually. This is very difficult to do because requires good skills in manual control using the radio controller of the drone.

Another solution to regain the control of the drone if it is still in a stable flight condition, is to switch the flight mode to *hold mode* when the drone is still flying and is, for example, about to collide with another object. This is an autonomous mode that controls the drone position. The mode change can be triggered both from the radio controller, where a switch is directly mapped to the mode change, and from the ROS 2 environment. To perform a mode change using ROS 2, a message is published to the */fmu/in/vehicle_command* topic, instructing a mode change to the desired one. After the drone is hovering in hold mode, multiple options are available, for example using manual control or performing an autonomous landing.

The last available option to use is flight termination. This command instantly stops all the motors from spinning and it has to be used only as a last measure. When using this command the drone receiving it will stop all motors from spinning in any condition is in, both in flight or not. If the drone receive a flight termination command while it is flying, it will to fall to the floor instantly, receiving possible damage. For this reason, it should be avoided in most cases, using it only if the drone already crashed. When the drone crashes the motors should stop spinning to avoid hitting the floor and causing more damage but it doesn't always happen. This command is very useful in this situation. The flight termination command can be sent from the radio controller with a suited button that is intended only for this task or also from a ROS 2 node. In order to send this command to PX4 from ROS 2, the command *VEHICLE_CMD_DO_FLIGHTTERMINATION* on the same topic cited before. The main reason to have a radio controller connected to each drone when using the offboard control is the availability of the flight termination button on a different communication channel other than WiFi.

The main purpose of the presented measures is to regain control of the drones. In case of a crash, in fact, the drones can suffer a lot of damage, depending on the situation, and the aim is to try to reduce it to a minimum. During the setup of the swarm and the development phases, is quite frequent to encounter situation of loss of control since a lot of unseen issues can arise unexpectedly. An example is the setup phase related to integrating the Vicon measurements to correctly estimate the drone position. The estimate was not stable or correct at all at the beginning. A lot of test flights were performed and sometimes they resulted in uncontrolled drifts of the drone that crashed into the walls of the cage.

The safety related to the human operators is already taken care by using the cage: it is completely enclosed so the drones can't manage to escape and the only thing that can happen is the drones suffering damage.

Chapter 4

Target Estimation and Tracking

After the first part of this thesis project was finished, the drones were ready to fly and be localized by the Vicon system. In order to test the setup, an offboard application was deployed onto the drones, after being developed and tested with the help of a simulation environment offered by *Gazebo*. The implemented application includes two main subjects. The first is a diffused estimation, using a information form Kalman filter, of the position of an ArUco marker positioned on the floor. The position and velocity of the marker are estimated using the onboard camera of each drone as a source of measurement. The second subject of the task is the implementation of a flocking algorithm that aims to both control the relative position between the drones and to track the position and velocity of the marker. This means that the swarm of drones tries to stay above the marker and match its velocity.

This work was based on the paper [9], implementing the same algorithms for the flocking and for the Kalman filter. The used framework is ROS 2, and the communication with the drones is done using XRCE-DDS, using the setup described in the previous chapters. The code was tested in a *Gazebo* simulation, in particular the simulation was configured so that the developed ROS 2 nodes don't need any modification in order to be deployed on the real drones. More details on the simulation environment and the obtained results are described in chapter 5.

Different nodes have been developed for the different parts of the algorithms. In order to work on multiple drones using the same source code, the nodes use parameters and name-spaces. All the topics related to a single drone start with a unique identifier, so the same topic can be differentiated across multiple drones. For example, the topic */estimation*, that outputs the estimate of the Kalman filter, is present on every drone and a unique prefix is added (e.g. */drone1/estimation*, */drone2/estimation*, ...). The nodes can be instantiated with different parameters, for example the drone name-space, and this allows to reuse the same code and to render trivial the modification of the number of the drones. For each drone the main nodes that had been developed in order to execute the previously mentioned task are:

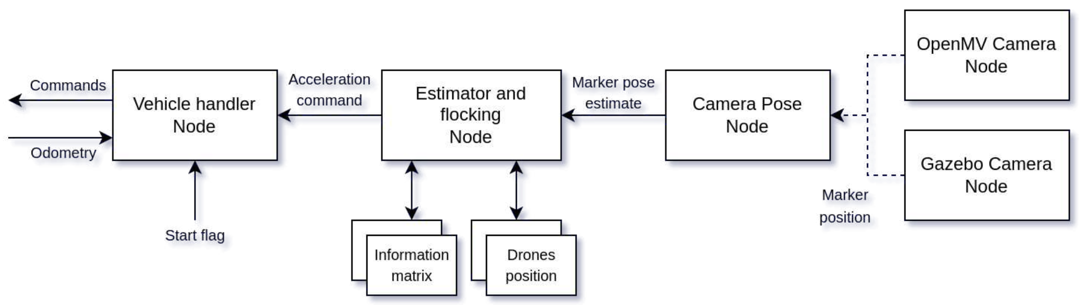


Figure 4.1: *The ROS 2 nodes instances for a single drone.*

- The *Vehicle Handler node* interacts with XRCE-DDS and is the responsible to send commands to the drone like arming and setpoints. Additionally reads the odometry given by PX4 and translates it to TF2, which will be used by the other nodes to read information about any transformation. TF2 is a transoform library tool built into ROS 2.
- The *Estimator and Flocking node* executes the Kalman filter algorithm, taking as input the camera readings and outputs the estimate of the marker position. In this also the flocking controller is executed too. It that reads the drone position (using TF2) and outputs the control input as an horizontal acceleration. The vehicle handler node will then forward the acceleration control input to the drone.
- The *Camera Pose node* calculates the position in the 3D space of the ArUco marker, taking into consideration the camera intrinsic parameters, the drone position and the reading of the camera sensor. The readings can came from a node that interact with the *Gazebo* camera or with the real OpenMV camera, depending if the nodes are executing in the simulation or on the real drones.

In order to easily start the required nodes with the correct parameters, the launch system of ROS 2 has been used. This tool allows to specify a configuration through a file where the required nodes are described, including the respective parameters to set for each node. For example, in case of the *Gazebo* simulation, all the nodes are executed on the same computer, so the launch file executes all the nodes for all the drones, each configured with a different name-space, in order to refer to the correct simulated drone. In case the algorithms are executed on the companion computer of the real drones, the launch file will only execute the nodes related to the drone they are launched on.

4.1 Vehicle Handler Node

Interacting with the drones using the XRCE-DDS interface requires to subscribe to the topics published by PX4, create the messages to send to the drone and receive the odometry messages. The role of the vehicle handler node is to interact directly with the assigned drone (via the name-space) in order to perform the takeoff, landing and other tasks. This node is the only one that subscribes to the PX4 related topics. For each drone a node of this type is instantiated so each drone has an independent control logic. In order to couple a drone to the vehicle handler node, since the PX4 topics have the same name in each instance, a name-space is set on the topic names published by PX4 and the same name-space is assigned to the node. This allows the node to send and receive messages only to the desired target, avoiding mismatches.

Regarding the odometry of the drone, all the nodes described before need the position information in order to perform calculations and, instead of directly reading the PX4 topic, they use the ROS 2 tool *tf2*. The odometry message retrieved from PX4 uses a FRD coordinate system, different from the standard FLU convention. The odometry message is read from the topic *fmu/out/vehicle_odometry*, then the position information is transformed into a FLU reference frame, both the translation and the rotation quaternion. Finally the transform of the drone position relative to the origin is published to the other nodes using *tf2*. Other nodes, to retrieve the position of a drone, read the tf2 transform from the *map* frame (fixed frame) to the drone frame, for example the frames *drone1* or *drone2*. Depending on what frames are specified the tool automatically takes care of concatenating the correct transformations.

The commands and the topics needed to be published in order to perform an offboard control are all handled in this node. The vehicle commands are needed to perform actions on the drone state. The main performed actions are: arming, disarming, flight mode change, landing, flight termination. The commands are published to the topic *fmu/in/vehicle_command*. Another important aspect to handle which enables offboard control is the steady publication rate of the *OffboardControlMode* message to the topic *fmu/in/offboard_control_mode*: this message acts as a keep-alive message to ensure the liveliness of the offboard control computer and also sets the desired control mode. The rate of publication of this message is handled by a timer offered by the ROS 2 Node class, that every time it expires publishes the message (in this case 100ms period). Different control modes can be set, in particular there is the possibility to set different modes on different axes at the same time. For example, position control can be requested so the drone reaches the specified coordinates, or acceleration control can be enabled. The acceleration control mode is used with the flocking controller: it outputs an acceleration command and the vehicle handler node sets the control mode to acceleration in the X and Y axes and position in the Z axis. Doing that the altitude is maintained automatically and the controller only acts in the horizontal plane. The control setpoints are formatted into a *TrajectorySetpoint* message and published to PX4 on the topic *fmu/in/trajectory_setpoint*.

