# POLITECNICO DI TORINO

**Master's Degree in Mechatronics Engineering**

**Master's Degree Thesis**

# Position Stabilization of Drones in Indoor Environments: Evaluation of Sensors and Techniques and Integration into the PX4 Flight Control Software.

**Supervisors**

**Prof. Alessandro RIZZO**

**Candidate**

**Dario CATANIA**

**OCTOBER 2023**

**Abstract**

In the last years, unmanned vehicles are gaining a lot of interests all over the world, due to their capability of adapting functionalities to always different and various applications. For example, just think that, in some countries, drones deliver food directly at your home. So, the application of drones is mainly intended to optimize already existing processes, to reduce the risk of injuries or even save lives (like search and rescue applications), covering a wide range of scenarios, such as smart agriculture, emergency situations and safety inspections. This master's thesis investigates the critical aspects of indoor drones, focusing on the development and enhancement of position stabilization techniques, sensor evaluation, and the integration of these advancements into the PX4 flight control software. As most of the standard solutions implemented in drone navigation rely on GNSS positioning, indoor navigation represents one of the main challenges to be managed. The UAV uses a camera to give the pilot an overview of the drones surroundings, and a set of IMUs and Barometers to give the drone a reliable reference system. This thesis was born in order to improve the already existing positioning system for simplifying the inspection of the UAV in endangered situations. A way to do that is the use of range sensors to regulate the distance of the drone from a wall or from certain objects.

# Table of Contents

**List of Figures** IV

**Acronyms** VII

**1 Introduction** 1
- 1.1 Objective of the Thesis . . . 1
- 1.2 Organization of the thesis . . . 1

**2 PX4 Autopilot, Architecture and Control System** 3
- 2.1 PX4 System Architecture . . . 3
  - 2.1.1 Flight Stack . . . 4
  - 2.1.2 Middleware . . . 5
  - 2.1.3 Runtime Environment . . . 6
  - 2.1.4 uORB . . . 7
  - 2.1.5 MAVLink . . . 13
  - 2.1.6 QGroundControl . . . 15
- 2.2 PX4 Control Architecture . . . 18
  - 2.2.1 Position Control . . . 19
  - 2.2.2 Velocity Control . . . 19
  - 2.2.3 Attitude Control . . . 20
  - 2.2.4 Angular Rate Control . . . 21

**3 Configuration, Model of the Drone and Sensors** 23
- 3.1 Gazebo . . . 27
- 3.2 SDF File . . . 31
- 3.3 Gazebo and PX4 connection . . . 34

**4 Custom Control Algorithm, "Horizontal-Lock"** 37
- 4.1 Definition of the Orientation Angle . . . 39
- 4.2 Angle_correction Topic . . . 45
- 4.3 Creation of the Flight Task Modules . . . 45

4.4 The Altitude Algorithm . . . . . . . . . . . . . . . . . . . . . . . . . . 49
4.5 The Angle Algorithm . . . . . . . . . . . . . . . . . . . . . . . . . . . . 56
4.6 The Position Algorithm . . . . . . . . . . . . . . . . . . . . . . . . . . 58

**5 Testing: Results and Troubleshooting** 61
5.1 Problems in the Position Mode . . . . . . . . . . . . . . . . . . . . . . 61
5.2 Problems in the Angle Measurement . . . . . . . . . . . . . . . . . . . 62
5.3 Measurements and Results . . . . . . . . . . . . . . . . . . . . . . . . 64

**6 Conclusions** 68

**A Codes** 70
A.1 AngleCorrector.cpp . . . . . . . . . . . . . . . . . . . . . . . . . . . . 70
A.2 AngleCorrector.hpp . . . . . . . . . . . . . . . . . . . . . . . . . . . . 74
A.3 FlightTaskDistanceAltitude.cpp . . . . . . . . . . . . . . . . . . . . . . 76
A.4 FlightTaskDistanceAltitude.hpp . . . . . . . . . . . . . . . . . . . . . . 85
A.5 FlightTaskDistancePosition.cpp . . . . . . . . . . . . . . . . . . . . . . 89
A.6 FlightTaskDistancePosition.hpp . . . . . . . . . . . . . . . . . . . . . . 92

**Bibliography** 94

# List of Figures

2.1 PX4 system that includes both a flight controller and a companion computer . . . 4
2.2 Building blocks of the flight stack . . . 5
2.3 uORB Publication/Subscription Graph. For full graph see the reference 8
2.4 Over-the-wire format for a MAVLink 2 packet . . . 14
2.5 Example of a paramenters list on QGC . . . 16
2.6 Output of the QGC console of the command ls/obj . . . 17
2.7 Output of the QGC console of the command uorb status . . . 17
2.8 Multicopter Control Architecture . . . 18
2.9 Position controller block scheme . . . 19
2.10 Velocity controller block scheme . . . 20
2.11 Attitude controller block scheme . . . 21
2.12 Angular rate controller block scheme . . . 22

3.1 STM32h7x board . . . 24
3.2 ST-Link . . . 24
3.3 Block scheme of a HITL simulation environment . . . 26
3.4 Model of the iris_hitl on Gazebo . . . 28
3.5 Gazebo empty world . . . 29
3.6 Gazebo world modified for simulation purposes . . . 31
3.7 Model of the iris_hitl after the integration of the 2 distance sensors 34
3.8 Chart of the communication lines in the Gazebo SITL simulation . 34

4.1 Handwritten scheme to understand a possible configuration drone-wall 40
4.2 A possible sensors-wall configuration used to calculate the generic formula for the angle of inclination . . . 42
4.3 From above, physical scheme of the sensors placed on top of the drone 43
4.4 Flow chart of the angle correction algorithm . . . 44
4.5 Operating scheme of the altitude algorithm . . . 50
4.6 Where the angle algorithm is added in the altitude operating scheme 56
4.7 Operating scheme of the angle algorithm . . . 57

5.1 Manual control inputs for X/Pitch (in green) and Y/Roll (in red) over time . . . 63
5.2 Diagrams of the local position of the drone in the X and Y axis over time . . . 63
5.3 Diagram of the yaw_setpoint over time for a wrong algorithm . . . 64
5.4 Yaw input from the joystick over time . . . 65
5.5 Yaw_setpoint over time, in red the drone is flying in manual, in yellow in altitude . . . 65
5.6 Yaw angle over time. This refers to current heading of he drone and it is due yaw inputs from joystick . . . 66
5.7 Yaw angle over time . . . 66
5.8 Yaw_setpoint over time . . . 67

# Acronyms

**GPS**
: Global Positioning System

**HITL**
: Hardware In The Loop

**HW**
: Hardware

**IMU**
: Inertial Measurement Unit

**INT**
: Integer

**MAVLink**
: Micro Air Vehicle Link

**NAN**
: Not A Number

**PID**
: Proportional Integral Derivative

**QGC**
: QGroundControl

**RC**
: Radio Control

**SDF**

(Simulation Description Format

**SITL**

Software In The Loop

**SW**

Software

**UAV**

Unmanned Aerial Vehicle

**uORB**

Micro Object Request Broker

# Chapter 1

# Introduction

## 1.1 Objective of the Thesis

In this thesis, I will evaluate the effectiveness of a position stabilization technique for drones in indoor environments, such as industrial buildings or covered structures. Lidar sensors will be mostly used, and their performance in terms of accuracy and stability will be assessed. These sensors will be analysed and modified to suits the aim of the control system. Another crucial aspect of this thesis will be the integration of this position stabilization technique into the PX4 flight control software. I will examine how the flight control system can utilize signals from the sensors to stabilize the drone's position and execute inspection missions safely and efficiently. Additionally, the performance of the flight control system will be evaluated concerning the stability of the drone's position.

## 1.2 Organization of the Thesis

- **Chapter 1 - PX4 Autopilot, Architecture and Control System**: in this chapter will be held a description of the PX4 environment, including the architecture of the hardware and of the Control system.
- **Chapter 2 - Configuration, Model of the Drone and Sensors**: in this chapter will be held a description of the simulation environment and on how to customize it.
- **Chapter 3 - Custom Control Algorithm, "Horizontal-Lock"**: in this chapter will be held an analysis of the parts that compose the custom control algorithm.
- **Chapter 4 - Testing: Results and Troubleshooting**: in this chapter

will be presented all the issues encountered during the development and the obtained results.

- **Chapter 5 - Conclusions.**

# Chapter 2

# PX4 Autopilot, Architecture and Control System

PX4 is a platform-independent autopilot software, or firmware, that can drive drones or unmanned vehicles. It can be programmed on some hardware (such as Pixhawk), and together with the ground control station to form a completely independent autopilot system. It is Developed by world-class developers from industry and academia and supported by an active world-wide community.

One of the key strengths of PX4 is its hardware extensibility. This means that it can be seamlessly integrated with a wide range of devices, including cameras and sensors, to enable the development of diverse functionalities. By interacting with the firmware, these integrated devices can enhance the capabilities of the autopilot system and provide valuable data for advanced features and applications.

PX4 also offers different types of application programming interfaces (APIs) to facilitate interaction with the system, like ROS (Robot Operating System) or MAVSDK (MAVLink Software Development Kit). The one used in this work was MAVSDK, a collection of libraries available for various programming languages to interface with MAVLink systems. [1]

## 2.1 PX4 System Architecture

The PX4 organization of hardware (HW) and software (SW) is typically divided into two high-level system configurations. The first configuration consists of a standalone flight controller, while the second configuration includes both a flight controller and a companion computer. The project at hand utilizes the latter

configuration, which offers increased capabilities and flexibility.

At its core, PX4 is composed of two main layers: the flight stack and the middleware. These layers provide distinct functionalities within the overall system architecture The flight stack serves as the estimation and flight control layer of PX4. The middleware is a general robotics layer of PX4 and it focuses on hardware integration and internal/external communications.

By organizing PX4 into distinct layers, the system achieves modularity, extensibility, and reusability. This architecture allows for customization and adaptation to specific requirements, making PX4 a versatile platform for various unmanned systems applications. [2]

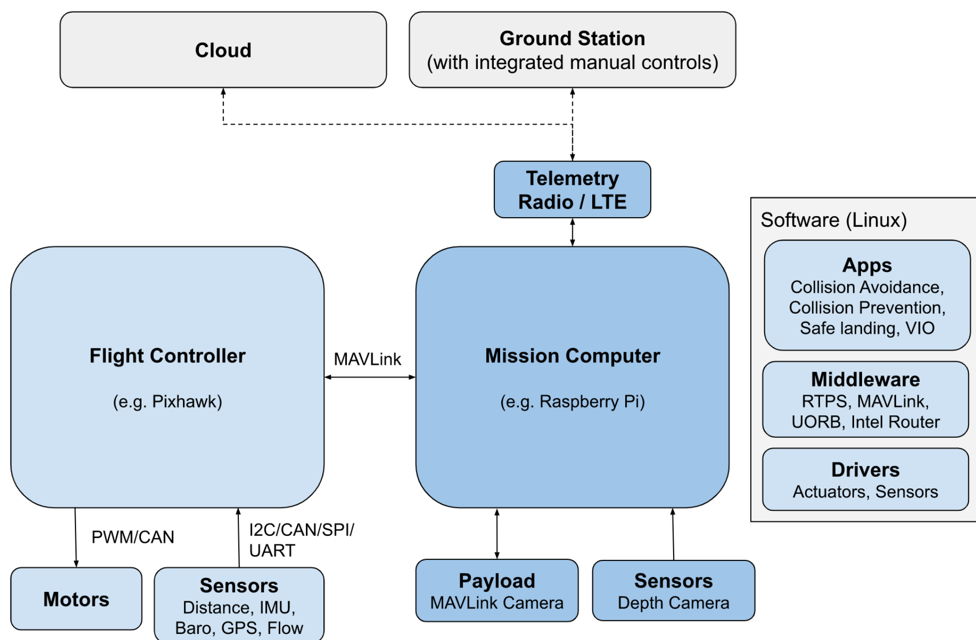



**Figure 2.1:** PX4 system that includes both a flight controller and a companion computer

## 2.1.1 Flight Stack

"The flight stack is a collection of guidance, navigation and control algorithms for autonomous drones. It includes controllers for fixed wing, multirotor and VTOL airframes as well as estimators for attitude and position.

The following diagram shows an overview of the building blocks of the flight stack. It contains the full pipeline from sensors, RC input and autonomous flight control

(Navigator), down to the motor or servo control (Actuators).

An **estimator** takes one or more sensor inputs, combines them, and computes a vehicle state (for example the attitude from IMU sensor data).

A **controller** is a component that takes a setpoint and a measurement or estimated state (process variable) as input. Its goal is to adjust the value of the process variable such that it matches the setpoint. The output is a correction to eventually reach that setpoint. For example, the position controller takes position setpoints as inputs, the process variable is the currently estimated position, and the output is an attitude and thrust setpoint that move the vehicle towards the desired position.

A **mixer** takes force commands (such as "turn right") and translates them into individual motor commands, while ensuring that some limits are not exceeded. This translation is specific for a vehicle type and depends on various factors, such as the motor arrangements with respect to the centre of gravity, or the vehicle's rotational inertia" [3]

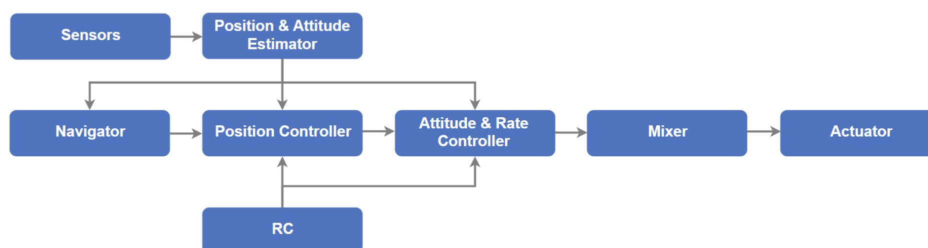



**Figure 2.2:** Building blocks of the flight stack

## 2.1.2 Middleware

"The middleware consists primarily of device drivers for embedded sensors, communication with the external world (companion computer, GCS, etc.) and the uORB publish-subscribe message bus. In addition, the middleware includes a simulation layer that allows PX4 flight code to run on a desktop operating system and control a computer modelled vehicle in a simulated world." [3]

So, the device drivers within the middleware are responsible for managing embedded sensors, allowing the autopilot system to gather data from various sensors such as accelerometers, gyroscopes, magnetometers, and other peripheral devices. These drivers provide a standardized interface for accessing and interpreting sensor data. Moreover, the middleware facilitates communication between the autopilot system and external entities, including companion computers and Ground Control Stations

(GCS) which is essential for exchanging mission plans, telemetry data, and control commands. Also, the uORB publish-subscribe message bus is a key component of the middleware, providing a lightweight and efficient communication mechanism within the PX4 system. It enables different modules and components to exchange information by publishing messages to specific topics and subscribing to relevant topics of interest. This publish-subscribe model ensures loose coupling between components and allows for efficient data sharing and inter-module communication.

For the first part of the work, modifications will be made to the middleware to incorporate the required sensors and establish the communication protocol with the flight controller. This entails adding device drivers for the relative sensors and implementing the necessary protocols to ensure seamless data exchange and integration with the autopilot system. Instead, in the second part of the work, modifications will be made to the flight stack itself. Specifically, a control system for position stabilization techniques will be added. This involves enhancing the existing flight stack with algorithms and modules that enable precise control of the vehicle's position, ensuring stability and accurate positioning during flight operations.

### 2.1.3 Runtime Environment

"PX4 runs on various operating systems that provide a POSIX-API (such as Linux, macOS, NuttX or QuRT). For our application NuttX will be used. It should also have some form of real-time scheduling. The inter-module communication (using uORB) is based on shared memory. The whole PX4 middleware runs in a single address space and memory is shared between all modules. The system is designed such that with minimal effort it would be possible to run each module in separate address space. There are 2 different ways that a module can be executed:

- **Tasks**: The module runs in its own task with its own stack and process priority. All the tasks must behave co-operatively as they cannot interrupt each other.
- **Work queue tasks**: The module runs on a shared work queue, sharing the same stack and work queue thread priority as other modules on the queue. Multiple work queue tasks can run on a queue, and there can be multiple queues

The advantage of running modules on a work queue is that it uses less RAM, and potentially results in fewer task switches. The disadvantages are that work queue tasks are not allowed to sleep or poll on a message." [3]

**NuttX**

NuttX is a real-time operating system (RTOS) designed for embedded systems and IoT devices. It provides a range of features, including pre-emptive multitasking, inter-process communication, file systems, device drivers, and networking. Nuttx has a modular architecture, allowing developers to choose which features to include in their application and to build custom configurations to meet specific requirements. The most relevant features are:

- NuttX supports pre-emptive multitasking, which allows multiple tasks to run simultaneously and provides fast context switching.
- NuttX is designed to be lightweight and portable, with a small memory footprint, making it suitable for use in systems with limited resources.
- NuttX is POSIX-compliant
- NuttX supports multiple file systems, including FAT, ROMFS, and NFS, which makes it easier to work with data and store files.
- Support for a wide range of microcontrollers and SoCs including ARM, MIPS, and AVR architectures. [4]

### 2.1.4 uORB

uORB is an asynchronous publish-subscribe messaging framework used in the PX4 flight control software stack, which is a popular open-source autopilot system for unmanned aerial vehicles (UAVs). [5] uORB is designed to provide a lightweight and efficient mechanism for inter-process communication (IPC) between modules within the PX4 system. Before going ahead, here is an overview of the terminology used to work with uORB in PX4:

- **Message**: defines the basic information format with a name and internal values, like a language grammar. That information is stored into a struct.
- **Topic**: is a communication funnel where the message gets sent and received
- **Publish / Subscribe**: Sending out messages on a topic is called publishing, and listening to a topic is called subscribing. Every topic therefore has at least one publisher and one subscriber.

The publish-subscribe model of uORB allows modules to publish messages to specific topics, and other modules to subscribe to those topics to receive messages. This

asynchronous communication approach reduces the need for modules to explicitly communicate with each other, which can improve system responsiveness and reduce latency.

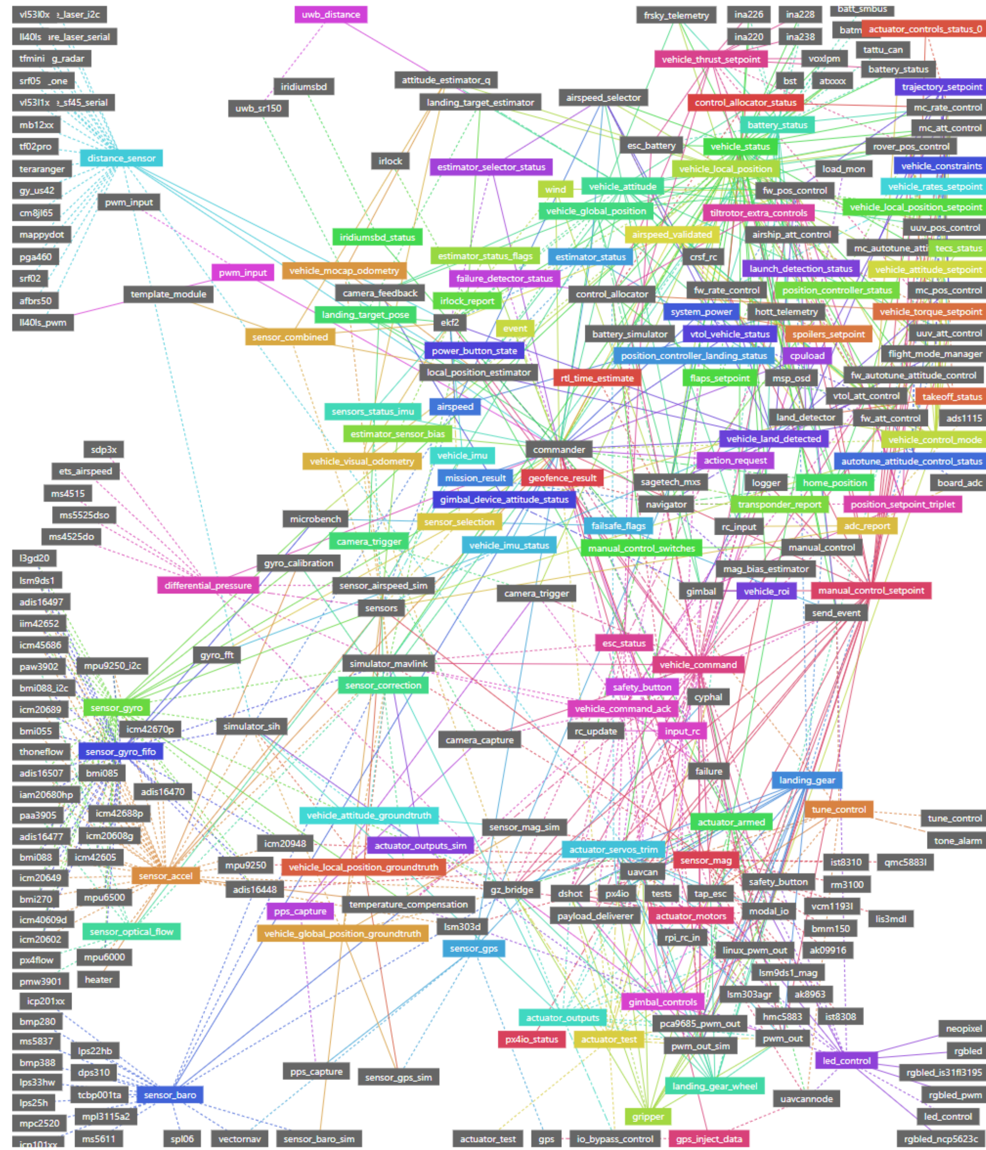



**Figure 2.3:** uORB Publication/Subscription Graph. For full graph see the reference

In the PX4 ecosystem, numerous topics are already defined and connected with their respective publishers and subscribers. A comprehensive list of these topics

can be found on the PX4-Autopilot GitHub repository [6]. However, there may be situations where a new topic needs to be created from scratch. To add a new topic, one must create a new .msg file in the msg/ directory and add the file name to the msg/CMakeLists.txt list. This triggers the automatic generation of the necessary C/C++ code. Each generated C/C++ struct includes a uint64_t timestamp field, which is used by the logger. It is important to ensure that this field is properly filled in when publishing a message since that variable plays a crucial role in maintaining temporal information and synchronization within the PX4 system. To use a topic in the code, it is sufficient to include the corresponding header using the format:
**#include <uORB/topics/topic_name.h>**

The entirety of the connections between modules and topics can be seen here:
https://docs.px4.io/main/en/middleware/uorb_graph.html
where in colour we see the topics and in grey the modules. If a module is subscribed to a topic the connecting line will be a solid line. Instead, if a module is publishing in a topic, the line will be dashed.(see figure 2.3)

ORB stands for "Object Request Broker" and that explains very well the role of this system. But since PX4 is running on an embedded system like a micro-controller on a flight control board, the ORB needs to be small and lightweight. That's why it's named u-ORB, since it's a micro (symbolized as u) sized Object Request Broker. We can generalize it saying that "uORB works like a librarian who keeps track of information, while communicating with people. Customers have a notion of 'Topic' as the book they are writing / reading from. But only the librarian can actually touch, write, and read the book. Therefore, customers never actually get to interact with the physical book itself." [7] For identifying and differentiating between different topics within the system, uORB uses the ORB_ID(topic_name) macro. A macro is a programming construct that allows for the definition of reusable code snippets or statements. In reality, it's just a pointer to the topic's metadata instance. This is a struct which includes information about the specific topic:

```
/**
* Object metadata.
*/
struct orb_metadata{
    const char *o_name;             /**< unique object name */
    const uint16_t o_size;          /**< object size */
    const uint16_t o_size_no_padding;       /**< object size
    w/o padding at the end (for logger) */
    const char *o_fields;           /**< semicolon separated list
    of fields (with type) */
```

```
    uint8_t o_id;                   /**< ORB_ID enum */
};  }
```

Full code: Link

When we build PX4, all the uORB topic headers (.h) and source files (.cpp) are generated in the build folder in 'uORB/topics/' and 'msg/topics_sources/' respectively.

