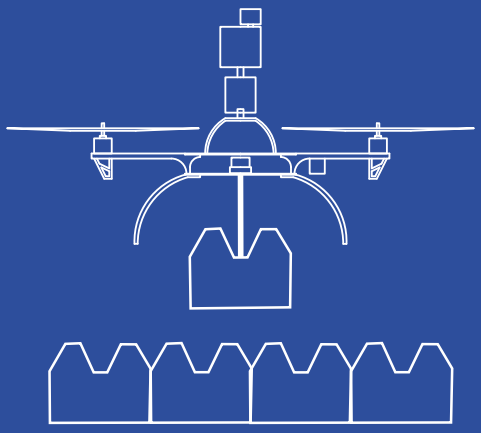
UCLouvain

epl

École polytechnique de Louvain

Design of a positioning and a mission management system for builder drones



Authors: **Julien AMALABERQUE, Marie-Marie VAN DER BEEK , Nicolas VANVYVE**
Supervisors: **Pierre LATTEUR, Ramin SADRE**
Readers: **Sébastien GOESSENS, François WIELANTS**
Academic year 2018-2019
Master [120] in Computer Science and Engineering

Abstract

This master's thesis is part of the research project led by Pr. Pierre Latteur and his team at the Civil Engineering Department of UCLouvain, on the use of drones for building real-life structures in order to robotize construction processes.

In previous years, drone compatible construction elements and different positioning systems for the drone were developed using multiple technologies: ultra-wideband, total station and computer vision. The next step was to integrate those systems together in a single more robust and reliable positioning system that could send its information to the flight controller unit.

Additionally, a guidance system had to be developed to allow the planning of flight missions and their correct execution by the drone.

In this report we will explain the development of these two steps and show experimental results of flight missions.

Acknowledgements

We would like to thank:

Pierre Latteur, for providing us with invaluable supervision and guidance even from the other side of the world.

Ramin Sadre, for his guidance and useful insights regarding our work.

Sébastien Goessens, for his support, advice and availability throughout the year.

Mathieu Monnart, with whom we shared a great working atmosphere which encouraged us to carry out the project successfully.

Samuel Laurent, for his expertise in electronics and his assistance in the development of the lifting system.

Geoffrey Mormal and Christophe Balthasart, from ALX Systems, for their expertise in the field of drones, their advice and the equipment they have lent us.

Our families and friends, for their presence, their words of encouragement and their moral support.

Contents

Glossary

BIM Building information modeling. 9

DOF Degrees of freedom. 37

ENU East, North, Up (Axes conventions). 47

FCU Flight Controller Unit. 37

FishEye Ultra-wide angle lens, able to capture an image around 180 degrees. 40

GPIO General Purpose Input/Output. 14

IEEE Institute of Electrical and Electronics Engineers. 14

IMMC Institute of Mechanics, Materials and Civil Engineering. 14

IMU Inertial Measurement Unit, usually a gyroscope and an accelerometer. 15

IP Internet Protocol. An IP address is a label assigned to each device connected to a network. 76

LPE Local Position Estimator. Extended Kalman filter for 3D position estimation. 46

LR-WPAN Low Rate Wireless Personal Area Network. 14

MAVLink Messaging protocol that has been designed for the drone ecosystem. 18

Multilateration Finding a position from multiple distances. Called trilateration with 3 distances. 24

NED North, East, Down (Axes conventions). 47

RCP Remote Procedure Call is a protocol that one program can use to request a service from a program located in another computer on a network without having to understand the network's details. 17

RGB Red-Green-Blue, primary colors that can be used to represent a broad array of colors. 38

ROS Robot Operating System. 15

SLAM Simultaneous localization and mapping. 40

TCP Transmission Control Protocol. 22

UDP User Datagram Protocol. 16

UI User Interface. 78

UWB Ultra-wideband. 8

VPU Vision Processing Unit, a specialized chip. 15

Chapter 1

Introduction

1.1 Context

The construction sector has been marked by several revolutions over time. One example is the Industrial Revolution with the systematic use of steel in the 19th century. Another is the revolution linked to computers and finite element methods, which makes it possible to simulate and test very complex structures.[1]

By taking inspiration from nature, we could induce a new revolution. Birds or bees spend their time flying between the nests they are building and their surroundings, where they can find building elements. What if we imitate them?

This project aims to gather all elements enabling a drone to transport construction elements and autonomously construct a masonry wall. As expected, we cannot simply replace a worker with a drone, we need to rethink the manner our society builds. The following aspects need to be taken into account:

- **Construction** aspect i.e., the development of drone-compatible elements compensating the fact drones might not be able to provide the same precision as human workers.
- **Computer Science** aspect i.e., the systems that will provide data and control the drone.

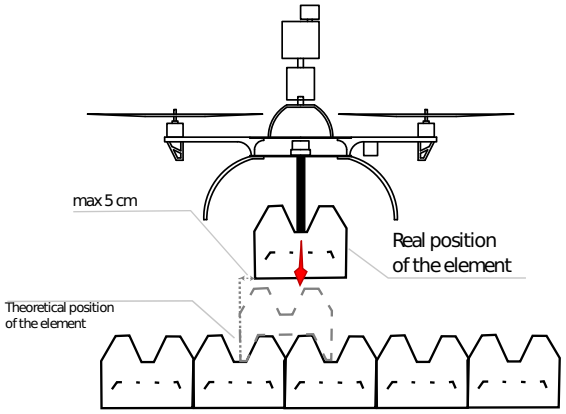


Figure 1.1: Illustration of the inaccuracy allowed to the drone for the placement of constructives

The first aspect was researched by Professor Latteur's team, who have developed several block designs that allow for inaccuracy during installation [2, 3, 4, 5], as can be seen in Figure 1.1. They are called "**dricks**", a mix between drone and brick.

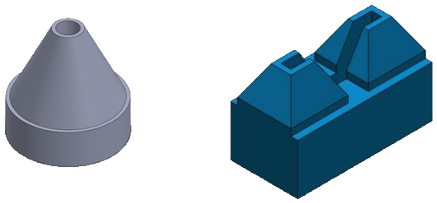


Figure 1.2: Two different types of dricks among others

Once we have bricks that can be dropped from a drone, we have to make sure that the drone can come and pick them up, move them and drop them off at the right place. A solution is to call a skilled pilot. It is in this context that the idea of automating the process appeared, replacing the pilot with an onboard computer.

Before giving a flight plan to the drone, it must know where it is in the working environment. Last year, 5 students worked on 3 different systems to locate the drone. N. Sorensen and R. Vermeiren developed a solution using a total station [6], Z. Bouhaddi uses UWB tags [7] and F. Kaczynski and T. Jacques used computer vision to know the position, orientation of the drone and the distance that separates the drone from the target when hovering above it [8].

Our job is to take these positioning systems and operate them together to know the exact position of the drone and command it to fly according to a list of instructions.

1.2 Overview of the Project

The complete construction process by drone is summarized hereunder and is represented in Figure 1.3.

1. As with any construction, **plans must be designed** by architects and engineers.
2. Then **this plan must be transformed** into a BIM model.
3. Then there is a transformation of this model into a **series of instructions** that will allow a drone to execute the construction.
4. The last part consists in forwarding these different instructions to the drone for the **construction process**.

We focus on the fourth step.

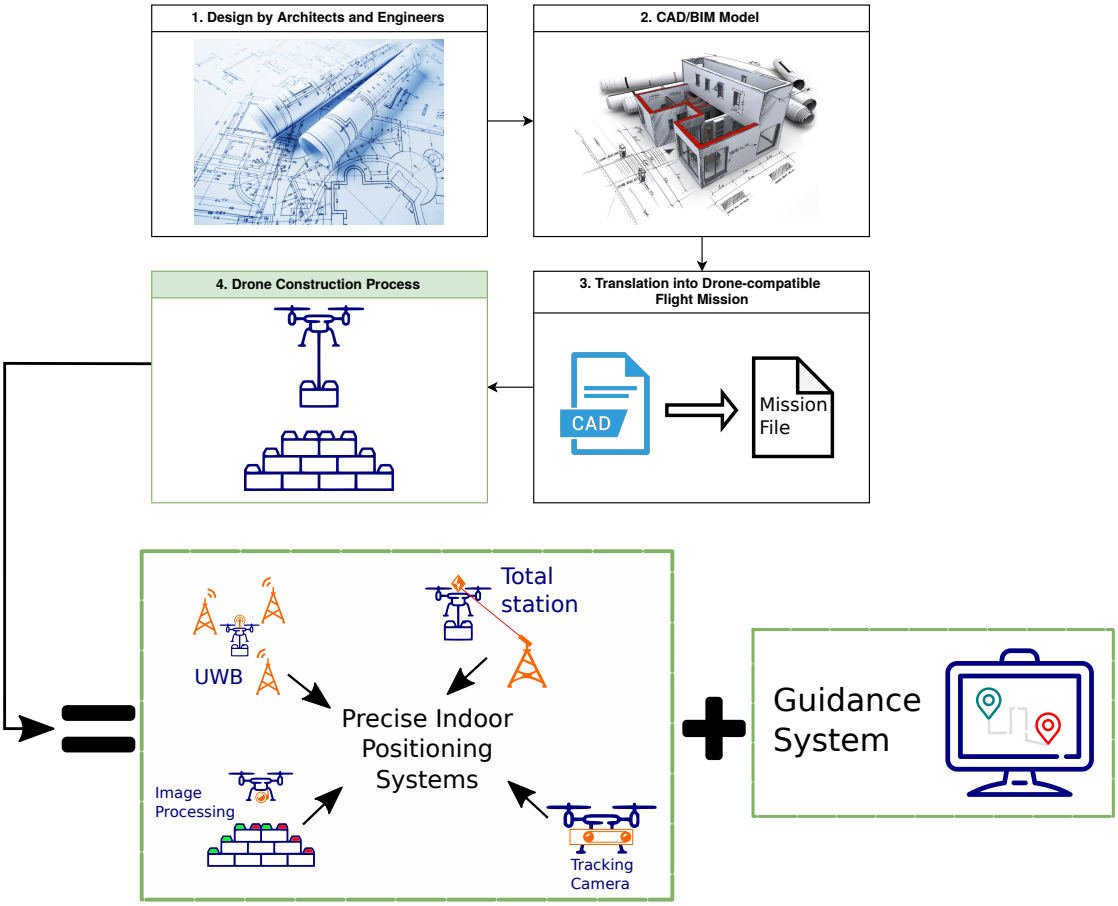


Figure 1.3: Project Overview

1.3 Objectives

The first part of this project was the development of a robust positioning system based on multiple developed technologies and with respect to some constraints. It must be capable of guaranteeing a minimum level of accuracy due to the structural elements: a deviation of maximum 5 cm in each direction is tolerated. It must also be capable of working in indoor environments.

Our first objective was to create one program combining the different positioning systems developed last year resulting in a more robust and reliable software than each single system taken individually.

Our second objective was to develop the second part of the computer science aspect: a system capable of giving the drone instructions along the way, making it act without manual control to reach target waypoints in space and to lift or drop construction elements.

Chapter 2

System Overview

2.1 Hardware Resources

2.1.1 Drones

For this project, there are three types of drones that allow the development and testing of the various stages of the project in complete safety.

Big drone

This model is the full-scale prototype capable of transporting masonry units weighing up to 30 kg. Built on a custom chassis and composed of carbon components, it should only be used if the systems have been proved to be reliable and robust by having been tested on the two smallest drones.

Figure 2.1: Big drone

Medium drone

The medium drone allows the systems developed on the small drone to be tested on a more realistic scale. It has a large enough chassis to carry a lot of instruments and is able to lift elements weighing up to 5-8 kg.

Figure 2.2: Medium drone

Small drone

The smallest of the three drones is used to develop the different technologies of the project. It has the advantage of being a low-cost alternative compared to the other two. It is built upon a DJI F550 chassis as a basis, which serves as a printed circuit board. The rest of the contraption is completely printable in 3D, which makes it quite flexible.

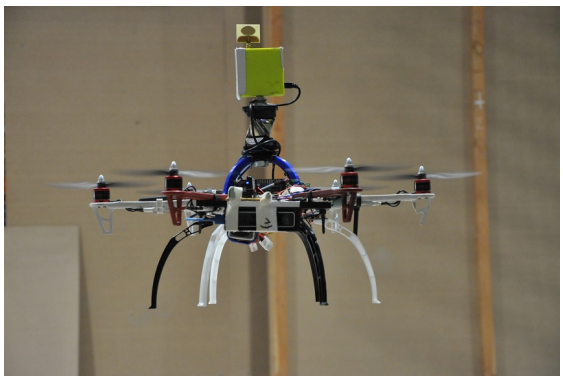
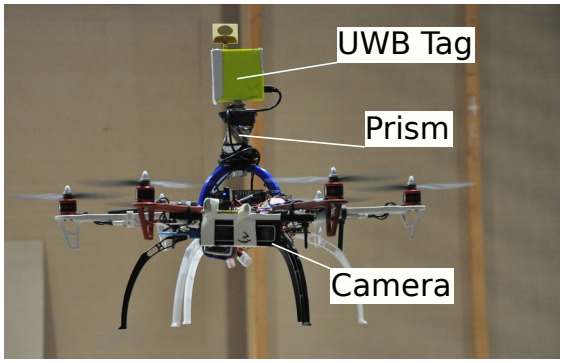


Figure 2.3: Small drone

2.1.2 Onboard Equipment

We will focus here on the elements that are present on the small drone because it is the one we extensively tested this year.



Flight controller The controller is the brain of the drone, it receives the driving instructions and transmits them to the engines according to the calibration made beforehand. We use the Pixhawk 2 and the PX4 autopilot firmware on it, all under Creative Commons license. The Pixhawk has several internal sensors such as a compass, gyroscopes, accelerometers and barometer. Depending on its configuration it takes into account the information coming from its own sensors and the position information we give it. All input data is merged together to obtain a single position.

A simplified version of the PX4 flight controller can be seen on page 19. It has 3 major components: a position and attitude estimator, a state machine and a position controller. The first will combine sensors and/or external inputs to compute the position and attitude state of the drone, the second stores the entire current state (position&attitude, armed/disarmed, flight mode, etc.) and the third computes the correction the drone must make to reach a given setpoint based on the state provided by the estimator. It is thus the position controller that will output an attitude and thrust setpoint that will move the vehicle towards the desired position. [9]

Onboard computer A central element of the global system, it is an Odroid XU4. It is preferable to a Raspberry Pi 3 because it has, among other things, 2GB of RAM and with an 8-core ARM processor. There are also 3 USB ports, including two USB 3.0. A disadvantage is the absence of an integrated wifi card which requires the use of a wifi dongle.

Prism We use the Sokkia ATP1 360 mini prism as a reflector for the total station. It returns the laser in exactly the same direction, allowing the total station to calculate the travel time and therefore the distance. The position returned by the station is the center of the prism. It's fixed in the center of the drone, slightly elevated. The prism height is a compromise between the accuracy of the position in the event of a drone tilt and the visibility from the station. Indeed, if it is too low, the laser will be disturbed by the drone's arms.

UWB tag Just above the prism is an EVB1000 beacon from DecaWave. As described in section 4, this beacon is used as a receiver to compute the position using UWB technology.

Realsense camera We installed an Intel T265 camera at the front of the drone, which combines gyroscopes, accelerometers with two fisheye lenses and SLAM technologies to obtain a third position information. It is also the only system that allows us to know the orientation of the drone. We will develop all this in section 5.

Shifter shield, electronic card and electromagnet In order to control the lifting system from the onboard computer, we added a XU4 Shifter Shield on top of the odroid. The latter allows us to increase the output voltage of the Odroid GPIO pins and to command a switch which is located on an electronic board provided by the electronics laboratory of the IMMC department. This card enables us to activate/deactivate the current flow from the battery to the lifting system which in our case is an electromagnet.

2.2 Positioning Systems

2.2.1 UWB anchors and tag

UWB is a technology that belongs to the LR-WPANs family, which is governed by the IEEE 802.15.4-2011 standard. Its main interest is its robustness to interference and noise, its accuracy and its real-time operation.

By using four EVB1000 beacons, it is possible to determine a position. Three beacons are fixed to known positions, they will be called anchors, and the fourth is fixed on the drone and will be called tag. The firmware installed on the tag computes the distance between each anchor and the tag. Using a trilateration algorithm, the position in the three-dimensional space of the tag is determined. It

was requested to obtain the accuracy of 50 cm, at the end of the year, the accuracy was 18.7 cm in x & y axes and 39.6 cm in z-axis.

This precision is not sufficient when it comes to laying bricks, but it can be used as a sufficient approximation for long flights and to refine the total station's search for the drone when it has lost the prism.

2.2.2 Onboard camera

We have worked with two Intel cameras, the T265 and the D435i. Only the first one will finally be integrated into the drone this year.

The Intel RealSense T265 tracking camera includes two fisheye lens sensors, an IMU and an Intel Movidius Myriad 2 VPU. All SLAM algorithms work directly on the VPU, allowing very low latency and extremely efficient power consumption (1.5 W).

The Intel RealSense D435i is a depth camera that includes a Bosch BMI055 6-axis inertial sensor in addition to the depth camera that measures linear accelerations and angular speeds.

Both cameras require Realsense SDK, which supports T265 on both Windows and Linux and via a ROS wrapper.

The T265 is designed to produce high-precision, low-latency, 6-degree-of-freedom position data without using additional computing resources. It does not contain a depth camera but can be used in combination with all D400 series depth cameras (including the D435i). [10, 11]

2.2.3 Total station

Nicolas Sorensen and Rémy Vermeiren developed a location system using a Leica total station (TCRP1203) and a prism from the same brand, the Sokkia ATP1 360 mini prism. The prism acts as a reflector and the total station can follow it. By knowing the angles of the total station and the time it takes for the laser to make the round trip, it is possible to know the relative position of the drone.

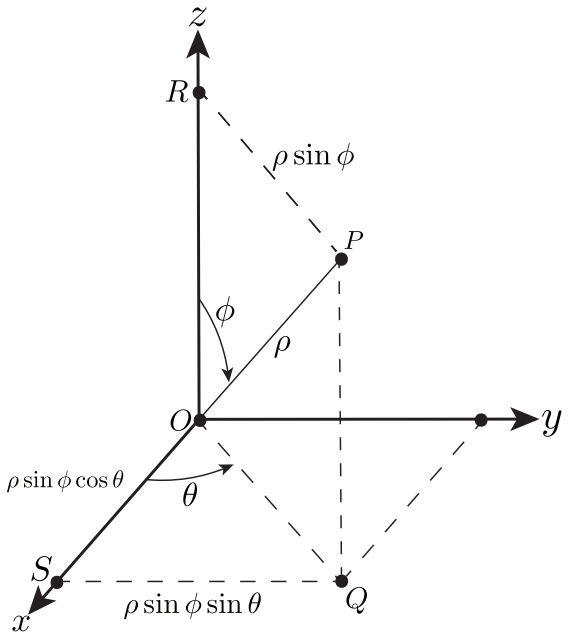


Figure 2.4: Trigonometry behind position calculation via angles

The main disadvantage of this system is that it requires wireless communication with the computer onboard the drone, a priori using a UDP server and wifi. This can cause an impact of variable intensity on latency depending on the quality of the wifi.

It should also be noted that the total station and drone must always be in visual contact, this is a constraint that must be taken into account when creating the instructions given to the drone, for example, it will be necessary to build the different elements starting from the bottom of the area to avoid losing the laser contact.

Also the station has a limited rotation speed, so if the drone is too close and it goes too fast, the station will not be able to follow it. The recommended distance between the prism and the station is 20m.[6]

Last, it is natively impossible for this total station to track several prism. Thus redundancy of this system is not possible without additional units.