The drone behaviour is controlled exclusively from this node, using a state machine to cycle between different states and depending on the state perform different actions. The states transitions are performed by reading the current state variables of the PX4 autopilot, allowing to reliably perform the intended logic.

4.1.1 State Machine

In order to handle the various flight events, like arming, takeoff, landing etc. a simple state machine has been used. The implemented state machine uses one state variable that represent the current state, and a timer that on each expiration execute the current state actions and checks the transition condition. To transition to a new state, the state variable that represent the current state is overwritten with the new value, so on the next execution the code relative to the new state will be run. The conditions that are checked in order to transition to another state are based on the current state reported by the PX4 autopilot on the topic */fmu/out/vehicle_state* so, even if some commands are not correctly executed by PX4, there is an active check on the wanted behaviour of the drone. There are 5 different states implemented in the state machine and each one represents a different part of the flight:

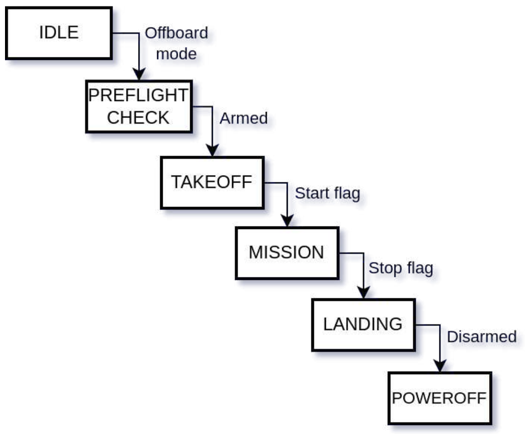


Figure 4.2: *State machine logic that controls the drone behaviour.*

- *IDLE* state is executed at startup, here the flight mode is set to offboard and the actual drone flight mode is monitored to trigger the next state.
- *PREFLIGHT_CHECK* state waits for the drone to pass the internal checks and be ready to be armed. After sending the arming command the state transitions at the same time when the drone state changes to armed.
- *TAKEOFF* state sets the control mode to position and impose a setpoint in order to make the drone rise to the desired altitude without an horizontal position change. Then the start flag for flocking allows transition to the next state. This condition is checked by waiting for a message on the topic */start_flocking*. The idea is to manually send a steady rate of messages of type Bool on this topic so the state transitions to the next one only when intended.
- *MISSION* state is relative to the execution of the flocking algorithm. The acceleration command computed by the flocking algorithm on the topic *acceleration_cmd* is forwarded to PX4 as an acceleration setpoint, so the algorithm can fully control the drone. The flocking node, in fact, keeps running even if the state machine is not in this state and its output are simply ignored. In order to remain in this state a steady rate of publication on the topic */start_flocking* is expected then, and as

soon as it stops, the state is transitioned to a landing. This is done because from the ground station, where this command is generated, the landing can be triggered in any moment manually simply by stopping the messages publication.

- *LANDING* state performs the landing of the vehicle. An automated landing routine is present and can be activated by sending a *VehicleCommand* message with command set to *VEHICLE_CMD _NAV_LAND*, but it was not working correctly due to a bug in the PX4 source code. The bug consists in the landing of the drone in the coordinates (0,0) even if other landing coordinates are imposed. When working with multiple drones this is an important issue since all of them want to land in the same spot. This issue was confirmed also with a PX4 developer on the official forum. To work around this issue the landing is performed by imposing a position setpoint with the current X and Y coordinates of the drone and altitude equal to 0 m. In this way the drone reaches the ground by slowly descending vertically from the current position and automatically disarms after has landed. Note that the topic */start_flocking* is read by all the active drones, so when the message stream is stopped all the drones will detect this event and perform a landing.
- *POWEROFF* is the last state and it sends a flight termination command after the drone touched the floor. This step is not necessary but it is an added safety since sometimes the drone does not correctly detect the landing and, consequently it doesn't disarm. Additionally, after this command the drone can't be armed anymore without a power reset. This prevents the drone from re-arming and taking off in case of a bug in the code that restarts the state machine in the IDLE state.

4.2 Computer Vision

The aim of the developed task is to estimate accurately the position and velocity of a certain target and then, with a flocking algorithm, move the drone swarm to track the estimated target position and velocity. The chosen target is an ArUco marker that has be to located in the space using the onboard cameras, obtaining a measurement of its position from each drone.

The first step is to identify the ArUco marker in each video frames captured by the camera: ArUco markers are easily recognisable from an image since they are made exactly for this purpose. The library OpenCV offers functionalities to identify and retrieve the marker position in pixel coordinates, that's why it has been used with the *Gazebo* camera, to identify the marker in the image retrieved from the simulation in pixel coordinates, that are then fed to the *Camera Pose node*. The OpenMV camera on the drone directly allows to perform computer vision algorithms onto its microcontroller, so the marker pixel coordinates are directly calculated inside the camera without the need for the OpenCV library. From figure 4.1 it's shown that both the *Gazebo* camera and the OpenMV camera sends information to the *Camera Pose node*: the marker position in pixel coordinates is published on the topic *camera/marker_pos*. The message type is *Float64MultiArray* and in the first position stores the marker ID, in the second and third position it stores the marker (u, v) pixel coordinates. An ArUco marker in fact is also associated with a numeric

ID, that can be used to filter the output of the computer vision algorithm and discard any unwanted results.

The task of obtaining the marker 3D position from a 2D sensor image is performed taking into consideration some internal characteristic of the camera. A geometric approach can be used analyzing the projection of a 3D point onto the camera sensor. The projected point is represented by the sensor with a 2D coordinate (u, v) expressed in pixels, and the objective is to obtain the space coordinates of the object that generated that projection on the sensor. Using only a single camera there is still one free variable in the solution: the depth information is lost when the is image captured, so in order to obtain a unique solution also the distance from the point to the sensor is a needed datum. In order to avoid this problem a common solution is to use a stereo camera, that adds more information and is able to estimate the distance to the target. Another way to estimate the depth information is knowing the real world size of the target, thus adding an additional equation to the system to solve and obtaining a unique solution. A more in depth explanation on the used algorithm to estimate the pose of the marker is explained in subsection 4.2.1. After obtaining the position of the marker relative to the camera, there is still to consider the camera position relative to the fixed frame in order to obtain the position of the marker in the fixed reference frame. In our case in fact the camera position and orientation is highly variable because is attached to the drone that is constantly moving. The camera position and orientation are called extrinsic parameters. The *tf2* tool is used to aid this concatenation of transformations, since it allows to specify the single transformations and automatically it concatenates them in the correct order.

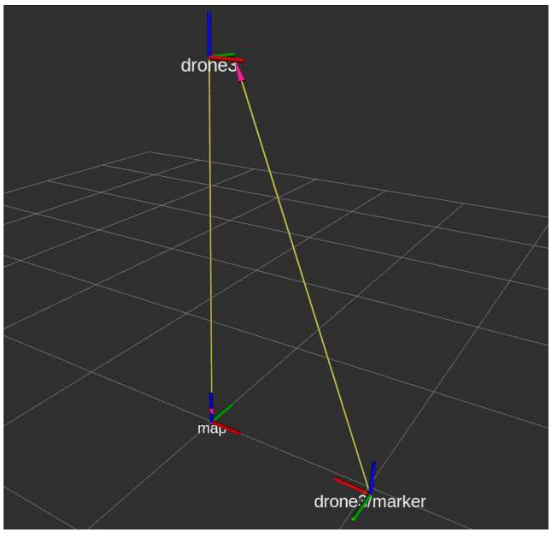


Figure 4.3: *Diagram of the different reference frames handled with tf2. The camera frame is not represented.*

The transformation chain is composed by three reference frames:

1. "*drone*" represents the transform between the fixed frame (*map*) and the current

drone position. It is constantly updated by the *Vehicle Handler node* when a new odometry message is received.

2. "*drone/camera*" is the frame associated with the camera: the camera is not in fact coincident with the odometry frame and also has a different orientation since it points downward. A static transformation is published from the drone frame to this frame, representing the location of the camera on the drone.

3. "*drone/marker*" is the frame associated with the ArUco marker, is calculated with the transformation retrieved from the computer vision algorithm that links the camera to the marker. Also with this transformation the different axes convention used for optical pose estimation is handled.

Each transformation is published independently and the tool automatically handles the calculation. In order to obtain the position of the marker in fixed coordinates, simply requesting to the tool the two frames will result in the transformation (from *map* to *drone/marker*) where the translation represent the marker coordinates.