### uORB Publishing

To publish messages, we use the uORB::Publication class. In the code we develop a struct and fill it, then we publish it with the .publish() function.

```
bool publish(const T &data)
{
    if (!advertised()){
        advertise(); // Advertise publishing of the topic
    }

    return (Manager::orb_publish(get_topic(), _handle, &data) ==
    PX4_OK);
}
```

Full code: Link

In the function there is a line where the advertised( ) function is called. If it returns false, then advertise( ) is called. Basically the advertise() function calls the 'orb_advertise_queue()' function to get the ORB topic advertiser handle. The advertised() function simply checks if we have a valid 'handle' for advertising.

There the concept of queue is used. To keep a queue of data instead of overwriting the data every time there's a new publish event ensures that published messages won't be lost so easily since we have some buffer to preserve past publications. However, having a queue takes more memory. Then, the publish() functions calls the orb_publish() function through the return.

```
int uORB::Manager::orb_publish(const struct orb_metadata *meta,
orb_advert_t handle, const void *data)
{
    // ...
    return uORB::DeviceNode::publish(meta, handle, data);
}
```

Full code: Link

The orb_publish takes three arguments: the topic's metadata, the advertiser handle and the pointer to the data itself. And this information gets passed straight to the DeviceNode class, which is the 'instance' of the device file. This basically controls the read/write of the data to the uORB topic in the memory. The orb_advert_t handle is actually a pointer to the DeviceNode object tied to the specific topic's device file. So, every time publish() is called, a specific node will be created ad stored into the /obj folder.

### uORB Subscribing

To subscribe to a message, we use the uORB::Subscription class. In our code we generate a struct and fill it through the .update() function

```
bool update(void *dst)
{
    if (!valid()){
        subscribe();
    }

    return valid() ? Manager::orb_data_copy(_node, dst,
    _last_generation, true) : false;
}
```

Full code: Link

It is very similar to the Publication class. The _node object is the DeviceNode pointer for the topic instance, just like the advertisement handle. It has a similar check logic to verify if the _node is a valid pointer, and if not it subscribes to the topic. Let's figure out what the subscribe() function does.

```
    // Subscribes to the topic
bool Subscription::subscribe()
{
    // check if already subscribed
    if (_node != nullptr) {
        return true;
    }

    if (_orb_id != ORB_ID::INVALID && uORB::Manager::get_instance()){
        unsigned initial_generation;
```

```
        void *node = uORB::Manager::orb_add_internal_subscriber(
        _orb_id, _instance, &initial_generation);

        if (node){
             _node = node;
             _last_generation = initial_generation;
            return true;
        }
    }

    return false;
}
```

Full code: Link

Basically the 'uORB::Manager::orb_add_internal_subscriber' gets called and _node gets updated to the DeviceNode pointer returned from the function.

```
void *uORB::Manager::orb_add_internal_subscriber(ORB_ID orb_id,
uint8_t instance, unsigned *initial_generation)
{
    uORB::DeviceNode *node = nullptr;
    DeviceMaster *device_master =
    uORB::Manager::get_instance()->get_device_master();

    if (device_master != nullptr){
        node = device_master->getDeviceNode(get_orb_meta(orb_id),
        instance);

        if (node){
            node->add_internal_subscriber();
            *initial_generation = node->get_initial_generation();
        }
    }

    return node;
}
```

Full code: Link

The function 'add_internal_subscriber' of the DeviceNode updates the subscriber count inside the DeviceNode object, to keep track of how many subscribers it has

for the topic. Then it fetches the ‘generation’ number. This is an internally tracked number inside DeviceNode, and it increases whenever new data is published on the topic. By keeping track of this number internally on Subscriber’s side, it is possible to detect whether new data is available or not. This is how uORB knows whether new data is available to the Subscriber or not. This is why it needs to internally store the value.[8, 7, 9]

### 2.1.5 MAVLink

MAVLink is a communication protocol used in the PX4 autopilot system for unmanned aerial vehicles (UAVs) and other robotic applications. It is a lightweight messaging protocol that enables communication between different components in the system. In fact, PX4 uses MAVLink to communicate with QGroundControl, and as the integration mechanism for connecting to drone components outside of the flight controller: companion computers, MAVLink enabled cameras etc. So, the main difference from uORB is that it is used as a communication method for internal component of the drone. MAVLink, instead, is responsible for the communication of the entire ecosystem. In the PX4 system, MAVLink is used to send and receive messages between different modules or components. These messages can contain information such as sensor data, GPS coordinates, or control commands.[10]

The advantage of the MAVLink protocol is that it supports different types of transport layers. It can be transmitted through serial telemetry low bandwidth channels. Another alternative is to use a network interface, which is typically Wi-Fi or Ethernet, and stream the MAVLink messages through IP Networks. In this case, the autopilot running the MAVLink protocol typically supports both UDP and also TCP connections at the transport layer between the ground station and the drone, depending on the reliability level required by the application.

UDP is a datagram protocol that requires no connection between the client and the server, and it has no mechanism to ensure that messages are reliably delivered but provides a fast lighter weight alternative for real-time and loss-tolerant message streaming.

TCP is a reliable connection-oriented protocol that provides better reliability of transfer thanks to its acknowledgment mechanism but could be subject to congestion and heavy management of the connection.

The messages used to communicate are binary serialized. That means that the parts of the messages are packed into bytes and then transmitted. There are 2 main versions: v1 and v2. The first one was released in 2009, the second one in 2017 and it is backward compatible with the previous one. By default, MAVLink

v2 is used, but the version can be switched to v1 according to the needs. The message structure for v2 is proposed:

**Figure 2.4:** Over-the-wire format for a MAVLink 2 packet

- **STX**: it describes start of frame and will always be 0xFE as in official documentation of MAVLink 1.0.
- **LEN**: it represents the message length in bytes and is encoded into 1 Byte.
- **INC FLAGS**: incompatibility flags that indicate whether the packet contains some features that must be considered when parsing the packet. For example, an Incompatibility flag equal to 0x01 means that the packet is signed and that a signature is appended at the end of the packet. Those flags affect the message structure.
- **CMP FLAGS**: compatibility flags that indicates flags that can be ignored if not understood and it does not prevent the parser from processing the message even if the flag cannot be interpreted. For example, this may refer to flags that indicate the priority of the packet as it does not affect the packet structure. Those flags don't affect the message structure.
- **SEQ**: it denotes the sequence number of the message. It is encoded into 1 Byte and takes values from 0 to 255. The sequence number of message enabled to detect message losses in the receive.
- **SYS ID**: it represents the System ID. Every unmanned system should have its System ID, in particular, if they are managed by one ground station.
- **COMP ID**: it identifies the component of the system that is sending the message. If there is no subsystem or component, then it is not used
- **MSG ID**: it refers to the type of the message embedded in the payload. For example, the message ID equal to 0 refers to a message of type HEARTBEAT, which indicates that the system is alive and is sent every one second. The message ID is the essential information that allows to parse the payload and extract the information from it. It differs between the two versions. It is encoded into 24 bits instead of 8 bits in the previous version, which allows a much higher number of message types, reaching up to 16777215 different ones.

- **PAYLOAD**: this carries out the real data of the message, which depends on the message type. It contains up to 255 bytes.

- **CHECKSUM**: it is formed by CKA and CKB giving one byte each. These represent the Cyclic Redundancy Check (CRC) calculated with seed values A and B, respectively. The CRC ensures that the message has not been changed during its transmission and that both the sender and the receiver have the same message.

- **SIGNATURE**: optional field used to ensure that the link is tamper-proof. It allows the authentication of the message and verifies that it originates from a trusted source. The signature of the message is appended if the incompatibility flags are set to 0x01. The 13 bytes of the message signature contain the following fields: LinkID (1 byte that represents the ID of the link/channel used to send the packet), timestamp (encoded with 6 bytes in 10-microsecond units, it is used to avoid replay attacks) and signature (encoded in 6 bytes for the message and is calculated based on the complete message, timestamp, and the secret key, which is a shared symmetric key of 32 bytes stored on both ends, namely the autopilot, and the ground station or the MAVLink AP).[11, 12]

### 2.1.6 QGroundControl

QGroundControl is a versatile open-source software application used for the comprehensive control of unmanned aerial vehicles (UAVs) and other unmanned systems. Its primary purpose is to interface with various autopilot systems, including PX4. This software offers a user-friendly interface that enables users to configure, calibrate, and govern UAVs efficiently and provides extensive customization options for a wide range of flight parameters, such as altitude, speed, and mission waypoints. Additionally, QGroundControl facilitates real-time telemetry data transmission, empowering users to monitor their UAVs' flight status and make necessary adjustments when required. QGC also stands out for its compatibility with a wide array of hardware components and systems. This adaptability enables users to seamlessly integrate the control station with different UAV platforms, autopilot systems, and payloads and supports multiple communication protocols, such as MAVLink.

One of the notable capabilities of QGroundControl is its support for advanced features, including autonomous flight modes. These autonomous modes enable UAVs to execute pre-planned missions without direct intervention from the user, being particularly valuable for tasks like mapping, surveying, and inspections.[13]

Furthermore, QGC empowers users to personalize nearly all aspects of their drones,

encompassing the airframe, radio, sensors, flight modes, power tuning, camera settings, and various other parameters. Through QGC we can access the list of topic nodes maintained by the publishing/subscribing system of uORB and stored into the /obj directory. We can do that by typing the command: ls /obj (see figure 2.6).

To have a better overview of the topics, you can use the command uorb status (see figure 2.7).

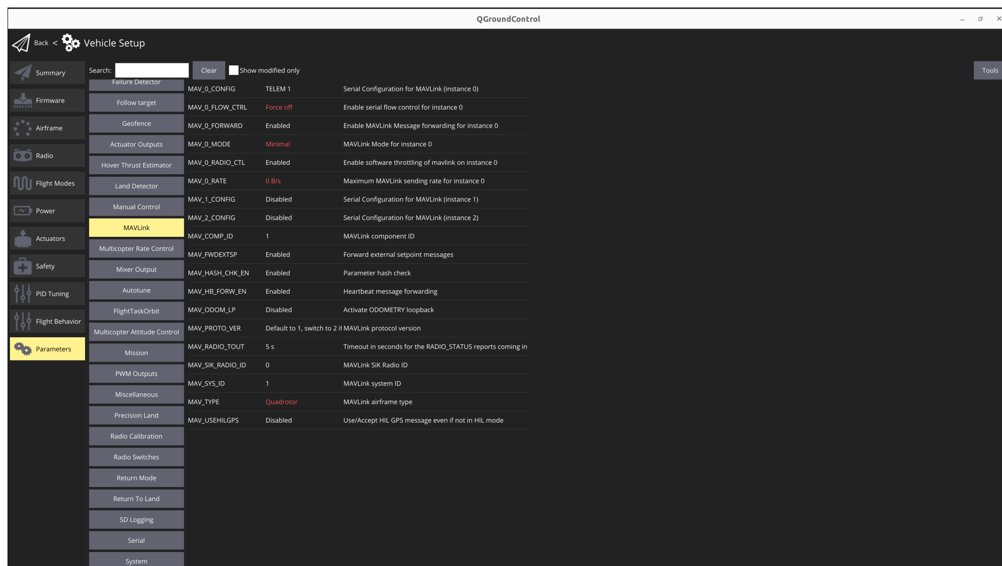



**Figure 2.5:** Example of a paramenters list on QGC

```
NuttShell (NSH) NuttX-11.0.0
nsh> ls /obj
/obj:
 actuator_armed0
 actuator_controls_00
 actuator_controls_status_00
 actuator_motors0
 actuator_outputs0
 actuator_outputs_sim0
 actuator_servos0
 actuator_servos_trim0
 autotune_attitude_control_status0
 battery_status0
 commander_state0
 control_allocator_status0
 control_allocator_status1
 cpuload0
 distance_sensor0
 distance_sensor1
 ekf2_timestamps0
 estimator_aid_src_baro_hgt0
 estimator_aid_src_fake_hgt0
 estimator_aid_src_fake_pos0
 estimator_attitude0
 estimator_baro_bias0
 estimator_ev_hgt_bias0
 estimator_event_flags0
 estimator_global_position0
 estimator_gnss_hgt_bias0
```

**Figure 2.6:** Output of the QGC console of the command ls/obj

```
uorb status
TOPIC NAME                      INST #SUB #Q SIZE PATH
actuator_armed                     0   10  1   16 /obj/actuator_armed0
actuator_controls_0                0    3  1   56 /obj/actuator_controls_00
actuator_controls_status_0         0    0  1   24 /obj/actuator_controls_status_00
actuator_motors                    0    3  1   72 /obj/actuator_motors0
actuator_outputs                   0    2  1   80 /obj/actuator_outputs0
actuator_outputs_sim               0    2  1   80 /obj/actuator_outputs_sim0
actuator_servos                    0    1  1   48 /obj/actuator_servos0
actuator_servos_trim               0    0  1   40 /obj/actuator_servos_trim0
autotune_attitude_control_status   0    4  1  104 /obj/autotune_attitude_control_status0
battery_status                     0    5  1  168 /obj/battery_status0
commander_state                    0    1  1   16 /obj/commander_state0
control_allocator_status           0    2  1   80 /obj/control_allocator_status0
control_allocator_status           1    1  1   80 /obj/control_allocator_status1
cpuload                            0    5  1   16 /obj/cpuload0
distance_sensor                    0    3  1   56 /obj/distance_sensor0
distance_sensor                    1    3  1   56 /obj/distance_sensor1
ekf2_timestamps                    0    1  1   24 /obj/ekf2_timestamps0
estimator_aid_src_baro_hgt         0    0  1   56 /obj/estimator_aid_src_baro_hgt0
estimator_aid_src_fake_hgt         0    0  1   56 /obj/estimator_aid_src_fake_hgt0
estimator_aid_src_fake_pos         0    0  1   88 /obj/estimator_aid_src_fake_pos0
estimator_attitude                 0    1  1   56 /obj/estimator_attitude0
estimator_baro_bias                0    1  1   40 /obj/estimator_baro_bias0
estimator_ev_hgt_bias              0    1  1   40 /obj/estimator_ev_hgt_bias0
estimator_event_flags              0    1  1   56 /obj/estimator_event_flags0
estimator_global_position          0    1  1   64 /obj/estimator_global_position0
estimator_gnss_hgt_bias            0    1  1   40 /obj/estimator_gnss_hgt_bias0
estimator_gps_status               0    1  1   40 /obj/estimator_gps_status0
estimator_innovation_test_ratios   0    1  1  136 /obj/estimator_innovation_test_ratios0
estimator_innovation_variances     0    1  1  136 /obj/estimator_innovation_variances0
estimator_innovations              0    1  1  136 /obj/estimator_innovations0
estimator_local_position           0    1  1  168 /obj/estimator_local_position0
estimator_odometry                 0    1  1  112 /obj/estimator_odometry0
estimator_optical_flow_vel         0    1  1   72 /obj/estimator_optical_flow_vel0
estimator_rng_hgt_bias             0    1  1   40 /obj/estimator_rng_hgt_bias0
estimator_selector_status          0    6  1  160 /obj/estimator_selector_status0
estimator_sensor_bias              0    6  1  120 /obj/estimator_sensor_bias0
estimator_states                   0    1  1  216 /obj/estimator_states0
estimator_status                   0    4  1  120 /obj/estimator_status0
estimator_status_flags             0    2  1   96 /obj/estimator_status_flags0
estimator_visual_odometry_aligned  0    1  1  112 /obj/estimator_visual_odometry_aligned0
```

**Figure 2.7:** Output of the QGC console of the command uorb status

## 2.2 PX4 Control Architecture

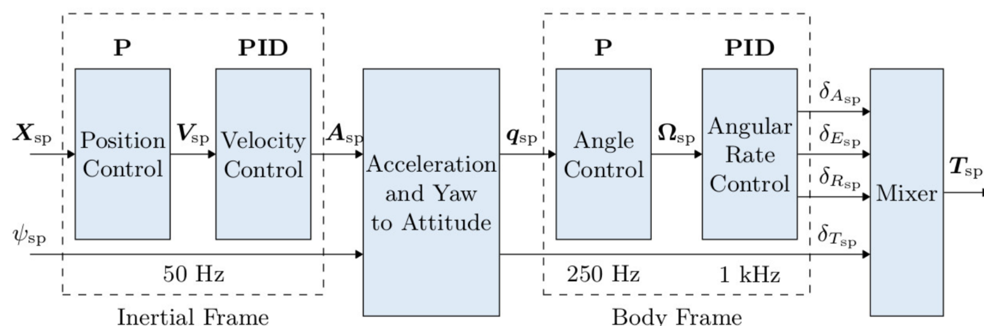



**Figure 2.8:** Multicopter Control Architecture

PX4 controller is formed by 2 main parts. The first, the position controller, controls the inertial frame, taking as input the position setpoint and as output the acceleration setpoint. The second one, the attitude controller, controls the body frame, taking as an input the quaternion generated by the previous part and giving as outputs the Aerodynamic control surface angular deflection setpoints. The two are connected by the Acceleration and yaw to attitude block, which takes acceleration and yaw setpoint to generate the quaternion. This is a standard cascaded control architecture. Depending on the mode, the outer (position) loop is bypassed (shown as a multiplexer after the outer loop). The position loop is only used when holding position or when the requested velocity in an axis is null. The acceleration setpoints generated by the velocity controller will be converted to thrust and attitude setpoints.

The controllers are a mix of P and PID controllers and the estimates come from EKF2:

- **PROPORTIONAL GAIN (P)**: The P gain is used to minimize the tracking error and it is responsible for a quick response and thus should be set as high as possible, but without introducing oscillations. If the P gain is too high, you will see high-frequency oscillations while if the P gain is too low, the vehicle will react slowly to input changes.
- **DERIVATIVE GAIN (D)**: The D gain is used for rate damping. It is required but should be set only as high as needed to avoid overshoots. If the D gain is too high, the motors become twitchy (and maybe hot), because the D term amplifies noise. If the D gain is too low, overshoot is seen after a step-input.

- **INTEGRAL GAIN (I)**: The I gain keeps a memory of the error. The I term increases when the desired rate is not reached over some time. It is important, but it should not be set too high since we would see slow oscillations.

Here is a detailed description of how each block is made and which is its role.

### 2.2.1 Position control

The position control sub-controller aims to control the vehicle's position in three-dimensional space. Based on the position error, the position control generates a velocity command that aims to drive the vehicle towards the desired position. This error is the difference of desired and actual distance setpoints. The latter is obtained from the vehicle's sensors (e.g., GPS).