2.3 Choice of Framework

In order to implement our systems, two frameworks were considered. On the one hand the ROS framework, a set of open source tools for robotics and, on the other

hand, ALX OS, a proprietary software developed by the start-up ALX Systems, which they made available to us.

During the year, we worked simultaneously with both projects but ended up focusing on the ROS development because it was open source and gave us more flexibility as well as extensive documentation from the community. Additionally support for external positioning sources was only partial in ALX OS which we could not modify ourselves due to the black box nature of the software.

In this thesis, we mainly develop the work done with the ROS framework. The work performed on the ALX system can be found in Appendix E.

2.4 ROS Framework Description

To develop our systems, we decided to use ROS, a widely adopted, flexible framework designed for writing robot software. This framework aims to simplify the creation of complex and robust robot behavior. It is technically not an operating system but rather a collection of tools, libraries, and conventions, running on top of an actual operating system[1].

Our onboard computer is equipped with the ROS Melodic distribution and is running on top of a Ubuntu Mate distribution. Since our positioning system and guidance system are developed as ROS components, it is important to understand the base concept of the framework.

ROS is designed as a loosely coupled system[2] in which each computation process is a component, called a node. These nodes can communicate with one another, by exchanging messages using streaming topics, RCP services, and the ROS parameter server.

In our implementations we will both use topics and services. We can distinguish them as topics have an anonymous publish/subscribe semantic while services have a request/reply semantic.

The publisher/subscriber pattern works in the following way:

1. A node that has data it wishes to share can do so by publishing it on a relevant topic. This node is a **publisher** of that topic.

2. A node that is interested in that data can subscribe to that topic. This node is a **subscriber** of that topic.

[1] Ubuntu Mate 18.04 in our case.

[2] In computing and systems design a loosely coupled system is one in which each of its components has, or makes use of, little or no knowledge of the definitions of other separate components.[12]

Nodes can both be subscribers of multiple topics and publisher of other topics at the same time. Each topic can also have multiple publishers and subscribers. Due to their semantic, topics are a unidirectional streaming communication bus.

The request/reply pattern works in the following way:

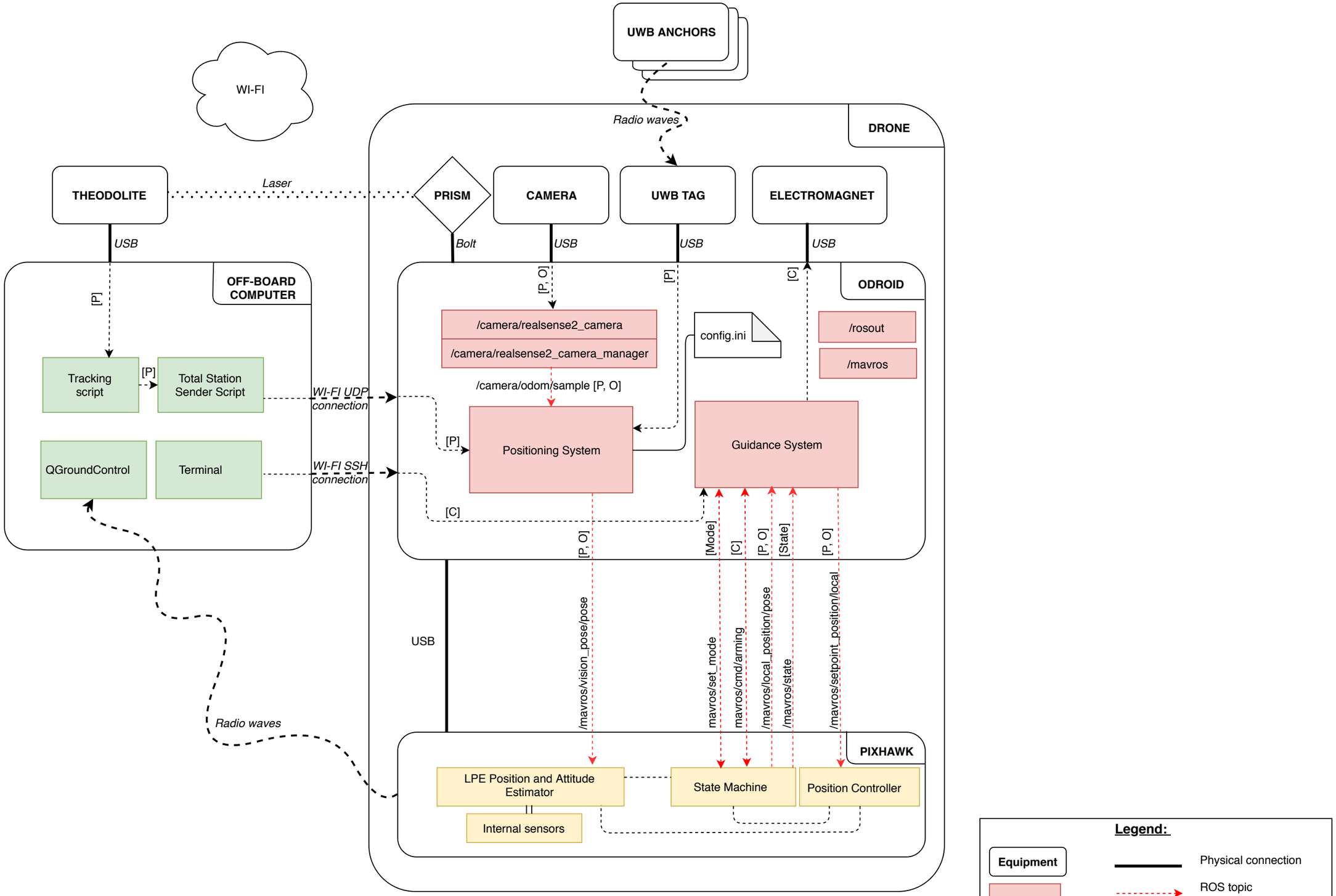
1. A node calls the appropriate service by sending a **request message** and awaiting the reply.
2. Once the request is received, the **server will respond** to the demand.

The use of the ROS framework has a number of interesting advantages:

- Its fault tolerance as crashes are isolated to individual nodes.
- Less complex code compared to monolithic programs due to the separation in different nodes.
- Interesting packages for our use case. In particular, the mavros package enables communication between a ROS enabled computer with any MAVLink enabled autopilot. Our project uses the Pixhawk as flight controller, the associated autopilot PX4 uses the MAVLink messaging protocol and the standard messages defined by it for exchanging data.[13]

2.5 Communication Scheme

All the components described above are linked together and communicate with each other in different ways. The following scheme shows all the components, their communication links and information streams. The links will be explained in later chapters.


UWB ANCHORS
WI-FI
Radio waves
DRONE
THEODOLITE
Laser
PRISM
CAMERA
UWB TAG
ELECTROMAGNET
USB
Bolt
USB
USB
USB
OFF-BOARD COMPUTER
[P]
ODROID
[P, O]
[P]
[C]
/camera/realsense2_camera
/camera/realsense2_camera_manager
config.ini
/rosout
/mavros
/camera/odom/sample [P, O]
Tracking script
[P]
Total Station Sender Script
WI-FI UDP connection
[P]
Positioning System
Guidance System
QGroundControl
Terminal
WI-FI SSH connection
[C]
[P, O]
[Mode]
[C]
[P, O]
[State]
[P, O]
USB
/mavros/vision_pose/pose
mavros/set_mode
mavros/cmd/arming
/mavros/local_position/pose
/mavros/state
/mavros/setpoint_position/local
Radio waves
PIXHAWK
LPE Position and Attitude Estimator
State Machine
Position Controller
Internal sensors
Legend:
Equipment
ROS Node
off-board program
PX4 software
Physical connection
ROS topic (Publisher -> Subscriber)
ROS service
[P] = Positions
[P,O] = Positions, Orientations
[C] = Commands

Chapter 3

Total Station

3.1 Introduction

A total station is an instrument capable of measuring distances to a point. It works by emitting an infrared signal which is reflected on a glass prism placed at the target. By emitting and receiving multiple frequencies, the station can determine the distance by estimating the integer number of wavelengths to the target.

It is usually used for surveying and building, but in this project the prism is mounted on a drone, whose position can then be tracked in real time with the station.

In this chapter, we will present the different modifications and additions we made to the total station project that we have presented in section 2.2.3 in order to integrate it with the drone to actually use this technology during flight.

3.2 Adaptations

We made a few changes to the script using the total station. Firstly, we changed the automatic setup to a manual setup.

- With automatic setup, the x-axis must be pointed with the station and once this is done, it will look for the prism starting from around the x-axis. The problem with this method is that the drone must be close to the x-axis or the station will abandon its search.
- With manual setup, you also have to point the x-axis with the station, but then we must target the prism. The search for the prism then begins, it is

much faster, almost instantaneous. It is more interesting to use the station in this way, because it is possible to define the x-axis as desired, without any constraint on the position of the drone.

Secondly, we modified the part actually returning the position. The raw output of the script is, for the z-axis, the difference in height between the station and the prism. The station (the rotating part) is calibrated as being at z=0 during the setup process. However, as can be seen in Figure 3.1, the station is actually at a height of a, so the value of z we received was $-a + b$.

In order to solve this problem, when setting up we measure the coordinates of the drone when it is placed on the ground i.e., at height 0. The vertical component $(-a+b)$ is kept in memory and will be subtracted from each new calculated position. This modification allows us to know the actual height of the reference point of the drone, located at ground level as can be seen in figure 6.1.

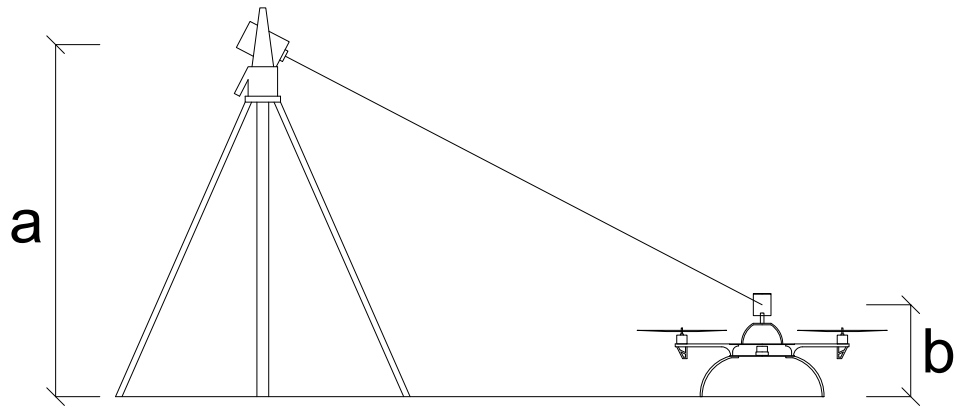


Figure 3.1: Disposition of the total station and the drone at the time of the setup. a height of the station, b the height of the prism.

We have also fixed an error in the code, this will be explained in section 8.1.1.

3.3 Server

The total station provides the estimated positions to the ground computer, whom are connected by USB cable. The positioning system running on the onboard computer, explained in chapter 6, needs to process this data. Thus a communication

channel is required between the ground computer and the Odroid. This channel can be seen on the scheme page 2.5.

We wrote a program that performs the following actions:

1. It **opens a UDP connection** to the IP address of the Odroid on a specific port number.

2. It **launches the tracking script** that retrieves the positions from the total station.

3. It **sends the reliable positions** via the UDP connection and discards unreliable ones. The tracking script developed in [6], returns a precision factor followed by some data. The data, i.e. the position is considered reliable if the precision factor is either 0 = accurate position or 1 = lower accuracy. The data is discarded when the factor is either 3 = only angle measurement or 4 = error.

The UDP protocol was selected over the TCP protocol since we do not require reliability. Due to timing constraints, we do not want to recuperate an old position that is lost, but rather a current position.

Chapter 4

UWB Positioning for Drones

4.1 Introduction

As part of his master's thesis presented in 2018, Zakariya Bouhaddi [7] presented a study of UWB positioning algorithms. He measured the accuracy of the algorithm distributed by the company DecaWave as part of the TREK1000 evaluation kit, a set of antennas, in the DroneZone of the civil engineering laboratories at UCLouvain. The initial idea was to reuse the code he wrote to compute its results and adapt it to send them to the positioning system. However, for multiple reasons detailed below, this proved to be difficult. While the algorithm itself was of good quality and efficient for our use, the code surrounding it had flaws that needed to be fixed in order to provide the positioning service in a reliable way and to further expand it. Additionally, the documentation and the structure of the program were poorly suited for a multi-year project like the builder drones of the IMMC.

With that in mind, a new program was written this year that aimed to fix the problems encountered and allow new usages of the UWB tag. Better development practices were followed diligently to create software that is both maintainable and robust in use, two qualities greatly important for this project.

The following sections will detail the prior work, where additional improvements were necessary for the good integration of the positioning system, and the new structure of our final program.

4.2 UWB Technology

Ultra-wideband technology is radio-based and uses large frequency ranges. It is aimed at sending information at close range while avoiding interfering with other

already occupied frequencies.

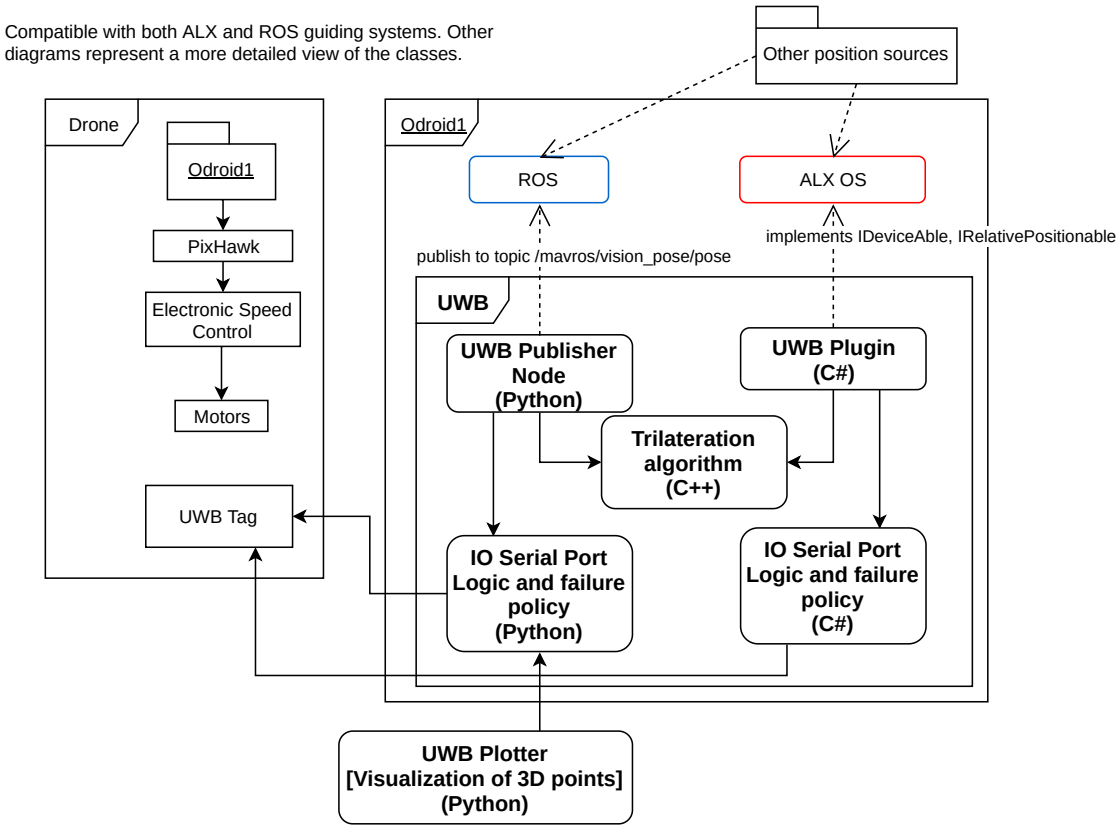
To find the position using this technology requires multiple steps. Firstly, an antenna (the "tag") is placed on the object whose position is of interest, in our case the drone. Afterwards, the goal is to find the distances between the tag and other antennas (the "anchors") placed around the object. While there are multiple techniques[14] to find these, the one used in our system is called Two-Way Ranging. The tag first polls the anchors whose responses include the time of arrival. Once the tag receives the information, it can be directly converted to distances by multiplying the time difference by the speed of light in the air.

Once we have the distances between the tag and the anchors, it is possible to compute a position through a mathematical problem called multilateration.

This technology was chosen for the project as an emergency backup positioning system because of its robustness. Its precision is less than what is offered by other positioning system. It was manually assessed last year to ~ 20 cm x & y and ~ 70 cm z, and you can find new manual and in-flight measurements in the "Testing" chapter of this master thesis.

4.3 Structure Overview

UWB positioning overview



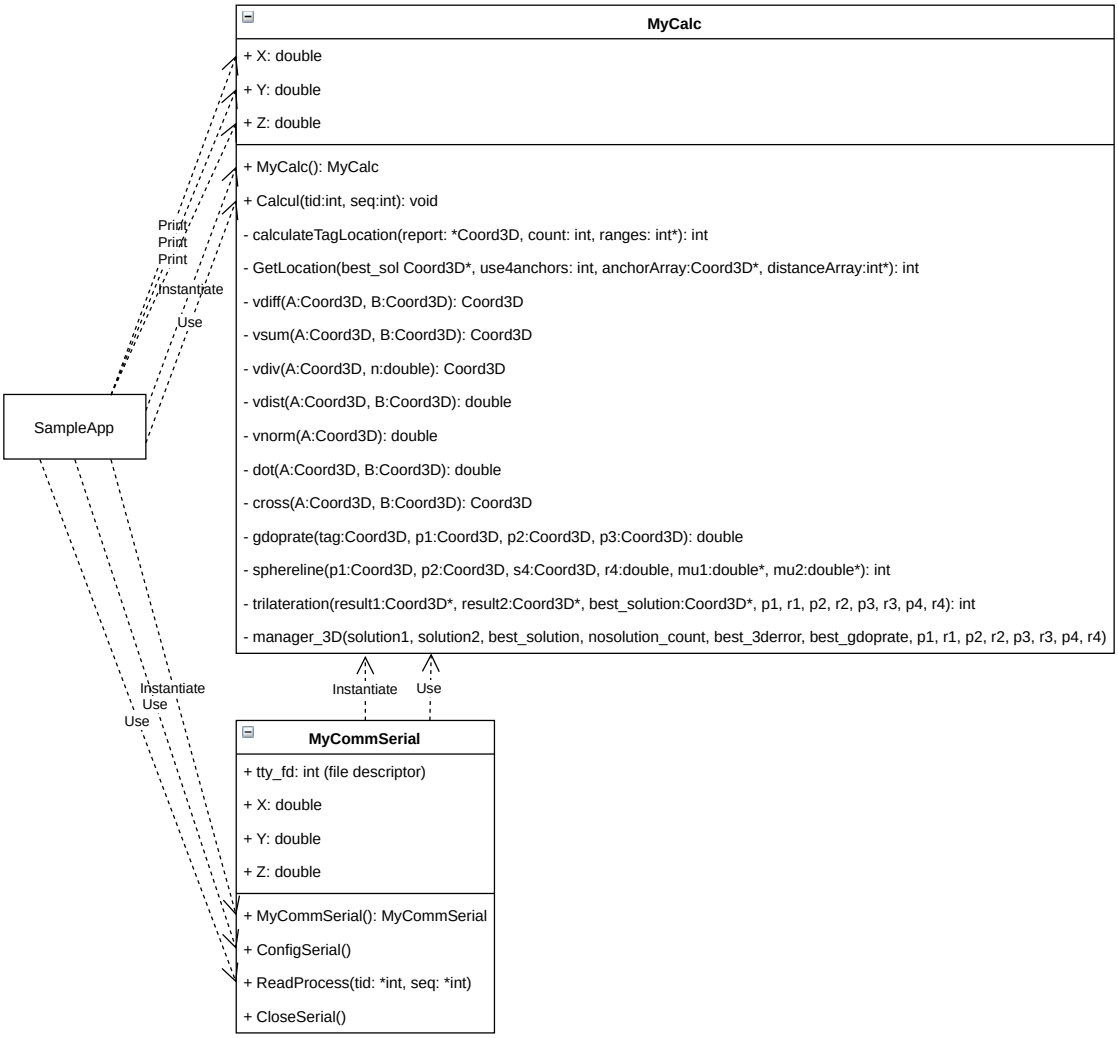
The UWB positioning system is essentially composed of three distinct components:

- the **I/O component** which interacts with the DecaWave device to get the distances from the tag to the anchors;
- the **algorithms/computation strategy**, that tries to compute an xyz position for each tag out of three or four tag-anchor distances, and angular information when there are multiple sources;
- the **Drone integration** part, which calls other components and knows how to pass the coordinates in a way the system understands.