4.2.1 ArUco marker pose estimation

An AruCo marker is a square delimited by a black border, and presents a black and white matrix that encodes its ID. These characteristics allow to easily recognise it from the scene, and accurately position the 4 corners in the pixel coordinates. The aim of these markers is in fact to be easily recognised and accurately estimate their pose only using a camera. After identifying the pixel coordinates of the four corners of the marker, a correspondence from the 2D points to a 3D point in the space is needed. The mapping of the points from the camera reference frame only depends on the camera *intrinsic parameters*, for example focal length, etc. The 3D point coordinates can also be expressed in a fixed reference frame (different from the camera frame), especially if the camera is not stationary for example when mounted on a drone. The camera translation and rotation matrices are used to express the point coordinates from the camera frame to the fixed frame, these are referred to as *extrinsic parameters*. These latter parameters are easily retrieved knowing the drone reference frame and the position of the camera onto the drone. The transformation matrix that defines the camera position in the fixed coordinate frame is:

$$T = T_{\text{drone}} \cdot T_{\text{camera}} = \begin{bmatrix} & R_d & & x_d \\ & & & y_d \\ & & & z_d \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & 0.08 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.1}$$

where $\mathbf{R_d}$ is the rotation matrix associated with the drone attitude, and T_{camera} is the static transformation from the drone frame to the camera frame. These calculations are automatically handled by the transform tool mentioned in the previous section.

The intrinsic parameters describe the mapping from the 2D image plane to the 3D world. This mapping is expressed using the camera matrix as:

$$C = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{4.2}$$

where f_x and f_y are the focal distances, c_x and c_y represent the optical center. In order to retrieve this matrix, a process named camera calibration is needed. These parameters depends on the actual camera and parameters used, and the values needs to be extrapolated from experimental data.

To retrieve the space coordinates of the point projected on the image, an additional information is needed. The distance from the point needs to be known, since a single camera looses the depth information. The following equation is used in the algorithm in order to calculate the position of the marker relative to the camera frame:

$$\mathbf{x_m} = \begin{bmatrix} x_m \\ y_m \\ z_m \end{bmatrix} = C^{-1} \cdot \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} z_d \tag{4.3}$$

The marker coordinates in the image are represented with u and v. The distance from the marker to the camera is assumed as equal to the height of the drone z_d (this is an approximation), since the marker height is zero. When a new frame is available from the camera, the marker is identified in the image and the equation 4.3 is used to calculate the marker position relative to the camera. Also each time the calculation is performed the current height of the drone z_d is retrieved.

The identification of the marker inside each camera frame is done using the library *OpenCV*. The frame is passed to the function *ArucoDetector::detectMarkers()*, that returns the position of all the identified markers in the image. Other functions are available from the library that can directly perform the pose estimation, given the camera parameters. The function *cv::SolvePnP()* can be used to map a point from the image to the 3D coordinates. This function however was not used because it didn't work correctly. Multiple tries with different functions and different parameters were performed, but the functions of the OpenCV library were not able to provide any working results. Trying different version of the library didn't solve the issue either.

In order to retrieve an estimate, the equation 4.3 was directly used to perform the calculation without using the computer vision library. This approach is much simpler than the algorithm used to refine the estimate that are implemented into the library, nonetheless it provided satisfactory results, both in simulation and with the real cameras.

4.2.2 Camera Calibration

The camera matrix that maps the image points to the 3D world, is fundamental in the calculation of the marker pose. The structure of this matrix is represented in equation 4.2. The camera calibration process aims at retrieving these parameters, and should be done for each camera to achieve accurate results, even if multiple cameras are the same model and have the same configuration. In order to estimate the intrinsic parameters, a set of points with a known world position is needed, along with their respective projections in the image plane. If the camera perfectly respects the pinhole model and the two set of points are exactly known, then the solution exists and is unique. However in reality all of these conditions are not met, in particular the position of the points is known with a limited accuracy and noise also affects the measure. Depending on how the error related to each constraint is measured, it is possible to determine or estimate an essential matrix which optimally satisfies the constraints for a given set of corresponding image points. The most straightforward approach is to set up a total least squares problem.

In practice, the calibration of the OpenMV camera was done by leveraging some functionalities of the *OpenCV* library that implements the aforementioned estimation algorithms to retrieve the camera intrinsic parameters. A chessboard pattern is used: knowing the real size of the grid, the squares corners are easy to identify and are used to relate the real point coordinate to its projection on the image. The function *cv::FindChessboardCorners()* is used to identify the corners from an image, and then these points are fed to the function *cv::calibrateCamera* that retrieves the intrinsic camera parameters, including the distortion coefficients.

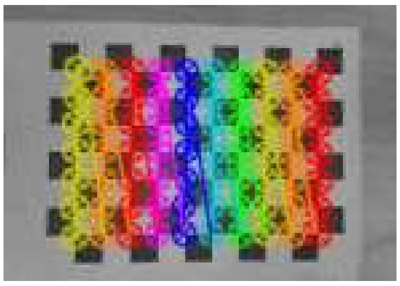


Figure 4.4: *Image from the OpenMV camera of the chessboard, overlayed with the recognised pattern by OpenCV*

4.3 Information Form Kalman Filter

Using the onboard camera each drone can now provide a measurement on the position of the marker, as explained in previous sections of this chapter. This measurement is affected by noise, and fusion of all the drone measurements is done in order to retrieve an accurate estimate of the marker position and velocity. To accomplish this a Kalman filter is used, in particular formulated in the information form. Each drone uses a different instance of the filter (one estimation node for each drone) and the main inputs for the algorithm are the measurement provided by the drone camera and the information vector and matrix shared by the other drones. The estimation node is setup so it subscribes to the *information* topic of the other drones and when a new matrix is published, it is copied to a local variable of the node in order to be used when the calculation is done. The different estimation nodes in fact don't execute the filter update in the same exact moment, as it's executed only when a new measurement is available from the camera of the same drone. In particular in the deployment on the real platforms there could also be random delays due to the network, that affect the real-time share of the information matrix between the nodes. The estimates of each instance of the filter rely on only the data provided by the camera of the drone they are deployed on, but they share the information matrices and vectors. Since multiple estimator nodes are executed, each one of them provide a different estimate of the marker state. The expected result of the information form filter is that the multiple estimates should converge and be very close. Examples of the performance in a simulation environment will be presented in section 5.1.1.

The marker is represented with a simple dynamical system, using 4 states that include the planar position and velocity $\xi = [x_t, y_t, \dot{x}_t \dot{y}_t]^T$. The the dynamical model is represented by:

$$\xi(k+1) = F \cdot \xi(k) + w(k) \tag{4.4}$$

where $w(k)$ is a zero mean white noise and

$$F = \begin{bmatrix} 1 & 0 & \Delta T & 0 \\ 0 & 1 & 0 & \Delta T \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.5}$$

Note that ΔT is the time elapsed between two updates of the filter, and the accuracy of this value directly influence the accuracy of the velocity estimate. In the algorithm in order to guarantee a precise time step calculation, each measurement is coupled with a timestamp of the instant it is created. In particular during the simulation we expect a certain frame rate but the simulation can slow down and the samples are more spaced in time. Using a hard coded value for the time between samples leads to a inaccurate velocity estimation. In order to improve the reliability of the velocity, the time step is calculated as a difference of timestamps of two successive measurements. Correctly estimating the velocity is important also for the flocking algorithm, since the control input has a term relative to the velocity matching of the marker. If the velocity estimate is wrong then the drone speed will be different from the marker one, resulting in an increase of the distance from the marker and the drone.

The marker position measurements taken from the camera are directly expressed in the fixed frame coordinate, but the filter is configured to work with a particular type of measurement model called bearing sensor. The relative X and Y coordinates between the drone and the marker are transformed into a distance and an angle. The distance is calculated using the Euclidean norm and the angle is calculated using a 4-quadrant arc-tangent function [11]. This function, given the two coordinates, returns the angle in the range $(-\pi, +\pi)$. The model of the bearing sensor is:

$$h = \begin{bmatrix} \sqrt{(x_{\text{marker}} - x_{\text{drone}})^2 + (y_{\text{marker}} - y_{\text{drone}})^2} \\ \arctan\left(\frac{y_{\text{marker}} - y_{\text{drone}}}{x_{\text{marker}} - x_{\text{drone}}}\right) \end{bmatrix} \tag{4.6}$$

and the measurement model is

$$z(k) = h(\xi(k)) + v(k) \tag{4.7}$$

where $v(k)$ is a zero mean white noise.