The commanded velocity is saturated to keep the velocity in certain limits, to prevent excessive or unsafe velocity commands that the vehicle may not be able to achieve or that could compromise its stability.

This system is implemented as a proportional (P) controller. The gain of the P determines the strength of the control response. Higher gain values result in stronger corrective velocity commands for larger position errors.

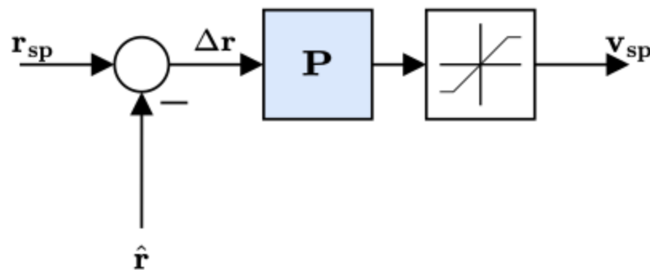



**Figure 2.9:** Position controller block scheme

### 2.2.2 Velocity Control

It receives the velocity setpoint generated in the previous controller and uses the difference between the desired and actual velocities to calculate the control commands. Uses a PID controller to stabilise velocity and commands an acceleration.

To prevent integral wind-up, the integrator of the PID controller incorporates an anti-reset windup mechanism using a clamping method. This restricts the

integrator output within specified bounds, limiting the impact of integration when the system is saturated or unable to actuate the desired control commands.

The output of the PID controller is a commanded acceleration that is used to control the vehicle's motion. This acceleration represents the desired rate of change of velocity to achieve the desired velocity setpoints. Although the commanded acceleration is not saturated, saturation is applied to the converted thrust setpoints. This saturation, in combination with the maximum tilt angle, ensures that the thrust commands and tilt angles remain within safe and feasible limits for the vehicle.

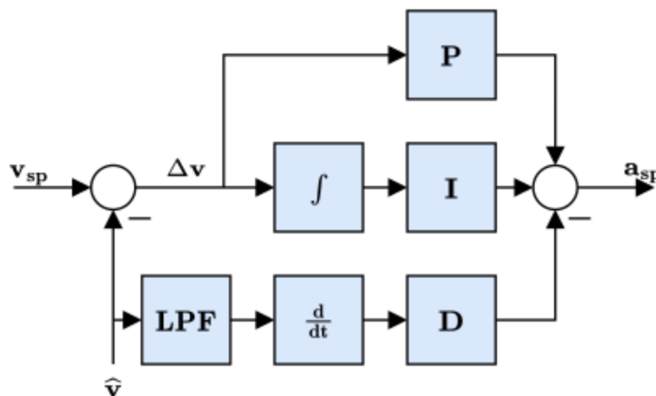



**Figure 2.10:** Velocity controller block scheme

### 2.2.3 Attitude Control

The attitude control typically works with a quaternion representation of attitude. Quaternions provide a concise and computationally efficient way to represent rotations in three-dimensional space. For that reason, the desired attitude setpoint and the current attitude measurements from the vehicle's sensors (e.g., IMU) are represented as quaternions.

The first step in attitude control is to compute the error between the desired attitude setpoint and the current attitude measurement. This error is usually obtained by taking the quaternion multiplication of the conjugate of the current attitude quaternion and the desired attitude quaternion. The resulting quaternion error represents the angular difference between the desired and current attitudes.

When tuning this controller, the only parameter of concern is the P gain. The P gain determines the strength of the control response based on the quaternion error. Careful tuning of the P gain is crucial to achieve desired performance and stability in attitude control. Once the desired control torques or angles are computed, they

need to be mapped to motor commands or control surface deflections to actuate the vehicle.

The rate command can be saturated. This occurs when the desired angular rate exceeds the maximum limit that the vehicle's actuators or control surfaces can achieve. Saturation mechanisms are in place to limit the rate command and ensure it remains within the capabilities of the actuators or control surfaces.

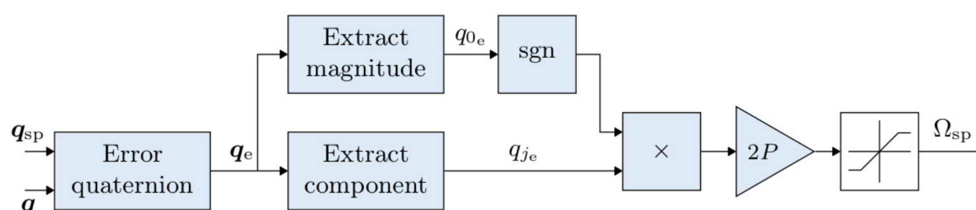



**Figure 2.11:** Attitude controller block scheme

### 2.2.4 Angular Rate Control

The angular rate control subcontroller plays a crucial role in maintaining the desired rotational speed of the vehicle. This is a K-PID controller, and it is part of the attitude controller. The angular rate control sub-controller typically receives angular rate measurements from the vehicle's sensors, such as gyroscopes. These sensors provide information about the current rotational speed around each axis of the vehicle. It then calculates an error signal based on the difference between the desired and actual rates.

The integral authority is limited to prevent wind up, also the outputs are limited around -1 and +1. A low pass filter is used in the derivative path to reduce the noise so that the gyro driver provides a filtered derivative to the controller.

Once the desired motor commands or control surface deflections are computed, they need to be mapped to the specific hardware configuration of the vehicle. This mapping depends on the number and arrangement of motors or control surfaces. [14]

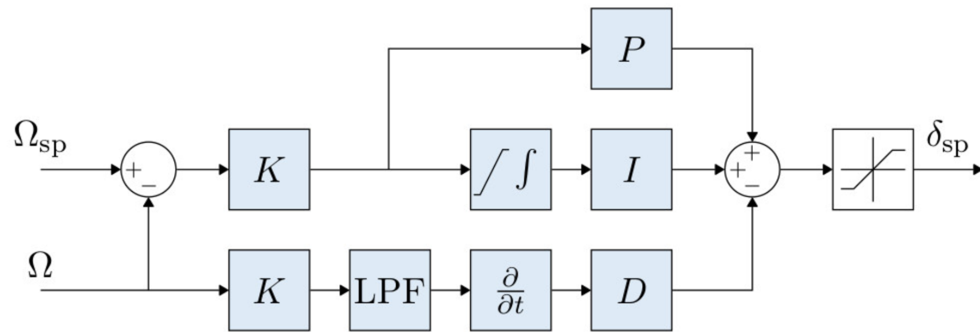



**Figure 2.12:** Angular rate controller block scheme

# Chapter 3

# Configuration, Model of the Drone and Sensors

This chapter's aim is to describe and explain all the used devices and all the operations made to develop a good environment for the drone to fly in.

The first action performed was to create a partition of my hard disk, where I installed ubuntu 22.04. Then I downloaded from GitHub the source code of PX4-autopilot. To access and modify that code I used VS Code, a code editor redefined and optimized for building and debugging. Then I flashed the code into an STM32h7x board through an ST-Link. This is a debugging and programming interface commonly used to flash and debug microcontrollers and development boards manufactured by STMicroelectronics. It provides a bridge between the development host (usually a computer) and the target device (such as an STM32 microcontroller). I followed a sequence of action to flash the board. First I built the default code and the bootloader through the make command. I took the generated .elf files and put them into a specific folder into the Build directory. To flash, OpenOCD has been used, which is an Open On-Chip Debugger. Then I connected the board to the ST-Link and flashed it using the command:

```
~/openocd/bin/openocd -f ~/openocd/scripts/interface/stlink.cfg -f
~/openocd/scripts/target/stm32h7x_dual_bank.cfg -c "init; reset
halt; flash erase_sector 0 0 last; flash erase_sector 1 0 last;
program fc-v3_bootloader.elf verify; program fc-v3_default.elf
verify; reset; exit"
```

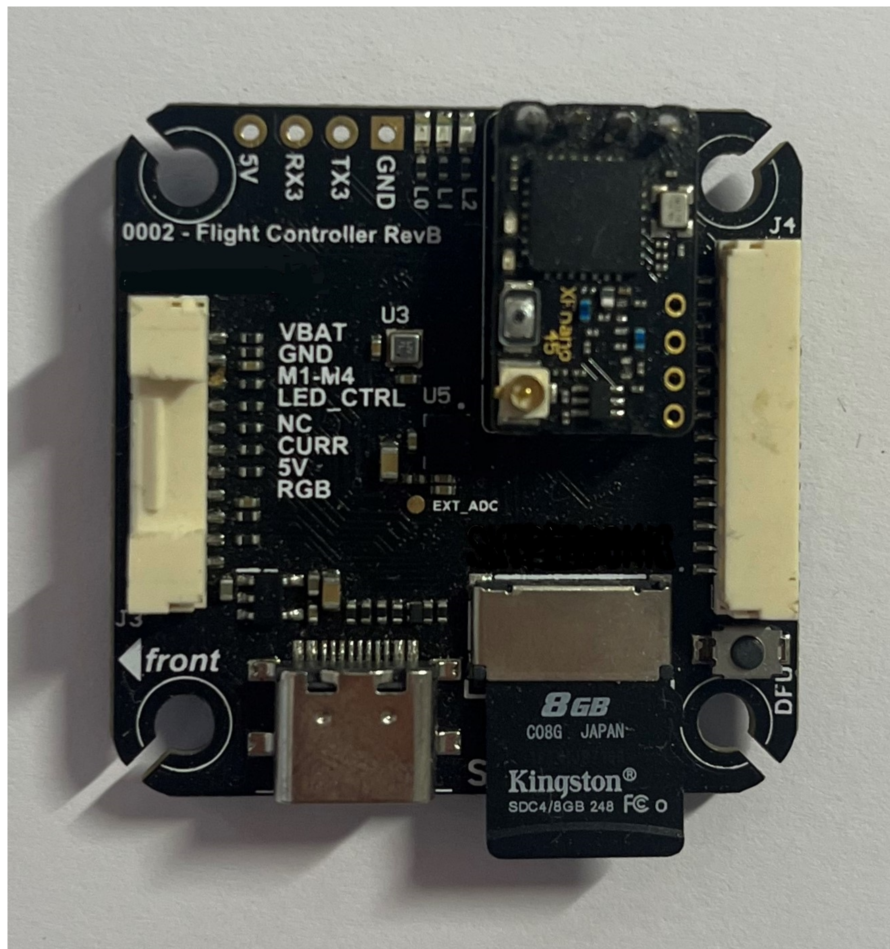



**Figure 3.1:** STM32h7x board

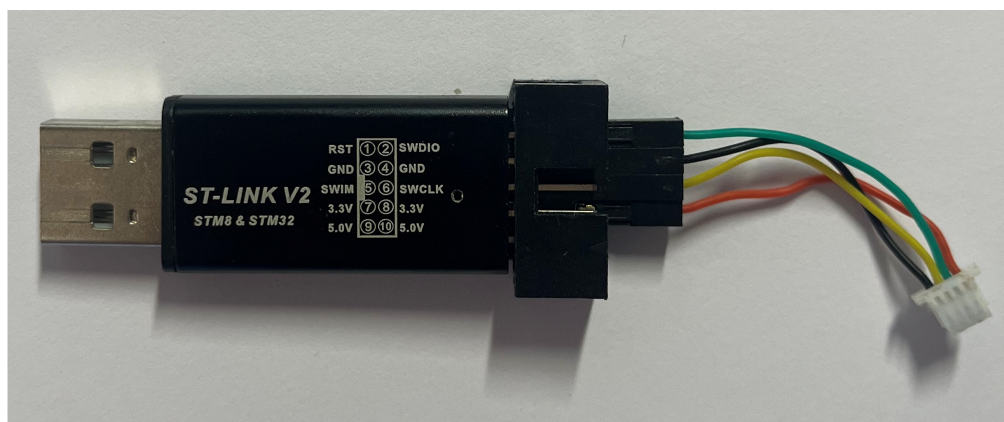



**Figure 3.2:** ST-Link

Here's a breakdown of what each part of the command does:

- **/openocd/bin/openocd**: this specifies the path to the OpenOCD binary executable that will be executed.

- **-f /openocd/scripts/interface/stlink.cfg**: this option specifies the path to the configuration file for the ST-Link interface. It provides the necessary settings to establish a connection between the debugger and the target device.

- **-f /openocd/scripts/target/stm32h7x_dual_bank.cfg**: This option specifies the path to the configuration file for the STM32H7x dual bank target. It provides the necessary settings specific to the target device, in this case, an STM32H7x microcontroller with dual bank memory.

- **-c "init; reset halt; flash erase_sector 0 0 last; flash erase_sector 1 0 last; program skypersonic_fc-v3_bootloader.elf verify; program skypersonic_fc-v3_default.elf verify; reset; exit"**: this part of the command specifies a series of OpenOCD commands to be executed: .

  - **init**: this command initializes the connection to the target device.
  - **reset halt**: this command halts the target device, bringing it to a halted state.
  - **flash erase_sector 0 0 last**: this command erases the flash memory sector 0 of the target device. The "0 0 last" specifies the range of sectors to erase.
  - **flash erase_sector 1 0 last**: this command erases the flash memory sector 1 of the target device.
  - **program fc-v3_bootloader.elf verify**: this command programs the "fc-v3_bootloader.elf" file onto the target device's flash memory. The "verify" option ensures the verification of the programming process.
  - **program fc-v3_default.elf verify**: this command programs the "fc-v3_default.elf" file onto the target device's flash memory, similar to the previous command.
  - **reset**: this command resets the target device, allowing it to start executing the newly flashed program.
  - **exit**: this command exits the OpenOCD tool

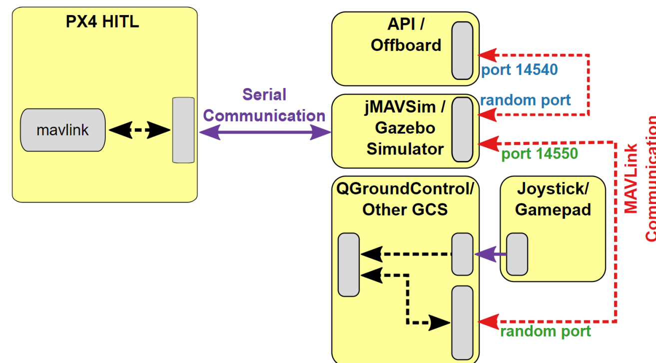



**Figure 3.3:** Block scheme of a HITL simulation environment

The next step was to modulate the system in order to test it through Hardware-in-the-loop (HITL) simulation. “With Hardware-in-the-Loop (HITL) simulation the normal PX4 firmware is run on real hardware. Gazebo (running on a development computer) is connected to the flight controller hardware via USB/UART. The simulator acts as gateway to share MAVLink data between PX4 and QGroundControl.

To enable HITL a few passages need to be followed:

1. Connect the autopilot directly to QGroundControl via USB.
2. Enable HITL inside QGC (setup/safety).
3. Select Airframe ( I chose HIL Quadcopter).
4. Calibrate the Joystick.
5. Setup UDP
6. Configure Joystick and Failsafe. Set the following parameters in order to use a joystick instead of an RC remote control transmitter:
    - COM_RC_IN_MODE to "Joystick/No RC Checks". This allows joystick input and disables RC input checks.
    - NAV_RCL_ACT to "Disabled". This ensures that no RC failsafe actions interfere when not running HITL with a radio control.“ [15]

To start the QGroundControl station and the gazebo environment you need to use a few lines of code. First connect the board through USB to the computer . Notice that, while PX4 is ran into an HITL simulation, gazebo will instead perform a SITL simulation, since it is a simulated environment. To run gazebo, move into

the source code folder through the terminal and launch, separately, this sequence of lines:

```
DONT_RUN=1 make px4_sitl_default gazebo
```

- **DONT_RUN=1**: this tells the system not to start the simulation automatically after the build, giving the possibility to set up additional features after that command.
- **make px4_sitl_default gazebo**: it builds the gazebo SITL simulation using the default settings.

```
source Tools/simulation/gazebo/setup_gazebo.bash $(pwd)
$(pwd)/build/px4_sitl_default
```

- **source Tools/simulation/gazebo/setup_gazebo.bash**: is used to execute the setup_gazebo.bash script within the current shell environment. The second contains the path of the .bash.
- **$(pwd)**: used to refer to the current working directory.
- **$(pwd)/build/px4_sitl_default**: this refers to the path of the SITL gazebo simulation build with default configuration.

```
gazebo Tools/simulation/gazebo/sitl_gazebo/worlds/hitl_iris.world
```

This is used to run the gazebo environment with a specific world file (in this case hitl_iris.world) in HITL mode. As for the other lines, it is necessary to specify the path of the world file.

## 3.1 Gazebo

"Simulators have played a critical role in robotics research as tools for quick and efficient testing of new concepts, strategies, and algorithms. Gazebo is designed to create a 3D dynamic multi-robot environment capable of recreating the complex worlds that would be encountered by the next generation of mobile robots. Its open source status, fine grained control, and high fidelity place Gazebo in a unique

position to become more than just a stepping stone between the drawing board and real hardware: data visualization, simulation of remote environments, and even reverse engineering of blackbox systems are all possible applications." [16]

To generalize, the drone and the environment are not physical, but are generated by gazebo. Instead, the way they behave is developed by the px4 firmware, which takes the values generated in the simulation and uses them to sustain and activate the control operations. Gazebo can provide both SITL (Software-in-the-loop) and HITL (Hardware-in-the-loop). In my work the version of gazebo used is Gazebo Classic, due to incompatibility issues encountered with other versions and ubuntu 22.04 (Jammy Jellyfish).

In order to create a simulation in Gazebo, developers typically start by defining the world and the objects within it using a file format such as SDF (Simulation Description Format). This file describes the physical properties of the objects in the simulation, such as their size, shape and mass.

The world of the simulation is defined by a .world file. In this file can be included different models, which implement the objects of major interest, like the drone, or some environment objects, like walls or blocks. These models are also defined in an SDF file. Then we can apply some small changes to the objects, like the pose or the colour.

**Figure 3.4:** Model of the iris_hitl on Gazebo

In the PX4 original folder, a lot of premade worlds and models are present, like the world of a warehouse, or many implementations of the iris drone, different

from each other by means of integrated sensors/devices. In this thesis, the already made iris_hitl.sdf has been used, but 2 more distance sensors had been added for development purposes. This has been tested into a new generated world, where only some walls had been added. This due to the necessity of testing the drone control behaviour with respect to a wall.