Beyond functionality, these roles must be easily distinguished in the program to facilitate the maintenance of the existing codebase, the evolution of additional features and most importantly to easily try new solutions without rewriting all the other parts. For instance, it is a good idea to test many different algorithms and trying a new one should be a small and easy task: someone should not need to

pay attention to other components when doing so. This is also a general principle called the **separation of concerns**. Additionally, sometimes components can be reused elsewhere. A concrete example is to only produce visual plots of obtained points, ranges and angles: this is completely unrelated to the guiding system logic and that should be reflected in the project structure.

4.4 Previous Program Description and Analysis



4.4.1 Global control flow

Let us take a look at the control flow of the original testing program written by Zakariya Bouhaddi last year and its code to understand its strengths and weaknesses.

The entry point is the `Main` method of the `SampleApp.cs` class. This class is written in C# and through a wrapper (`LibUWB.cs`) C++ functions are called.[1]

The first instruction is the instantiation of an object of class `MyCommSerial` (declared in `myCommSerial.cpp/h`) upon which the `ConfigSerial` method is called. `ConfigSerial` opens the file stream and sets file descriptor settings like the baudrate. This is done using the `termios.h` definitions which are part of the UNIX specifications. This means the program is **not portable** to other platforms like Windows.

The method also loads the configuration file for the anchor positions. The configuration file ("conf_Ancres.txt") is composed of 9 numbers separated by spaces, corresponding in order to the xyz values of the anchors 0, 1 and 2. This is not self-descriptive and could be improved for user-friendliness.

Afterwards an infinite, condition-less loop (`while(true)`) then starts and continuously read the data with the `ReadProcess` method before printing it to `stdout`, the standard output (usually the terminal screen).

A call to the `ReadProcess` on the `CommSerial` object reads all data available at some time from the serial port, interprets it, creates a `MyCalc` (which will be detailed below) object whose x, y, z attributes will be populated by a call to `MyCalc::Calcul`. x, y and z attributes on the `CommSerial` object are then assigned the value of those of the `MyCalc` object which is already deleted after. This is a very unusual way of returning values that involves a lot of object creation/destruction overhead compared to a simple function call.

Back in `SampleApp::Main`, after the infinite loop a call to the method `CloseSerial`, whose role would be to close the file stream to the DecaWave device is present. However since the loop is infinite this means this code is never executed and **the device is never properly closed**. This has an impact on any other software that would try to access the device as the operating system will not allow a second connection to be open, even after the program has been forcefully closed.

Additionally, the separation of concerns is not respected in this program. The part of code reading data from the device should not be aware of how such data will be used. Here a tight coupling can be seen between I/O reading from the serial

[1] Since the names of the functions are changed during the wrapping process, we will use the original C++ names for simplicity of the description.

port and the calls to the actual algorithms which would make it difficult to change one part without having to dive into the code of another.

4.4.2 `MyCalc` and the positioning algorithm

Taking a look at `MyCalc`, we can observe the actual trilateration algorithm in the aptly named `trilateration` method. This algorithm was initially written by the DecaWave company for their Real Time Location System software and is distributed as a part of the TREK1000 kit. You can download it at the following page of the official DecaWave website: `https://www.decawave.com/content/trek-source-code`. Zakariya Bouhaddi's master's thesis mentions adding further optimization to the algorithm in particular with the problem of finding a solution with 4 anchors, but a comparison of both codes revealed no change on this aspect.

However, he did evaluate extensively this algorithm and the findings concluded with a precision error inferior to 20 cm in x & z and inferior to 70 cm in z [7]. Further tests done this year confirmed the results (more details in Chapter 8).

An inconvenient with the implementation of the algorithm is that it hides the detailed reasons of failure that may arise. Its end error messages are only `TRIL_3SPHERES`, `TRIL_4SPHERES` or a generic -1 meaning none could be found. This might have been fine in its original use case (the DecaWave RTLS PC software) of simple visualization on a map, but to use the results for positioning it is important and useful to know why a solution could not be found. Are the anchor ranges inconsistent? Has one of the anchors stopped responding? This will be reflected upon in the following section.

Lastly, the `MyCalc` class incorporates vector mathematics methods (which you can find a list on the related class diagram). These should be handled on their own as functions defined in a separated file as they are a typical example of reusable code.

4.4.3 Drone integration

No interfacing with any guiding system was done last year, although there were attempts to integrate with ALX OS. According to Geoffrey Mormal (CEO&Founder of ALX Systems) the UWB program was tried in May 2018 but crashed instantly, and the causes could not easily be determined.

4.4.4 Other difficulties encountered

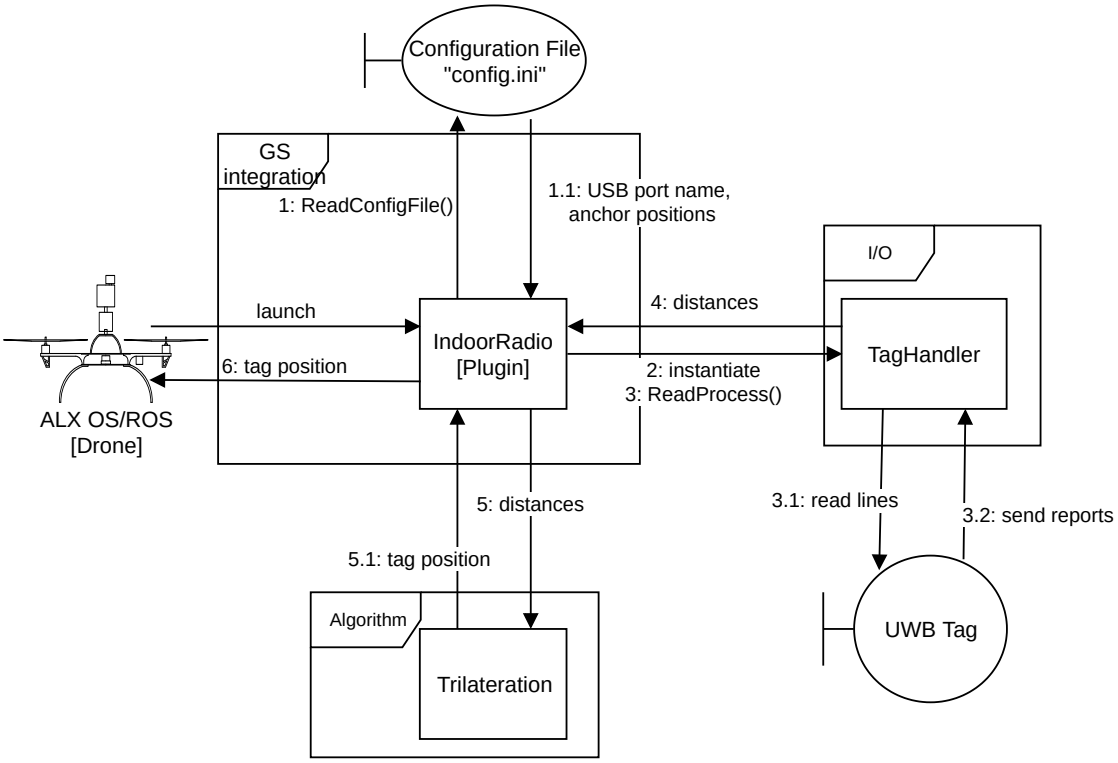
```
//Change by void if not working
void CommSerial::ReadProcess(int* tid, int* seq){
 unsigned char temp ='D'; //09/05
 //unsigned char temp;
 //vector<string> TOF; //09/05
 //string report;
 //cout << " ReadProcess before read tty = "<< this->tty_fd << endl;
 /* IF INTERRUPT, ENTER WHILE*/
 //if(this->isReady){
   //cout << " iM hERE " << endl;
   //if(read(this->tty_fd,&temp,1)>0){

  //for(int i=0; i<0; i++){ // Added for 25/04/2018
   while(read(this->tty_fd,&temp,1)>0){
   //09/05
   //while(read(this->tty_fd,&temp,1)>0){ // && != INTERRUPT
    //cout << " ReadProcess after read " << endl;
    //read(this->tty_fd,&temp,1);

    this->report += temp;
    //cout<<this->report<<endl;
    //TOF.clear();//NEED TO CLEAR AT SOME TIME!!!!!
    //cout<< *&(temp)<<endl;
    //Push into vector the value from the pointer to buffer temp
    if(temp=='\n'){
    //if(*&(temp)=='\n') {
     //NEED TO MODIFY IF WE DON'T WANT TO STORE DATA
     //cout << "Im in /n"<<endl;
```

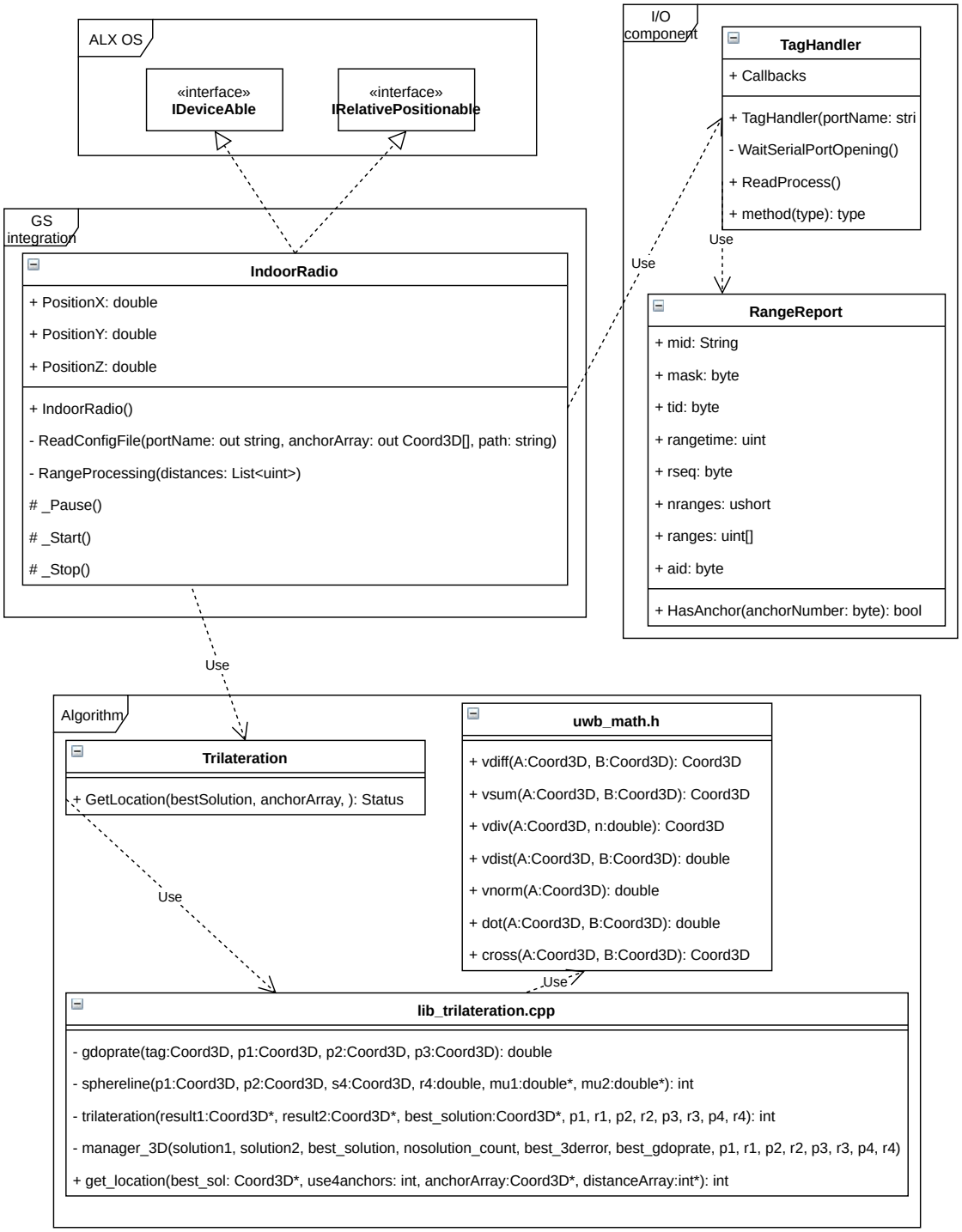
Figure 4.1: This is both hard to read and hard to maintain

- Last year updates of the code were done by copy-pasting said code, commenting the old version and applying edits on the other. This resulted in exceptionally long files containing useless old code which impedes readability, makes difficult to see the evolution of changes over time and to improve the existing code. Instead, a VCS like Git should be used to properly record the history of changes.
- Extremely sparse documentation was provided about the role, the inputs and the effect of each function and no convention was followed. Easily generating documentation (using standard tools like Doxygen) would be an asset, especially for a project which is used successively by new persons each year.

4.5 Overhaul of the UWB Positioning Program

These diagrams represent the new structure of the program, which is much more modular, with each component being easy to replace or written independently, allowing parallel development when needed. How each component was implemented is detailed in further sections.


Configuration File
"config.ini"
GS
integration
1: ReadConfigFile()
1.1: USB port name,
anchor positions
I/O
launch
IndoorRadio
[Plugin]
4: distances
TagHandler
6: tag position
2: instantiate
3: ReadProcess()
ALX OS/ROS
[Drone]
5: distances
3.1: read lines
3.2: send reports
5.1: tag position
Algorithm
Trilateration
UWB Tag


ALX OS
«interface»
IDeviceAble
«interface»
IRelativePositionable
I/O
component
TagHandler
+ Callbacks
+ TagHandler(portName: stri
- WaitSerialPortOpening()
+ ReadProcess()
+ method(type): type
Use
Use
GS
integration
IndoorRadio
+ PositionX: double
+ PositionY: double
+ PositionZ: double
+ IndoorRadio()
- ReadConfigFile(portName: out string, anchorArray: out Coord3D[], path: string)
- RangeProcessing(distances: List<uint>)
_Pause()
_Start()
_Stop()
RangeReport
+ mid: String
+ mask: byte
+ tid: byte
+ rangetime: uint
+ rseq: byte
+ nranges: ushort
+ ranges: uint[]
+ aid: byte
+ HasAnchor(anchorNumber: byte): bool
Use
Algorithm
Trilateration
+ GetLocation(bestSolution, anchorArray,): Status
uwb_math.h
+ vdiff(A:Coord3D, B:Coord3D): Coord3D
+ vsum(A:Coord3D, B:Coord3D): Coord3D
+ vdiv(A:Coord3D, n:double): Coord3D
+ vdist(A:Coord3D, B:Coord3D): double
+ vnorm(A:Coord3D): double
+ dot(A:Coord3D, B:Coord3D): double
+ cross(A:Coord3D, B:Coord3D): Coord3D
Use
Use
lib_trilateration.cpp
- gdoprate(tag:Coord3D, p1:Coord3D, p2:Coord3D, p3:Coord3D): double
- sphereline(p1:Coord3D, p2:Coord3D, s4:Coord3D, r4:double, mu1:double*, mu2:double*): int
- trilateration(result1:Coord3D*, result2:Coord3D*, best_solution:Coord3D*, p1, r1, p2, r2, p3, r3, p4, r4): int
- manager_3D(solution1, solution2, best_solution, nosolution_count, best_3derror, best_gdoprate, p1, r1, p2, r2, p3, r3, p4, r4)
+ get_location(best_sol: Coord3D*, use4anchors: int, anchorArray:Coord3D*, distanceArray:int*): int

4.6 Usage

Complete instructions to use the drone with all positioning systems including UWB are available in Appendix A. This section explains compilation of the programs themselves.

4.6.1 Position visualization in terminal

1. Make sure to specify the anchor positions and the correct USB port in the config.ini file next to the binaries.
2. Open a terminal. On Windows, look for the program "Command prompt". On Linux, use Ctrl+Alt+T. Move to the directory of the program (using the cd command).
3. Start the executable: on Windows, by typing UWB_App.exe, on Linux, ./UWB_App.exe.

4.6.2 Compiling the project

Please follow the subsections in order to compile the project from the source code.

Trilateration algorithm

While the binaries themselves are compatible with Windows, Linux and MacOS, the C++ trilateration algorithm implementation still works with a Linux-only Makefile. The instructions below were tested on Ubuntu 18.04 Bionic Beaver.

1. Open a terminal. The default shortcut is Ctrl+Alt+T.
2. Using the `cd` command, move to the root directory of the source code.
3. Move inside the directory `lib_trilateration`
4. Type the command `make`

ALX OS plugin

1. Compile the trilateration algorithm as instructed above.
2. Launch a C#-capable IDE such as *Visual Studio* on Windows or *Rider* on any platform.

3. Open the solution (sln file at the root directory of the source code).
4. Right click on ALX.Host.Common.Devices.Drivers.UCLIndoorRadio and (on Rider) click on "Build Selected Projects".
5. From a file explorer, the compiled binaries and all their dependencies should be ready in the directory:
   ALX.Host.Common.Devices.Drivers.UCLIndoorRadio/bin/Debug/.

Position visualization in terminal

1. Compile the trilateration algorithm as instructed above.
2. Launch a C#-capable IDE such as *Visual Studio* on Windows or *Rider* on any platform.
3. Open the solution (sln file at the root directory of the source code).
4. Right click on UWB_App and (on Rider) click on "Build Selected Projects".
5. From a file explorer, the compiled executable and all their dependencies should be ready in the directory:
   UWB_App/bin/Debug/.

4.7 I/O Component

An ideal implementation of this complement should have the following characteristics:

- Ability to read data received through the UWB tag;
- Interpret this data as distances and return them to the caller;
- Offer robustness against malformed messages sent by the device;
- Fully recover when distances to some or all anchors cannot be received temporarily;

Messages from the UWB tag are sent through the USB port using serial communication interface (one bit sent at a time).

To stay compatible with both ALX and ROS, two implementations of this component were done: one in Python using pyserial, and another in C# using the

standard library SerialPort. These are the standards in their respective languages, meaning they are well documented and widely used. The logic behind these two implementations is absolutely identical.

4.7.1 Receiving range reports

Reading from the DecaWave EVB1000 unit requires to understand what message it sends through the port.

Range reports are 65 bytes-long messages and sent through the serial port as long as the anchor 0 is detected by the tag. Their format is described as follows: [15]

<MID>␣<MASK>␣<RANGE0>␣<RANGE1>␣<RANGE2>␣<RANGE3>␣<NRANGES>␣<RSEQ>
␣<DEBUG>␣<t0:0>\r\n

Field	Bytes	Description
MID	2	Report type, "mr" for raw ranges or "mc" for ranges where the anchor range bias was corrected.
MASK	2	A bitmask where bit N (starting from right) is 1 if there is a range to anchor N.
RANGE<N>	8	Distance in mm between the tag and the anchor N. 32-bit hex number. If anchor N is not present, unspecified (usually 0).
NRANGES	4	Number of ranges completed by the unit. 16-bit hex number.
RSEQ	2	An increasing sequence number. 8-bit hex number.
DEBUG	8	Time of the last range reported. 32-bit hex number.

```
r 05 00000509 00000000 000003b3 00000000 0291 b9 404d404d t0:00
mcmc 05 000005ab 00000000 0000045f 00000000 02b3 ca 00082641 t0:0
mr 05 000004f7 00000000 00000397 00000000 02b3 ca 404d404d t0:0
```

Figure 4.2: Malformed messages

Arriving messages can be malformed. In the example on Figure 4.2, we can see the first two lines are malformed messages. The "m" of the first message could not be read, and in the second one the range type part is too long. This usually only happens at the start of the reading process, and these few malformed reports are discarded.