The bearing model is calculated in the *Camera pose node*. After calculating the marker pose from a new frame, the node will calculate the bearing measurement using equation 4.6, using the relative coordinates between the frame *drone* and the frame *drone/marker*. These data are published on the topic *sensor_measure* with a *Float64MultiArray* message, and then read by the estimation node. Each time a new sample is received the filter algorithm is executed and the estimate is updated. The algorithm is composed of two sections, the first is the prediction step and the second is the update step. In the prediction step only the system dynamical model is used to have an initial estimate of the system states. Then the bearing sensor measurements is transformed into fixed frame coordinates and the update step is performed, where the information matrix is calculated and the new measurement is integrate into the final estimate. The information matrix and information vector are calculated as

$$I = \nabla h(\xi(k|k-1))^T \cdot R^{-1} \cdot \nabla h(\xi(k|k-1)) \tag{4.8}$$

$$i = \nabla h(\xi(k|k-1))^T \cdot R^{-1} \cdot (z(k) - h(\xi(k|k-1)) + \nabla h(\xi(k|k-1)) \cdot \xi(k|k-1)) \tag{4.9}$$

then are published on the topic *information* as a *Float64MultiArray*. During the setup of the filter algorithm with the help of the *Gazebo* simulation environment a particular edge case in the calculation of the information vector caused significant oscillations in the position estimate (around 1 meter of error), that later caused also the flocking algorithm to react to the wrong estimate with oscillations in the drone position. This issue was caused by not correctly handling the $(-\pi, \pi)$ periodicity of the angle in the bearing sensor model. The angle $-\pi$ and π represents the same angle, but when performing the simple difference in equation 4.9 of $(z(k) - h(\xi(k|k-1))$, if the two terms equal two opposite angles near π then the result is very big. This induces the filter to include in the estimate a lot of "new" information and results in big oscillations of the final estimate. An example of this situation experienced during the *Gazebo* simulation is shown in figure 4.5. The difference between these two angles can be correctly handled in the following way:

$$\Delta\theta = \begin{cases} \theta_z - \theta_h - 2\pi & \text{if } \theta_z - \theta_h > \pi \\ \theta_z - \theta_h + 2\pi & \text{if } \theta_z - \theta_h < -\pi \\ \theta_z - \theta_h & \text{otherwise} \end{cases} \tag{4.10}$$

After implementing these considerations when performing the difference, the correct distance between the angles is calculated and the result doesn't present any unexpected oscillation.

To perform the final estimate then the information matrices and vectors published from the other drones are read and the final estimate is calculated with:

$$Y(k|k) = Y(k|k-1) + I + \sum_j I_j(k) \tag{4.11}$$

$$y(k|k) = y(k|k-1) + i + \sum_j i_j(k) \tag{4.12}$$

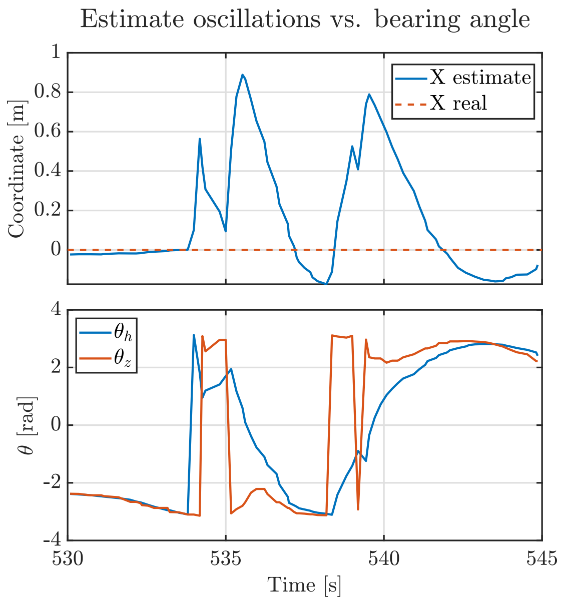


Figure 4.5: *The estimate is compared to the bearing angle provided by the measurement θ_z and the one predicted θ_h. At second 534 one angle jumps from $-\pi$ to π so the resulting difference without considering the periodicity is very high.*

$$\hat{\xi}(k) = Y(k|k)^{-1} \cdot y(k|k) \tag{4.13}$$

An advantage of the information form formulation can be seen also in equation 4.11 and 4.12, that are used to calculate the final estimate. The contribution of the other drones is taken into consideration by simply summing the information vector and matrix published by each drone. In this way a lot of other agents can be added to the estimation easily. Finally the estimate provided by the filter is now directly available to the flocking algorithm (since it runs on the same node) and also is published to the *estimation* topic to be recorded and analyzed later. The estimation will be used by the flocking algorithm as a target to follow.

4.4 Flocking Algorithm

An important aspect to consider when dealing with a swarm of drones is the control of their position. The flocking behaviour can be defined using the 3 Reynolds rules. These can be a starting point to define a controller for the position. The flocking algorithm implemented is taken from [9]. The aim of this control algorithm is to create a flocking behaviour, by controlling the inter-drone distance and matching their velocities. The secondary objective of the controller is moving the swarm to track the position of a defined target that, in this case, is the ArUco marker. The position of the marker is the estimation provided by the Kalman filter described in the previous section.

Each drone is modeled as a double integrator, where $\mathbf{q_i} = [x_i, y_i]^T$ and $\mathbf{p_i} = [\dot{x}_i, \dot{y}_i]^T$. The control input $\mathbf{u_i} = \dot{\mathbf{p}}_\mathbf{i}$ imposes the planar acceleration of the drone. Note that this algorithm manages only the horizontal position and velocities. The vertical dynamics are not considered and during the flight a constant altitude is imposed and is managed directly by the autopilot. PX4 allows to independently setup the level of control of each of the three axes independently. To achieve the horizontal acceleration control, the position and velocity controllers of the X and Y axes are disabled and they are kept enabled on the Z axis. This configuration is imposed by correctly generating the *TrajectorySetpoint* message. In particular in this situation the message fields are set as follows:

Parameter name	**X**	**Y**	**Z**
position	NaN	NaN	Z
velocity	NaN	NaN	NaN
acceleration	a_x	a_y	NaN

Table 4.1: Fields of the TrajectorySetpoint message to impose a position height and a horizontal acceleration setpoint.

The control input is composed of multiple terms:

$$\mathbf{u_i} = (\mathbf{u_{i,d}} + \mathbf{u_{i,d,int}}) + (\mathbf{u_{i,v}} + \mathbf{u_{i,v,int}}) + \mathbf{u_{i,t}} \tag{4.14}$$

The first term is relative to the inter-drone distances. The aim is to maintain a predefined distance between neighbouring drones, in particular a *communication radius* is defined and a drone considers as neighbours all the drones inside this circle. The term $\mathbf{u_{i,d}}$ is proportional to the inter-drone distance of only the neighbours drones, and also an additional function is included in order to make the attractive force weaker than the repulsive one. The term $\mathbf{u_{i,d,int}}$ is an integrator action on the distance between the drones. Its objective is to remove the steady state offset on the distance created by the former mentioned controller.

The second term is related to the velocity matching. Again this term is composed by a component proportional to the difference in velocities of neighbouring drones and an integral action on the error between velocities. The result is that the drones match their velocities that is a requirement to create a flocking behaviour. The integral action is used to reduce the oscillations and overshoots.

Finally the last term $\mathbf{u_{i,t}}$ includes the target following objective. The control input is proportional to the distance from the drone to the target position, and also it's proportional to the difference in velocities. This results in the drone following, in this case, the position of the ArUco marker and also matching its velocity.

This algorithm is executed by the *Estimator and Flocking node* that is instantiated for each drone. This means that each drone runs a separate instance of this same control algorithm. Each instance is configured to correctly label the drone is running on and the neighbouring drones. The algorithm is executed with a rate of 20 Hz, using a ROS 2 timer, so the timing is always constant and is not related to the update rate of the Kalman estimates. At each execution of the control algorithm the latest target position estimate is used, even if the estimate was not updated by the filter. This configuration allows to avoid a situation where the algorithm does not run because the estimate is not updated anymore when the marker is not in sight. Decoupling the control algorithm execution from the filter update frequency, provides a steady rate of control inputs that, in case the estimate is not updated, simply tracks the last known position of the marker. The resulting control input generated by the algorithm is published on the topic *acceleration_cmd* for each drone. This topic is read by the *Vehicle Handler node* that only forwards the message to PX4 only in the correct moment of the flight.

4.4.1 Parameters tuning

Each term used for the computation of the control input in equation 4.14 represent a different objective of the overall control algorithm. Each term is multiplied by a weight (not shown in the equation) in order to tune the performance and prefer some objectives respect to others. The weight values are hard-coded into the source code, so when multiple instances of the node are executed, each drone has the same weights in the control algorithm. Since the drones composing the swarm are all equal, then using the same tuning for all of them is not an issue. On the other hand using different kind of drones may required a different tuning for each one.