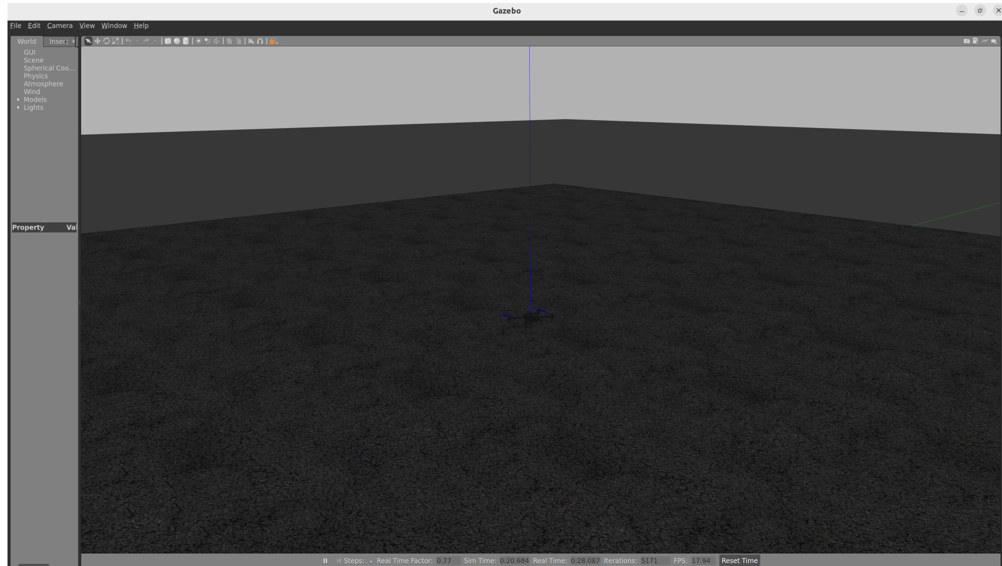



**Figure 3.5:** Gazebo empty world

The custom implementation of the ‘warehouse_wall.world’ file is:

```
<?xml version="1.0" ?>
<sdf version="1.5">
  <world name="default">
    <!-- A global light source -->
    <include>
      <uri>model://sun</uri>
    </include>
    <!-- A ground plane -->
    <include>
      <uri>model://ground_plane</uri>
    </include>
    <include>
      <uri>model://asphalt_plane</uri>
    </include>
<include>
```

```
      <uri>model://iris_hitl</uri>
    </include>
    <physics name='default_physics' default='0' type='ode'>
      <gravity>0 0 -9.8066</gravity>
<ode>
   <solver>
          <type>quick</type>
          <iters>10</iters>
          <sor>1.3</sor>
          <use_dynamic_moi_rescaling>0</use_dynamic_moi_rescaling>
        </solver>
        <constraints>
          <cfm>0</cfm>
          <erp>0.2</erp>
          <contact_max_correcting_vel>100</contact_max_correcting_vel>
          <contact_surface_layer>0.001</contact_surface_layer>
        </constraints>
      </ode>
      <max_step_size>0.004</max_step_size>
      <real_time_factor>1</real_time_factor>
      <real_time_update_rate>250</real_time_update_rate>
      <magnetic_field>6.0e-6 2.3e-5 -4.2e-5</magnetic_field>
    </physics>

    <include>
      <name>grey_wall</name>
      <uri>model://grey_wall</uri>
      <pose>-8.34545 0 0 0 0 -1.57</pose>
    </include>

    <include>
      <name>grey_wall2</name>
      <uri>model://grey_wall</uri>
      <pose>-3.5 9 0 0 0 0</pose>
    </include>

  </world>
</sdf>
```

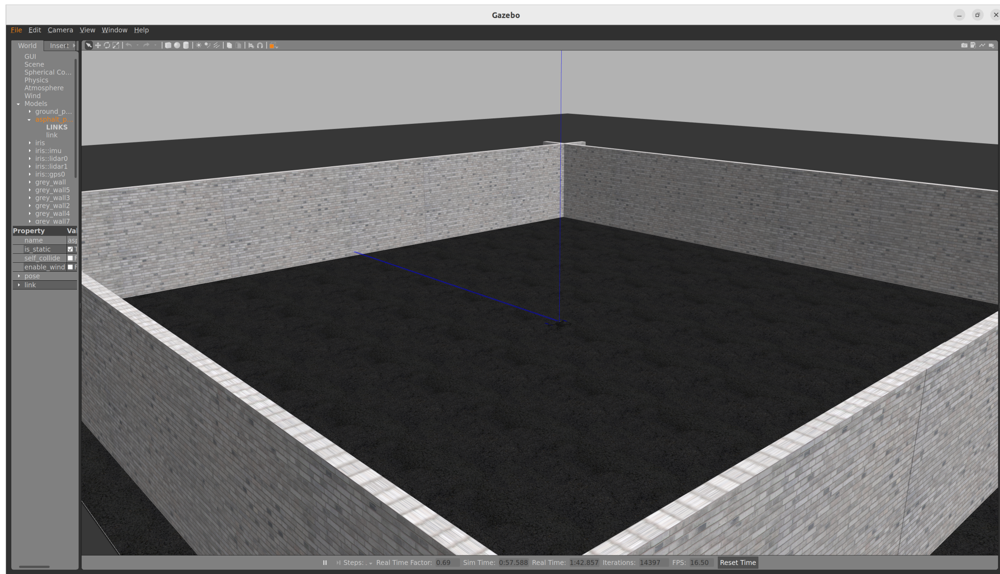



**Figure 3.6:** Gazebo world modified for simulation purposes

## 3.2 SDF File

"The Simulation Description Format (SDF) is a representation formats for robots employed by the Gazebo Simulator. SDF is also designed for more general purpose use with support for environment entities such as lighting, scene-objects, and sensors. SDF use XML language."[17] There are several recursive elements that allow for hierarchical structuring and composition of models:

- **<model>**: it represents a complete model within the SDF file. It contains all the element listed below and more. Models can be nested within other models, allowing for complex model hierarchies.

- **<link>**: it represents a rigid body within a model. It defines the visual and physical properties of the body, such as mass, inertia and collision properties. Additionally, <link> can contain child elements recursively, such as:

  - **<visual>**: it defines the visual representation of the object.
  - **<collision>**: it defines the physical properties and collision behaviour.
  - **<sensor>**: it is used to define a sensor attached to a link within a model. Gazebo supports a variety of sensors such as cameras, depth sensors, lidars, contact sensors and more.

  - **<frame>**: it represents a coordinate frame that can be attached to a link or another frame. It defines the relative position and orientation with respect to the parent frame.

- **<joint>**: it describes the connections between links in a model (e.g., revolute, prismatic, fixed)

- **<plugin>**: it allows for the addition of custom functionality to models and are often used to integrate external software components or implement specific behaviours.

- **<include>**: it allows for the inclusion of external SDF files within the current SDF file.

the SDF file I generated to build the two lidar sensor is (only representing one of the two):

```
<model name='lidar0'>
  <link name='base'>
    <pose>0 0.01 0.06 0 0 0.033</pose>
      <inertial>
       <mass>0.000002</mass>
       <inertia>
        <ixx>0.001087473</ixx>
        <iyy>0.001087473</iyy>
        <izz>0.001092437</izz>
        <ixy>0</ixy>
        <ixz>0</ixz>
        <iyz>0</iyz>
      </inertia>
    </inertial>
    <collision name='base_collision'>
      <geometry>
        <cylinder>
          <radius>.004267</radius>
          <length>.005867</length>
        </cylinder>
      </geometry>
    </collision>
    <visual name='base_visual'>
      <geometry>
        <cylinder>
```

```
          <radius>.004267</radius>
          <length>.005867</length>
        </cylinder>
      </geometry>
    </visual>
    <sensor type='ray' name='lidar0'>
      <pose>0 0 0 1.5707 0 0</pose>
      <visualize>true</visualize>
      <update_rate>30</update_rate>
      <ray>
        <scan>
        <horizontal>
          <samples>1</samples>
          <resolution>1</resolution>
          <min_angle>0.01</min_angle>
          <max_angle>0.01</max_angle>
         </horizontal>
      </scan>
       <range>
         <min>0.06</min>
          <max>35</max>
          <resolution>0.02</resolution>
        </range>
      </ray>
    <plugin
name='lidar_plugin' filename='libgazebo_lidar_plugin.so'>
      <robotNamespace></robotNamespace>
        <customSubTopic>lidar0</customSubTopic>
        <min_distance>0.2</min_distance>
        <max_distance>15</max_distance>
     </plugin>
    </sensor>
  </link>
</model>
<joint name='lidar0_joint' type='fixed'>
  <pose>0 0 -0.036785 0 0 0</pose>
  <parent>base_link</parent>
  <child>lidar0::base</child>
</joint>
```

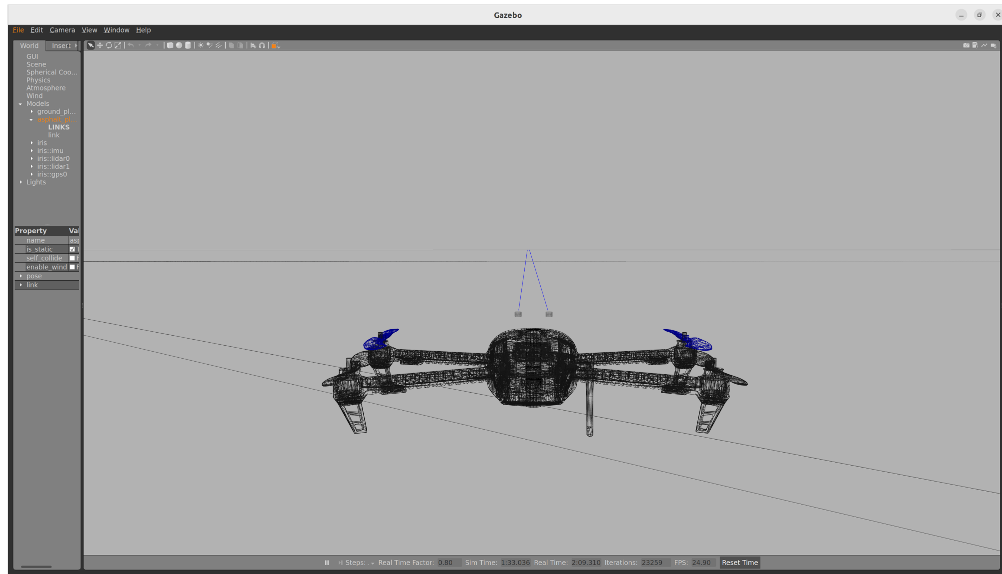



**Figure 3.7:** Model of the iris_hitl after the integration of the 2 distance sensors

## 3.3 Gazebo and PX4 connection

**Figure 3.8:** Chart of the communication lines in the Gazebo SITL simulation

To let the data pass from gazebo to the PX4 firmware, some specific actions need to be performed. First, the data of interest are defined in the plugins on the SDF files. In fact, plugins are software components that extend the functionality of the simulator by adding custom features, behaviours or interactions. So, they interact with the simulation environment, models, sensors or other components of Gazebo to provide additional capabilities. Plugins can be used in Gazebo to establish communication with the PX4 firmware using the MAVLink protocol, a feature that is used in this thesis. These enable the exchange of sensor data, control commands and system status between Gazebo and PX4.

When the simulation starts, each .sdf is analysed by the respective gazebo plugin, which is a c file inside the PX4 firmware, i.e., if we have a gps sensor, the gazebo plugin will be called 'gazebo_gps_plugin'. In each of these files all the data of interest are extracted with the same procedure. The compiler will search for a

certain sequence of word in the .sdf and, if it finds it, it will save the data after that sequence trough the c file. For example, under the plugin line inside the gps.sdf we can find a sequence like:

```
<gpsNoise>1</gpsNoise>
```

To extract this '1' and generate the respective variable, a code like the one below will be implemented inside the c file of the corresponding plugin (gazebo_gps_plugin in the case of this example):

```
// Get noise param
if (_sdf->HasElement("gpsNoise")) {
    getSdfParam<bool>(_sdf, "gpsNoise", gps_noise_,
    gps_noise_);
} else {
    gps_noise_ = false;
}
```

This match will be done for each line inside <plugin> into the .sdf. Then a structure will be created and published as a topic. For every sensor declared inside the iris model, this procedure will be followed.

The 'gazebo_mavlink_interface' module plays a vital role in subscribing to these topics. It uses specific SensorSubscription functions to establish connections with matching published topics. The matching process employs a recursive text pattern of the type:

**" /" + model name + "/link/" + nested sensor name**

Every time a topic is published, or a subscription is attempted, a text like that is created. If the pattern of the subscribed topic matches the published one, the connection is established, and call-back functions associated with the subscribed topics are invoked. These call-backs generate messages with specific structures representing the structure of the specific sensor data. The messages are then sent to MAVLink, which facilitates the transfer of data to the PX4 firmware. In cases where multiple sensors of the same type are used, differentiation can be achieved through an ID assigned by the developer. This ID helps distinguish between different instances of the same sensor type.

Upon receiving the data structure from Gazebo, the 'mavlink_receiver' module comes into play. It processes the received structure and fills the relative Uorb topic. The data is transformed into a new structure that aligns with the PX4 firmware's requirements. This transformed structure is then published to the relevant UORB

topic through the publish() function, enabling other modules within the firmware to subscribe to and utilize the sensor readings.

Now, considering what has been done for this project, all the previous passages will be analysed for the publication of two lidar sensors on the iris_hitl.sdf model.

After the generation of the two lidar sensors .sdf files, I added a line to both the plugins, to distinguishing between the two sensor used:

```
<customSubTopic>lidar0</customSubTopic>
<customSubTopic>lidar1</customSubTopic>
```

To let the data pass, a few lines of code had been added to the 'gazebo_lidar_plugin':

```
// get lidar topic name
if(_sdf->HasElement("topic")) {
    lidar_topic_ = parentSensor_->Topic();
}else if (_sdf->HasElement("customSubTopic")) {
    lidar_topic_= _sdf->GetElement("customSubTopic")->Get<std::
    string>();
}else {
    // if not set by parameter, get the topic name from
    the model name
    lidar_topic_ = parentSensorModelName;
}
```

The match of the topic name is now done also for a different line called "customSubTopic". This means that a new subscription method has been added to the predefined ones, to understand more in depth the mechanism of the communication protochol and how it behaves when using two sensors of the same type. So, I did not use the already implemented 'Lidar_callback' but I generated a new callback from scratch, very similar to the lidar one, called 'Custom_callback'. This function needs to be called twice to pass both the sensor's current values.

The custom_callback publishes then these structures for the MAVLink receiver, that builds automatically the relative uORB topic. In this case, two uORB topic of the same message are created: distance_sensor0 and distance_sensor1. That means that, when a subscription to the distance sensor topic is made, I need to specify which one of the two topics I am going to use.

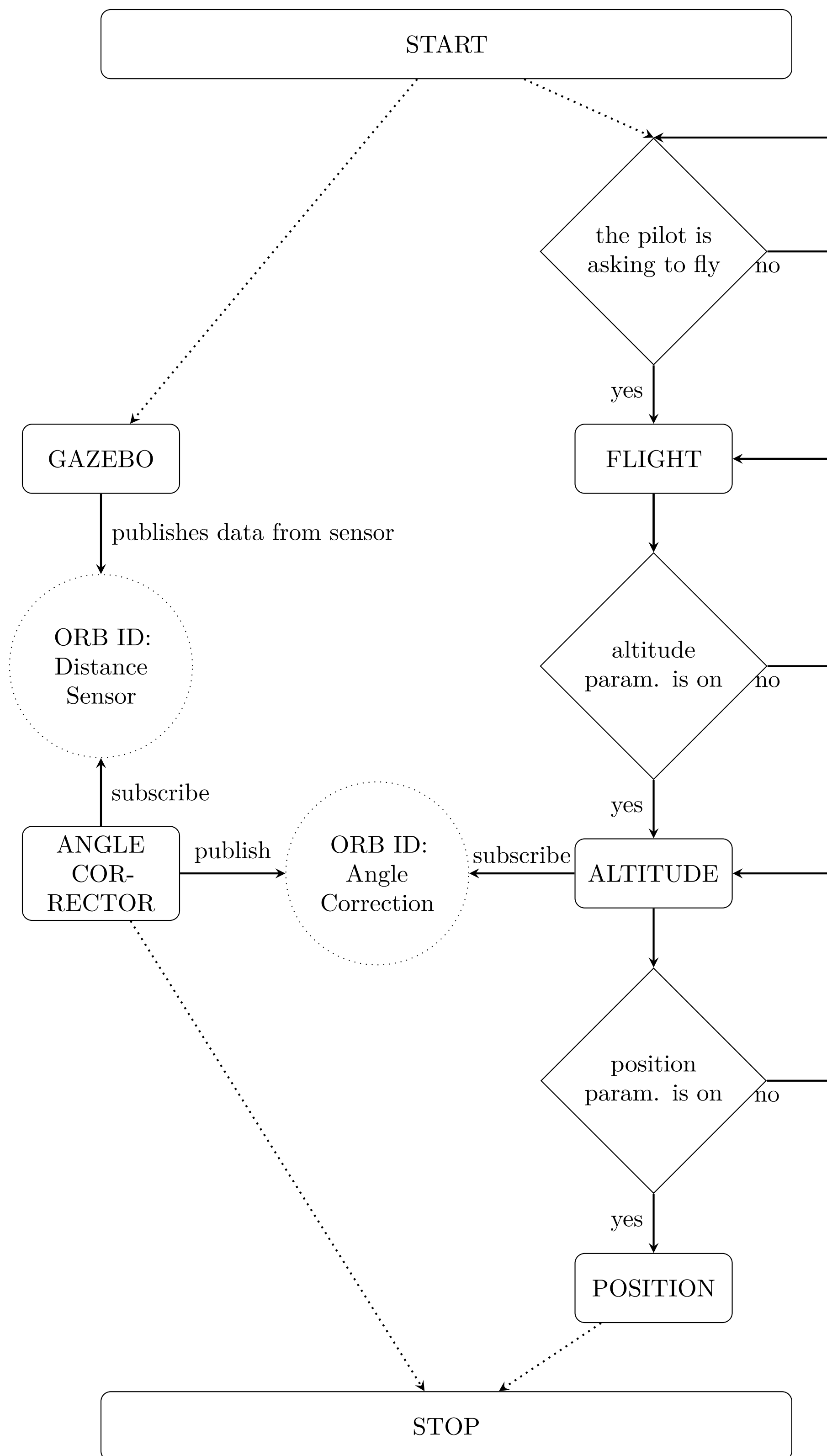
# Chapter 4

# Custom Control Algorithm, "Horizontal-Lock"

The final object of this project is to develop a control system that aligns the drone with a wall and locks its position vertically and horizontally. To do that a deep understanding of the PX4 autopilot system and its communication protochol is needed and this is explained in the previous chapters. The aim of this chapter is to explain and analyze the code sections that implements the control. Before doing that, an overview of the entirety of the modules and how they interact with each other is needed to understand better the passages done to reach the final conclusion.

The various modules already present in the original code implement all the various functionalities of the drone. These are primarily written in the C/C++ programming languages which are a common choice for embedded systems and real-time applications like drone autopilots because of their efficiency and low-level control over hardware. As already seen in the previous chapters, they communicate with each other using a publish/subscribe mechanism, messages, function calls or parameters. The modules that will be mostly used are the ones which implement the flight modes, like altitude or position and a custom driver.

Before beginning explaining the code in all its parts, a little overview of the environment is necessary to have a better understanding of the system. The flow chart below illustrates the various phases of the control algorithm, neglecting the parts not related to that due to problems of dimensions and difficulty. in fact, the processes represented in the graph are ran in parallel with many other ones, which carry out the most diverse operations for maintaining the system.


START
the pilot is asking to fly
no
yes
GAZEBO
FLIGHT
publishes data from sensor
ORB ID: Distance Sensor
altitude param. is on
no
yes
subscribe
ANGLE COR-RECTOR
publish
ORB ID: Angle Correction
subscribe
ALTITUDE
position param. is on
no
yes
POSITION
STOP

As can be seen from the chart, there are three main processes/modules that I developed and modified: angle corrector, flight task altitude and flight task position. Those will be fully explained in the next sections. The last two modules are represented on the right part of the chart. Going down from "FLIGHT", the modules are dependent on the previous ones. So, "POSITION" is dependent from "ALTITUDE" and "FLIGHT" since those generate outputs needed for the module to maintain itself. These implement the algorithm to let the drone fly in different modes, respectively manual, altitude and position modes. Instead, "ANGLE CORRECTOR" is an independent module that generates a value called angle correction that represents an orientation angle. It runs in parallel to the other three and publishes this value in the Angle_correction topic, a topic that as been created for the purpose of this thesis. The subscription to the distance_sensor topic (see chapter 3) and the generation of angle correction could have been done directly on the module of the altitude mode. However, having a process that runs in parallel is better in terms of computing time and availability of the data. As a matter of fact, putting this series of operations directly into this module could cause a slowdown due to those additional operations and the arrival of an outdated data to the output, thus dephasing the drone.

In this project, I will use a kind of variable called “setpoint”. This encapsulate a desired value/state that the flight controller should strive to reach. The control system then will adjust the actuators to maintain these states. So, this is a very advantageous variable, since it manages in a very simple way abstract complex control commands into user-friendly references for the vehicle’s behavior, simplifying the interaction pilot-drone. To confirm this, many flight modes are based on the use of setpoints to define certain types of behavior. There are various type of setpoints that can refer to position, velocity and many other relevant control variables. The most important and the more used ones in the project are:

- **Position setpoints**: defines the coordinates, with respect to a global reference system, that the UAV should maintain. It is a three-element vector in which each element represents, in order, the x, y and z axes.
- **Yaw setpoints**: typically expressed as an angle in radians, defines the direction that the drone should point at.

## 4.1 Definition of the Orientation Angle

As said in the previous paragraph, the purpose of this algorithm is to control the drone and to lock it accordingly to predefined criteria. Those are the alignment of the UAV to the wall, meaning that the drone’s head has to point perpendicularly

to the wall, and the lock of the distance from the wall. A setpoint in the z axis will also be generated, but the current height can always be modified through RC inputs. That means that the drone has only two degrees of freedom. The z and the y axis, that lead the drone to move vertically, and sideways through roll angle modification.

The first choice for the type of sensor was a UWB. That would have scanned the whole area in front of the drone and calculate a distance from any object through the returning waves. However, the generation of the sdf file of the UWB was unclear and difficult to interpret. The choice of the two lidar sensor come up to simplify the process of data acquisition. As can be seen in the chapter 3, lidar sensors are very easy to implement and generate an output of linear distance from an object without the need to implement a conversion algorithm, like for the UWB. The only disadvantage is that the lidar sensors used for this scope are linear, so the acquisition of the distance can happen only with respect to one object/wall. However, this was a forced choice, since the linearity of the sensors makes much more easier the acquisition of the orientation angle with respect to the object/wall.

The first thing to do was to analyse the measures of the two sensor and generate a yaw angle using the difference of the current distances. This is one of the most important parts of the algorithm. In fact, when the drone requests a distance lock, it first needs to be correctly aligned with the wall/obstacle. If not, when the lock happens, the drone can crash on the wall while moving sideways.