Further information could be obtained by reading from the RTLS software source code. It is possible to send a special message "deca?" through the serial port to notify the device we need the information. This was not handled by the original program that would fail.

The USB port still needs to be specified in the project configuration file (see usage section). It could be possible to detect it. Based on the DecaWave code (`DecaRangeRTLS_PC/network/SerialConnection.cpp`), the idea would be to use the port with the description "STMicroelectronics Virtual COM Port". However, getting port description is difficult to write in a multiplatform way.

Problems with improperly closed ports were fixed, meaning the program can be stopped and restarted without removing and reconnecting the device.

4.7.2 Loss of anchor signal and partial data

The loss of an anchor manifests itself in two ways, depending on which anchor fails:

- **For anchors >0**, the mask field in reports indicate which anchor distances are reported. That means we can get partial data, for example ranges to only two anchors. This is useless for the trilateration algorithm which requires all ranges to work. However, other uses might be found. It could give a global indication of where the drone is, which could for instance be used to facilitate the work of a total station that needs to pinpoint the drone in space. Two ranges might also still be useful if the tag is part of a pair of units used to detect the angle of the drone. However, the large angular error may make it a sub par solution compared to the Pixhawk's compass.
- But **if the anchor that fails is the anchor 0**, the device completely stops sending data. Sometimes, the signal of this anchor might not be available, for instance when the drone is close to it. This is a special case that needs to be handled by the program. In that scenario, the program will wait a fixed amount of time (1 second) and then it will re-check if reports are sent. In the meantime, not even partial data is available.

4.8 Positioning Algorithm

The mathematics behind the trilateration algorithm are unchanged from last year. It is an intersection of multiple spheres, each centered around an anchor with the radius being the distance anchor-tag[7].

However the program would support new implementations in the future. To ensure this, an example of optimization-based multilateration algorithm was quickly written. It was, however, subpar in precision and accuracy for moving objects and thus discarded for the project, and extensive analysis of a range of algorithms would be a master's thesis on its own anyway. But it means new, improved solutions could quickly be added.

While DecaWave's IO code and trilateration algorithm is very useful and widely distributed, its implementation was hiding precise error messages to the caller. This was corrected and it is now possible for example to know if the number of detected anchors is not sufficient or if their ranges were too inconsistent to find a solution.

Error	**Code**	**Meaning**
TRIL_3SPHERES	3	Success with 3 spheres
ERR_TRIL_CONCENTRIC	-1	Some spheres have the same center. Two or more anchors are indicated to be at the same position.
ERR_TRIL_COLINEAR_2SOLUTIONS	-2	Some anchors are co-linear and the algorithm cannot agree on a single solution
ERR_TRIL_SQRTNEGNUM	-3	Algorithm has no solution. Usually happens with wrong UWB anchor positions that are completely inconsistent with received distances

That way, exploitation of partial data could also be added in the future. Potential improvements could include deduction of the orientation angle, even when not enough anchors are detected.

4.9 Summary

Despite the initial obstacles with the basis we had to work upon, we accomplished our goals for the UWB technology by writing maintainable and robust program that handles the positioning task in a reliable way. While its precision is always limited, it fills its role as a backup system for the flight of drones.

Chapter 5

Intel RealSense Depth and Tracking Cameras

While a total station or UWB trilateration can be used to compute xyz positions of a drone, those positioning systems cannot give any information about its orientation: yaw, pitch and roll. The combination of a position and an orientation is called a pose, usually meaning a 6 DOF pose.

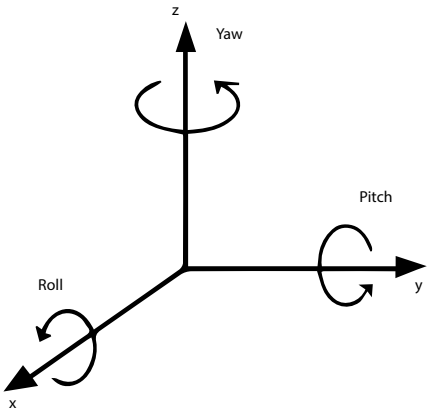


Figure 5.1: Yaw, Pitch, Roll

Orientation is not a prerequisite for simple navigation in space. Indeed, multicopters are able to move in all directions regardless of where they are facing. However, our use case involves the lifting of possibly non-circular objects which requires to be approached with the correct yaw.

Additionally, feeding the FCU this information helps it to improve stability,

although the Pixhawk model we used includes a built-in accelerometer and gyroscope which is enough for simple loitering. For the yaw, it is usually guessed using the compass, but we need precision and most importantly resistance to electromagnetic interference. Indeed, the powerful electromagnet used to carry the blocks causes perturbations on magnetic sensors and the flight controller unit does not allow their usage under that circumstance (See section 8.3).

For all these reasons, we decided to integrate a new positioning system based on visual data. This chapter will describe our solution.

5.1 Camera Models

In early 2019, Intel announced the release of two cameras as part of its RealSense 2 collection for an entry price of US$199 each. They were both generously lent to our project by the company ALX Systems.

Figure 5.2: D435i (left) and T265 (right)

The first one, the D435i, is an infrared-capable depth camera and is an upgrade of a previous model, the D435, with an added inertial measurement unit (IMU, in this case an accelerometer and a gyroscope). Its visual streams include RGB, infrared and depth at various resolutions and framerates.

The second model, the T265, has only two black-and-white cameras but holds a very powerful hardware unit dedicated to process all its data streams that are detailed in the sections below. Its field of view is also much larger (163 degrees[16], against 86 degrees[17] for the D435i), thanks to two fisheye lens sensors.

Feature	**D435i**	**T265**
IMU Sensor	Bosch BMI055	Bosch BMI055
VPU	None	Intel Movidius Myriad 2
Additional sensors	Depth, Infrared, RGB	Two Fisheye Lens
6DOF Pose stream	None	200Hz

5.2 Position from Visual Data: SLAM

Simultaneous localization and mapping is the problem of finding the location and the orientation of an object by mapping the environment around it. The algorithms solving SLAM are the subject of extensive research by many universities, including UCLouvain.

The idea is that the environment is mapped as a cloud of 3D points created from features/details extracted from successive frames sent by the sensor. With multiple frames received over time, it is possible to compare the common features to guess what movement happened.

On Figure 5.3, you can see an example of the cloud of 3D points that was generated when moving inside a circular building[1]. The turquoise points compose the path computed by a SLAM algorithm from the cloud.

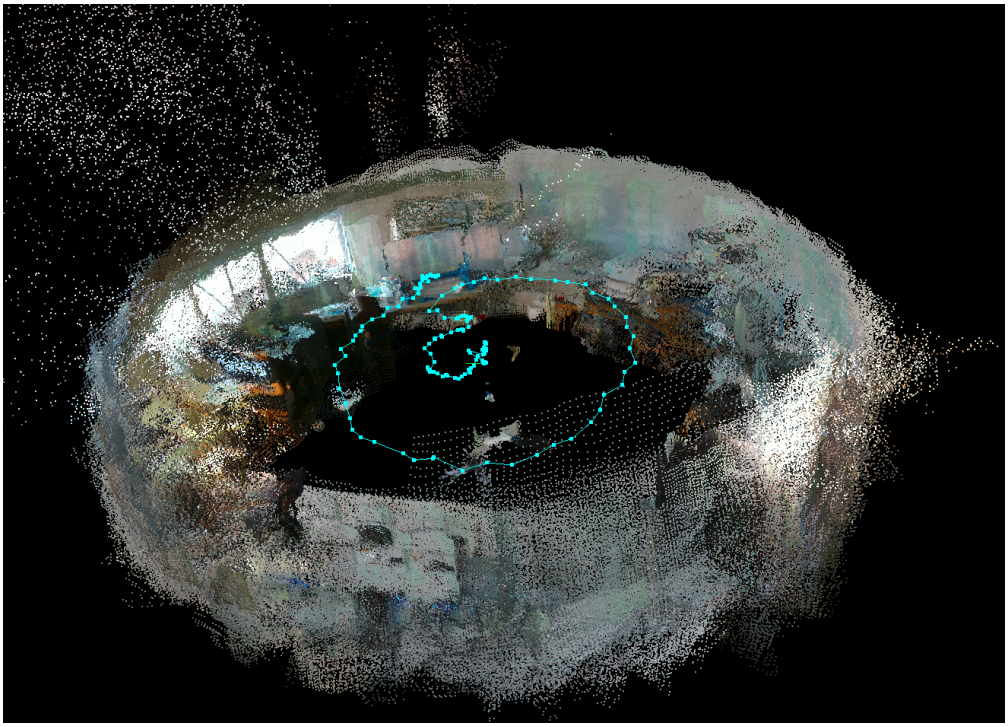


Figure 5.3: Example of a building interior mapped with 3D Cloud

There are many different ways to solve this problem. Important points in our choice were whether direct usability within ROS (to be compatible with our other programs) was possible or of it would require additional work; and whether

[1]RTAB-Map and the D435i were used for this picture

the settings and output data were sufficiently open to allow tweaking and further transformation. We went with different approaches depending on the camera model.

Both the cameras we used are accessible through the RealSense library by Intel[2] which offers a C++ API and ROS wrappers to manipulate settings and data streams of various types. The availability of those streams via the RealSense SDK depends on the device used.

The D435i does not have a direct 6DOF pose available from its streams, but we can compute one from the other streams available. As indicated above, it still has many sensors like colored RGB video, infrared and depth which means more details can be extracted than the T265 (which only has black&white video data) especially in low-light environments.

The second model, the T265 contains an Intel Movidius Myriad 2.0 VPU, a processing unit purpose-built to efficiently execute a proprietary SLAM algorithm on data received from its FishEye lens but also from its IMU (gyroscope and accelerometer). The advantage is double: the camera has a wider field of view and can detect more features in the environment, and the computation of position and orientation is done **inside** the camera, meaning no computational power at all is required from the Odroid. A data stream directly sends 6DOF poses through the USB cable.

5.3 Software Realized

5.3.1 D435i

As explained above, to assess the capabilities of the D435i, we needed to find a way to execute a SLAM algorithm. For that task, we chose to try RTAB-Map, an open-source graph-based SLAM approach maintained by the University of Sherbrooke in Québec. It did not support the D435i out of the box (which is understandable as it is very recent), but we implemented it by editing RTAB-Map to use the proper calls to the Intel RealSense API in order to get data from all streams including the inertial measurement unit. While the results seemed precise, the problem was the performance of the system. Indeed, the high quality of video and depth streams implies this trade-off. Even on a desktop computer, the low speed of the computations was an issue even for testing and we preferred to rather find an alternative and focus on the T265 for general use.

However, it could still be used as a low-light backup system. Indeed, we tried different combinations of resolutions and enabled/disabled certain streams and we

[2] `https://github.com/IntelRealSense/librealsense`

found out that most of the computational workload came from the framerate (90 Hz) and the too high resolution 1280x720 of RGB images. When we decreased the framerate of the video stream to 30 Hz and the resolution to 424x240 (while keeping to take into account the other streams like infrared and depth), speed of the program dramatically increases which leads us to believe that it should be tested as a positioning system. It would be especially interesting to see how it performs in low-light conditions. However, due to a lack of time we could not do this test ourselves.

5.3.2 T265

The T265 Tracking Camera was specifically designed by Intel for the task of positioning.

Intel already published a ROS node to access the data from the device. However, it is not targeted specifically at drones. Indeed, the camera can be used for a variety of usages like vacuum cleaners or automated arms. Thus we had to write a script that remapped the output of Intel's node to the MavROS topic in order to feed the camera data to the Pixhawk.

While we initially encountered `LIBUSB_ERROR_OVERFLOW` on the Odroid, an error meaning too much data was sent through the USB port, it was quickly fixed with additional settings to only send relevant positioning information. Initially, Intel's node was also transferring full video streams through the USB which is useless in our case since the Odroid is a screen-less onboard computer.

Afterwards we were finally able to begin the tests that would assess the capabilities of this camera.

5.4 Interpretation of Tests

The T265 camera has been tested both in real flight and through manual movements. Detailed test procedures can be found in Chapter 8.

On most tests (example below in Figure 5.4), by taking the total station as a reference (which is reasonable given its high precision) an offset of xyz position can be observed. This difference almost always increases steadily with time.

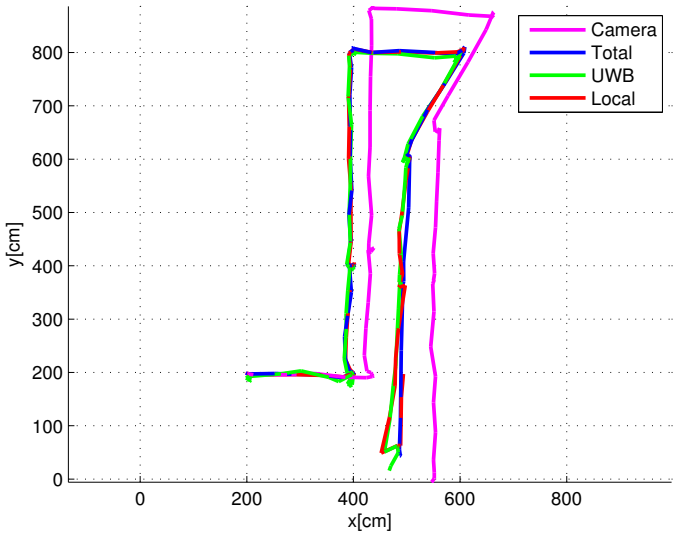


Figure 5.4: Path of the drone in the XY plane during manual movement.

We first needed to determine whether the problem came from the positions sent by the camera itself, or from the transformations we apply to align reference frames between all positioning system.

To do so, we tested multiple times the camera on its own, on similar paths as the other tests. There is indeed also a comparable offset that increases with time. This means the issue is related to the camera itself and not the transformation we apply to the received positions or the positioning system program.

A first possibility would be that the environment of the testing area does not have enough details or features for the SLAM algorithm to compare video frames and recognize its movements. Indeed, if there was a problem with the visual positioning, the camera would rely more on its IMU which would estimate its position based on the velocity computed by the accelerometer, which is well known for its error accumulating over time.

A second would be an error in the software or firmware provided by Intel. The camera is still recent (March 2019) and very experimental, and many issues and fixes are published every week. It is possible that future updates drastically improve the quality and stability of the tracking.

Chapter 6

Global Positioning System

The objective is to create a single positioning system based on all individual positioning systems previously described: local and global image detection, the total station and ultra-wideband technologies. We can decompose that objective as we need to develop a system capable of providing a single global position i.e., relative to a predefined fix frame and a local position relative to the brick we would like to pick up. In this chapter, we will develop a first version of the positioning system that only provides a global position.

To accomplish that task, we asked ourselves the following questions:

- Given we have multiple distinct sources, each providing position, how can we obtain a single one?
- Once an estimated position obtained, how do we communicate it to the drone's flight controller so it takes it into account?

6.1 A Single Position

The constraint stating a single position is required comes from the laws of physics: in a point in time, the drone can physically only be in one exact spot. It is very important for our system to estimate that spot precisely as the autopilot will base itself on that estimation to correct its course towards the destination. The deviation of the drone's course from the intended path grows with the imprecision of the estimated position.

To obtain a single position, the two following strategies are considered:

- Combine the multiple positions to create a weighted mean.

– Select one source at each time, depending on the conditions at that time.

Since the different sources have significant different accuracy values, the first strategy will produce a solution that is less accurate than the most accurate position available as it will also take into account less accurate sources. As opposed to the second solution, which only takes the most accurate source available at each time. The second strategy is the one we used in our implementation.

6.1.1 Frame and reference point synchronization

We previously mentioned that our output position should be relative to a predefined reference frame. This requirement comes from the drone's autopilot, which only uses a single frame to represent the space around it. Since all positioning systems use different frames and we have another frame for the destination points, we must choose a frame of reference to work in and convert all positions to that frame. All information given to the drone i.e. destination points by the guidance system and positioning information will be relative to that chosen frame.

All positioning systems also use different reference points on the drone for which they provide position. We also must choose a single reference point to make all sources cohesive.

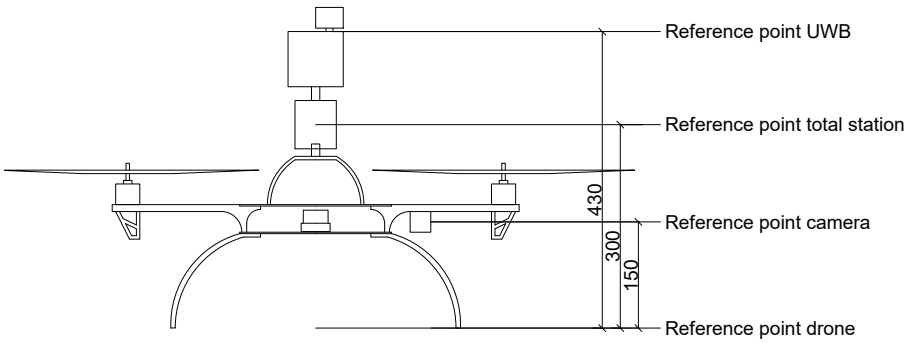


Figure 6.1: Reference point offsets

The reference points and frames of each system are:

– **Total station** The frame is defined during the station's setup by pointing towards the x-axis and at ground level. The reference point is the center of the prism.

– **UWB** The frame is defined by the configuration file with the positions of UWB anchors. The reference point is the base of the UWB tag antenna.

– **T265 Camera** The frame is defined at start-up (it considers itself at (0, 0, 0)), the x-axis is perpendicular to the front of the camera. The reference point is the center of the camera.

We choose to apply transformations to the provided positions such that the reference frame is the one provided by the total station and the reference point is the center of the drone at ground level as can be seen in figure 6.1.

To convert the position received from the UWB tag to that frame, we use the total station to measure the positions of the UWB anchors. Then we generate the configuration file and store it on the onboard computer where it will be used to compute the positions. For the reference point, we apply a translation along the z-axis.

The camera defines its frame at start-up, it will send positions relative to its starting position and orientation. We decided to align the camera with the total station's x-axis so they have the same x-axis. We then apply a translation from the center of the camera's frame.

As the camera has an offset from the Pixhawk, if the drone stays in place but rotates around the z-axis, the camera will describe a circle with a radius equal to this shift, while the reference point stay in place. The position of the camera changes but not the position of the reference point. In order to remove this effect, we apply the following transformation to the position given by the camera:

$$\begin{aligned} x =& \text{cam}.x + \text{home}.x + \text{diff} - \text{diff} \cdot \cos(\text{yaw}) \\ y =& \text{cam}.y + \text{home}.y \qquad\quad - \text{diff} \cdot \sin(\text{yaw}) \end{aligned}$$

Where home.x and home.y are starting position, cam.x and cam.y, relative position given by camera, diff is the distance between the reference point of the camera and the center of the drone and yaw the rotation angle around the z-axis.

Since the drone starts in z=0, no translation in z is necessary, for the effect of pitch and roll on the position given by the camera, we have decided to neglect it because it is negligible compared to the variation due to the yaw.

6.1.2 Decision Process

The decision process defines the source we will choose at each time. It is based on the estimated accuracy of each source, as this is naturally the factor we want to

maximize.