The aim of this phase is to obtain a stable control and also a good performance in terms of transient, overshoot and oscillations. A trial and error process is used, changing one parameter at a time and evaluating the effect on the overall performance. This process has been done in the simulation environment. The weights of the control algorithm are:

	Swarming
K_P	Drones distance proportional term
K_I	Drones distance integrator term
K_D	Drones velocity matching term
	Target following
C_1	Distance to target term
C_2	Target velocity matching term
D_I	Integral term on velocity matching

Table 4.2: *The list of weights used in the flocking algorithm.*

Increasing the value of a weight has the effect of accentuating the importance of that term in the overall calculation of the control input. For example, increasing the factor K_P increases the strength of the distance control between the drones, imposing higher accelerations in order to reach the desired distance. Firstly the parameters relative to the swarming terms were tuned until a stable swarm was created in simulation by 3 drones. An important aspect considered was to avoid big overshoots in the inter-drone distance, to avoid the drones to be too close that can lead to a collision. This was accomplished working mostly on the parameters K_I and K_D.

After the swarm part of the controller was properly tuned, then the coefficients that regulate the target tracking objective were considered. The parameter with the most effect is C_1. This term attracts each drone to the marker position, so increasing this gain will result in a stronger attraction. This attraction force works against the inter-drone distance controller that tries to separate the drones: increasing the attraction to the target position also affects the distance between the drones during the transient, promoting undershoots that bring the drones too close. This effect is countered at steady state by the integrator action $\mathbf{u_{i,d,int}}$.

After the tuning phase in the *Gazebo* simulation with three drones, the resulting parameters are:

Parameter	**Value**
K_P	0.5
K_I	0.01
K_D	0.6
C_1	0.5
C_2	5
D_I	0.005

Table 4.3: *Parameters values after the tuning, with target spacing between drones of 1 meter.*

These parameters were used as a staring point when testing the algorithm on the real drones. Only 2 real drones were used and the tuning of the parameters didn't need any major modification. The results of the tests are presented in chapter 5.

Chapter 5

Simulation and Testing

The development of the algorithm described in chapter 4 was aided with the use of a simulation environment that was essential to validate the correct behaviour of the programs before trying to test them on the real drones. In particular, the simulation environment is setup to emulate as close as possible the real setup, so that the implemented programs can be executed both on the real drones and in the simulation without needing any changes. The PX4 software is executed as a software in the loop, running on the computer along with the Gazebo simulator. This chapter presents the simulation environment and the performances of the flocking algorithm and the diffused Kalman estimator, both in simulation and when deployed on the real drones.

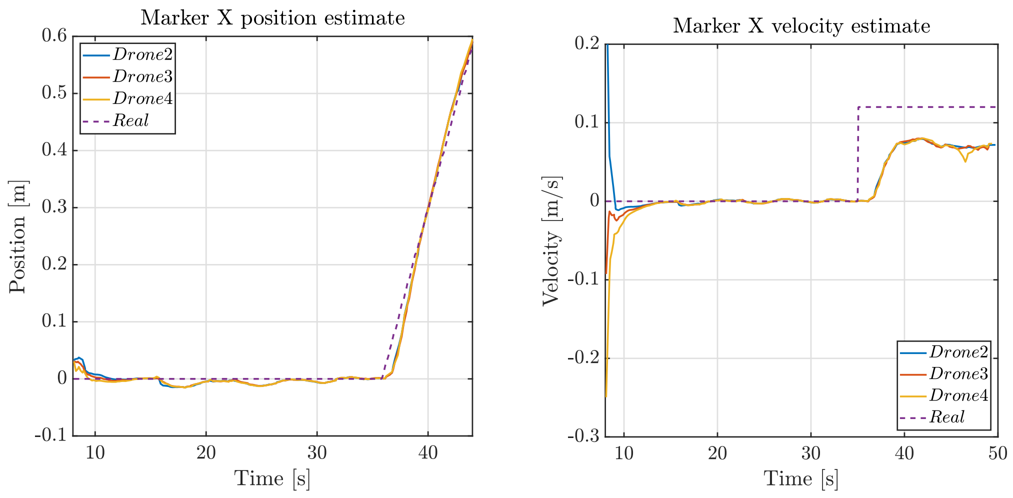
5.1 Simulation

The PX4 software supports being simulated as Software In The Loop (SITL) and offers a wide variety of supported simulators like Gazebo, jMAVsim, JSBSim and others. The chosen simulator is *Gazebo-Classic*, because it is well integrates with ROS 2 and also is well supported by PX4 with a lot of features and documentation that are not available with other simulators. A good integration with ROS 2 is mandatory since it is used as the communication framework with the real drones, and as a way to test the correct interface between ROS 2 nodes and the autopilot.

The build system used by PX4 offers multiple compilation targets, where most of them supports different kinds of flight controllers and others are intended to compile the source code to be executed on the computer and integrate it with the chosen simulator. This feature allows to execute the same source code that will be executed on the real flight controller, offering the same features and the same behaviours. In particular also the same set of modules chosen for the flight controller are included in the SITL version to further resemble the real scenario. In order to enable the communication with ROS 2, the *XRCE-DDS* bridge has to be used: the client side is built into the PX4 executable, so it is already running with the simulation, then the *XRCE-Agent* side is executed on the computer to allow the interface with ROS 2. This setup is the same used in the real drones, with the only difference being running everything on the same computer. Note that the same topics and messages types are exposed in the ROS 2 environment, allowing to validate the correct interaction between the developed nodes and PX4. A lot of work has been done in correctly interacting with the autopilot to perform the desired actions, like changing modes, experimenting with features etc. and doing it on the simulation is much easier and safer than directly testing with the real drones.

5.1.1 Simulation Results

The flocking algorithm and the diffused estimation have been tested with only 3 simulated drones due to practical reasons of speed of the simulation. The drones are spawned in 3 different positions on the ground and then the nodes relative to the vehicle handler, estimator and camera are executed for each drone. After the takeoff, each drone keeps a steady position and when the marker is in the field of view of the cameras the estimation starts. In figure 5.1 the estimation of each instance of the filter is shown, only for X axis, which was the only one used to test the movement of the marker. Although the position estimate is very close to the real position, the velocity estimate is not. A crucial component for the velocity estimate is the accurate measurement of the time elapsed between two measurements and that is probably the cause of the wrong velocity calculation. Moreover, the simulation does not run in real time, but the experienced speed was around 60% of the real time speed so if the messages use the system timestamp, the calculated elapsed time is more that it should be, resulting in a lower velocity estimate.



(a) *X position* (b) *X velocity*

Figure 5.1: *Estimation result for the X position and velocity in the Gazebo simulation environment*

In an effort to improve the accuracy of the velocity calculation, during the simulation phase, each node was started with the parameter *use_sim_time* set to *true*. This parameter should make the nodes use the simulation time, so even if the simulation is slower than real time, the time differences between samples should be coherent with the simulation. Improvements were visible after this step in the accuracy of the velocity, but the final result was still not accurate (visible in figure 5.1).

After letting the estimate of the marker stabilize from the initial transient, a message is sent manually on the topic */start_flocking* that makes each drone start the flocking algorithm to control the position.

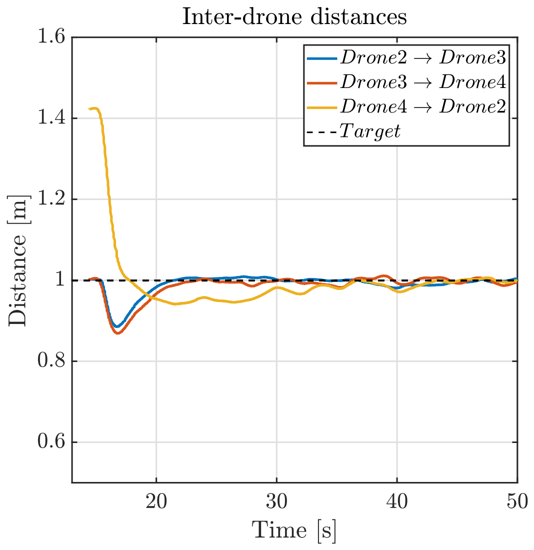


Figure 5.2: *Distance between the drones controlled by the flocking algorithm during the simulation. The target distance between each drone is 1 meter.*

In order to make the flocking algorithm work, a phase of tuning of the control parameters was necessary, otherwise random parameters cause the behaviour to be completely unstable. This process is described in section 4.4.1. In figure 5.2 is shown the actual measured distance between the drones during the simulation and is visible the correct behaviour where each drone distance from each other by 1 meter. Also note that the after the initial transient, the steady state error between the target distance and the actual distance tends to zero thanks to the integrator term, shown in equation 4.14. The velocity matching works correctly as represented in figure 5.3 where is visible that the velocity of the drones follows the estimated velocity of the marker.