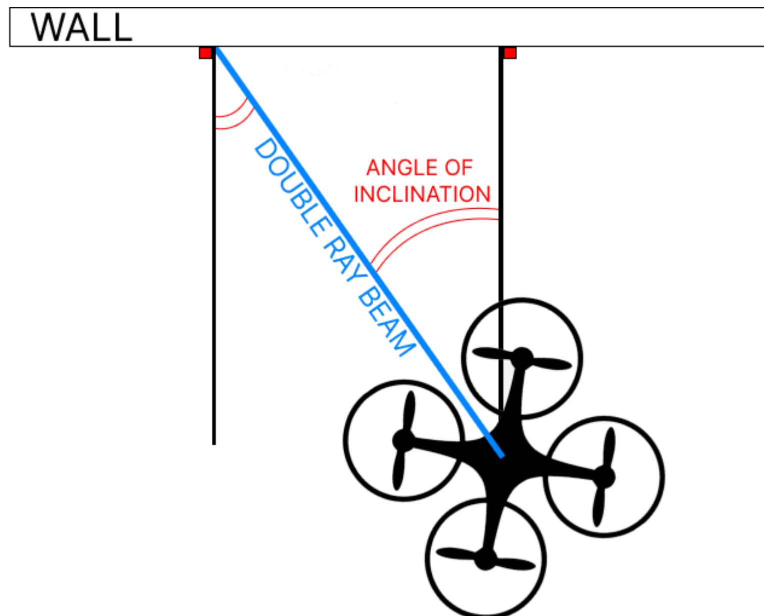



**Figure 4.1:** Handwritten scheme to understand a possible configuration drone-wall

So, the first action that the algorithm has to perform is the definition of a yaw angle correction. To do that a new module has been created and added to the PX4 source code. This is called "AngleCorrector" and is located in the ‘drivers’ folder of the firmware. The structure of the module is very simple. It has a constructor and a destructor for the class. We’ll put in the constructor any possible advertise() function, necessary when publishing a topic. The main functions are:

- **Init()**: initializes the module and sets it to run at a scheduled interval of 10 milliseconds.
- **Run()**: is the main execution loop of the module. It checks for the program exit condition, updates sensor data from subscribers and performs file I/O operations.
- **Task_spawn()**: creates an instance of the class, initializes it and returns the result.
- **SensorCheck()**: it generates the angle of the misalignment.

The remaining functions handle the printing of the module status, the custom commands and the usage information, and serve as entry point for the module.

In the .h file, the subscriptions to the two distance sensors are made:

```
uORB::Subscription
_distance_sensor_left_sub{ORB_ID(distance_sensor),0};
uORB::Subscription
_distance_sensor_right_sub{ORB_ID(distance_sensor),1};
```

The two new generated structures are called, respectively, _distance_sensor_left_struct and _distance_sensor_right_struct. A distance sensor struct is composed of the following components:

```
struct distance_sensor_s {

    uint64_t timestamp;
    uint32_t device_id;
    float min_distance;
    float max_distance;
    float current_distance;
    float variance;
    float h_fov;
```

```
    float v_fov;
    float q[4];
    int8_t signal_quality;
    uint8_t type;
    uint8_t orientation;
    uint8_t _padding0[1];
}
```

The variable of interest is current_distance, that gives the actual distance from the corresponding sensor to the first point detected (in this case the wall). The angle is generated by the SensorCheck() function, that is called inside the run() loop. This will take as inputs the 2 distance sensors structs. The calculation of the angle is relatively simple. I drew a possible configuration of the system drone-wall and, through trigonometry, I obtained a general formula.

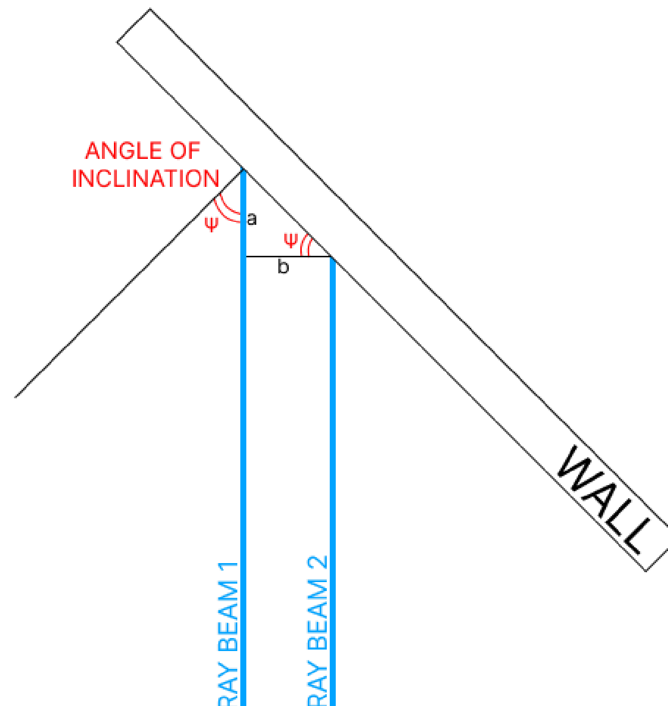



**Figure 4.2:** A possible sensors-wall configuration used to calculate the generic formula for the angle of inclination

$$angle_correction = arctan\left(\frac{a}{b}\right) \tag{4.1}$$

Where:

- **'a'** is the difference between the two current distances. It is negative if the right beam is longer then the left one, meaning that the drone is oriented with an angle opposite to the one of the figures.
- **'b'** is a fixed value. It indicates the distance between the centers of the two laser beams. Its value is 4cm. It has been calculated following the scheme in the figure below.

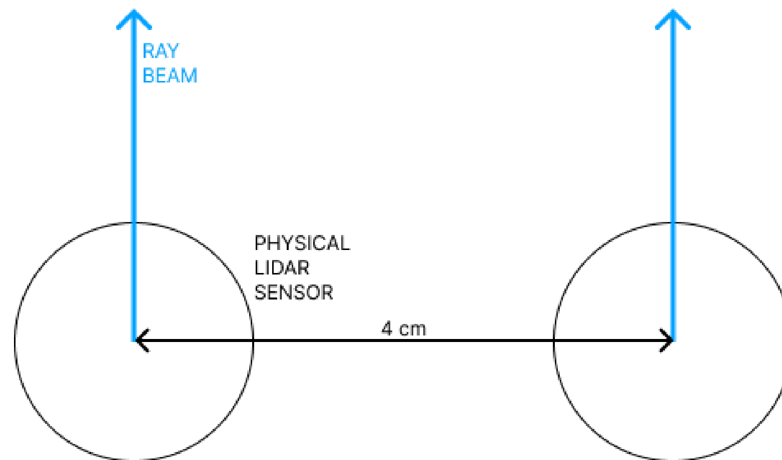



**Figure 4.3:** From above, physical scheme of the sensors placed on top of the drone

After the description of the parts of the code, how they work and the results they obtain, here is a flow chart that explain how the algorithm of the angle corrector module works in terms of flow of operations.

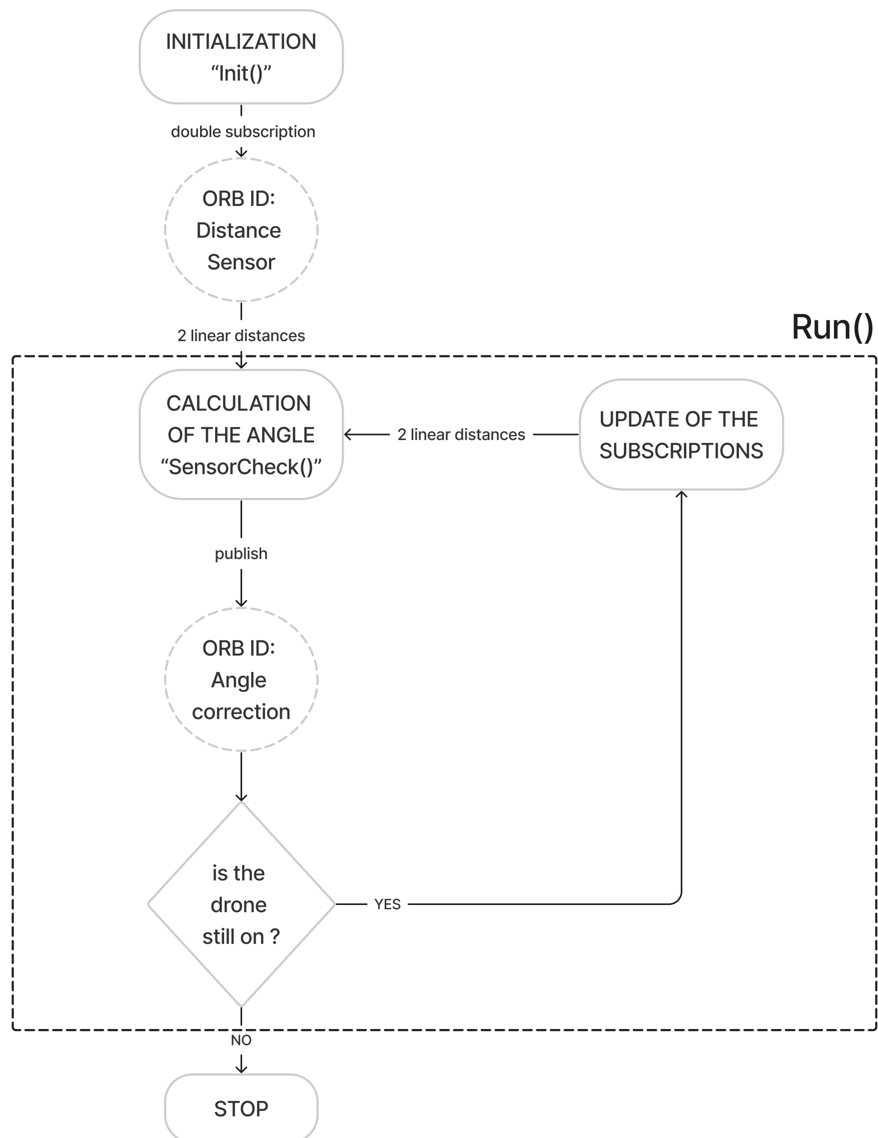



**Figure 4.4:** Flow chart of the angle correction algorithm

## 4.2 Angle_correction Topic

Since the control algorithm is implemented in a different module, I created a new topic called angle_correction in order to make that measure available for any module. To do that I followed the guidelines on the PX4 user guide. First, I created an angle_correction.msg file in the msg/ directory. The file is composed of the following lines:

```
uint64 timestamp          # time since system start
(microseconds)

float32 yaw_correction    # yaw value to be used to correct
the setpoint (rad)
```

Those are all the variables that are going to form the structure of this new topic. Then I added the file name to 'msg/CMakeLists.txt' that contains the list of all the topics' messages. Recompiling the project will trigger the creation of the C/C++ files needed for the topic. Then I included the new topic in the module I was developing and published it inside the SensorCheck() function.

```
angle_correction.timestamp = hrt_absolute_time();
angle_correction.yaw_correction = angle_inclination;
_angle_corr_pub.publish(angle_correction);
```

## 4.3 Creation of the Flight Task Modules

This control algorithm works in altitude and position mode. When the altitude flight task is enabled, the drone will maintain the height it was flying at, without maintaining the throttle input in the RC. This mechanism works using a position setpoint for the z axis. Every time the altitude task is triggered, the module of the task will save the actual height inside this setpoint, and the drone will move maintaining this value until a new throttle input is received. In that case, the setpoint will be updated with respect to how much throttle is given by the pilot, both positive and negative ones.

The position task, instead, will generate setpoints in the x directions, so that the drone remains still with respect to the wall without inputs from the joystick.

The angle algorithm I made is implemented inside the altitude flight task. Since I did not want to modify the original code for the two modules and having so a

backup in case of inappropriate modifications, I created a copy of these and called them DistanceHoldAltitudeAltitude and DistanceHoldAltitudePosition. However, these new modules need to be inserted inside the flight mode manager in order to be activated.

To do that, a parameter needs to be modified inside the 'mc_pos_control_params.c' file, where all the parameters used by the PX4 source code are present. The one relative to my objective is 'MPC_POS_MODE'. This is defined as an 'int' and contains a maximum of 5 different values (it counts also the value 0). Each value is associated to a specific sub-mode which is a combination of altitude and position tasks with different behaviours. For example, for the value 3, the altitude and the position modes will be smooth. The new flight mode I added is activated with the value 2 and is called DistanceHold.

```
/**
 * Manual-Position control sub-mode
 *
 * @value 0 Simple position control
 * @value 2 DistanceHold position control
 * @value 3 Smooth position control (Jerk optimized)
 * @value 4 Acceleration based input
 * @group Multicopter Position Control
 */
PARAM_DEFINE_INT32(MPC_POS_MODE, 4);
```

'Mpc_pos_mode' is changed directly inside QGC. Go to 'Vehicle Setup/Parameters', search for the corresponding parameter and select DistanceHold. In that way, every time the altitude mode is active, it will always refer to the DistanceHold sub-mode.

Now, I will show the procedure first for the Altitude mode, then I will present the Position one, but in less details, since the procedure is the same. In the directory that contains all the various flight modes, I made a new folder called DistanceHoldAltitude, and inserted there the copied 'FlightTaskManualAltitude.cpp' and .h (renamed as the folder). Then I created a new 'CMakeFile.txt' to define where this task is placed and interconnected with the others. The file contains this text:

```
px4_add_library(FlightTaskDistanceHoldAltitude
        FlightTaskDistanceHoldAltitude.cpp
)
target_link_libraries(FlightTaskDistanceHoldAltitude
    PUBLIC
    FlightTask FlightTaskUtility)
```

```
target_include_directories(FlightTaskDistanceHoldAltitude
    PUBLIC ${CMAKE_CURRENT_SOURCE_DIR})
```

The first 3 lines add a library target named FlightTaskDistanceHoldAltitude and specify the source file 'FlightTaskDistanceHoldAltitude.cpp' that will be compiled to create the library. The other two functions do link the FlightTaskDistance-HoldAltitude library with the FlightTask and FlightTaskUtility libraries (first function) and add the current source directory to the include directories for the FlightTaskDistanceHoldAltitude target (second function).

In the flight task mode manager are expressed all the criteria for the various tasks to activate. In my case, the altitude and the position tasks, as seen above, change their behaviour with respect to the number held by the 'mpc_pos_mode' parameter. Inside the function 'start_flight_task()' in the 'FlightModeManager.cpp' I added a new case for the parameter equal to 2 that will enable the distance hold as altitude mode.

```
if(_vehicle_status_sub.get().nav_state==vehicle_status_s::
NAVIGATION_STATE_ALTCTL || task_failure){
    should_disable_task = false;
    FlightTaskError error = FlightTaskError::NoError;

    switch (_param_mpc_pos_mode.get()) {
    case 0:
        error = switchTask(FlightTaskIndex::ManualAltitude);
    break;

    case 2:
        error =
        switchTask(FlightTaskIndex::DistanceHoldAltitude);
    break;

    case 3:
    default:
        error =
        switchTask(FlightTaskIndex::ManualAltitudeSmoothVel;
    break;
}
```

For the position mode we can repeat the same procedure, but a few small modifications need to be applied. When creating the new folder, the name now will be

DistanceHoldPosition. The CMakeFile.txt will have the following structure:

```
px4_add_library(FlightTask DistanceHoldPosition
    FlightTask DistanceHoldPosition.cpp
    FlightTask DistanceHoldPosition.hpp
)

target_link_libraries(FlightTask DistanceHoldPosition
        PUBLIC
        FlightTask DistanceHoldAltitude
)

target_include_directories(FlightTask DistanceHoldPosition
    PUBLIC ${CMAKE_CURRENT_SOURCE_DIR})
```

Instead, in the flight mode manager, I added another case as did for the altitude task:

```
if (_vehicle_status_sub.get().nav_state ==
vehicle_status_s::NAVIGATION_STATE_POSCTL || task_failure) {
    should_disable_task = false;
    FlightTaskError error = FlightTaskError::NoError;

    switch (_param_mpc_pos_mode.get()) {
    case 0:
        error = switchTask(FlightTaskIndex::ManualPosition);
    break;

    case 2:
        error =
        switchTask(FlightTaskIndex::DistanceHoldPosition);
    break;

    case 3:
        error =
        switchTask(FlightTaskIndex::ManualPositionSmoothVel);
    break;

    case 4:
    default:
        if (_param_mpc_pos_mode.get() != 4) {
```

```
            PX4_ERR("MPC_POS_MODE %" PRId32 " invalid,
            resetting", _param_mpc_pos_mode.get());

            _param_mpc_pos_mode.set(4);
            _param_mpc_pos_mode.commit();
        }
        error =
        switchTask(FlightTaskIndex::ManualAcceleration);
    break;
```

That means that when the 'mpc_pos_mode' parameter is equal to 2, both the altitude and position tasks of the DistanceHold sub-mode will be activated.

## 4.4 The Altitude Algorithm

The aim of this module is to control the vehicle's altitude above ground level by generating and maintaining the z setpoint, relying on sensors such as barometers to determine the altitude. This is a very useful mode since it simplifies the piloting process, allowing operators to focus on horizontal navigation and mission-specific tasks without constantly adjusting the drone's altitude.

The behaviour of this algorithm will now be described. To give a quick description of it, can be said that, when the module is activated, it will get the current height the drone is flying at and save it inside the _position_setpoint(2) variable. Then this variable will be subjected to few functions that will analyse the correctness and operability of this value. Below there is a flow chart to have an overview of all the component functions and how they interact each other.

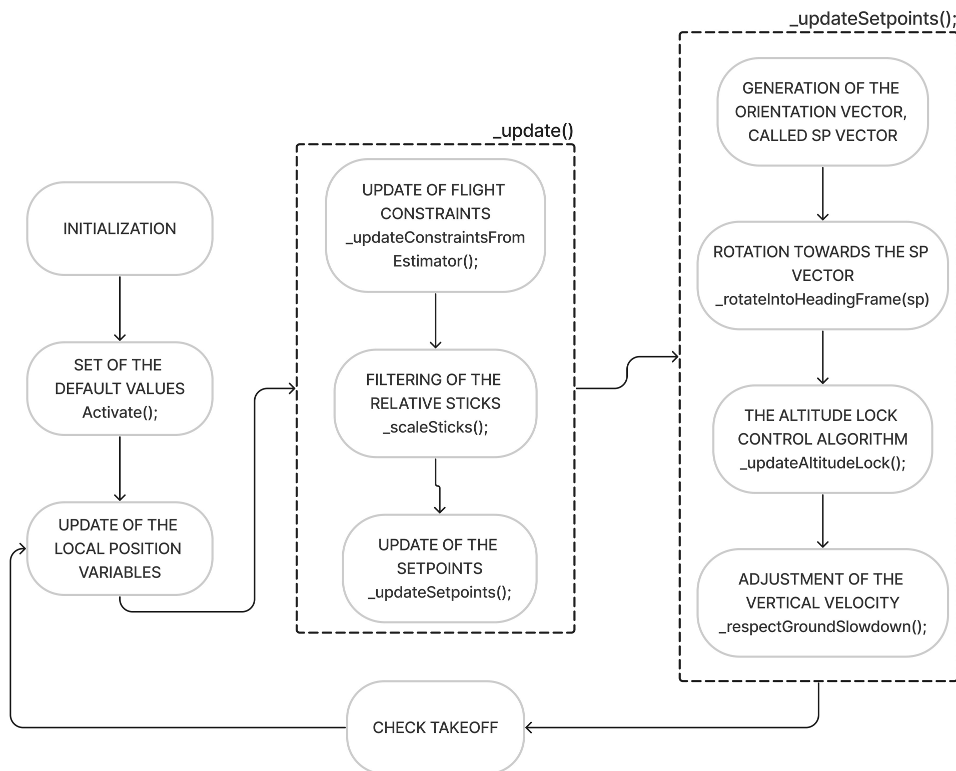



**Figure 4.5:** Operating scheme of the altitude algorithm

As can be seen from the chart, the module is a loop that updates certain values and controls the height through designed functions. When the altitude task is enabled, through command or parameters, a preliminary initialization of all the variables and the check of the validity of local position needs to be done. Then, the first function to be called is activate(). This sets to default all the setpoints (position, velocity, yaw, and acceleration), the constrains (speed up/down, max/min distance to ground) and the sticks inputs. These steps are very important for the maintenance of the system, since initializing variables to a known state before any processing occurs is critical to prevent undefined behavior, unexpected results, or errors caused by unexpected data. In other words, generates a null initial state which will be the basis for the generation of subsequent states. Having uninitialized values as the first state, and therefore not containing any value, would cause problems in the calculations of the subsequent ones and the crash of the system.

The main function of the loop is the _update() which is likely responsible for updating the state of the altitude control performing recursive function-calls. Here are, in order, the functions that are called going through update():

**_updateConstraintsFromEstimator()**: it updates the constraints of minimum and maximum distance to ground.

**Algorithm 1** _updateConstraintsFromEstimator()

```
if hagl_min (minimum height above ground level) exist then
    _min_distance_to_ground = hagl_min;
else
    _min_distance_to_ground = - infinity;
if hagl_max (maximum height above ground level) exist then
    _max_distance_to_ground = hagl_max;
else
    _max_distance_to_ground = + infinity;
// these are useful constraints to set how much higher and lower the drone can move during the flight
```

**_scaleSticks()**: This function scales the stick inputs to determine the desired yaw speed and vertical velocity setpoints. The yaw speed is low-pass filtered helping to smooth out rapid changes and provides a more stable control output.