The different positioning systems are ranked by decreasing precision:

1. **Total station** less than 1 cm[6]
2. **UWB** around 20 cm for XY and 40 cm for Z [7]
3. **T265 Camera** over 0.5 m

The positions received from all these systems are monitored continuously but only the current best is used to send data. The two others serve as back-up and verification. If the source currently considered as the best becomes unavailable, the next in line takes over. As soon as one with a higher precision becomes available, the current best is replaced. We implemented a first version that only takes the first 2 positioning systems into account, since they are tested to be reliable. This version allowed us to securely fly. We implemented a second version that adds the camera for manual testing purposes.

Furthermore, since we have 3 systems, we should also check if the best position is coherent with the other two. The reason a minimum of 3 systems is required is to be able to confirm which system should be discarded if two systems are available and they contradict each other. If we detect that a system is incoherent with the two others, it should be replaced by the next in line. Again, if a more precise system becomes coherent again, it will take over. This feature was not implemented due to a yet unreliable camera position.

In the second implementation, we add the orientation provided by the camera at all times since it is the only source of information providing that type of data.

6.2 External Position Estimation

Now that we have decided which position will be taken into account, we need to be able to give that position to the flight controller. The documentation of the PX4 shows position estimates can be sent from an onboard computer using vision capture systems. Any system providing positions can use the same communication channels i.e., ROS topics streaming specific message types provided by the mavros package mentioned in section 2.4.

As can be seen on page 19, our onboard computer is a publisher of the topic `/mavros/vision_pose/pose` and the flight controller's LPE position and attitude estimator is a subscriber. The type of message that is sent on that topic is `PoseStamped`, containing a position and orientation.[1].

[1] A Pose with reference coordinate frame and timestamp

```
PoseStamped = {
      Header header = {
               uint32 seq
               time stamp
               string frame_id}
      Pose pose = {
               Point position = {
                        float64 x
                        float64 y
                        float64 z}
               Quaternion orientation = {
                        float64 x
                        float64 y
                        float64 z
                        float64 w}}}
```

As discussed in the previous section, this message is updated when data is received and it is published at a frequency of 20 Hz i.e., the recommended frequency.

On the other end, the LPE position and attitude estimator will receive these messages. This module is an extended Kalman filter for 3D position and velocity states. It means this module applies sensor fusion on the messages with the values of its internal sensors[2]. The filtered result is used to update the local position of the autopilot i.e., update its internal state.

For the LPE estimator to integrate the vision position we provide, parameter tuning is required (see section B.2). Other estimators are available but according to the PX4 documentation, only the LPE estimator is compatible with vision integration[19].

An interesting fact, our frame follows the ENU convention while the drone's PX4 frame follows the NED convention but mavros does the conversion automatically, so no need to manually convert.

6.3 Implementation Details

We implemented this system as a ROS node in the Python language.

We use a multithreaded architecture containing one data reader thread per

[2]Defined as the "combining of sensory data or data derived from disparate sources such that the resulting information has less uncertainty than would be possible when these sources were used individually."[18]

data source and a single publisher thread. We use multiple variables protected by locks, such as the messages of the different sources and the best source currently available.

All reader threads are launched at the start of the program. The publisher thread is launched by the first thread receiving data.

The first thread reads the total station data. It opens a socket through USB and waits for data to arrive. Once data arrives, it updates the total PoseStamped message. If the wait reaches a timeout, the current source is updated until the next data message.

The second thread reads the data received from the UWB system. It starts by setting up the tag handler and then waits for data. Like the total station reader, it will update its PoseStamped message.

The third thread reads the information from the camera, applies the transformations to the positions, and updates its own message.

The publisher thread loops at a constant rate (20 Hz). Each cycle, it picks the best currently available source for both positions, and uses the camera's orientation. This pose is then sent to the `/mavros/vision_pose/pose` topic to feed the flight controller unit.

Chapter 7

Guidance System

We can define a guidance system as a process that controls the navigation path of a moving object in real time. This system allows the drone to autonomously execute a flight mission, while simultaneously maintaining certain safety measures. Since the objective of this project is to enable a drone to build a masonry wall, we must create a guidance system for it to be able to move around autonomously.

In this chapter, we start by defining the execution of a standard flight mission during the construction process. We then develop a simplified version of this execution and its implementation in the ROS environment.

7.1 A Standard Construction Mission

The objective of the mission is to build a structure. Since this problem is pretty complex, we need to decompose it into smaller parts. This will make it easier to conceive, program and later on to maintain. Equally it should improve general readability and ease of understanding.

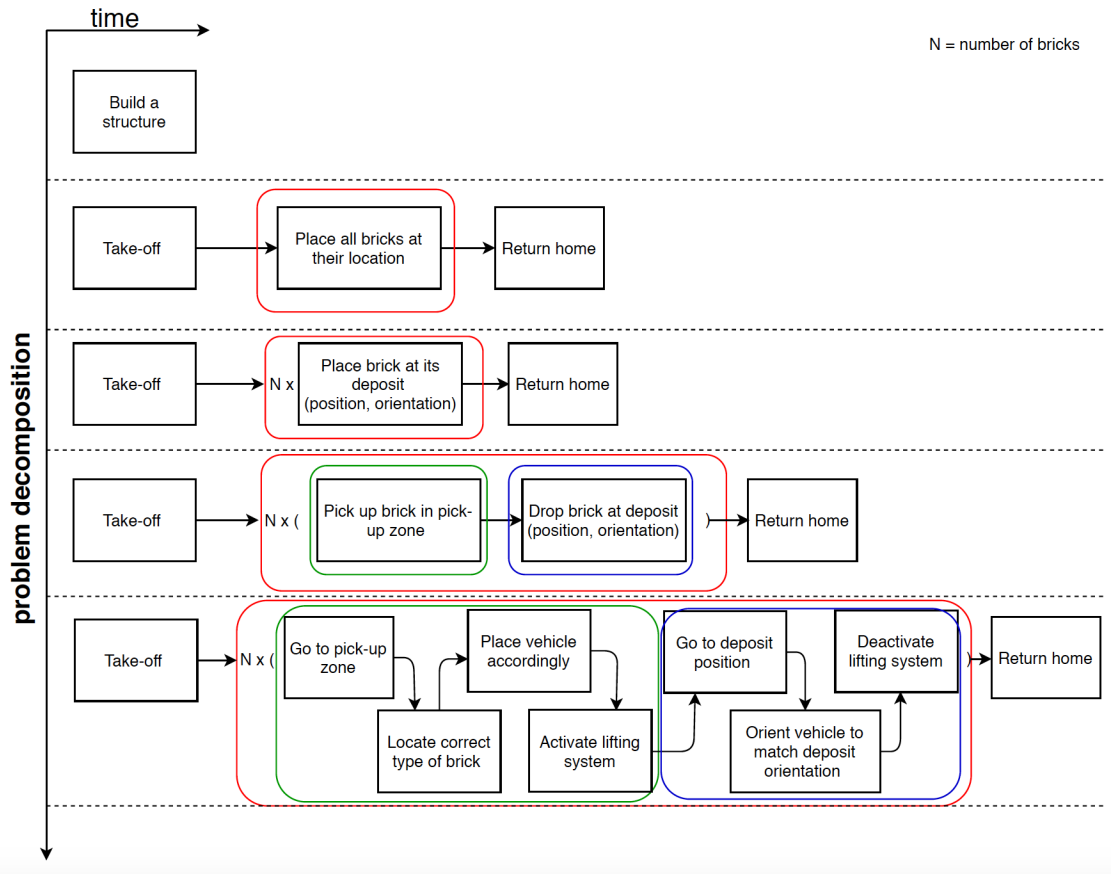


Figure 7.1: problem decomposition of flight mission

In figure 7.1, we observe the different actions we must be able to accomplish with our guidance system, graphically represented in figure 7.2. There are basically 3 types of actions:

1. Going to a position and orientation i.e., a setpoint;
2. Activating/deactivating the lifting system;
3. Locating a brick by scanning the pick-up zone;

Each of these actions has requirements to be able to function: a functional positioning system to calculate the movement it needs to make to go to the setpoint, a functional lifting system and an image detection system to scan/detect the bricks.

We can also deduce the different pieces of information we must provide the system to be able to accomplish those actions.

Mission = {
home position: $\{p_H\}$
flight height: $\{h\}$
scanning height: $\{sh\}$
pick-up zone: $\{p_1, p_2, p_3, p_4\}$
the 4 corner positions of the zone where the bricks are stored in
brick info:$[\{type_i, p_i, o_i\}]$
for each brick, the brick type the deposit position and orientation where the brick will be dropped, aka an ordered list }[1]

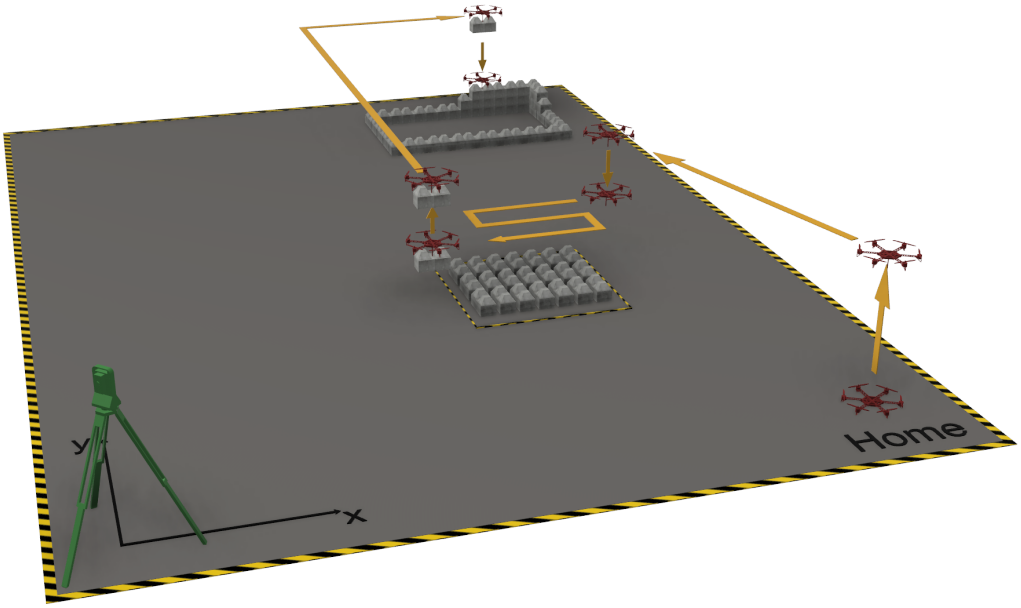


Figure 7.2: Standard mission execution

7.2 The Simplified Version

In section 7.1, we stated the 3 types of actions our final system must be able to do. Our development strategy was to implement actions one by one starting with the most important: going to a position, orientation (see section 7.3.3). Once completed, we added the second one: activating/deactivating the lifting system

[1]where p is a position in space $p_i = [x_i, y_i, z_i]$ and o is an orientation in space $o_i = [yaw_i, pitch_i, roll_i]$

which in our case is an electromagnet (see section 7.3.4). For the third action, the image detection algorithm developed in [8] needs to be integrated in the positioning system described in section 6. This is future work.

The simplified version can be used for basic flights and is a base structure for a more complex system. It shows how to use the different communication channels with the PX4, which are often among the most complex steps to figure out.

We worked with a list of commands, rather than the mission structure defined above as this simplified testing. Adapting the project to the above structure would not require much change but we needed to be more free with our actions. Instead of going from a pick-up point to a deposit point, we follow a list of commands, which is either going to a position or activating/deactivation the lifting system. Equally, we defined the home position as the starting position.

Algorithm 1 Simplified mission execution

Input:

`State` =		vehicle's current state
`CommandList` =		ordered list of the `Command`s to execute
`Command` = {	{X Y Z Y P R},	Go to position (X, Y, Z), orientation (Y, P, R)
	'D',	Deactivate electromagnet
	'A'	Activate electromagnet}

1: `Home` ← *GetCurrentPosition*()
2: `State` ← *Armed*
3: *Execute*(`Home.x`,`Home.y`,`Home.z+1`,0,0,0)Take-off
4: **for all** `Command` in `CommandList` **do**
5: *Execute*(`Command`)
6: **end for**
7: `State` ← *Disarmed*

Currently a command can either be to move to a given position or to activate/deactivate the electromagnet to pick up/drop a brick. We use pattern matching to differentiate the different commands.

7.3 ROS Implementation

The first step of understanding how to implement the execute function of the guidance system is to understand the information the flight controller uses to make the drone fly.

Based on the definitions in section 2.1.2, we know that we will need to give the correct data to the position controller. The parameters that will affect the position controller are the vehicle's state and the setpoint. At each point in time, these must be coherent to what we need in the mission.

Similar to the positioning system, the guidance system is a ROS node that communicates with the flight controller over the MAVLink protocol using the ROS mavros package. The different communication channels between the guidance system and the Pixhawk can be seen on page 19.

7.3.1 PX4 State: Flight Mode

The manner in which the drone will react to remote control input and the way it will manage its movement during fully autonomous flight missions are defined by its flight mode. The PX4 software provides a number of different modes, each providing different levels of autopilot assistance: from automated takeoff and landing to holding the vehicle's position. During a flight, one might want to transition between modes, which is possible using switches on the remote control, with a ground control station or via a specific ROS service: `/mavros/set_mode`

An extensive list describing all existing flight modes for our flight controller is provided in the PX4 documentation. [20]

Our system uses the following modes :

1. `Offboard`: "Vehicle obeys a position, velocity or attitude setpoint provided over MAVLink (often from a companion computer connected via serial cable or wifi)."[20]

2. `Land`:"Vehicle lands at the position where the mode was engaged."[20]

3. `Hold`: "Vehicle hovers at the current position and altitude".[20]

When reading the documentation, we notice another interesting mode: `Mission`. A flight mode designed for executing a predefined flight plan. We decided, however, on using the `Offboard` mode instead since it provides more flexibility during mission execution. For example, it gives us the liberty of adding mission commands such as activating the electromagnet in the mission.

Thus before starting the flight mission, we change the state of our vehicle to put it in `offboard` mode. We do this by sending a message to the `/mavros/set_mode` service.

7.3.2 PX4 State: Arming

To be able to fly the drone must be armed. This can be done using the ROS service `/mavros/cmd/arming`.

7.3.3 Setpoints

We provide the setpoints i.e., the positions the drone should reach in our mission, to the position controller via the ROS topic `/mavros/setpoint_position/local`. The type of message that we send over that topic is PoseStamped: a Pose with reference coordinate frame and timestamp.

```
PoseStamped = {
      Header header = {
              uint32 seq
              time stamp
              string frame_id}
      Pose pose = {
              Point position = {
                      float64 x
                      float64 y
                      float64 z}
              Quaternion orientation = {
                      float64 x
                      float64 y
                      float64 z
                      float64 w}
}}
```

7.3.4 Electromagnet

We developed this part in close collaboration with Mathieu Monnart. More technical details can be found in his thesis.[21]

The lifting systems of the different types of bricks have the common point of being driven by an electromagnet. We therefore focused on the possibility of operating an electromagnet from the onboard computer. Our investigations began using only an Arduino MEGA 2560 board. On this type of board, it is possible to load a program that runs in a loop as long as the board is powered. This allowed us to test that it was possible to trigger a magnet with a relatively low voltage.

We actually use the voltage delivered by the Arduino to operate a switch, which allows the battery current to operate the electromagnet.

The first solution we considered was to create a communication between the Odroid and the Arduino board. By using an Odroid Shifter Shield, especially pin 10 and connecting it to the Arduino, on pin RX1 (19), it is normally possible for the Arduino to react according to the signal. Please note that the Arduino must be powered by the Odroid in order for these two to be synchronized.

This solution did not succeed because only the reading of a signal from the Arduino to the Odroid to work. Following this, we decided not to use the Arduino anymore.

We have therefore focused our research in two different directions. The first was to use a USB switch and the second was to use only the Odroid Shifter Shield.

The USB switch solution was promising, because it only asked the Odroid to provide the power needed for the switch and not the power needed for the magnet. This would have allowed us to do without the circuit we used with the Arduino.

For the solution involving only the Odroid Shifter Shield, we realized that by using the WiringPi library specific to Odroid, it was possible to obtain a result similar to what we had with the Arduino, with the difference that here, we can interact with the electromagnet using a Python or C script. Given the results involving this method, we did not pursue our research with the USB Switch.

With the small drone, we use a 12 V electromagnet, and in the future, the magnets used will have a voltage of 24 V. It is obviously impossible for the Odroid to directly provide either tension. We use an electronic board where there is a switch that allows or not the current coming from the battery to pass through. It is this switch that is controlled by the 3.3 V that provides the Odroid.

During our first tests with the electromagnet, we encountered a problem that we did not have with the Arduino. It was impossible to demagnetize the electromagnet. This is due to the fact that Odroid has a floating state and therefore a parasitic voltage prevents the electromagnet from cutting itself off. To solve this, an optocoupler has been added just after the input signal and allows two circuit parts to be galvanically isolated. It is therefore an infrared LED and a receiver that allow the signal to be transmitted. Without this device, it is impossible to get rid of the parasitic voltage and therefore to control the electromagnet.

A scheme of this board can be found in Appendix F

Since it is possible to control the triggering of the magnet from a Python script, we were able to add this action in the possible actions for the flight missions realized by the drone.

7.4 Safety

This is implemented as a thread that takes input from the ground computer, via an SSH connection. We use pattern matching to detect the command to execute.

At any point during the execution of a mission, we must be able to take control of the drone. This is vital for safety reasons, as the drone may present a strange undesired behavior and might harm persons in its direct environment, cause damage to construction materials or even damage itself. These accidents can have moral and financial repercussion we definitely want to avoid. We might simply want to cancel the mission, for other reasons.

We provide the following commands, any of which interrupt the ongoing mission and execute instead. We must note that certain commands, in certain cases might also damage the drone.

- **DISARM**: This command will force the drone to cut off all its motors. As a consequence, the drone will fall to the ground and most likely break. This command is to be used if the drone presents a real danger to its environment and nobody is underneath or in proximity to the drone! Pieces might bounce on the ground and scatter all around. This is the quickest way to neutralize the drone.

- **LAND**: This forces the drone to land at its current position. If the drone carries a drick or is above an obstacle, this can be problematic. This command can either be used when there is an indirect danger (for example wind) or to get the drone to safety (when no brick or obstacle is present). The flight mode is switched to `Land`.

- **LOITER**: This command will command the drone to stay idle at its current position. Obviously, this command must be followed by any other command to get the drone on the ground. The flight mode is switched to `Hold`.

- **RETURN TO HOME**: This will ask the drone to follow its way home, the starting point of the drone. The setpoint is updated to 1 m above the home position and then to the home position.

- **DEACTIVATE ELECTROMAGNET**: The electromagnet will stop being powered. As a consequence, any attached drick at the time will fall to the ground.

- **MANUAL**: This command will force the drone to switch to a manual mode, allowing a person to regain control with the remote control.

Chapter 8

Experiments

In order to test the global system, we performed different types of tests. The first type of tests, which we will not develop in this document, was performed with the remote control to ensure that the drone is functional, that the engines are able to lift the weight of the drone and its various equipment.

We then tested the accuracy of the positioning systems by moving the drone by hand. Once this step was validated, we carried out increasingly sophisticated autonomous flights, always aiming to quantify the accuracy and the correct functioning of the guidance system. Lastly, we executed a flight with a block that was placed on another.

It should be noted that we only performed these tests with the small drone, simply because we did not have guarantees on the safety of the vehicle. The less expensive and dangerous drone was the better solution.