5.1.2 Gazebo Environment

By default the when simulating PX4, the instance of PX4 in SITL is attached to a model of a drone in the Gazebo simulation, so the results are shown with the behaviour of the simulated drone. The drone model used is an *Iris* drone (shown in figure 5.4), that was customized to add a down facing camera that streams the images on a ROS 2 topic, in order to resemble the real drones. Another important aspect that has been simulated is the use of an external source for the positioning of the drone: by default the simulation

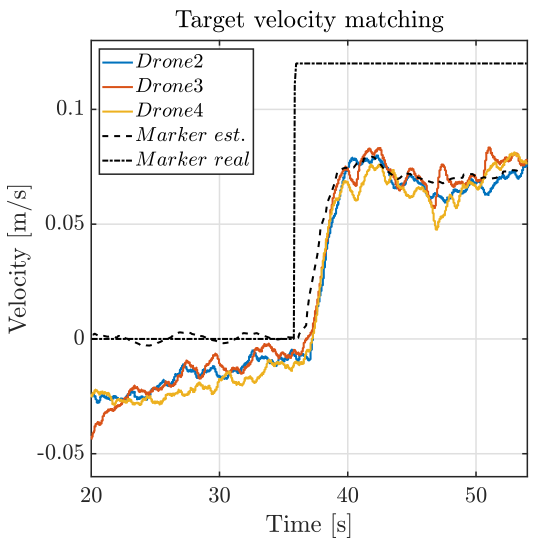


Figure 5.3: *Velocity in X axis of the drones matching the velocity estimate. Note that the velocity estimate is not accurate as explained in section 5.1.1.*

uses a fake GPS sensor, but it was removed and the PX4 autopilot was setup with the same configuration as the real drone in order to correctly support the external position estimates. The measurements are taken directly from Gazebo and then are sent on the *fmu/in/vehicle_visual_odometry* topic like the Vicon data. In order to customize the simulation and to add the mentioned modification, two plugins made for the integration with ROS 2 are added. The real position of the simulated drone is published on a ROS 2 topic by the plugin *libgazebo_ros_p3d.so* that was attached to the base link of the drone. This plugin publishes on a specified topic the position of the link it is attached to (i.e. the drone's position ground-truth). Then a node reads this information, performs the transformation from the Gazebo reference frame FLU to the PX4 reference frame FRD and finally sends it to the autopilot, emulating the same role of the Vicon. Implementing this feature was also very useful in order to correctly setup the autopilot to rely on an external measurement for the position, since the choice of the correct parameters wasn't trivial.

Adding a down-facing camera is implemented again by adding a gazebo camera sensor and a plugin to the model description of the drone. The camera sensor is attached to the base link of the drone, with the proper rotation and translation to imitate the position in the real case and also the focal length and resolution are set in order to match the OpenMV camera module. The plugin *libgazebo_ros_camera.so* is added to the camera sensor in order to stream the image and other camera information directly on a ROS 2 topic. The image is available on the topic *camera/image_raw*, and the information including the camera intrinsic parameters are available on the topic */camera/camera_info*.

Another important element of the simulation environment is the ArUco marker model. The presence of the both the marker and simulated cameras in Gazebo allows to test

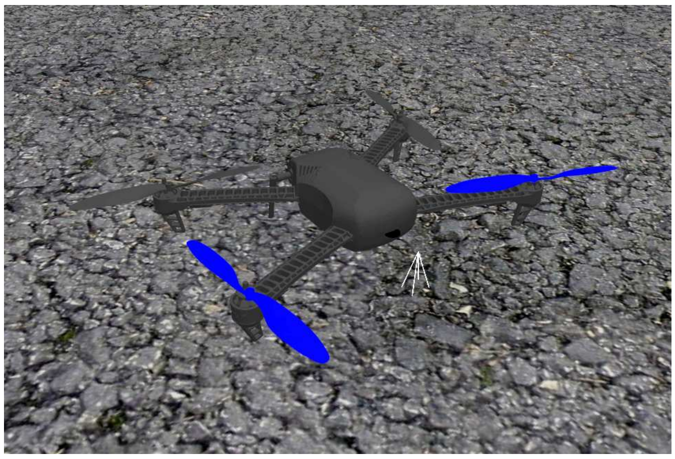


Figure 5.4: *Model of the Iris drone used in the Gazebo simulation. The white lines in the front represent the field of view of the camera.*

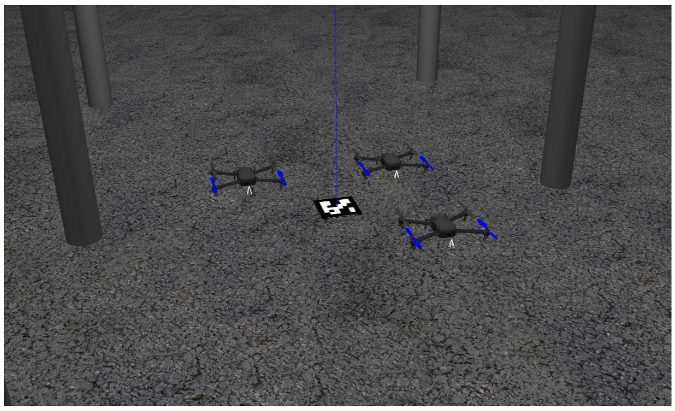


Figure 5.5: *Gazebo world used to test the ArUco position estimation and the flocking algorithm. The vertical pillars represents a mock-up of the real cage.*

the estimation algorithm in a realistic way, testing also the nodes responsible to identify and calculate the position of the marker. The model is implemented as a simple box, with the texture showing an image of a marker. By default it is placed in the origin of the simulation reference frame and is visible in figure 5.5. Furthermore the plugin *libgazebo_ros_planar_move.so* is attached to the marker model in order to both impose linear velocities to it and also retrieve the real position. The velocities are set by publishing a message on the topic */aruco_marker/cmd_vel*, and the position can be retrieved by reading the topic */aruco_marker/odom*. This plugin was very useful when testing the estimation performance in case of marker movements, with the possibility of comparing the result of the calculations with the real position.

Finally the last important aspect of the simulation is the possibility of simulating multiple drones at the same time, that is an essential feature for testing the flocking algorithm and the diffused estimation. Using a script that is available directly from the PX4 repository, it is possible to simulate multiple drones in the same simulation. There are two main steps: the first is to start multiple instances of PX4 in SITL mode, one for each simulated drone, then the required number of drone models are spawned into the Gazebo simulation. Some customization was needed in order to correctly setup the name-spaces of each drone and configuring the interface for ROS 2. This was performed with the templating tool *Jinja* that is already used for this task in the PX4 repository. In particular the drone *Gazebo* model file (e.g. *iris.sdf*) presents parameters inside that are substituted by *Jinja* before spawning the model into the simulation. For example the ROS 2 name-space was set as a parameter and substituted accordingly for each spawned drone, creating a different *sdf* file for each one. This customization was necessary because this script uses the same approach to configure *MAVlink* communication, but it is not still supporting the new XRCE-ROS 2 bridge.

5.2 Real implementation

After validating the flocking algorithm and the Kalman filter in the simulation environment the next step was to execute this nodes using the real drones. The tests were performed using only 2 drones in order to be able to validate the correct exchange of information between the nodes. An important difference from the simulation environment is the presence of a real network between the drones and the ground station, so there may be present random delays in sending or receiving the messages.

The intended way of running the ROS 2 nodes would be to execute them on the *NanoPi* companion computer of each drone as ROS 2 has been deployed on them, but this was not accomplished for two main reasons. Firstly running the nodes from the ground station is much easier if recompilation is needed when changes are made, while on the *NanoPi* the compilation is very slow and the only way to interact with it is using a *ssh* console. The second and most important reason is that the *NanoPi* board doesn't have the necessary computational power in order to execute the ROS 2 framework. The nodes can be executed successfully but the initialization uses so many resources that also other processes stop due to lack of computational power. In particular, it was encountered that the *XRCE-DDS-Agent* stops executing for about 30 seconds when the ROS 2 framework is started. This results in a total loss of communication between PX4 and the offboard control node, so running the nodes onto the *NanoPi* was not feasible. In order to avoid this situation the nodes are simply executed on the ground station and they are able to work without issues.