**Algorithm 2** _scaleSticks()

```
                    // filters stick yaw and throttle inputs to avoid deadzone
1 yawspeed_target = yaw stick position;
2 _yawspeed_setpoint = filtered yawspeed_target;
3 _velocity_setpoint(2) = bounded throttle stick position;
```

**_updateSetpoints()**: This function is responsible of calling all the needed updates. Before, a new vector object called 'sp' is defined:

```
Vector2f sp(_sticks.getPosition().slice<2, 1>(0, 0));
_man_input_filter.setParameters(_deltatime,
         _param_mc_man_tilt_tau.get());
_man_input_filter.update(sp);
sp = _man_input_filter.getState
```

Sp contains two variables. Those are extracted from the stick object, which is a vector of 4 variables that contains the inputs from sticks of pitch, roll, throttle and yaw. getPosition() will get all the 4, while .slice<2, 1>(0, 0) will extract only the first two, roll and pitch. So the 'sp' vector will contain the two input variables of pitch and roll. The next lines of code defines an object, man_input_filter, through

the parameters DELTATIME and MC_MAN_TILT_TAU and update its state through sp. The vector sp updates using the state of the filter. This represents the filtered output of the joystick sticks. This is an important procedure, in fact joysticks and other input devices can produce noisy signals due to imperfections in the totality of the system. Also, unfiltered joystick inputs can lead to oscillations or instability in the vehicle's response. Filtering helps smooth out these noisy signals, providing more stable system and reliable control inputs.

Inside the _updateSetpoints() all these functions will be called:

**_rotateIntoHeadingFrame()**: takes as input the 2D vector sp, and it is responsible of aligning the vector with the vehicle's current heading.

**Algorithm 3** _rotateIntoHeadingFrame()

```
   // this function uses the yaw or the yaw setpoint if available to modify the
   sp vector (which point only in x and y) with respect to the current heading of
   the drone
 4 INPUT: V[2]                                          // this is the sp vector
 5 if _yaw_setpoint is finite then
 6 |  yaw_rotate = _yaw_setpoint;
 7 else
 8 |  yaw_rotate = _yaw;
                            // the vector v_r is a 3D vector while the V only 2D
 9 Vector v_r[3] = [0 , 0 , yaw_rotate] * [V(0), V(1), 0.0f];
10 V(0) = v_r(0);
11 V(1) = v_r(1);
```

**_updateAltitudeLock()**: in this function will be implemented the core of the control algorithm. To reach the final objective, the funtion will use two variables called 'apply_brake' and 'stopped' to check the drone movement status. Those are calculated in that way:

```
- const bool apply_brake =
        fabsf(_sticks.getPositionExpo()(2)) <= FLT_EPSILON;
```

This variable is used to check if the user wants to brake. The code returns a boolean value on 'apply_brake' comparing two quantities: _sticks.getPositionExpo()(2) and FLT_EPSILON. The former represent the input thrust from the joystick, the later is instead a fixed value. If the input given by the stick to move along the z-axis is less or equal to a certain small value FLT_EPSILON, it means that the pilot is asking to brake and the variable will be set to true.

```
- const bool stopped=(_param_mpc_hold_max_z.get()
    < FLT_EPSILON || fabsf(_velocity(2)
    < _param_mpc_hold_max_z.get())
```

This variable is used to check if the vehicle stopped through a double condition. The functions uses 3 main data: The parameter MPC_HOLD_MAX_Z, that defines the maximum vertical velocity for which position hold is enabled, the actual velocity in z direction (_velocity(2)) and FLT_EPSILON, which is the same parameter as before. It checks whether the parameter MPC_HOLD_MAX_Z is lower than FLT_EPSILON or the actual vertical velocity is lower than MPC_HOLD_MAX_Z. If one of these condition is true it means the vehicle has a vertical velocity so low that the vehicle can be considered stopped.

The algorithm has 3 different behaviours depending on which value the parameter MPC_ALT_MODE is set to. The one used in this project is equal to '0' corresponding to the 'Altitude following' mode, that controls the height with respect to the earth frame origin. For simplicity, only this behaviour will be analyzed. However, in the appendix A the full code will be provided.

This part of the code is composed of an if structure with 3 different conditions (one excludes the others). In the algorithm 5 is provided the structure and the outcome of each condition.

This algorithm works checking the physical status of the drone and sets the setpoints accordingly. Each condition, in fact, corresponds to a different operative situation of the UAV. It sets the z position setpoint to the actual height if it has not been set yet and if the vehicle is stopped. Practically this happens when we release the throttle stick for the first time after the activation. Then it checks if the distance to the bottom is valid comparing it to the min_distance_to_ground constraint. If true, the function _terrainFollowing() will be called.

Here is an overview of how this function works.

**Algorithm 4** _TerrainFollowing()

```
// This function is responsible of maintaining the altitude lock to
the minimum distance to the ground (constraint). It will use a varible called
_dist_to_ground_lock that contains the height to ground at which the UAV will
be locked
12 if apply_brake = 1 && stopped = 1 && _dist_to_ground_lock is not defined then
       // User wants to break and vehicle reached zero velocity. Lock height to
       ground
13     z_position_setpoint = z_actual_position;
14     if _dist_to_bottom is finite && (_dist_to_bottom <
       _min_distance_to_ground) then
15         z_position_setpoint = z_actual_position - (_min_distance_to_ground -
           _dist_to_bottom);
       // lock distance to ground but adjust first for minimum altitude
16     _dist_to_ground_lock = _dist_to_bottom - ( z_position_setpoint -
       z_actual_position );
17 else if apply_brake = 1 && _dist_to_ground_lock is already defined then
       // in this case the vehicle needs to follow terrain. It calculates
       difference between the current distance to ground and the desired distance
       to ground and adjust position setpoint for the delta
18     delta_distance_to_ground = _dist_to_ground_lock - _dist_to_bottom;
19     z_position_setpoint = z_actual_position - delta_distance_to_ground;
20 else
21     do not lock in altitude;
```

This function will maintain the minimum altitude constraints while the drone is flying. So, if we fly below that threshold, the algorithm will set this contraint as a setpoint, forcing the drone to return at that height. This is a useful constraint since it prevent the drone from running into obstacles while flying very close to the ground in uneven terrain. It's basically as if we set a safe distance from ground.

The other condition, instead, checks if the setpoint is already set and if the user is demanding to brake. The algorithm will check if a reset event happened, if true, the z setpoint will be updated.

The last condition actives when the user is demanding a velocity change. The z setpoint will be set to Nan. A check on the altitude will also be done, ensuring it does not overcome the maximum achievable height. This is done through the _respectMaxAltitude() function, which works very similarly to _terrainFollowing(), with the only exception that the used constraint will be now referring to the

maximum height above the ground.

**Algorithm 5** _updateAltitudeLock()

```
    // this is the main algorithm of the altitude module, the one that evaluates
    the distance to ground and sets the relative z setpoint
22 if apply_brake = 1 && stopped = 1 && z setpoint is not defined then
23     z_position_setpoint = z_actual_position;
24     if distance to bottom < _min_distance_to_ground then
25         _terrainFollowing(...);
26     else
27         do not lock in altitude;
28 else if apply_brake = 1 && z setpoint is already defined then
29     _reset_counter = vehicle_local_position → z_reset_counter;
                                        // checks if a reset event has happened
30     if vehicle_local_position.z_reset_counter != _reset_counter then
31         z_position_setpoint = z_actual_position;
32         _reset_counter = vehicle_local_position.z_reset_counter;
                   /* updates the value of _reset_counter to the most recent one
           z_reset_counter in order to avoid a reset also on the next states */
33 else
34     do not lock in altitude;
35     _respectMaxAltitude();
```

**_respectGroundSlowdown()**:This function is responsible for adjusting the vertical velocity setpoint based on the distance to the ground (_dist_to_ground) and specific altitude parameters. It first calculates an ascend rate limit_up and a descent rate limit_down and then, trough the function below, ensures that the vertical velocity stays within the specified limits based on the distance to the ground:

```
velocity_setpoint(2) = math::constrain(_velocity_setpoint(2),
    -limit_up, limit_down)
```

The last action performed is the update of another constraint called 'want_takeoff' through the _checkTakeoff() function, that will do a check on the throttle stick input. If the input throttle is over a certain value, then the user is asking to take off.

## 4.5 The Angle Algorithm

Since the algorithm to obtain the physical angle has already been described, in this paragraph will be held a description on how those measures are used inside the firmware. The code will always be implemented inside the DistanceHoldAltitude module, so that when the altitude mode is selected, the drone will automatically align to the wall. To do that I deleted the previous yaw control implemented inside the original altitude module and substituted it with my custom one, but still the two are very similar one another. Here is how it insert in the altitude flow chart:

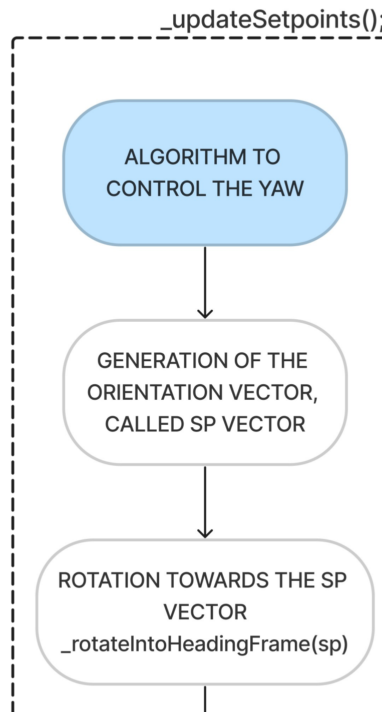



**Figure 4.6:** Where the angle algorithm is added in the altitude operating scheme

In the flow chart are represented the main processes that compose the algorithm.

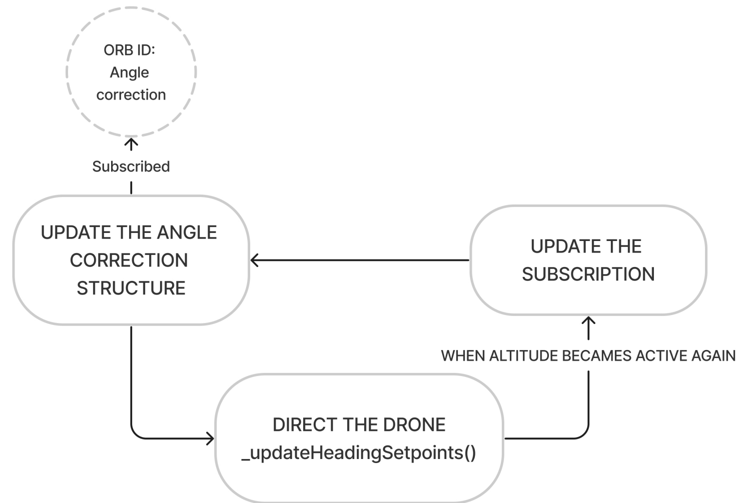



**Figure 4.7:** Operating scheme of the angle algorithm

First, the angle correction is defined. We do that through the subscription to the angle_correction topic:

```
uORB::Subscription _angle_corr_sub{ORB_ID(angle_correction)};
```

All the data will be stored inside the dedicated structure angle_correction_struct. The core of this algorithm is the _updateHeadingSetpoints() function. This is called inside the _updateSetpoints() function, right after the update of the angle_correction structure which is also passed as an argument of the function. The body of the function is:

**Algorithm 6** _updateHeadingSetpoints()

INPUT: yaw_correction
**if** *_yaw_setpoint is not set yet* **then**
⌊ yaw_setpoint = _yaw + _yaw_correction;

That says that if the yaw setpoint has not been set yet, the new setpoint will be given by the sum of the actual yaw the drone is heading at and the yaw needed for

the correction. Since the yaw is referring to a fixed and global reference of system, the actual yaw can be interpreted as the difference between the initial position (yaw = 0 rad.) and the actual position. For example, if the actual yaw is $\pi$ rad. (or 180° degrees) the UAV points in the reverse direction of the initial one.

Every time the module is activated for the first time, the setpoint of the yaw is set to Nan. After entering the loop, the new setpoint will be given updating the old one. However, due to the "if condition", this update will be done only once, increasing the chances of error in the calculations.

## 4.6 The Position Algorithm

As has been done for the altitude task, it is developed inside a new copied Position task module, which is directly connected to the altitude one. If the drone is flying in altitude and a change of mode in position is requested, the altitude will not be turned off, but it will continue to be active, independently from the position mode. This maintenance of the altitude mode is done through recursive functions that call the activate and the update of the task. The former is called inside the activate() function of the position mode:

```
bool ret = FlightTaskDistanceHoldAltitude::activate(last_setpoint);
```

Since the altitude setpoints need to be updated too, inside the update of the position a function similar to the one above will be called to update the yaw and the z-direction setpoints:

```
FlightTaskDistanceHoldAltitude::_updateSetpoints();
```

The aim of the position mode is to set the setpoints for the x and y axis too. In that way, if the drone is flying and the sticks of the RC are released, the drone will save the position in which this event happened and put the measures into the relative setpoints, so that the drone will stay still in this position. This module can be easily modified to work only in the x direction, leaving the y setpoint always set to NAN. In that way the drone will be free to move sideways, but it will be controlled in the x and z directions.

How the module works, and its parts will now be described. When we start the module for the first time, some preliminary operations will be done. The first operation that the algorithm does is to check if the actual position is valid or not. This is done inside the updateInitialize() function, that will also update the setpoints of the altitude mode for the first time:

```
bool ret = FlightTaskDistanceHoldAltitude::updateInitialize();
```

If everything is verified, the next function to be enabled is the activate(). It will turn on the DistanceHoldAltitude task for the altitude, as shown above, and then update the x-axis position and velocity setpoints respectively to the actual position and to zero.

The core of the module is the _updateSetpoints() function. This is the responsible of the updating of the altitude task and of the horizontal lock through the _updateXlock(). This is structured in this way:

```
const float vel_xy_norm = Vector2f(_velocity).length();
const bool apply_brake = _velocity_setpoint (0) < FLT_EPSILON;
const bool stopped =
    (_param_mpc_hold_max_xy.get() < FLT_EPSILON || vel_xy_norm
    < _param_mpc_hold_max_xy.get());
```

This part defines three variables. The first one is called 'vel_xy_norm' and it gives a norm between x and y velocities. The other two variables are called 'apply_brake' and 'stopped' and their meaning is equal to those used in the altitude task. However, the first one is defined in a different way, not using the sticks input, but the velocity setpoint in x.

**Algorithm 7** _updateXLock()

```
    // this is the main algorithm of the position module, the one that evaluates
    the distance to the wall and sets the relative x setpoint
36 if apply_brake = 1 && stopped = 1 && x setpoint is not defined then
37 |   x_position_setpoint = x_actual_position;
38 else if apply_brake = 1 && x setpoint is already defined then
   |                                           // checks if a reset event has happened
39 |   if vehicle_local_position.x_reset_counter != _reset_counter then
40 |   |   x_position_setpoint = x_actual_position;
41 |   |   _reset_counter = vehicle_local_position.x_reset_counter;
   |   |               /* updates the value of _reset_counter to the most recent one
   |   |     x_reset_counter in order to avoid a reset also on the next states */
42 else
43 |   do not lock in x;
```

It is visible that this algorithm is very similar to the one implemented inside the altitude module(_updateAltitudeLock()). The first condition is activated when

the vehicle is stopped and the setpoint hasn't been set yet. It sets the position setpoint equal to the actual position of the drone. Instead, if the setpoint has been already defined and the user is asking to apply brake, the setpoint will be updated if there is a reset condition verified. If none of the other is active, the setpoint will be set to NAN, since the user is asking to move the drone in another position.

# Chapter 5

# Testing: Results and Troubleshooting

In this chapter I will list the main problems I encountered during the development of the algorithm and the results I achieved.

## 5.1 Problems in the Position Mode

The main issue I encountered during my work regards the position task. Even though the code for the horizontal position lock has been developed, I did not have any way to test it in the gazebo-QGroundControl environment. For the task to be activated, some mandatory conditions need to be verified. The main one regards the GPS positioning system, which the drone I am using does not have. In fact, to enable the position mode the system requires a detailed information about the actual position of the drone. This information is not provided in the configuration of the drone I used, which uses a combination o IMUs and Barometers to estimate the current position. This problem does not involve only my sub-mode "DistanceHold", but it is extended to all the sub-modes of the MPC_POS_MODE parameter of the original source code.

A solution could have been to develop the horizontal lock in the same module where the altitude task is developed: 'FlightTaskDistanceHoldAltitude.cpp'. However, during the test phase, the drone was not able to start flying in altitude mode and, when switching from manual to altitude mode in flight, the drone would stop working too, falling to the ground. The problem regarding this is the same as the previous one. When an xy-plane setpoint is being defined, the system requires a

valid actual position of the drone.

## 5.2 Problems in the Angle Measurement

The major issue of this part of the control is the measurement error on the computation of the angle correction. As seen, this value is computed starting from the lidar sensors placed on the top of the drone. So, it depends on the accuracy of the .sdf file. However, when we simulate a flight and we activate the altitude, the angle is not very precise. This depends on various factors.

First, the conversion of the distance into an angle does not give a specific angle value, but always rounds the results to a finite set of values. For example, 12.5° and 15.3° degrees will be rounded into 0° and 19° degrees each. The precision of the angle can be enhanced by acting directly on the gazebo_lidar_plugin module, where the data from the .sdf are extracted. In this file, the order of magnitude of the measurements of the sensors can be modified, by multiplying x10,x100 or x1000 the extracted values. This leads to a current distance expressed on mm, 0.1mm or 0.01mm, depending on the multiplication factor. Doing that, the angle will have a wider and more precise range of values. However, decreasing the order of magnitude leads to an increase of the computational time making the simulation slower. A good compromise is obtained by using the data in mm, leading to a bit larger set of values and to a good simulation environment.

Another issue is the instability of the drone in the altitude mode. This may be caused by the use of the joystick instead of the RC. Below, a picture shows this wrong behaviour. In the first picture, the inputs from the joystick are represented, but only for the x/y axis. When the sticks are released, there will be an offset. The other one shows the physical result of this offset, leading to a movement in both x and y directions, making the drone move indefinitely when it should be still in its position. This will cause an error in the computation of the angle since the sensor can take a wrong measurement due to these continuous movement of the drone. Also, this makes it more difficult to control the drone in simulation and for the algorithm testing.

When the altitude task is activated, the angle control will be done only once, increasing the chances of moving with a wrong angle correction. However, this is the most performant solution that I obtained. In fact, rapidly disabling and enabling the altitude mode, will recalculate the angle leading to a better measure. Doing this for 2 or 3 times will almost ensure a correct alignment with the wall.

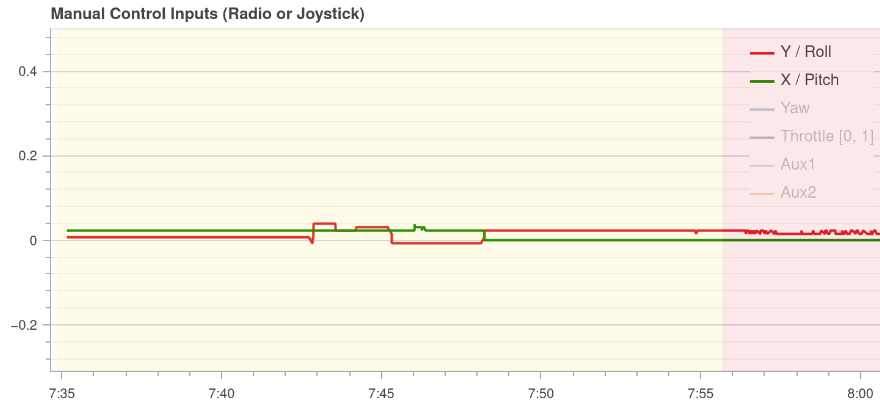



**Figure 5.1:** Manual control inputs for X/Pitch (in green) and Y/Roll (in red) over time

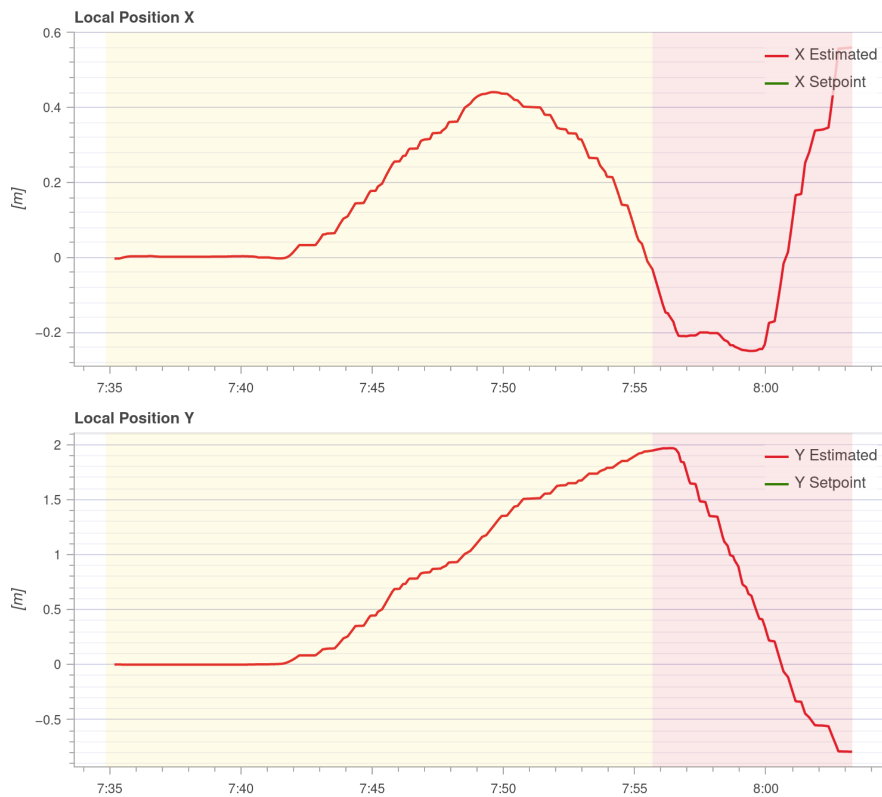



**Figure 5.2:** Diagrams of the local position of the drone in the X and Y axis over time

## 5.3 Measurements and Results

Due to the problems listed above, the control was almost always receiving wrong measurements, correcting the yaw in a wrong way, or started rotating indefinitely. A lot of different solutions had been implemented during the development phase. An example was a solution in which the control remains active, always adjusting the angle accordingly, if no yaw input from the joystick is provided. Instead, if input is provided the control is locked, and the drone can move the angle smoothly.