Local position

Before presenting the different tests we have performed, it is necessary to introduce a new position[1] that appears in our results: the local position. This is the position that is integrated by the flight controller which it determines based on the position the onboard computer provides and his internal sensors. This is represented in Figure 8.1. His movements will only be based on that estimated position. This position should be consistent with the position we give it, since the first is a filtered version of the later. During the first testing stages, we noticed that the integrated altitude of the drone was nonsensical because it did not integrate our position correctly. This was caused by the fact, the position we provided was too different from the one the drone had. It decided to ignore our position since the delta was too

[1]Even though it is called a position, it includes rotation data

big. By activating the barometer, the drone's position returns within an acceptable range to our position and it then decides to integrate it. Once integrated, we do not need the barometer input anymore.

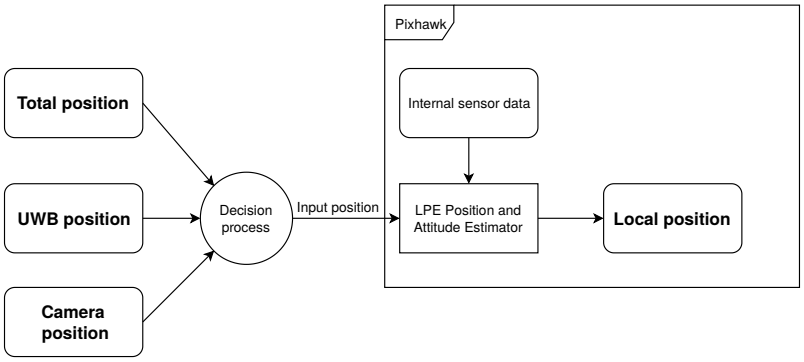


Figure 8.1: Local position computation

8.1 Positioning Tests

As mentioned above, these tests were performed with the drone engines off. We firstly tested the different positioning systems when the drone was stationary, then we repeated these tests while moving it by hand.

8.1.1 Fixing issues

Before flying the drone, it was very important to know if the FCU properly integrated the positions that our programs were sending. To verify that, we took the position given by the UWB (which was the first system we integrated) and the local position of the Pixhawk (which is the position the Pixhawk believes to be at). We took the drone by hand and lifted it before putting it on a table.

Fig. 8.2 shows the two position streams. The clear difference between the two curves indicates that the Pixhawk did not integrate at all the position of the UWB when we moved it. There are still modifications of its position, due to its internal instruments.

To fix that we activate some parameters for the data fusion, as explained in section B.2.

A second problem was revealed later on, still by moving the drone by hand. We realized that the value of y axis given by the total station was sometimes suddenly negative as we can see on Fig. 8.3. The value is exactly opposite of what was

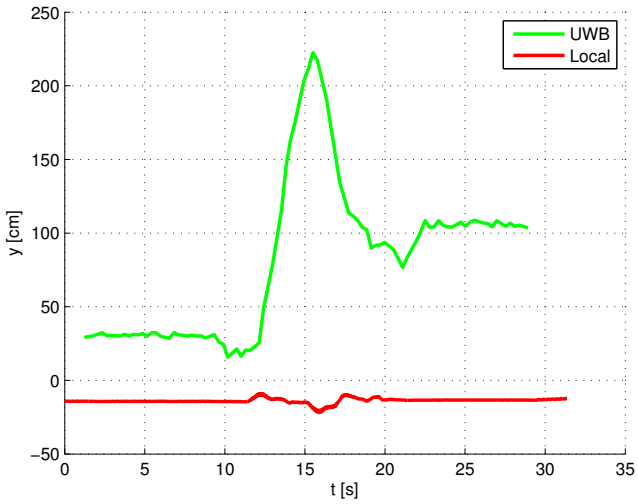


Figure 8.2: Incorrect integration, by the Pixhawk, of a given position by an external source

expected. Looking at the code, we realized that when the station could only track the prism with reduced accuracy, the value of y-axis was incorrect. By modifying the code, this problem was solved.

8.1.2 Quantification of accuracy

Static accuracy

We quantified the accuracy of the 3 positioning systems when the drone is static, on the ground. In figures 8.4 and 8.5, we can see the dispersion of positions with respect to the initial position for the three axes, for each system. This type of graph represents the dispersion of a variable. In red, the median and in blue, the first and third quartiles.

We can observe that for the total station, the variation is in the order of a millimeter. For the UWB, the inaccuracy increases by a factor of 10. For the camera, the precision appears quite fine but we will see later that once in motion, this is no longer really the case and that the station remains the most reliable source.

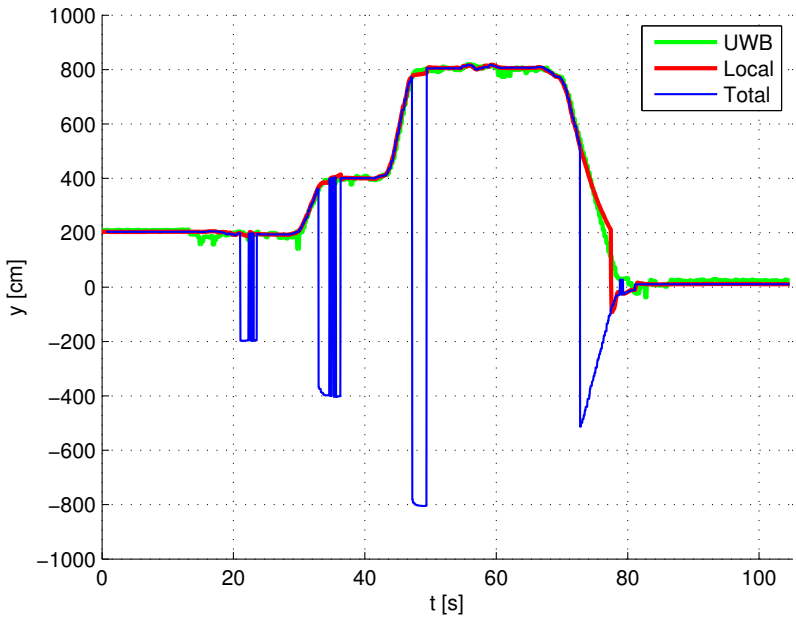


Figure 8.3: y position of the drone according to the total station

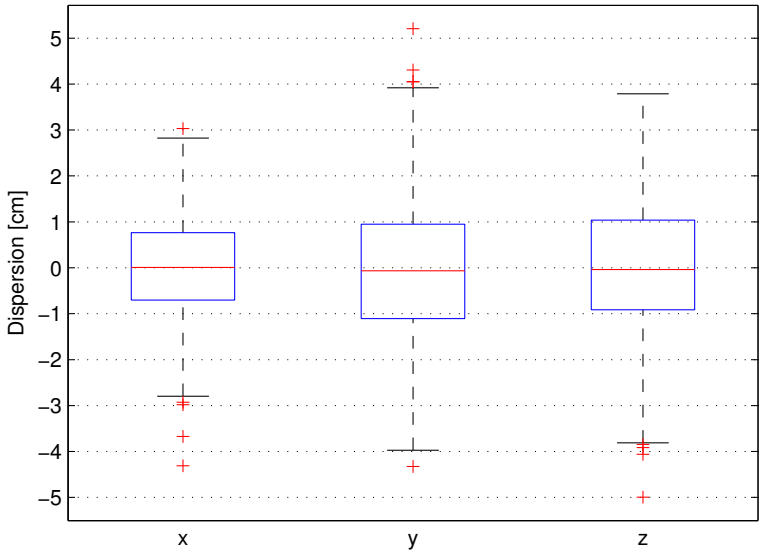


Figure 8.4: Dispersion of the UWB position, re-centered around zero, when drone is static

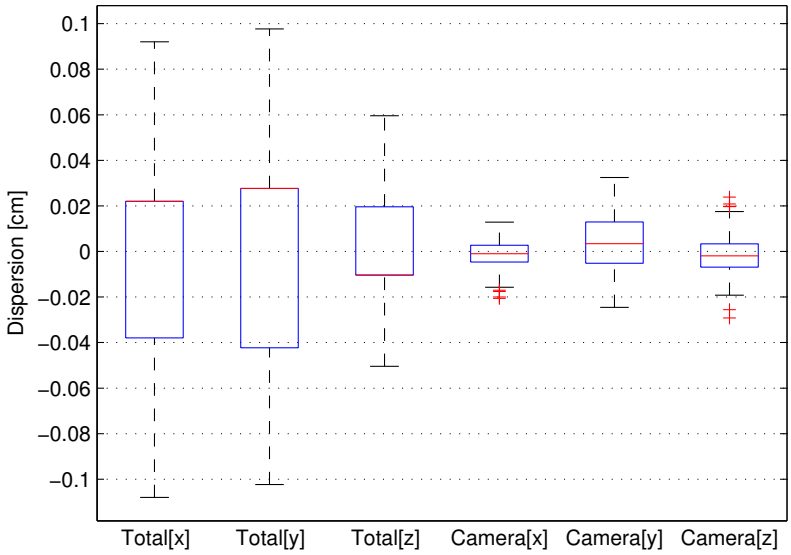


Figure 8.5: Dispersion of the total station and camera position, re-centered around zero, when drone is static

Dynamic accuracy

To compare the different positions we will refer to the total station as the reference position, since we know it is millimeter precise as specified in its documentation. This means, we trust the station to have correctly measured our route.

We defined different points and we moved the drone from point to point, putting it down a few seconds each time. In Figure 8.6 we present the variation in distance of all different positions from the position given by the total station.

We can see that the accuracy of the moving camera is not satisfactory at all, the deviation is more than 1.5 m when the route was a couple of meters long. In figure C.1 in the appendix, we can see the positions of the camera in relation to the others.

The expected performance of the camera was not met in our tests. We tried to record the positions with a laptop, while recording the positions of the total station to compare with a reference. Although the result is not perfect, we can notice (Fig. 8.7) a clear improvement in the accuracy of the camera, the deviation slightly decreases, as we can see on Fig. 8.8.

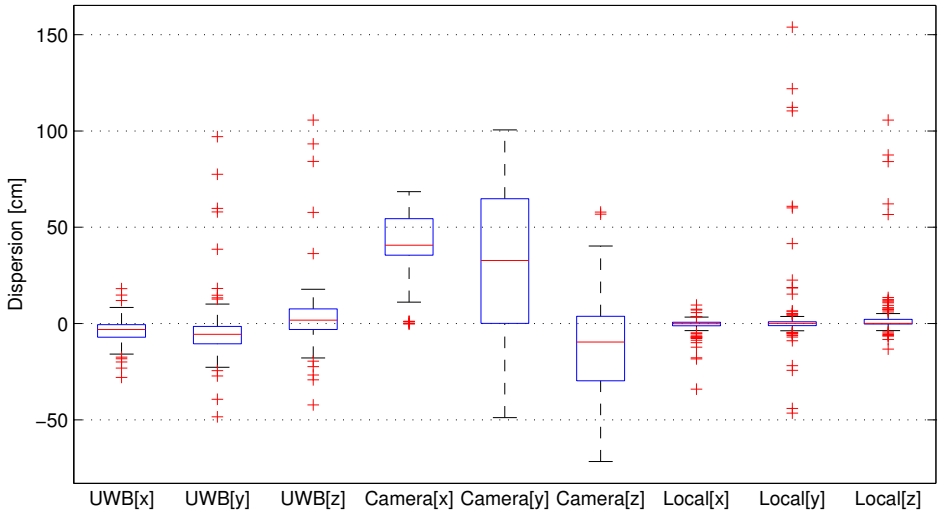


Figure 8.6: Dispersion for each axis and each system during the dynamic accuracy test in reference to the position given by the total station

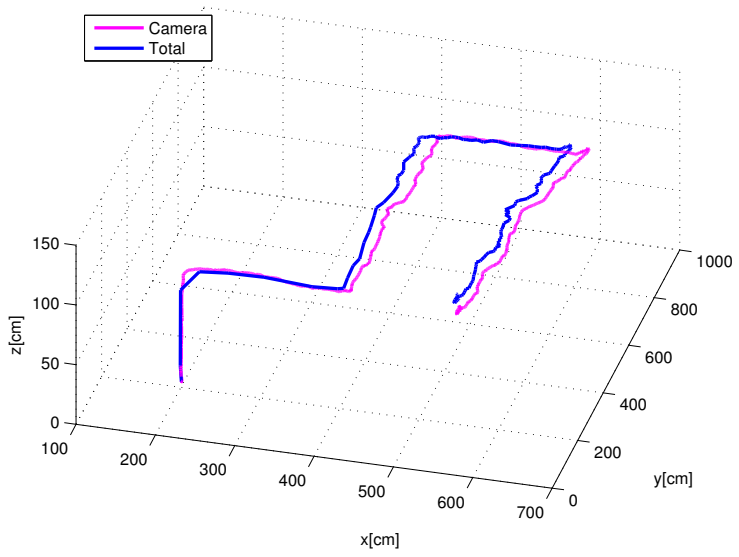


Figure 8.7: Prism and camera path when connected to a laptop

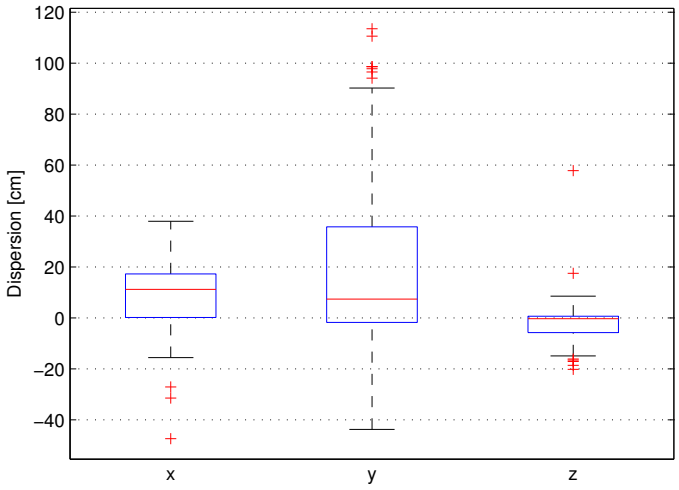


Figure 8.8: Dispersion of the camera position with respect to the total station during the same test as Fig. 8.7

8.2 Autonomous Flights

8.2.1 Stationary flight

In order to know the accuracy of the stationary drone, we recorded its positions once it was stationary. The trajectory of the drone during this flight is represented in 2D and 3D in figures 8.11 and 8.12, as well as the circle or sphere in which it was moving, their radius is 35.47 cm. We can see on Fig. 8.9 that in x and y we are accurate only at about fifty centimeters. This tells us that the global positioning system does not seem precise enough at the moment to pick up and deposit bricks accurately.

It is therefore necessary to use a system that allows the drone to orient itself locally with respect to the brick once it is hovering above them. A system based on computer vision was already developed last year by T. Jacques and F. Kaczynski[8].

During this experiment, we were again able to compare the different positioning systems. Figure 8.10 shows the position of the z-axis as a function of time, and we can see that there is an offset between the position given by the UWB, the total station and the camera. For the UWB, the offset is mainly present in z, while for the camera we observe it on the for each axis (See figures C.2 and C.3). Also this offset is constant for the UWB while it varies with the camera. We can therefore say with some confidence that the offset of the UWB is due to an error

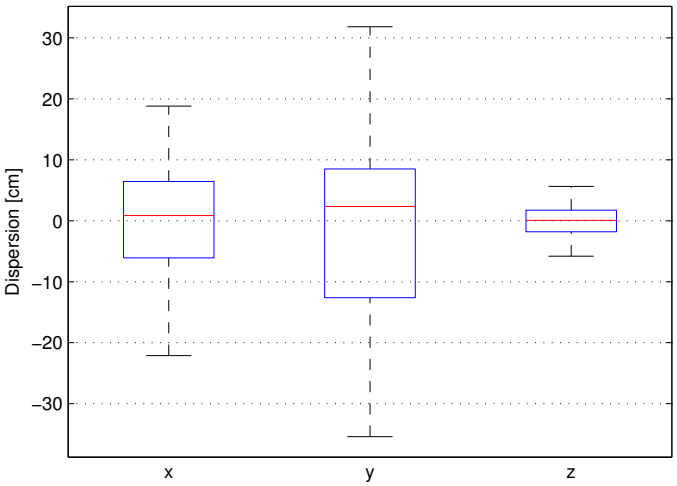


Figure 8.9: Dispersion for each axis measured by the total station during the stationary phase

in the configuration, while the camera accumulates measurement errors, since it is synchronized with the position of the total station at the beginning and at the end of the flight, there is an offset.

8.2.2 Mission

After the stationary flight, we simulated a mission at that time without the electromagnet. Here are the steps of the mission:

1. Takeoff up to 2 m.
2. Flight to the block storage area.
3. Entry into the area, descent to 1 m to be able to analyze the blocks.
4. Route through the storage area in search of a particular type of block (this simulates the block search performed by a local system such as image processing).
5. Block found, down to get it.
6. Climbed up to 2 m to fly to the landing area.

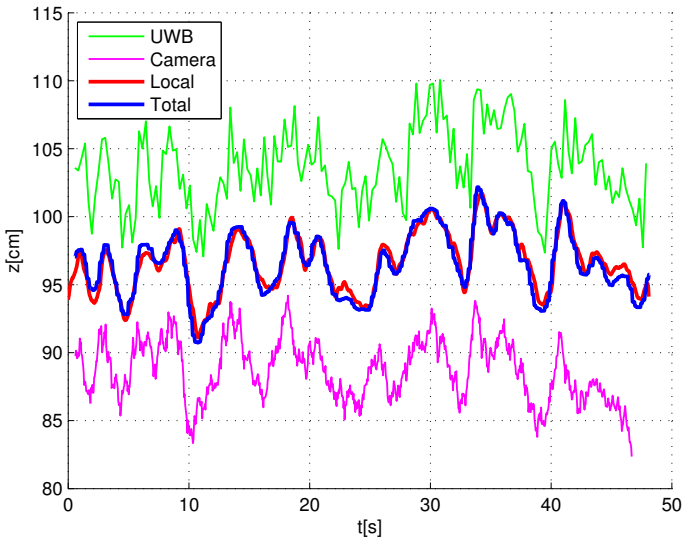


Figure 8.10: Position (z-axis) given by different location systems, only during the stationary phase of the flight

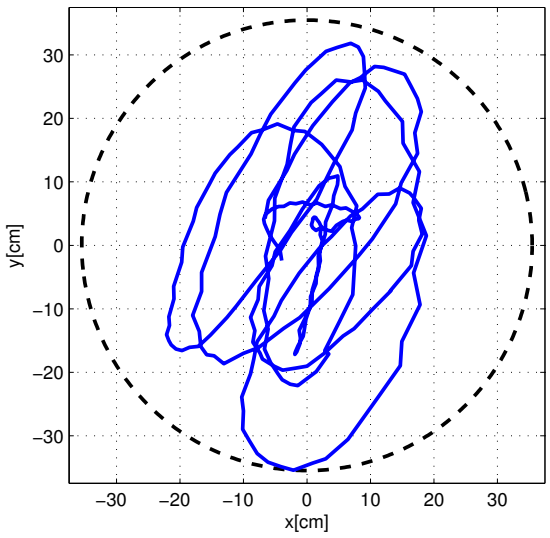


Figure 8.11: Trajectory during the stationary phase which lasts 48 sec.