The only remaining issue is that the execution of the node that handles the *OpenMV* camera is required to run on the *NanoPi* board. The camera module is in fact directly interfaced with the companion computer using a serial connection so a node that reads the serial port data and transmits it onto a ROS 2 topic is needed. This node's responsibility is just to read the serial port where the data containing the position on the camera sensor of the ArUco marker is sent and send it to the topic *camera/marker_pos*. Then other nodes will read the messages and calculate the 3D position of the marker. This node, even if very simple, faced the same issue of low computational power offered by the *NanoPi*: when started all the processes stopped up to 30 seconds resulting in a loss of communication between the offboard nodes and PX4. In order to resolve this problem and test the performance of the Kalman filter, fake measurements of the marker position can be generated and then corrupted with white noise. In this way the camera isn't used at all but the other nodes can be tested. Another approach can be to test only the Kalman filter while the drones are not flying but they are still positioned by the Vicon system so, from the point of view of the filter, the situation is the same as flight. Moreover, the drones are positioned in an elevated position with the help of cardboard boxes so the marker can be placed under them and be in the field of view of the cameras. When starting the nodes on the *NanoPi* the situation was always of temporary stop of all the processes but since the drones were not flying there was no issue related to loss of communication so, in this way the estimation can be fully tested with real data taken from the cameras.

The flocking algorithm nodes were again executed on the ground station but they were much easier to test since they don't rely on the camera module or on any node that needs to be run on the *NanoPi*.

5.2.1 Results

The tests performed in the real scenario were very incremental in order to avoid crashes and identify the issues as soon as possible. The first step is to test the correct functioning of a simple offboard control, that included takeoff and landing after a small time of hover in position. The *Vehicle Handler* node was used with a little modification in order to skip the *MISSION* state.

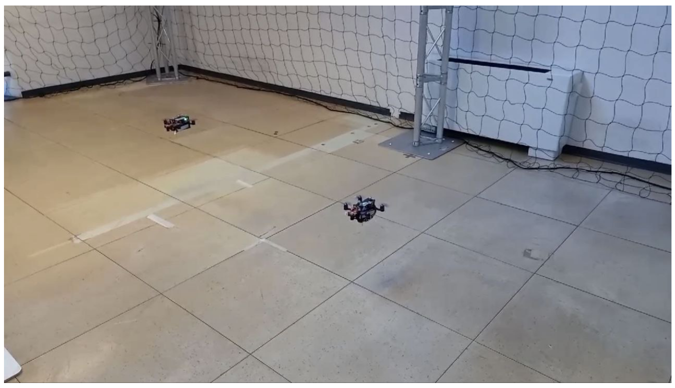


Figure 5.6: *Drones in flight during the test of the flocking algorithm.*

The test used 2 drones to prove that the multiple instances of the nodes didn't interfere and they communicate correctly with each other. The drones correctly responded to the offboard commands provided by the nodes, behaving exactly in the same way as in the simulation. The workarounds used for the landing routine (avoiding the built-in routine that had a bug) worked too in the same way as the simulation.

The second step is to test the distributed estimation and it is done as described in the previous section, by avoiding the flight and having the drones positioned on cardboard boxes. In figure 5.8 the estimation data from this test is shown. In particular, the two filters correctly exchange the information matrix and vector and the two estimates are very close. Until the time instant at 64 seconds, the marker position is not changed, then it is moved by hand and the estimate correctly tracks the changes. Sometimes the marker is out of the field of view of one drone due to its movement and this can be seen in the figure where some of the data points are missing. A comparison between the raw measurements of the cameras and the final estimate is shown in figure 5.7 where it is visible that the two measurements sources presents some offset between each other (8cm in the worst case) and the resulting estimate is a value between the two.

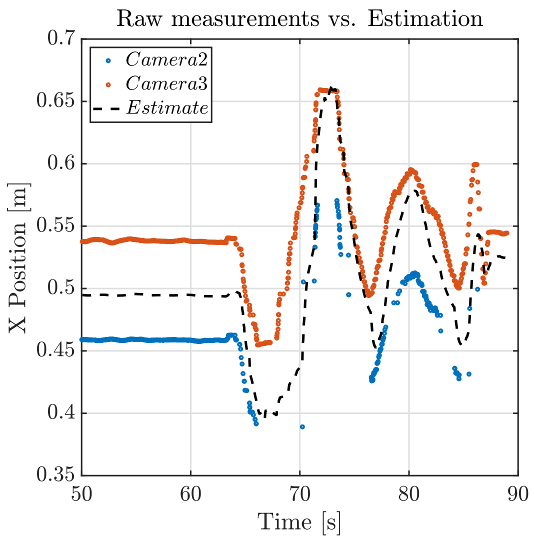


Figure 5.7: *Comparison between the raw measurements of the X position given by the cameras and the resulting estimate of the Kalman filter.*

During the test, drone's the movements, instead, didn't affect at all the estimation, validating the correct transformations described in section 4.2.1. The only encountered issue was that, depending on the position of the marker in the field of view of the camera, the calculated pose of the marker was not very precise. This effect was present mostly at the edges of the field of view and is probably due to the lens distortion of the camera as this issue was not present in the gazebo environment where the camera was ideal without distortion. .

Finally, the flocking algorithm was tested. In order to decouple this algorithm from the

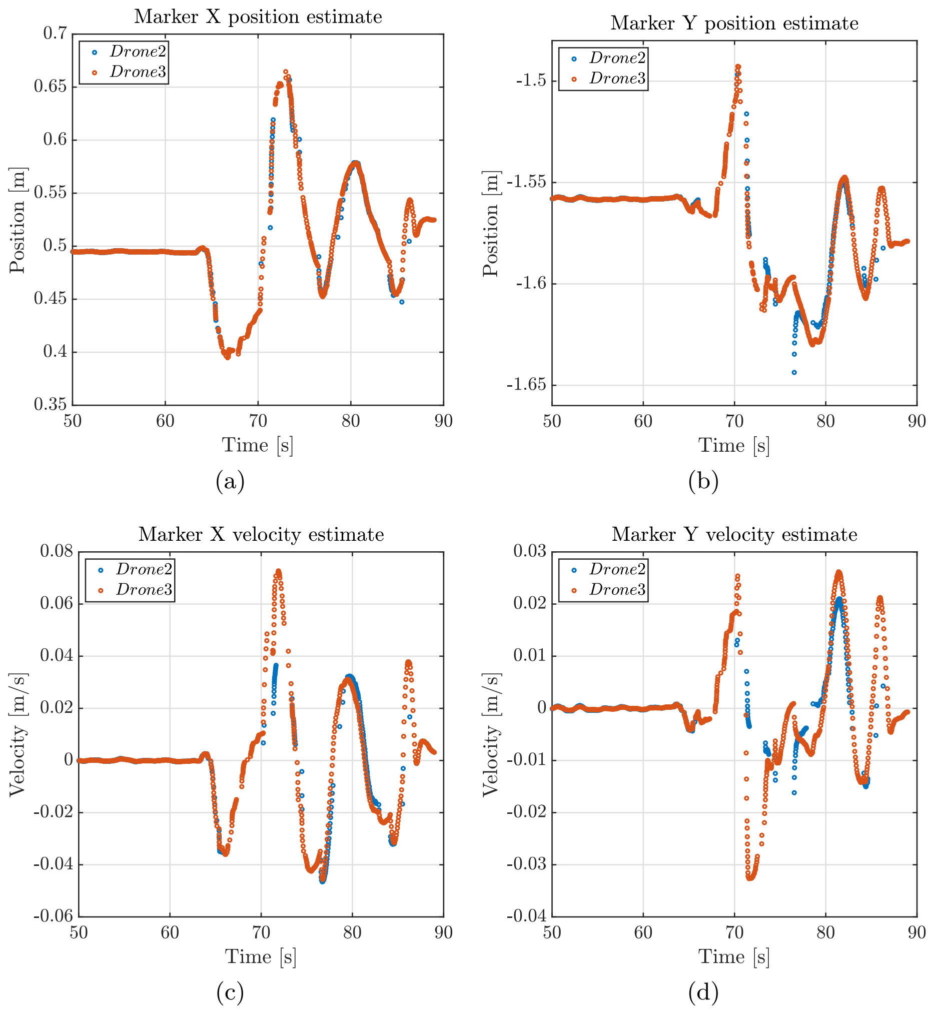


Figure 5.8: *Estimation results of the ArUco marker position and velocity using the OpenMV camera.*

marker position estimate, that could lead to unexpected behaviours in case of not accurate estimations, the target position used in the algorithm was set to $X = 0$ and $Y = 0$ and the velocities to zero. The expected behaviour is that the two drones should keep the target distance between each other and also both move close to the origin position (position of the "simulated" marker).

In figure 5.9 is shown the resulting distance between the two drones during the real flight. The weights of the control algorithm were not changed from the values set during the simulation and were also used for the real test. The behaviour of the control was already correct, so in order to achieve a better performance not much tuning is needed. The distance between the drones is correctly controlled, but more oscillations can be seen due to a lack of fine tuning of the control algorithm. Another important consideration is that when the two drones were flying at the same time in the cage, their precision of holding a certain position was reduced with respect to when only one drone was flying, possibly due to the turbulence produced by the two of them. The test was in fact performed flying at low altitude (around 1 meter above the ground) in order to reduce the risk of damage in case of a crash. In all likelihood, increasing the altitude probably can reduce the turbulence effects.