The developed code is provided below:

```
if(!PX4_ISFINITE(_yaw_setpoint) || !isYawInput()){
    _yaw_setpoint = _yaw + angle_correction_handle.yaw_correction;
}
```

This leads to a completely wrong behaviour, where the drone starts rotating indefinitely. This may be caused by the same problem analyzed before of the joystick inputs offset. As can be seen from the picture, the input is never zero, and this conflicts with the isYawinput() function, that will be always active, leading the algorithm to set another setpoint everytime.

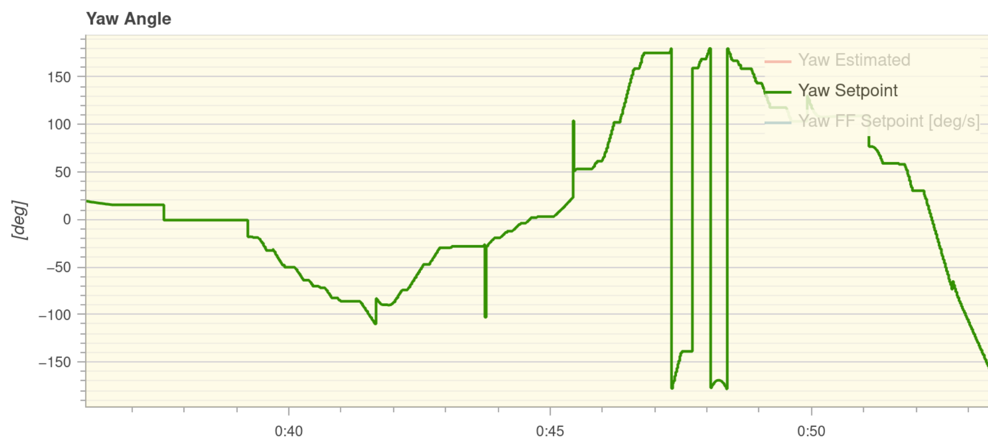



**Figure 5.3:** Diagram of the yaw_setpoint over time for a wrong algorithm

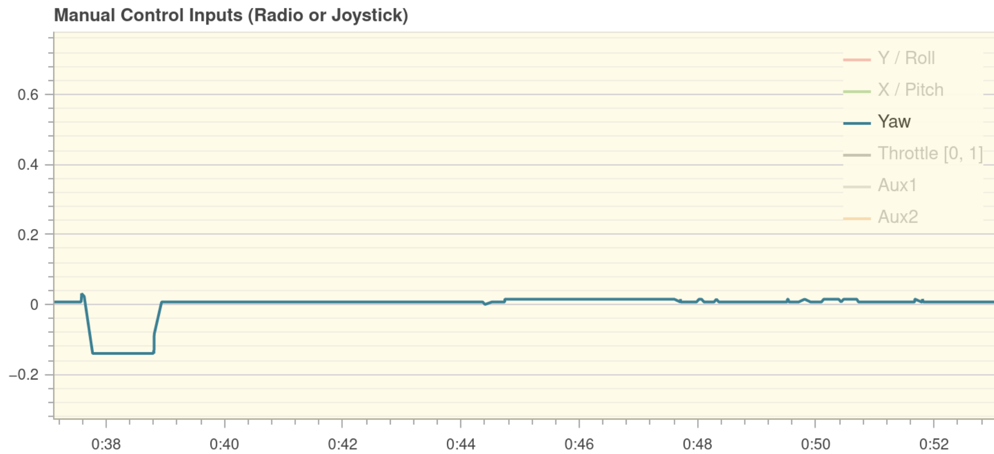



**Figure 5.4:** Yaw input from the joystick over time

Using the solution proposed in chapter 4, leads to a good, but not perfect, angle control. As can be seen in the picture, when calling for the first time the altitude (yellow zone) the drone does not align perfectly but has some degrees of error. Disabling and re-enabling the mode leads to a better angle, almost perfectly aligned with the target wall. The advantage of using the setpoint is that even though we apply a high yaw as input, the system will autonomously return to the setpoint. This is shown in the picture below, where the red line shows the actual heading of the drone.

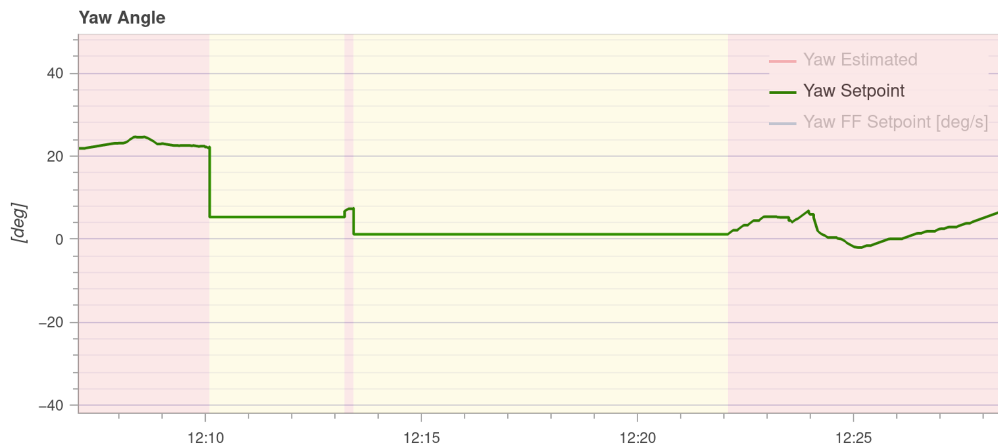



**Figure 5.5:** Yaw_setpoint over time, in red the drone is flying in manual, in yellow in altitude

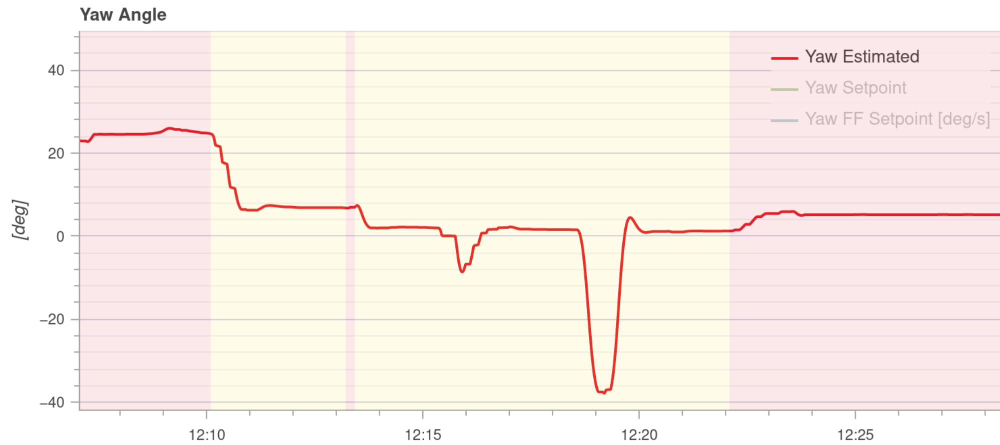



**Figure 5.6:** Yaw angle over time. This refers to current heading of he drone and it is due yaw inputs from joystick

Some issues arise when changing the actual setpoint. The environment where the drone is moving is formed of 4 perpendicular walls, as said in the previous chapters. To change the target to a near wall, the drone should perform a 90° degrees rotation and set the setpoint again taking this into account. During the simulation, I started from a 0° degrees setpoint, then entered in manual and rapidly changed the orientation of the drone. As can be seen from the picture, when I activate the altitude again, heading to the new wall, the setpoint is not very precise, but has an error of at least 5/6° degrees. As stated before, doing a quick re-calibration leads to an improved angle, which is almost 90° degrees.

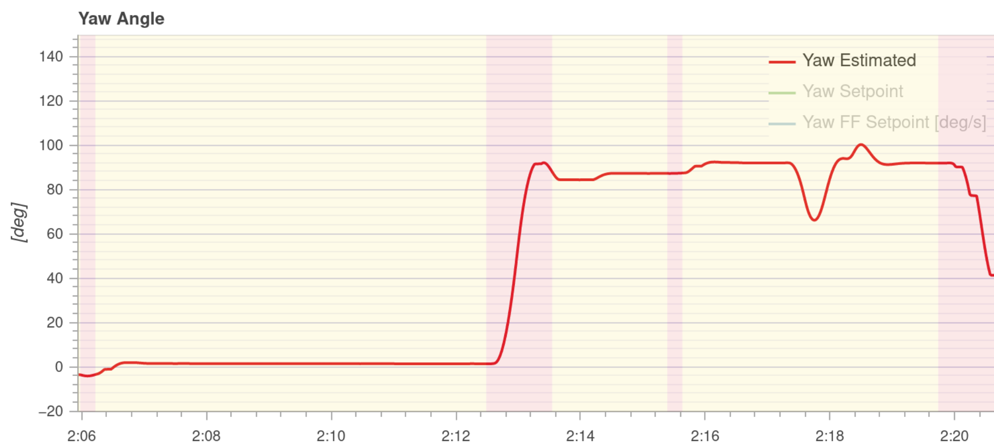



**Figure 5.7:** Yaw angle over time

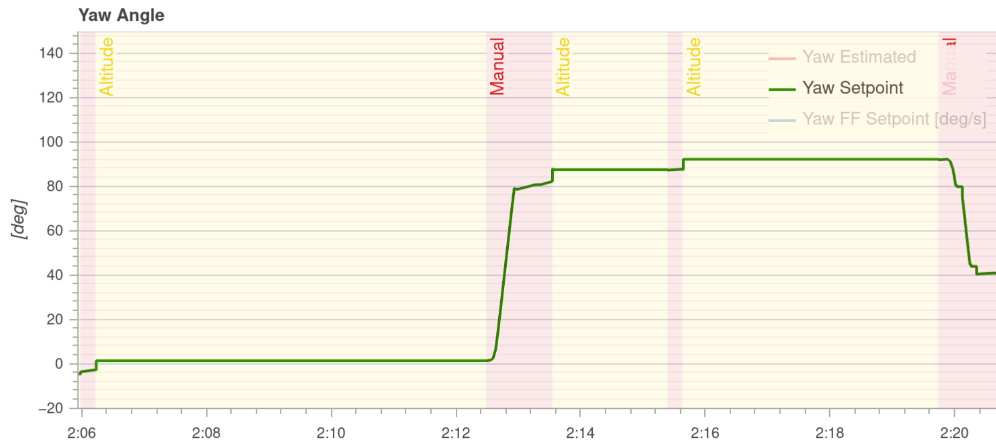



**Figure 5.8:** Yaw_setpoint over time

# Chapter 6

# Conclusions

This thesis has been developed to test and verify the effectiveness of a custom positioning algorithm inside the original PX4 autopilot firmware. The controller that has been created is good for what concern the orientation of the drone with an accuracy of about 5 degrees. However, to reach this accuracy, the drone should be calibrated from 2 to 3 times. This of course is a less autonomous system, but it can be enhanced adding some new features. One thing that I think could have been useful was the use of a direct button from the joystick to enable the calibration, to avoid the continuous enable/disable of the altitude to get a new calibration. Nonetheless, after the setpoint has been set, the drone moves in very smooth and controllable way.

Also, the algorithm is not fully tested, due to some problems regarding the position mode. So, the part about the distance lock has been implemented through code, but I didn't have the resources and knowledge to enable this task in flight. In fact, the position mode requires strict information of the actual position of the drone. This could have been achieved effortlessly by using a GPS device, but the drone I used was supposed to fly in GPS-denied environment, so it uses a combination of barometers and IMUs to get the position information. Those are less precise than the GPS ones, leading to a mismatch of the position activation conditions.

Despite those difficulties encountered during the end phase of the development, this project's contributions extend beyond the domain of positioning algorithms, dealing with the understanding and use of the PX4 system and of many internal functionalities of it. As a matter of fact, a deep analysis of the architecture and of the control system has been provided, with a particular focus on the mechanism of the communication system.

Also, the creation of custom environments and sensors had a great importance for

the development of my work. Indeed, the design and integration of the .sdf files into the original PX4 source code had a significant role to build and modify the world environment and equip the drone with different sensors for simulation and testing purposes.

# Appendix A

# Codes

## A.1 AngleCorrector.cpp

```
#include "AngleCorrector.hpp"

#include <fcntl.h>
#include <unistd.h>
#include <px4_platform_common/getopt.h>
#include <px4_platform_common/log.h>
#include <px4_platform_common/posix.h>

AMode::AMode() :
ModuleParams(nullptr),
ScheduledWorkItem(MODULE_NAME, px4::wq_configurations::test1)
{
_angle_corr_pub.advertise();
}

AMode::~AMode()
{
}

bool AMode::init()
{
PX4_INFO("INITIALIZATION");
ScheduleOnInterval(10000);
```

```
return true;
}

void AMode::Sensorcheck(
     distance_sensor_s distance_sensor_left_handle,
     distance_sensor_s distance_sensor_right_handle)
{
    float distance_diff = distance_sensor_left_handle.current_distance
    - distance_sensor_right_handle.current_distance;

    float angle_inclination = atan2f(distance_diff, sensor_spacing);

    angle_correction.timestamp = hrt_absolute_time();
    angle_correction.yaw_correction = angle_inclination;
    _angle_corr_pub.publish(angle_correction);
}

void AMode::Run()
{
    if (should_exit()) {
        ScheduleClear();
        exit_and_cleanup();
        return;
    }

    _distance_sensor_left_sub.update(&_distance_sensor_left_struct);
    _distance_sensor_right_sub.update(&_distance_sensor_right_struct);

int AMode::task_spawn(int argc, char *argv[])
{
    AMode *instance = new AMode();

    if (instance) {
        _object.store(instance);
        _task_id = task_id_is_work_queue;

        if (instance->init()) {
            return PX4_OK;
        }

    } else {
```

```
        PX4_ERR("alloc failed");
    }

    delete instance;
    _object.store(nullptr);
    _task_id = -1;

    return PX4_ERROR;
}

int AMode::print_status()
{
    PX4_INFO("Running");

    return 0;
}

int AMode::custom_command(int argc, char *argv[])
{
    return 0;
}

int AMode::print_usage(const char *reason)
{
    if (reason) {
        PX4_WARN("%s\n", reason);
    }
    PRINT_MODULE_DESCRIPTION(
    R"DESCR_STR(
    ### Description
    Angle Mode!
    )DESCR_STR");

    PRINT_MODULE_USAGE_NAME("AngleCorrector_main", "driver");
    PRINT_MODULE_USAGE_DEFAULT_COMMANDS();

    return 0;
}

extern "C" __EXPORT int AngleCorrector_main(int argc, char *argv[])
{
```

```
    return AMode::main(argc, argv);
}
```

## A.2 AngleCorrector.hpp

```
#pragma once

#include <commander/px4_custom_mode.h>
#include <drivers/drv_hrt.h>
#include <mathlib/mathlib.h>
#include <matrix/matrix/math.hpp>
#include <px4_platform_common/module_params.h>
#include <systemlib/mavlink_log.h>
#include <uORB/topics/distance_sensor.h>
#include <uORB/topics/angle_correction.h>
#include <uORB/SubscriptionMultiArray.hpp>
#include <uORB/Publication.hpp>
#include <uORB/PublicationMulti.hpp>
#include <uORB/Subscription.hpp>
#include <uORB/SubscriptionCallback.hpp>
#include <px4_platform_common/defines.h>
#include <px4_platform_common/module.h>
#include <px4_platform_common/module_params.h>
#include <px4_platform_common/posix.h>
#include <px4_platform_common/events.h>
#include <px4_platform_common/px4_work_queue/ScheduledWorkItem.hpp>



class AMode : public ModuleBase<AMode>, public ModuleParams,
    public px4::ScheduledWorkItem
{

public:
    AMode();
    ~AMode() override;

    /** @see ModuleBase */
    static int task_spawn(int argc, char *argv[]);

    /** @see ModuleBase */
    static int custom_command(int argc, char *argv[]);

    /** @see ModuleBase */
```

```
    static int print_usage(const char *reason = nullptr);

    bool init();

    double b;
    float sensor_spacing = 40f; // Distance between the
        two sensor rays in mms

    /** @see ModuleBase::print_status() */
    int print_status() override;

private:
    uORB::Publication<angle_correction_s>
            _angle_corr_pub{ORB_ID(angle_correction)};

    angle_correction_s angle_correction{};

    uORB::Subscription
            _distance_sensor_left_sub{ORB_ID(distance_sensor),0};
    uORB::Subscription
            _distance_sensor_right_sub{ORB_ID(distance_sensor),1};

    struct distance_sensor_s _distance_sensor_left_struct;
    struct distance_sensor_s _distance_sensor_right_struct;

    void Sensorcheck(distance_sensor_s,distance_sensor_s);
    void Run() override;
};
```