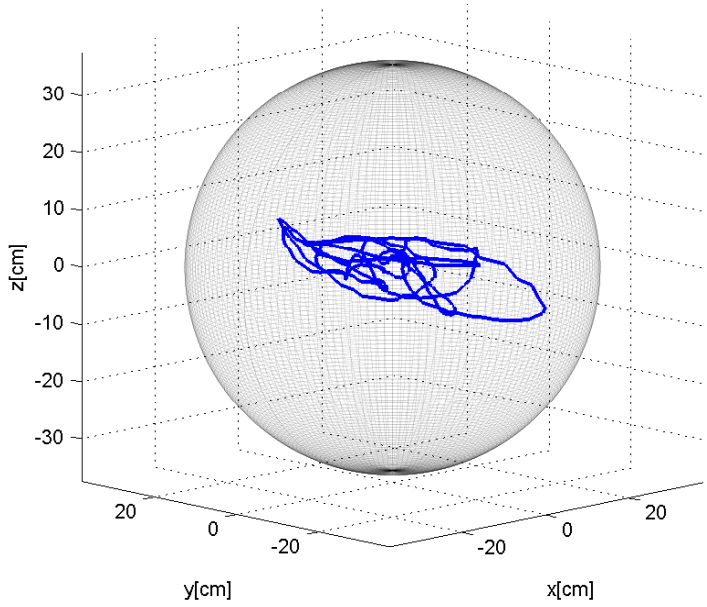


Figure 8.12: Trajectory during the stationary phase which lasts 48 sec.

7. Down to place the block (symbolized here by the landing of the drone).

In figures 8.13, 8.14, 8.15 we can see the positions given by the different systems, respectively for the x, y and z axes. The black curve represents the setpoints to be reached. (Three-dimensional trajectory: Fig.C.4)

We can see several things on these different graphs.

First, for the x axis, we can see that at each takeoff the drone tends to move a little less than one meter forward. One hypothesis is that the drone is not well balanced because of the presence of the camera, which induces a forward movement when the drone leaves the ground, which is then compensated by the flight controller.

Secondly, for the z-axis, we can see that when the drone climbs to reach a set point, it does so quite quickly until it reaches about thirty centimeters from the target, where it slows down to precisely reach its target. This behavior is related to the flight controller.

Thirdly, still for the z-axis, we can notice that when the drone makes a dive, when it descends quickly, the camera exaggerates the movement and, in this case, goes down more than two meters too low at t=80. The error is partially corrected during the upload but is repeated in t=100.

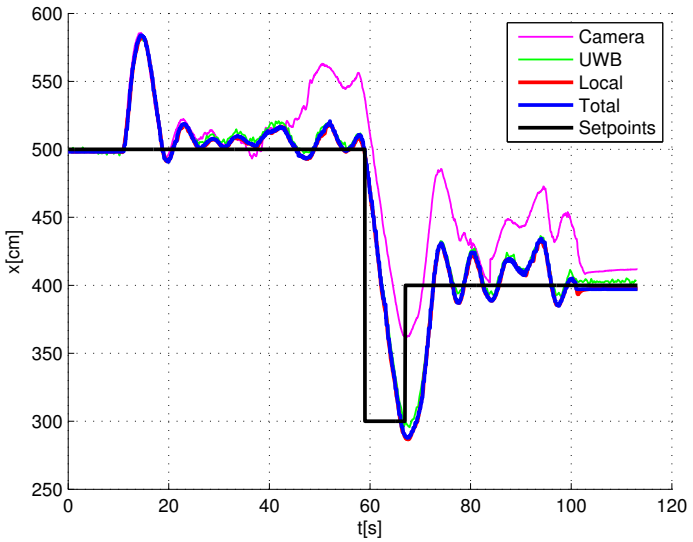


Figure 8.13: Positions and setpoints along the x-axis

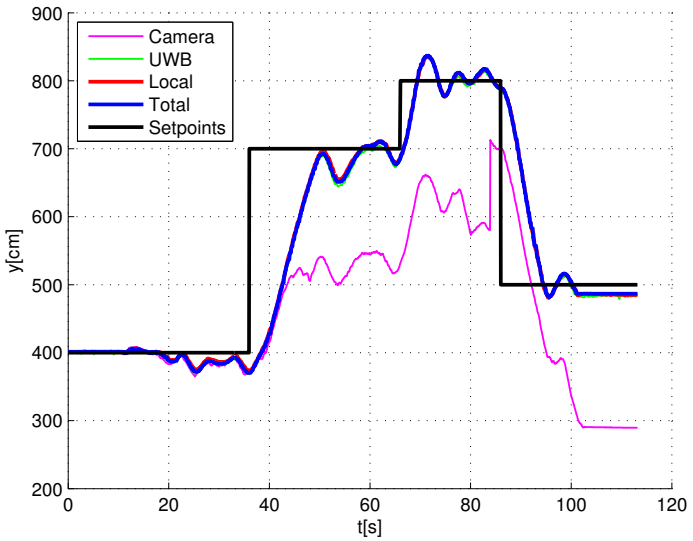


Figure 8.14: Positions and setpoints along the y-axis

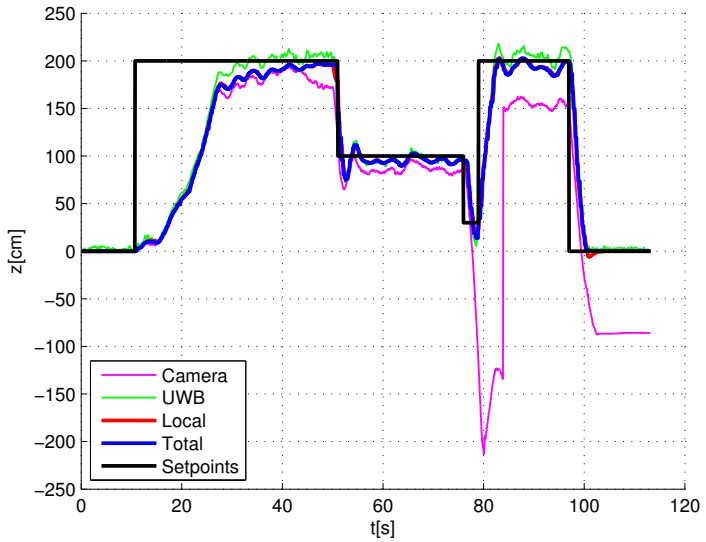


Figure 8.15: Positions and setpoints along the z-axis

8.3 Flight with a Drick

While it would have been a great experiment to approach a drick on the ground and lift it, this was not possible as the approach phase would require a separate local position system (using for example image processing) to align the drone. However dropping a drick could be tested by pre-attaching it before flight.

We initially encountered a problem: the compass of the Pixhawk suffered from interference. The compass being the only instrument to know the yaw (since the camera was not integrated yet), a safety measure from the FCU prevented the drone from arming and flying. Happening as a result of installing the electromagnet, we investigated different possibilities to prevent its magnetic field from interfering.

Two solutions are possible, both of which moving the electromagnet away from the Pixhawk when in use. The first is to use larger feet and a longer support. The second is to use a telescopic support that extends on takeoff and closes again during landing. The two systems are illustrated in figure 8.16.

This problem should not occur with the medium and large drones because their electromagnet is further away than on the small one.

When we tested these two devices, we realized that even by displacing further away the electromagnet, it was sometimes still impossible to take off. The real culprit was actually the metal plate that was used to fix the electromagnet. By replacing the entire device with a plastic equivalent, we succeeded in lifting off the

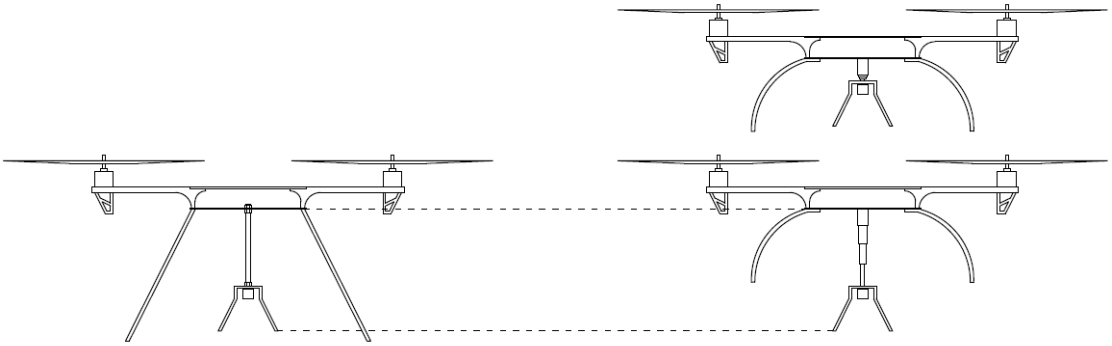


Figure 8.16: Difference between the two types of support

drone with a block and dropping it in flight.

8.4 Summary of Tests

We confirmed the total station to be our most accurate technology, and that the approximation provided by UWB units is of sufficient accuracy to act as a backup positioning system.

Concerning the T265 camera, its imprecision seems superior to the other two systems which was unexpected. However, as also discussed in Section 5.4, its capabilities could potentially improve depending on updates by the manufacturer and it would be interesting to investigate it further in a couple years.

In the current state of the project, the small drone is able to maintain itself within a sphere of 35 cm radius around its target position. This inaccuracy is too large to allow the manipulation of constructive elements. As our positioning system is precise at the millimeter level most of the time, we believe that this imprecision cannot be solved by using more precise positioning technologies. The stability of the drone is also influenced by its size, its motors, its overall structure and the physical placement of its components on the main frame. Reviewing these elements could yield valuable improvements.

The drone in its current configuration already works well for the very important tasks of flying between various predefined points, releasing a payload using the electromagnet and landing once it is done.

Chapter 9

Improvement Perspectives

Currently, the UWB system uses an algorithm based on the intersection of spheres with the anchors as centers. There are many variants of multilateration algorithms, for instance optimization-based solutions. It would be interesting to test a variety of them to find which one suits our usage the best.

When the total station looses the prism, for example when the drone moves too fast, it tries to find it around the last known position. This could be improved by using the other positioning system. The position computed with UWB units could be used as a clue for the Total Station to direct its searching area.

The T265 camera has potential but its software and firmware support by Intel is still very experimental. New fixes are published every week. We would recommend reassessing its capabilities in one or two years, when the technology will be more mature. Additionally, the D435i could be an interesting candidate for collision avoidance and perhaps a low light positioning system using its infrared sensors.

In our mission definition, we mention the scanning phase over a predefined pick-up zone. For this phase to be possible, the image processing software developed last year by T. Jacques and F. Kaczynski[8] should be integrated in our positioning system.

The precision in-flight of the drone still needs to be improved to meet requirements. Our positioning system is already very precise and further development would probably not yield a greater flight accuracy. Instead, further research should be directed towards improving the stability of the drone, for instance by reviewing the motors, its structure, the placement of components and finding the ideal weight.

A great addition to the guidance system would be an easy-to-use frontend. A graphical user interface (possibly web-based, accessible through any web browser) would be easier to manipulate.

Finally, new safety measures should be implemented. There is no precisely defined behavior at the moment when the drone looses its connection to the ground computer. It might also be a good idea to allow the parametrization of safety procedures, to offer choice to the user.

Chapter 10

Conclusion

With this master's thesis, we presented multiple technologies we used and the software we wrote to further approach the goal of building structures with drones.

Firstly, we reviewed what equipment and software we had at our disposal for the project before presenting our precise communication schemes for both ROS and ALX Systems, offering a clear overview of the complete system.

After that we extensively presented our three positioning systems: the total station, the UWB units and the cameras. For each of them, we described the prior work realized during past years and how we improved upon it to allow integrating all pieces together.

Those technologies were integrated together in the positioning system through a strategy that chooses the most trustworthy available source for position and orientation in order to send a unique information to the drone that is as accurate as possible. This master's thesis also went into details about how this information is sent to the flight controller unit.

Once this was finished, we presented the design and our implementation of a guidance system for construction missions. It allows sequences of flight waypoints and control of the electromagnet.

The multiple experiments we did to back our claims were detailed afterwards with each environment, testing procedure and results explained.

Finally, we saw future leads for improvement to build the project further upon the strong basis we have written, by perfecting the positioning system or expanding the usability of the guidance program.

This master's thesis aimed to integrate the various positioning systems in order to conduct successful flights without any GPS. We believe that we were successful in this endeavor.

Significant progress for the construction by drone project was achieved this year. While there could be improvements in precision and in decision strategies, our programs are already capable of directing a flight and activating the onboard electromagnet. Beyond that initial goal, we also reassessed the precision of positioning technologies that had never been tested in-flight so far, and we added a completely new source of position with the recently released T265 camera.

We have great expectations for the future of the project and for autonomous construction by drone, and hopefully the contributions of this master's thesis will help get us closer to this exciting dream.

Bibliography

[1] P. Latteur, "Building our cities with drones... A utopia? | TEDxUCLouvain." https://www.youtube.com/watch?v=tn9SnXz-c4g&ab_channel=TEDxTalks.

[2] J. Leplat and J.-S. Breton, "Feasability study for drone-based manufacturing of architectural," Master's thesis, UCLouvain, 2015.

[3] C. Coppieters de Gibson, "Environmental and financial feasibility of a drone-built architectural structure," Master's thesis, UCLouvain, 2016.

[4] A. Naveau and A. Moncourrier, "Development of a Drone-Compatible Masonry Construction System," Master's thesis, UCLouvain, 2016.

[5] M. Reniers, "Development of elementary BIM tools for drone-compatible construction systems," Master's thesis, UCLouvain, 2016.

[6] N. Sorensen and R. Vermeiren, "Development of a guidance and local automatic positioning system for builder drones with a theodolite," Master's thesis, UCLouvain, 2018.

[7] Z. Bouhaddi, P. Latteur, and C. Flémal, "Etude et développement d'un système de localisation et de guidage d'un drone basé sur la technologie UWB," Master's thesis, ECAM, 2018.

[8] T. Jacques and F. Kaczynski, "Development of a local positioning system for builder drones with image processing," Master's thesis, UCLouvain, 2018.

[9] "PX4 Architectural Overview." http://dev.px4.io/en/concept/architecture.html. Accessed: 2019-05-15.

[10] "T265 Tracking Camera." https://github.com/IntelRealSense/librealsense/blob/master/doc/t265.md. Accessed: 2019-06-01.

[11] "D435i." https://github.com/IntelRealSense/librealsense/blob/master/doc/d435i.md. Accessed: 2019-06-01.

[12] Wikipedia, "Loose coupling." https://en.wikipedia.org/w/index.php?title=Loose_coupling&oldid=880766284. Accessed: 2019-05-20.

[13] "MAVLink Messaging." https://dev.px4.io/en/middleware/mavlink.html. Accessed: 2019-05-22.

[14] A. G. Amigó, P. Closas, A. Mallat, and L. Vandendorpe, "Cramér-rao bound analysis of uwb based localization approaches," in *2014 IEEE International Conference on Ultra-WideBand (ICUWB)*, pp. 13–18, Sep. 2014.

[15] "TREK1000 User." https://www.decawave.com/wp-content/uploads/2018/09/trek1000_user_manual.pdf.

[16] Intel, "Intel RealSense Tracking Camera T265 Datasheet." https://www.intel.com/content/www/us/en/support/articles/000032422/emerging-technologies/intel-realsense-technology.html.

[17] A. Grunnet-Jepsen, J. N. Sweetser, and J. Woodfill, "Best-Known-Methods for Tuning Intel RealSense D400 Depth Cameras for Best Performance." https://www.intel.com/content/dam/support/us/en/documents/emerging-technologies/intel-realsense-technology/BKMs_Tuning_RealSense_D4xx_Cam.pdf.

[18] Wikipedia, "Sensor fusion." https://en.wikipedia.org/w/index.php?title=Sensor_fusion&oldid=899038066. Accessed: 2019-05-15.

[19] "Using Vision or Motion Capture Systems." https://dev.px4.io/en/ros/external_position_estimation.html.

[20] P. D. Team, "Flight Modes." https://docs.px4.io/en/flight_modes/, 2019.

[21] M. Monnart, "Développement et validation d'un système de guidage autonome pour l'assemblage d'éléments drone compatible," Master's thesis, UCLouvain, 2019.

[22] "Ubuntu install of ROS Melodic." http://wiki.ros.org/Installation/UbuntuARM.

Appendix A

Installation Instructions

A.1 Connect to Odroid

1. Connect the TP-LINK modem to power supply
2. Connect Odroid to modem by ethernet cable
3. Connect laptop to modem's wifi
4. Open web browser 192.168.0.1
5. Open DHCP list and get IP address "cable" of the Odroid
6. Open a terminal or ssh client:
7. Connect laptop to Odroid: `$ ssh odroid@ip-cable`
8. `$ sudo nmtui`
9. select "Activate connection"
10. Select modem wifi and connect
11. On modem website: fix the IP address of the wifi key (e.g. 192.168.0.150)
12. Close the first ssh connection
13. `$ ssh odroid@ip-wifi`

Next time, the IP address should be the same when the Odroid is connected on wifi. If laptop and Odroid are connected to the same wifi, you just have to connect with this command: `$ ssh odroid@ip-wifi`

A.2 Connect Odroid to internet

A.2.1 In DroneZone

1. Connect the TP-LINK modem to power supply
2. Connect Odroid with a cable to the modem
3. Connect laptop with the modem (either wifi or ethernet)
4. Generate a wifi network with a smartphone
5. `$ ssh odroid@ip-cable`
6. `$ sudo nmtui`
7. select "Activate connection"
8. Select smartphone wifi and connect
9. Test internet connection with `$ ping 8.8.8.8`

A.2.2 In computer room (UCLouvain - Reaumur)

1. Add MAC address of the Ethernet port on the list of authorized MAC address (Wiki INGI).You can find this MAC address when you're connected in SSH on Odroid with the command `ifconfig`
2. Connect the TP-LINK modem to power supply
3. Connect Odroid to modem's wifi
4. in terminal: `$ ssh odroid@ip-wifi`
5. Test internet connection with `$ ping 8.8.8.8`

A.2.3 On domestic network

1. Connect Odroid by cable at home's modem
2. Check on the user interface of the modem, the IP address of the Odroid.
3. `$ ssh odroid@ip-cable`

If you want to connect Odroid to home's wifi:

1. `$ sudo nmtui`
2. "Activate connection"
3. Select home's wifi and connect
4. Disconnect cable

Next time, network is recognized by Odroid and you can check that the IP address is displayed on the UI of the modem.