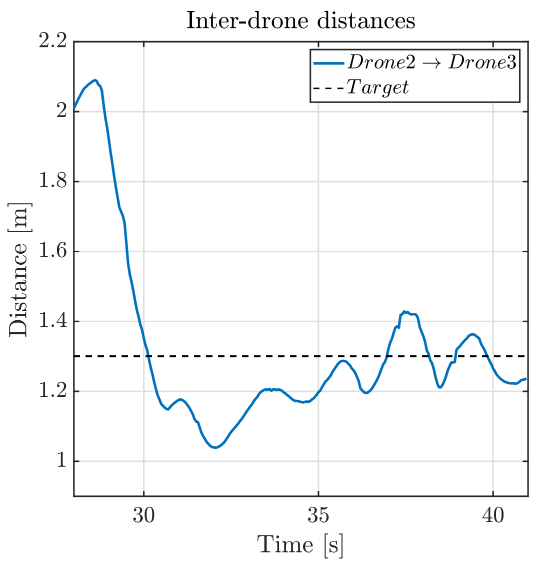


Figure 5.9: *Inter-drone distance during the test of the algorithm with the two real drones.*

Conclusions

The main aim of this thesis work was to obtain a working setup of the drones in the Robotics Laboratory of the Links Foundation. The final result is the complete setup of two drones, that were then used to test the flocking and estimation task. After this phase, two algorithms, a flocking controller and a diffused estimation with a Kalman filter, were developed in ROS 2 following the methods proposed by [9] with the aid of a simulation environment, configured in *Gazebo* to closely resemble the real case. The implementation of the estimation and flocking task is aimed at proving the proper operation of the complete setup. Finally testing of the flight performance and the validation of the developed algorithms was performed with the real drones in the laboratory.

The first phase of the setup of the drones required the most amount of work due to the time consuming troubleshooting, particularly the communication issues arising from the WiFi interfering with the radio controller and the correct reception the Vicon measurements by the drones. The final phase of testing of the developed algorithms was very straight forward without any issues relative to the drones. The flight performance of the drones is very satisfactory, as the flight is stable and the behaviour is exactly the same as the one experienced in the simulation. This allowed to focus the efforts only on the debugging of the algorithms, resulting in a quick and simple final phase of testing. This was also due to the extensive work performed in the simulation environment that was very useful to correctly validate the software before trying it onto the real platforms. The results of the tests validates the work presented in [9] in a real environment, without idealities of the simulation environment. Some workarounds had to be implemented due to limitations of the drone's hardware, leading to the impossibility of executing the nodes on the companion computers. As a result, this implementation partially excluded the interaction of network delays in the communication.

Future developements

Regarding the physical setup of the drones, the majority of difficulties encountered were caused by the companion computer used. Mainly, the computational power is very limited and also the 32-bit architecture is not suitable to execute useful software like ROS 2. The cross compilation was successful, but the computational power is not enough to run the nodes. Further research could find an alternative board to use as the companion computer that is more powerful and that can be correctly integrated with PX4. With the ability of running nodes on each drone's companion computer, the communication between nodes can also be tested against possible real network delays, instead of executing everything on the ground station.

During this project the *Modal AI VOXL* board was considered as a candidate to be used as companion computer. Some exploratory work was done with this board trying to understand if it can be a suitable replacement for the *NanoPi*, since it offers much more computational power. It was discarded since the NanoPi offered satisfactory performance to the scope of this project and because it required excessive work and research to correctly integrate it with the current configuration of the drones, both hardware and software wise. Further work aimed at integrating this board with the existing drones and PX4 can offer new possibility given the computational capability of the *Modal AI* boards.

Finally, further work can be done to complete the setup of all the drones present in the laboratory. Four drones are in fact available, of which only two were completely setup due to the availability of some components. Completing the setup of all the drones would be a good opportunity to implement algorithms that can leverage the presence of a more complex swarm composed by four drones.

Appendix A

ROS 2 Topics Description

Topic name	**Type**	**Description**
	PX4 topics	
fmu/in/vehicle_command	VehicleCommand	Used to send PX4 commands from the offboard node to the autopilot
fmu/in/offboard_control_mode	OffboardControlMode	Set the type of offboard control wanted, like acceleration, velocity, position ...
fmu/in/trajectory_setpoiny	TrajectorySetpoint	Impose to the autopilot the desired setpoints of different types, depending on the control mode
fmu/in/vehicle_visual_odometry	VehicleOdometry	Odometry sent from an external source to be fused
fmu/out/vehice_status	VehicleStatus	Status flags of PX4
fmu/out/vehicle_odometry	VehicleOdometry	Odometry of the drones including position velocity, attitude, angular velocity
	Estimation Node topics	
sensor_measure	Float64MultiArray	Measurement of the ArUco marker position relative to the drone in bearing sensor form
information	Float64MultiArray	Information matrix and array of the Kalman filter of each drone

acceleration_cmd	Float64MultiArray	Acceleration control input calculated from the flocking algorithm
flocking_info	Float64MultiArray	Detailed values of the flocking algorithm for plot and logging purposes
/start_flocking	Bool	Flag that enables the flocking algorithm after the takeoff
Camera topics		
camera/image_raw	Image	Uncompressed image streamed from the Gazebo camera
camera/marker_pos	Float64MultiArray	ArUco marker ID and coordinates in the image plane after recognition by CV algorithm
camera/camera_info	CameraInfo	Camera intrinsic parameters

Table A.1: *All the topics used in the algorithm described in chapter 4. Some PX4 topics that are not used are not included in this list. The topics that do not start with a "/" will have prefixed the namespace of the drone.*

Bibliography

[1] G. Bressan, D. Invernizzi, S. Panza, and M. Lovera. Attitude control of multirotor uavs: cascade p/pid vs pi-like architecture. 2019.

[2] Simon Daniel. *Optimal State Estimation: Kalman, H-infinity, and Nonlinear Approaches*. John Wiley & Sons, 2006.

[3] Frederick E. Daum. *Extended Kalman Filters*, pages 411–413. Springer London, London, 2015.

[4] Friendly Elec. Nanopi neo air wiki. `https://wiki.friendlyelec.com/wiki/index.php/NanoPi_NEO_Air`.

[5] Drone Foundation. Px4 repository. `https://github.com/PX4/PX4-Autopilot`.

[6] Drone Foundation. Px4-ros 2 user guide. `https://docs.px4.io/main/en/ros/ros2_comm.html`, 2023.

[7] Sang-Hoon Kim. Chapter 4 - modeling of alternating current motors and reference frame theory. In Sang-Hoon Kim, editor, *Electric Motor Control*, pages 153–202. Elsevier, 2017.

[8] Fausto Francesco Lizzio, Elisa Capello, and Giorgio Guglieri. Implementation and performance evaluation of a consensus protocol for multi-uav formation with communication delay. 2022.

[9] Fausto Francesco Lizzio, Stefano Primatesta, Haoyu Guo, and Giorgio Guglieri. Design and sitl performance of an online distributed target estimation for uav swarm. 2022.

[10] Steven Macenski, Tully Foote, Brian Gerkey, Chris Lalancette, and William Woodall. Robot operating system 2: Design, architecture, and uses in the wild. *Science Robotics*, 7(66):eabm6074, 2022.

[11] Mahendra Mallick. A note on bearing measurement model. 05 2018.

[12] Davide Morazzo and Ronald Cristian Dutu. swarm_uav_config. `https://github.com/links-cosero/swarm_uav_config`, 2023.

[13] Arthur G.O. Mutambara. *Decentralized Estimation and Control for Multisensor Systems*. CRC Press, 1998.

[14] Kwang-Kyo Oh, Myoung-Chul Park, and Hyo-Sung Ahn. A survey of multi-agent formation control. *Automatica*, 53, Mar 2015.

[15] Reza Olfati-Saber. A unified analytical look at reynolds flocking rules. 2004.

[16] Reza Olfati-Saber. Flocking for multi-agent dynamic systems: Algorithms and theory. *IEEE Transactions on automatic control*, 51, Mar 2006.

[17] OpenMV. Openmv cam micro-python documentation. `https://docs.openmv.io/`.

[18] Open Robotics. Ros 2 - rosbag2. `https://github.com/ros2/rosbag2`.

[19] Open Robotics. Ros 2 - understanding topics. `https://docs.ros.org/en/humble/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Topics/Understanding-ROS2-Topics.html#ros2-topic-hz`.

[20] František ŠolcRadek Baránek. Attitude control of multicopter. 2012.