## A.3 FlightTaskDistanceAltitude.cpp

```
/**
* @file FlightTaskDistanceAltitude.cpp
*/

#include "FlightTaskDistanceAltitude.hpp"
#include <float.h>
#include <mathlib/mathlib.h>
#include <geo/geo.h>

using namespace matrix;

FlightTaskDistanceAltitude::FlightTaskDistanceAltitude() :
    _sticks(this)
{}

bool FlightTaskDistanceAltitude::updateInitialize()
{
    bool ret = FlightTask::updateInitialize();
    _sticks.checkAndUpdateStickInputs();

    if (_sticks_data_required) {
        ret = ret && _sticks.isAvailable();
    }

    return ret && PX4_ISFINITE(_position(2)) &&
            PX4_ISFINITE(_velocity(2)) && PX4_ISFINITE(_yaw);
}

bool FlightTaskDistanceAltitude::activate(const
        trajectory_setpoint_s &last_setpoint)
{
    bool ret = FlightTask::activate(last_setpoint);
    _yaw_setpoint = NAN;
    _yawspeed_setpoint = 0.f;
    _acceleration_setpoint = Vector3f(0.f, 0.f, NAN);
    _position_setpoint(2) = _position(2);
    _velocity_setpoint(2) = 0.f;
    _setDefaultConstraints();
```

```
    _updateConstraintsFromEstimator();

    return ret;
}

void FlightTaskDistanceAltitude::_updateConstraintsFromEstimator()
{
    if (PX4_ISFINITE(_sub_vehicle_local_position.get().hagl_min)) {
        _min_distance_to_ground =
                _sub_vehicle_local_position.get().hagl_min;
    } else {
        _min_distance_to_ground = -INFINITY;
    }

    if (PX4_ISFINITE(_sub_vehicle_local_position.get().hagl_max)) {
        _max_distance_to_ground =
                _sub_vehicle_local_position.get().hagl_max;
    } else {
        _max_distance_to_ground = INFINITY;
    }
}

void FlightTaskDistanceAltitude::_scaleSticks()
{
    const float yawspeed_target = _sticks.getPositionExpo()(3) *
            math::radians(_param_mpc_man_y_max.get());
    _yawspeed_setpoint = _applyYawspeedFilter(yawspeed_target);

    const float vel_max_z = (_sticks.getPosition()(2) > 0.0f) ?
            _param_mpc_z_vel_max_dn.get() :
    _param_mpc_z_vel_max_up.get();
    _velocity_setpoint(2)= vel_max_z * _sticks.getPositionExpo()(2);
}

float FlightTaskDistanceAltitude::_applyYawspeedFilter(
        float yawspeed_target)
{
    const float den = math::max(_param_mpc_man_y_tau.get() +
            _deltatime, 0.001f);
    const float alpha = _deltatime / den;
    _yawspeed_filter_state = (1.f - alpha) * _yawspeed_filter_state
```

```
            + alpha * yawspeed_target;
    return _yawspeed_filter_state;
}

void FlightTaskDistanceAltitude::_updateAltitudeLock()
{
    const bool apply_brake = fabsf(_sticks.getPositionExpo()(2))
            <= FLT_EPSILON;

    const bool stopped = (_param_mpc_hold_max_z.get() < FLT_EPSILON
            || fabsf(_velocity(2)) < _param_mpc_hold_max_z.get());

    if (_param_mpc_alt_mode.get() == 2) {

        float spd_xy = Vector2f(_velocity).length();

        float stick_xy = Vector2f(_sticks.getPositionExpo().slice<2,
                1>(0, 0)).length();
        bool stick_input = stick_xy > 0.001f;

        if (_terrain_hold) {
            bool too_fast = spd_xy > _param_mpc_hold_max_xy.get();

            if (stick_input || too_fast ||
                    !PX4_ISFINITE(_dist_to_bottom)) {

                _terrain_hold = false;
                _terrain_follow = false;

                if (PX4_ISFINITE(_dist_to_ground_lock) &&
                        PX4_ISFINITE(_dist_to_bottom)) {
                    _position_setpoint(2) = _position(2) -
                        (_dist_to_ground_lock - _dist_to_bottom);

                } else {
                    _position_setpoint(2) = _position(2);
                }
            }
        } else {
            bool not_moving = spd_xy < 0.5f *
                _param_mpc_hold_max_xy.get();
```

```
        if (!stick_input && not_moving &&
                PX4_ISFINITE(_dist_to_bottom)) {

            _terrain_hold = true;
            _terrain_follow = true;

            if (PX4_ISFINITE(_position_setpoint(2))) {
                _dist_to_ground_lock = _dist_to_bottom -
                    (_position_setpoint(2) - _position(2));
            }
        }
    }

}

if ((_param_mpc_alt_mode.get() == 1 || _terrain_follow) &&
            PX4_ISFINITE(_dist_to_bottom)) {
        _terrainFollowing(apply_brake, stopped);
        _respectMaxAltitude();

} else {

    if (apply_brake && stopped &&
            !PX4_ISFINITE(_position_setpoint(2))) {

        _position_setpoint(2) = _position(2);

        if (PX4_ISFINITE(_dist_to_bottom) && _dist_to_bottom <
                _min_distance_to_ground) {
            _terrainFollowing(apply_brake, stopped);

        } else {
            _dist_to_ground_lock = NAN;
        }

    } else if (PX4_ISFINITE(_position_setpoint(2)) &&
            apply_brake) {

        if (_sub_vehicle_local_position.get().z_reset_counter
                != _reset_counter) {
```

```
                _position_setpoint(2) = _position(2);
                _reset_counter =
                    _sub_vehicle_local_position.get().z_reset_counter;
            }

        } else  {
            _position_setpoint(2) = NAN;

            _respectMaxAltitude();
        }
    }
}

void FlightTaskDistanceAltitude::_respectMinAltitude()
{
    if (PX4_ISFINITE(_dist_to_bottom) && (_dist_to_bottom <
            _min_distance_to_ground)) {

        _position_setpoint(2) = _position(2) -
                (_min_distance_to_ground - _dist_to_bottom);
    }
}

void FlightTaskDistanceAltitude::_terrainFollowing(bool apply_brake,
        bool stopped)
{
    if (apply_brake && stopped &&
            !PX4_ISFINITE(_dist_to_ground_lock)) {

        _position_setpoint(2) = _position(2);
        _respectMinAltitude();
        _dist_to_ground_lock = _dist_to_bottom -
                (_position_setpoint(2) - _position(2));

    } else if (apply_brake && PX4_ISFINITE(_dist_to_ground_lock)) {

        const float delta_distance_to_ground = _dist_to_ground_lock
                - _dist_to_bottom;

        _position_setpoint(2) = _position(2) -
                delta_distance_to_ground;
```

```
    } else {

        _dist_to_ground_lock = _position_setpoint(2) = NAN;
    }
}

void FlightTaskDistanceAltitude::_respectMaxAltitude()
{
    if (PX4_ISFINITE(_dist_to_bottom)) {

        if (PX4_ISFINITE(_max_distance_to_ground)) {
            _constraints.speed_up =
                    math::constrain(_param_mpc_z_p.get() *
                    (_max_distance_to_ground - _dist_to_bottom),
                    -_param_mpc_z_vel_max_dn.get(),
                    _param_mpc_z_vel_max_up.get());

        } else {
            _constraints.speed_up = _param_mpc_z_vel_max_up.get();
        }

        if (_dist_to_bottom > _max_distance_to_ground) {

            const float delta_distance_to_max = _dist_to_bottom
                    - _max_distance_to_ground;

            _position_setpoint(2) = _position(2) +
                    delta_distance_to_max;
        // limit speed downwards to 0.7m/s
            _constraints.speed_down =
                math::min(_param_mpc_z_vel_max_dn.get(), 0.7f);

        } else {
            _constraints.speed_down =
                    _param_mpc_z_vel_max_dn.get();
        }
    }
}

void FlightTaskDistanceAltitude::_respectGroundSlowdown()
```

```
{
// Interpolate descent rate between the altitudes MPC_LAND_ALT1
        and MPC_LAND_ALT2
    if (PX4_ISFINITE(_dist_to_ground)) {
        const float limit_down = math::interpolate(
                _dist_to_ground, _param_mpc_land_alt2.get(),
                _param_mpc_land_alt1.get(),
                _param_mpc_land_speed.get(),
                _constraints.speed_down);
        const float limit_up = math::interpolate(_dist_to_ground,
                 _param_mpc_land_alt2.get(),
                 _param_mpc_land_alt1.get(),
                 _param_mpc_tko_speed.get(),
                 _constraints.speed_up);
        _velocity_setpoint(2) =
                math::constrain(_velocity_setpoint(2), -limit_up,
                limit_down);
    }
}

void FlightTaskDistanceAltitude::_rotateIntoHeadingFrame(Vector2f &v)
{
    const float yaw_rotate = PX4_ISFINITE(_yaw_setpoint) ?
            _yaw_setpoint : _yaw;
    Vector3f v_r = Vector3f(Dcmf(Eulerf(0.0f, 0.0f, yaw_rotate)) *
            Vector3f(v(0), v(1), 0.0f));
    v(0) = v_r(0);
    v(1) = v_r(1);
}

bool FlightTaskDistanceAltitude::_isYawInput()
{
/*
* A threshold larger than FLT_EPSILON is required because the
* _yawspeed_setpoint comes from an IIR filter and takes too much
* time to reach zero.
*/
    return fabsf(_yawspeed_setpoint) > 0.001f;
}

void FlightTaskDistanceAltitude::_ekfResetHandlerHeading(float
```

```
        delta_psi)
{

    if (PX4_ISFINITE(_yaw_setpoint)) {
        _yaw_setpoint += delta_psi;
    }
}

void FlightTaskDistanceAltitude::_updateSetpoints()
{
    _angle_corr_sub.update(&angle_correction_struct);
    _updateHeadingSetpoints(angle_correction_struct);

// Thrust in xy are extracted directly from stick inputs. A
magnitude of 1 means that maximum thrust along xy is demanded. A
magnitude of 0 means no   thrust along xy is demanded. The maximum
thrust along xy depends on the thrust setpoint along z-direction,
which is computed in PositionControl.cpp.

    Vector2f sp(_sticks.getPosition().slice<2, 1>(0, 0));

    _man_input_filter.setParameters(_deltatime,
            _param_mc_man_tilt_tau.get());
    _man_input_filter.update(sp);
    sp = _man_input_filter.getState();
    _rotateIntoHeadingFrame(sp);

    if (sp.longerThan(1.0f)) {
        sp.normalize();
    }

    _acceleration_setpoint.xy() = sp *
            tanf(math::radians(_param_mpc_man_tilt_max.get())) *
            CONSTANTS_ONE_G;

    _updateAltitudeLock();
    _respectGroundSlowdown();
}

bool FlightTaskDistanceAltitude::_checkTakeoff()
{
```

```
// stick is deflected above 65% throttle (throttle stick is in
        the range [-1,1])
    return _sticks.getPosition()(2) < -0.3f;
}

bool FlightTaskDistanceAltitude::update()
{
    bool ret = FlightTask::update();
    _updateConstraintsFromEstimator();
    _scaleSticks();
    _updateSetpoints();
    _constraints.want_takeoff = _checkTakeoff();

    return ret;
}

void FlightTaskDistanceAltitude::_updateHeadingSetpoints(
        angle_correction_s angle_correction_handle)
{
    if(!PX4_ISFINITE(_yaw_setpoint)){
        _yaw_setpoint = _yaw +
                angle_correction_handle.yaw_correction;
    }
}
```

## A.4 FlightTaskDistanceAltitude.hpp

```
/**
* @file FlightTaskDistancePosition.cpp
*/

#pragma once

#include "FlightTask.hpp"
#include "Sticks.hpp"
#include <lib/mathlib/math/filter/AlphaFilter.hpp>
#include <uORB/topics/angle_correction.h>
#include <uORB/Subscription.hpp>
#include <uORB/SubscriptionCallback.hpp>

class FlightTaskDistanceAltitude : public FlightTask
{

public:
    FlightTaskDistanceAltitude();
    virtual ~FlightTaskDistanceAltitude() = default;
    bool activate(const trajectory_setpoint_s &last_setpoint)
            override;
    bool updateInitialize() override;
    bool update() override;

protected:
    void _updateHeadingSetpoints(angle_correction_s); /**< sets
            yaw or yaw speed */
    void _ekfResetHandlerHeading(float delta_psi) override; /**<
            adjust heading setpoint in case of EKF reset event */
    virtual void _updateSetpoints(); /**< updates all setpoints */
    virtual void _scaleSticks(); /**< scales sticks to velocity
            in z */
    bool _checkTakeoff() override;
    void _updateConstraintsFromEstimator();

/**
* rotates vector into local frame
*/
    void _rotateIntoHeadingFrame(matrix::Vector2f &vec);
```

```
/**
*  Check and sets for position lock.
*  If sticks are at center position, the vehicle
*  will exit velocity control and enter position control.
*/
    void _updateAltitudeLock();

    Sticks _sticks;
    bool _sticks_data_required = true;

    DEFINE_PARAMETERS_CUSTOM_PARENT(FlightTask,
    (ParamFloat<px4::params::MPC_HOLD_MAX_Z>) _param_mpc_hold_max_z,
    (ParamInt<px4::params::MPC_ALT_MODE>) _param_mpc_alt_mode,
    (ParamFloat<px4::params::MPC_HOLD_MAX_XY>)
            _param_mpc_hold_max_xy,
    (ParamFloat<px4::params::MPC_Z_P>) _param_mpc_z_p, /**<
            position controller altitude propotional gain */
    (ParamFloat<px4::params::MPC_MAN_Y_MAX>) _param_mpc_man_y_max,
            /**< scaling factor from stick to yaw rate */
    (ParamFloat<px4::params::MPC_MAN_Y_TAU>) _param_mpc_man_y_tau,
    (ParamFloat<px4::params::MPC_MAN_TILT_MAX>)
            _param_mpc_man_tilt_max, /**< maximum tilt allowed for
            manual flight */
    (ParamFloat<px4::params::MPC_LAND_ALT1>) _param_mpc_land_alt1,
            /**< altitude at which to start downwards slowdown */
    (ParamFloat<px4::params::MPC_LAND_ALT2>) _param_mpc_land_alt2,
            /**< altitude below which to land with land speed */
    (ParamFloat<px4::params::MPC_LAND_SPEED>)
    _param_mpc_land_speed, /**< desired downwards speed when
            approaching the ground */
    (ParamFloat<px4::params::MPC_TKO_SPEED>)
    _param_mpc_tko_speed, /**< desired upwards speed when still
            close to the ground */
    (ParamFloat<px4::params::MC_MAN_TILT_TAU>) _param_mc_man_tilt_tau
   )

private:
    bool _isYawInput();

/**
```

```
* Filter between stick input and yaw setpoint
*
* @param yawspeed_target yaw setpoint desired by the stick
* @return filtered value from independent filter state
*/
    float _applyYawspeedFilter(float yawspeed_target);

/**
* Terrain following.
* During terrain following, the position setpoint is adjusted
* such that the distance to ground is kept constant.
* @param apply_brake is true if user wants to break
* @param stopped is true if vehicle has stopped moving in D-direction
*/
    void _terrainFollowing(bool apply_brake, bool stopped);

/**
* Minimum Altitude during range sensor operation.
* If a range sensor is used for altitude estimates, for
* best operation a minimum altitude is required. The minimum
* altitude is only enforced during altitude lock.
*/
    void _respectMinAltitude();

    void _respectMaxAltitude();

/**
* Sets downwards velocity constraint based on the distance to ground.
* To ensure a slowdown to land speed before hitting the ground.
*/
    void _respectGroundSlowdown();

    void setGearAccordingToSwitch();

    float _yawspeed_filter_state{}; /**< state of low-pass filter
            in rad/s */
    uint8_t _reset_counter = 0; /**< counter for estimator resets
            in z-direction */
    bool _terrain_follow{false}; /**< true when the vehicle is
            following the terrain height */
    bool _terrain_hold{false}; /**< true when vehicle is
```

```
            controlling height above a static ground position */

    float _min_distance_to_ground{(float)(-INFINITY)}; /**< min
            distance to ground constraint */
    float _max_distance_to_ground{(float)INFINITY};  /**< max
            distance to ground constraint */

/**
* Distance to ground during terrain following.
* If user does not demand velocity change in D-direction and the
*vehcile is in terrain-following mode, then height to ground will
*be locked to _dist_to_ground_lock.
*/
    float _dist_to_ground_lock = NAN;

    AlphaFilter<matrix::Vector2f> _man_input_filter;

    uORB::Subscription
            _angle_corr_sub{ORB_ID(angle_correction)};
    struct angle_correction_s angle_correction_struct= {0} ;
};
```

## A.5 FlightTaskDistancePosition.cpp

```
/**
* @file FlightTaskDistancePosition.cpp
*/

#include "FlightTaskDistancePosition.hpp"
#include <mathlib/mathlib.h>
#include <float.h>

using namespace matrix;

bool FlightTaskDistancePosition::updateInitialize()
{
    bool ret = FlightTaskDistanceAltitudede::updateInitialize();

    return ret && PX4_ISFINITE(_position(0))
               && PX4_ISFINITE(_velocity(0));
}

bool FlightTaskDistancePosition::activate(const trajectory_setpoint_s
        &last_setpoint)
{

    bool ret = FlightTaskDistanceAltitudede::activate(last_setpoint);

    _position_setpoint(0) = _position(0);
    _velocity_setpoint(0) = 0.0f;

    return ret;
}

void FlightTaskDistancePosition::_scaleSticks()
{
/* Use same scaling as for FlightTaskDistanceAltitudede */
    FlightTaskDistanceAltitudede::_scaleSticks();

    Vector2f stick_xy = _sticks.getPositionExpo().slice<2, 1>(0, 0);

    Sticks::limitStickUnitLengthXY(stick_xy);
```

```
    if (_param_mpc_vel_man_side.get() >= 0.f) {
        stick_xy(1) *= _param_mpc_vel_man_side.get() /
                _param_mpc_vel_manual.get();
    }

    if ((_param_mpc_vel_man_back.get() >= 0.f) && (stick_xy(0) < 0.f)) {
        stick_xy(0) *= _param_mpc_vel_man_back.get() /
                _param_mpc_vel_manual.get();
    }

    const float max_speed_from_estimator =
            _sub_vehicle_local_position.get().vxy_max;

    float velocity_scale = _param_mpc_vel_manual.get();

    if (PX4_ISFINITE(max_speed_from_estimator)) {
// Constrain with optical flow limit but leave 0.3 m/s for repositioning
        velocity_scale = math::constrain(velocity_scale, 0.3f,
                max_speed_from_estimator);
    }

    Vector2f vel_sp_xy = stick_xy * velocity_scale;

/* Rotate setpoint into local frame. */
    _rotateIntoHeadingFrame(vel_sp_xy);

    if (_collision_prevention.is_active()) {
        _collision_prevention.modifySetpoint(vel_sp_xy, velocity_scale,
                _position.xy(), _velocity.xy());
    }

    _velocity_setpoint.xy() = vel_sp_xy;
}

void FlightTaskDistancePosition::_updateXlock()
{
/* If position lock is not active, position setpoint is set to NAN.*/
    const float vel_xy_norm = Vector2f(_velocity).length();
    const bool apply_brake = Vector2f(_velocity_setpoint).length() <
            FLT_EPSILON;
    const bool stopped = (_param_mpc_hold_max_xy.get() < FLT_EPSILON ||
```

```
            vel_xy_norm < _param_mpc_hold_max_xy.get());

    if (apply_brake && stopped && !PX4_ISFINITE(_position_setpoint(0))) {
        _position_setpoint(0) = _position(0);

    } else if (PX4_ISFINITE(_position_setpoint(0)) && apply_brake) {

        if (_sub_vehicle_local_position.get().xy_reset_counter !=
                _reset_counter) {
            _position_setpoint(0) = _position(0);
            _reset_counter =
                    _sub_vehicle_local_position.get().xy_reset_counter;
        }

    } else {
/* don't lock*/
        _position_setpoint(0) = NAN;
    }
}

void FlightTaskDistancePosition::_updateSetpoints()
{
    FlightTaskDistanceAltitudede::_updateSetpoints();
    _acceleration_setpoint.setNaN();

    _updateXlock();

    _weathervane.update();

    if (_weathervane.isActive()) {
        _yaw_setpoint = NAN;

        if (PX4_ISFINITE(_position_setpoint(0))) {
            _yawspeed_setpoint += _weathervane.getWeathervaneYawrate();
        }
    }
}
```

## A.6 FlightTaskDistancePosition.hpp

```
/**
* @file FlightTaskDistancePosition.hpp
*
* Flight task for manual position controlled mode.
*
*/

#pragma once

#include <lib/collision_prevention/CollisionPrevention.hpp>
#include <lib/weather_vane/WeatherVane.hpp>
#include "FlightTaskDistanceAltitudede.hpp"

class FlightTaskDistancePosition : public FlightTaskDistanceAltitudede
{
public:
FlightTaskDistancePosition() = default;
virtual ~FlightTaskDistancePosition() = default;
bool activate(const trajectory_setpoint_s &last_setpoint) override;
bool updateInitialize() override;

protected:
void _updateXlock(); /**< applies position lock based on stick and
        velocity */
void _updateSetpoints() override;
void _scaleSticks() override;

DEFINE_PARAMETERS_CUSTOM_PARENT(FlightTaskDistanceAltitudede,
(ParamFloat<px4::params::MPC_VEL_MANUAL>) _param_mpc_vel_manual,
(ParamFloat<px4::params::MPC_VEL_MAN_SIDE>) _param_mpc_vel_man_side,
(ParamFloat<px4::params::MPC_VEL_MAN_BACK>) _param_mpc_vel_man_back,
(ParamFloat<px4::params::MPC_ACC_HOR_MAX>) _param_mpc_acc_hor_max,
(ParamFloat<px4::params::MPC_HOLD_MAX_XY>) _param_mpc_hold_max_xy)

private:
uint8_t _reset_counter{0}; /**< counter for estimator resets in
        xy-direction */

WeatherVane _weathervane{this}; /**< weathervane library, used to implement
```

```
        a yaw control law that turns the vehicle nose into the wind */
CollisionPrevention _collision_prevention{this}; /**< collision
        avoidance setpoint amendment */
};
```

# Bibliography


[1] URL: https://px4.io/software/software-overview/ (cit. on p. 3).

[2] URL: https://docs.px4.io/main/en/concept/px4_systems_architecture.html (cit. on p. 4).

[3] URL: https://docs.px4.io/main/en/concept/architecture.html (cit. on pp. 5, 6).

[4] URL: https://nuttx.apache.org/docs/latest/introduction/about.html (cit. on p. 7).

[5] URL: https://docs.px4.io/main/en/middleware/uorb.html (cit. on p. 7).

[6] URL: https://docs.px4.io/main/en/middleware/uorb_graph.html (cit. on p. 9).

[7] URL: https://px4.io/px4-uorb-explained-part-2/ (cit. on pp. 9, 13).

[8] URL: https://px4.io/px4-uorb-explained-part-1/ (cit. on p. 13).

[9] URL: https://px4.io/px4-uorb-explained-part-3-the-deep-stuff/ (cit. on p. 13).

[10] URL: https://mavlink.io/en/ (cit. on p. 13).

[11] Francesco Malacarne. «PX4 autopilot customization for non-standard gimbal and UWB peripherals». MA thesis. Polytechnic of Turin, 2019-2020. URL: https://webthesis.biblio.polito.it/15919/1/tesi.pdf (cit. on p. 15).

[12] Anis Koubâa, Azza Allouch, Maram Alajlan, Yasir Javed, Abdelfettah Belghith, and Mohamed Khalgui. «Micro Air Vehicle Link (MAVlink) in a Nutshell: A Survey». In: *IEEE Access* 7 (2019), pp. 87658–87680. URL: https://ieeexplore.ieee.org/document/8743355 (cit. on p. 15).

[13] URL: https://docs.qgroundcontrol.com/master/en/index.html (cit. on p. 15).

[14] URL: https://docs.px4.io/main/en/flight_stack/controller_diagrams.html (cit. on p. 21).

[15] URL: https://docs.px4.io/v1.12/en/simulation/hitl.html (cit. on p. 26).

[16] N. Koenig and A. Howard. «Design and use paradigms for Gazebo, an open-source multi-robot simulator». In: *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*. Vol. 3. 2004, pp. 2149–2154. URL: https://ieeexplore.ieee.org/abstract/document/1389727 (cit. on p. 28).

[17] Adnan Munawar, Yan Wang, Radian Gondokaryono, and Gregory S. Fischer. «A Real-Time Dynamic Simulator and an Associated Front-End Representation Format for Simulating Complex Robots and Environments». In: *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2019, pp. 1875–1882. URL: https://ieeexplore.ieee.org/abstract/document/8968568 (cit. on p. 31).