A.3 Install ROS

Following instruction assume that you have an Odroid XU4 with Ubuntu mate 18.04

A.3.1 Installation on Laptop with Ubuntu

Based on tutorial found on `https://dev.px4.io`

```
$ wget https://raw.githubusercontent.com/PX4/Devguide/master/
    ↪ build_scripts/ubuntu_sim_ros_gazebo.sh
$ sudo usermod -a -G dialout \$USER
$ sudo reboot
$ source ubuntu_sim_ros_gazebo.sh
$ cd scr/Firmware
$ make posix_sitl_default gazebo
```

A.3.2 Run ROS & Gazebo on laptop

```
$ source ~/catkin_ws/devel/setup.bash
$ source Tools/setup_gazebo.bash $(pwd) \$(pwd)/build/posix_sitl_default
$ export ROS_PACKAGE_PATH=$ROS_PACKAGE_PATH:$(pwd)
$ export ROS_PACKAGE_PATH=$ROS_PACKAGE_PATH:$(pwd)/Tools/sitl_gazebo
$ roslaunch px4 mavros_posix_sitl.launch
```

A.3.3 Installation on Odroid

Following this tutorial (Melodic):[22]

```
$ sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release
    ↪  -sc) main" > /etc/apt/sources.list.d/ros-latest.list'
$ sudo apt-key adv --keyserver hkp://ha.pool.sks-keyservers.net:80 --
    ↪ recv-key 421C365BD9FF1F717815A3895523BAEEB01FA116
$ sudo apt update
$ sudo apt install ros-melodic-ros-base
$ apt search ros-melodic
$ sudo rosdep init
$ rosdep update
$ echo "source /opt/ros/melodic/setup.bash" >> ~/.bashrc
$ source ~/.bashrc
$ sudo apt install python-rosinstall python-rosinstall-generator python-
    ↪ wstool build-essential
$ mkdir -p ~/catkin_ws/src
$ cd catkin_ws/
$ catkin_make
$ sudo apt-get install ros-melodic-angles
$ sudo apt-get install ros-melodic-control-toolbox
$ sudo apt-get install ros-melodic-mavros ros-melodic-mavros-extras
$ cd ~/catkin_ws
$ sudo apt-get install python-wstool python-rosinstall-generator python-
    ↪ catkin-tools -y
$ wstool init ~/catkin_ws/src
$ rosinstall_generator --upstream mavros | tee /tmp/mavros.rosinstall
$ rosinstall_generator mavlink | tee -a /tmp/mavros.rosinstall
$ wstool merge -t src /tmp/mavros.rosinstall
$ wstool update -t src -j4
$ rosdep install --from-paths src --ignore-src --rosdistro melodic -y
$ cd ~/catkin_ws/src/mavros/mavros/scripts
$ sudo ./install_geographiclib_datasets.sh
$ sudo apt-get install python-serial
$ catkin build -j2 -l2
```

A.3.4 How remove ROS?

```
$ sudo apt-get remove ros-*
$ sudo apt-get purge ros-*
$ sudo rm -r /etc/ros
$ sudo apt-get autoremove
$ nano ~/.bashrc
```

Delete this two lines:

```
source /opt/ros/kinetic/setup.bash
source ~/catkin\_ws/devel/setup.bash

$ rm -rf catkin_ws/
$ rm -rf src/Firmware
$ rm -rf src
```

A.4 Installation of Librealsense2 on Odroid

There are instructions to install Librealsense2 on an Ordoid but when we tried it didn't work. So we followed the method for RaspberryPi 3[1]. `https://github.com/IntelRealSense/librealsense/blob/master/doc/RaspberryPi3.md`

```
$ cd ~
$ git clone https://github.com/IntelRealSense/librealsense.git
$ sudo apt-get install git libssl-dev libusb-1.0-0-dev pkg-config libgtk
    ↪ -3-dev
$ sudo apt-get install libglfw3-dev libgl1-mesa-dev libglu1-mesa-dev
$ ./scripts/setup_udev_rules.sh
$ cd ~/librealsense
$ mkdir build && cd build
$ cmake ../ -DBUILD_EXAMPLES=true -DBUILD_GRAPHICAL_EXAMPLES=false
```

Be careful it really takes a lot time. Advice: install and use `screen`

```
$ sudo make uninstall && make clean && make -j2 -l2 && sudo make install
```

To test the installation: Connect T265 and `$ rs-pose`

A.5 Generate `ROStotalSender.exe`

1. Copy the folder "ROStotalSender" from git to the laptop
2. Open `ROStotalSender.sln` with Visual Studio
3. Generate solution
4. `ROStotalSender.exe` is located in `ROStotalSender/bin/Debug`

[1] April 2019

Appendix B

User Manual

B.1 Drone Setup

1. Plug wifi key on the USB port next to the power supply.
2. Plug Pixhawk on the lower port 3.0 (blue).
3. Plug USB Hub on the upper port 3.0.
4. Make sure the prism is on top of the drone
5. Connect the two cables of the electronic board to the Shifter Shield. The ground on pin 9 and the second on pin 11.
6. Connect the power supply to the Odroid (and connect the drone battery if it has not already been done)
7. Connect the camera on the USB Hub, wait a few second, then connect the UWB device.
8. Place the drone at it's home position and align it with the x-axis of the total station.

B.2 Pixhawk Setup

This configuration must only be done once, but according to use cases might need to be updated for the desired behavior. You must have installed QGroundControl previously.

1. Connect your computer to the telemetry unit and launch **QGroundControl** (./QgroundControl.AppImage)
2. in the parameters of **QGroundControl**, activate LPE estimator `SYS_MC_EST_GROUP` = 1;
3. in the parameters of **QGroundControl**, enable vision usage `ATT_EXT_HDG_M` = 1;
4. in the parameters of **QGroundControl**, activate LPE position fusion `LPE_FUSION` = 148;
5. in the parameters of **QGroundControl**, reduce standard deviation `LPE_VIS_XY` and `LPE_VIS_Z` to minimum ;

B.3 Total Station Setup

1. Place the total station above the (0,0) point.
2. Shift+F12 to level.
3. Connect station to the computer with cable (station's drivers must be installed)
4. On computer, open command prompt and launch the server[1].
   `ROStotalSender.exe com3 57600`
5. Follow instructions on the screen

Possible issues

- If the station finds the prism but the program stops, check that the station is in LOCK mode (Shift + USER)[2]
- The station continues to search for the prism after leaving the program: turn the station off/on again
- The PC and the station do not communicate. Check if the baudrate of the station is correct (recommended value 57600)

[1] Instructions to generate ROStotalSender.exe can be found in the section A.5

[2] The screen that appears shows the values that the parameters will take if they are modified. Therefore, LOCK must be displayed in OFF for the station to lock the prism.

B.4 Launch Systems

Terminal 1: launch positioning system

```
$ cd catkin_ws/src/SmartDrone/launch
$ roslaunch smart_drone system.launch --screen
```

After launching the positioning system, in "Widgets", Mavlink Inspector of **QGroundControl**, we can see mavlink message `LOCAL_POSITION_NED` and `VISION_POSITION_ESTIMATE`. They should be similar as the first is the filtered version of the second. Do not attempt a flight if this is not the case! Rather go to parameters in **QGroundControl** and modify the `LPE_FUSION` parameter to include the barometer, wait until the values converge and deactivate it again.

Terminal 2: launch guidance system

```
$ nano guidance_system.py #add mission points
$ cd catkin_ws/src/SmartDrone/src
$ ./init.sh
$ python guidance_system.py
```

Terminal 3: launch measure script

```
$ cd catkin_ws/src/SmartDrone/src
$ ./script
```

Appendix C

Experiments

In figure C.1, we can observe the trajectories of the drone according to the different positioning systems in the XY plane. It is quite obvious that everyone, except the camera, is following the same path. For a reason unknown to us, the camera exaggerates each movement in all three directions.

In figures C.2 and C.3, we observe again the positions given by the different positioning systems in x and y, but this time as a function of time (the z-axis is represented in figure 8.10). As described in section 8.2.1, we observe that the position given by the UWB is the same as that of the total station, which confirms our hypothesis of a wrong setting. Unlike the camera which continues to show a shift in the 3 directions.

Figure C.4 shows in 3 dimensions the theoretical trajectory and that actually followed by the drone (Total Station) during the mission described in section 8.2.2.

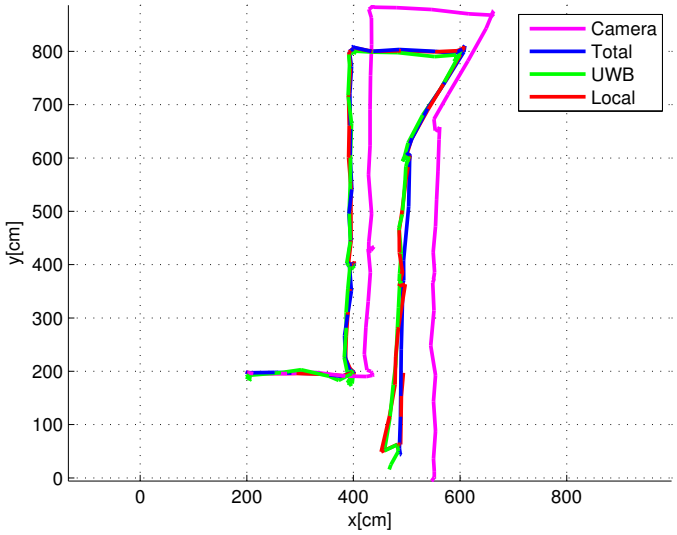


Figure C.1: Path of the drone in the XY plane during the manual movement analyzed in section 8.1.2.

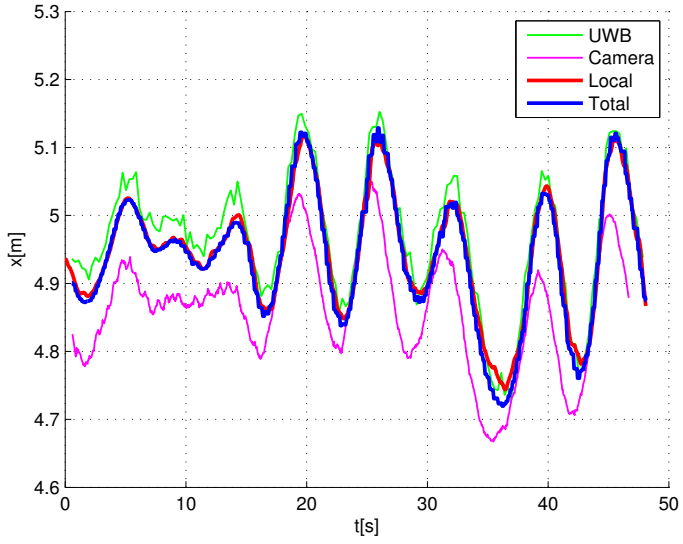


Figure C.2: Movement, along the x-axis, of the drone during the stationary part of its flight analyzed in section 8.2.1.

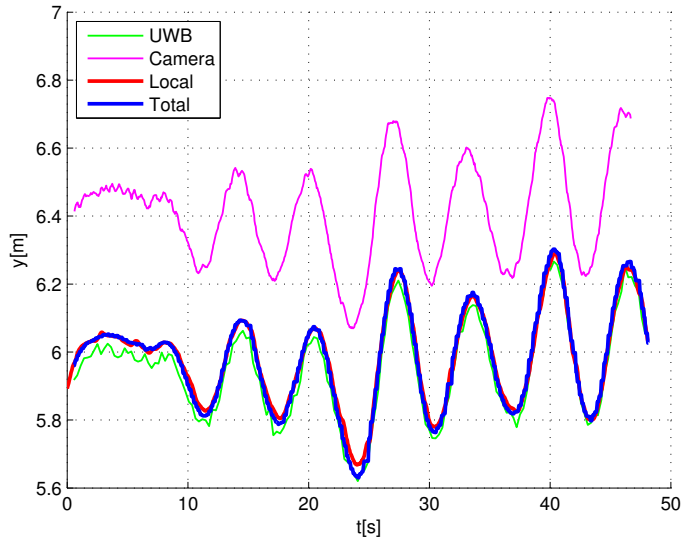


Figure C.3: Movement, along the y-axis, of the drone during the stationary part of its flight analyzed in section 8.2.1.

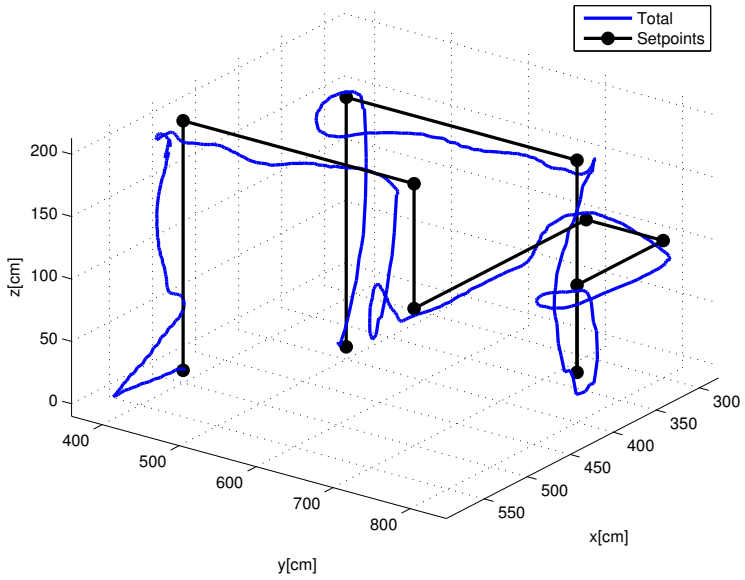


Figure C.4: Three dimension path during mission

Appendix D

Data Structures

These are the mavros messages used during communication with the Pixhawk.

```
geometry_msgs/PoseStamped.msg = {
        std_msgs/Header header
        geometry_msgs/Pose pose
}
std_msgs/Header header = {
        uint32 seq
        time stamp
        string frame_id
}
geometry_msgs/Pose pose = {
        geometry_msgs/Point position
        geometry_msgs/Quaternion orientation
}
geometry_msgs/Point position = {
        float64 x
        float64 y
        float64 z
}
geometry_msgs/Quaternion orientation = {
        float64 x
        float64 y
        float64 z
        float64 w
}
```

Appendix E

ALX Systems

During the year, we worked with ALX systems, a start-up from Liege whom develops autonomous drones for multiples use cases. Their system is split in different components:

- **Host** software on the onboard computer that will decide the behavior of the drone.
- **Ground Station** User interface on the offboard computer which allows the user to create a flight mission
- **Server** Software acting as a communication link between the offboard and onboard computer.
- **Plugins** Additional functionality for specific use cases.

These different components are shown in figure E.1 with the names of their executable files.

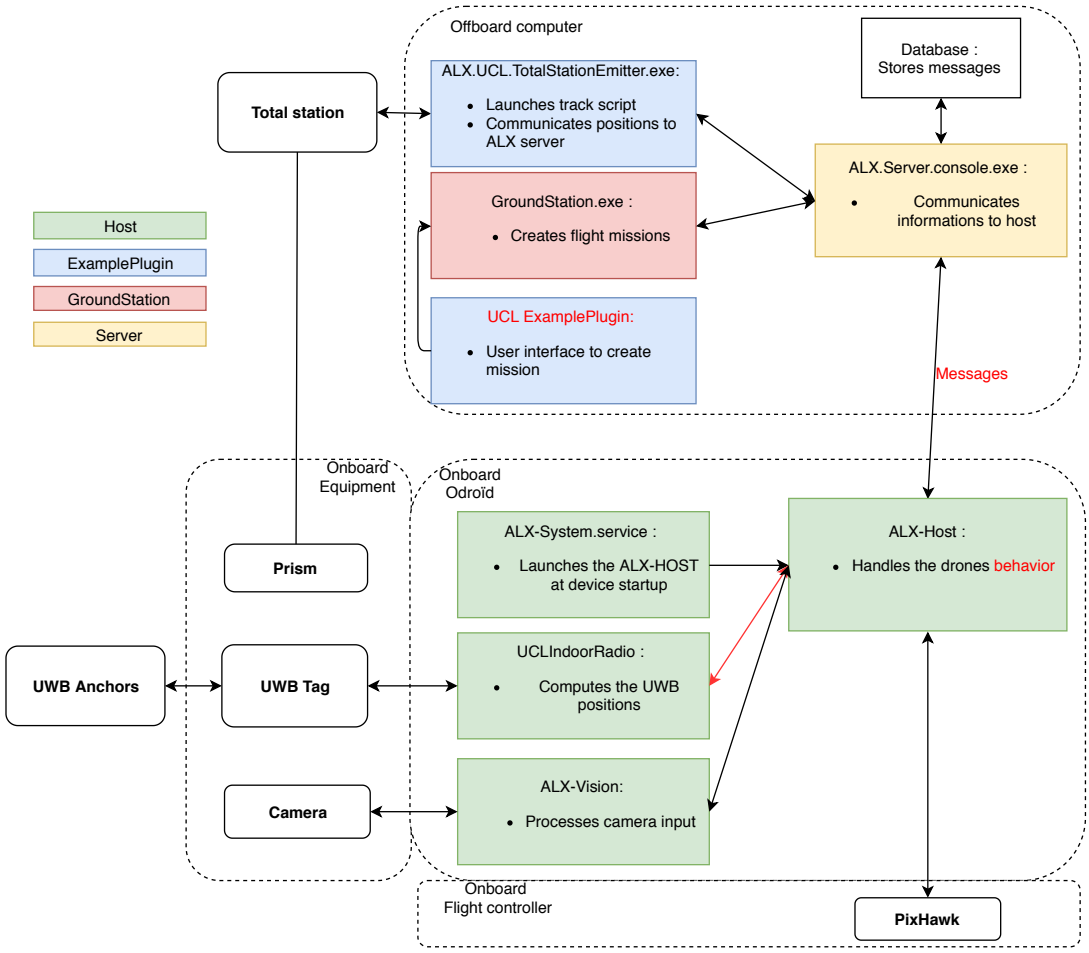


Figure E.1: ALX systems - system overview

We mainly participated in the development of a plugin for builder drones (UCL ExamplePlugin) and the integration of the positioning systems (UCLIndoorRadio).

The highlighted components in red in figure E.1, are the parts that were still missing. Since their system is proprietary software, we didn't have access to the code of the other components which caused difficulties during the development of the plugin. Equally, there were parts that needed to be defined in their system, we couldn't do ourselves.

We decided to develop our own open source version using the ROS framework. This gave us more flexibility and allowed us to be in control of all parts of the system.

Appendix F

Electronic Card

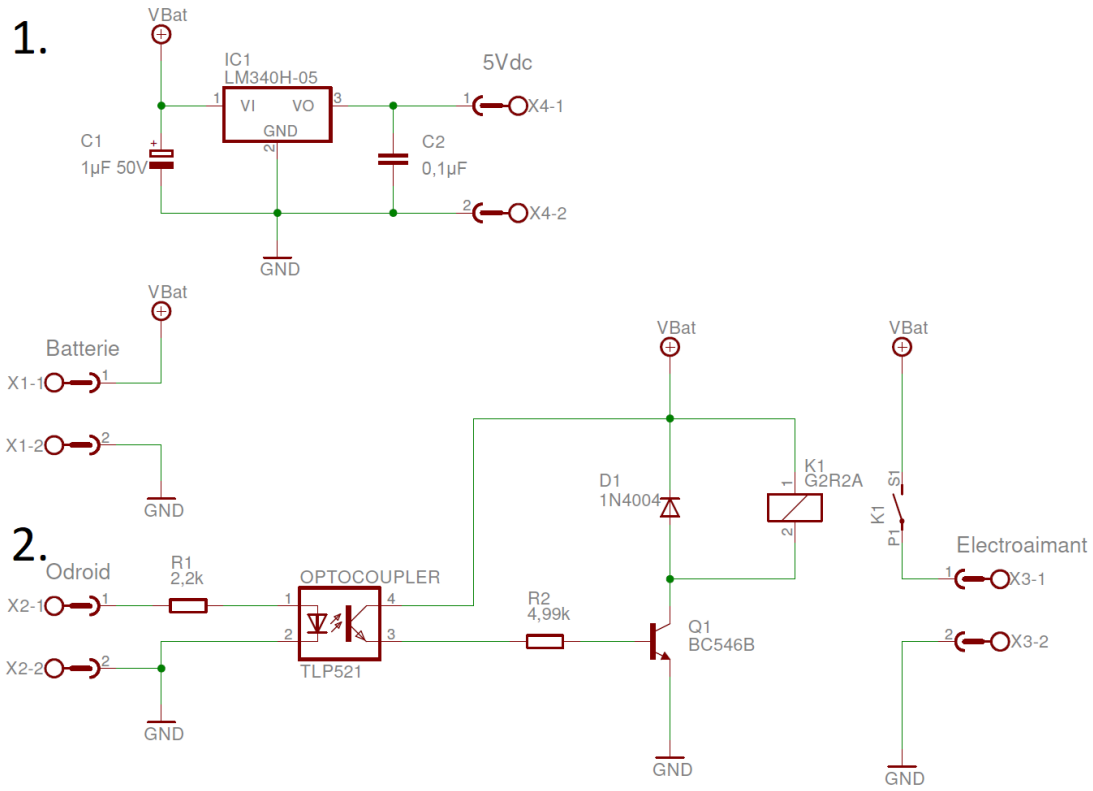


Figure F.1: Scheme of the electronic board that allows the Odroid to control the power supply of the electromagnet with the battery voltage

Appendix G

3D Models

We created a 3D model for a telescopic arm with on the upper side wings to attach to the drone and on the lower side a link to the electromagnet. This arm was 3D printed using PLA and was used during multiple test flights.

Figure G.1: Telescopic drone arm

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**