ENSTA BRETAGNE

RIGITECH

Master's thesis in industry with RigiTech

Improving simulation conditions and video of an Unmanned Aircraft Vehicle.

Antoine Wanctin
ROB 2023

Supervisor: Victor Delafontaine

August 25, 2023

Abstract

Drones are a rapidly growing sector. More and more companies are developing them for very specific tasks. RigiTech is a Swiss start-up that is developing a whole ecosystem around fast, autonomous delivery with a wide range of action. The drone's autonomy is ensured by a whole range of functions with different roles, but all working together to make the drone operator's job easier. Ultimately, a single operator should be able to manage multiple delivery routes simultaneously. I have found my place in this working group in two ways.

Firstly, my work on the video part of the drone has led to improvements in all aspects of streaming and recording. My research into optical streaming has helped to make the method more feasible. I was also able to test the possibility of implementing a Detect and Avoid solution using video, which could be implemented on the drone itself in the more or less long term.

On the other hand, my work on the simulation will have increased its realism. I have effectively implemented the integration of wind into the simulation and made great progress in creating a height map to simulate the elevation of the world. This paves the way for further improvements and new features.

Lastly, I've been asked to do some additional work. Whether it was fixing bugs, implementing new features or Research and Development, it all contributed to the continuous improvement of the drone and its cloud interface.

Acknowledgements

I'd like to express my sincere thanks to Victor Delafontaine for his unfailing support, his patience with my numerous questions and the many answers he was able to give me. I would also like to thank the entire RigiTech team, who have provided me with a friendly and serious working environment and whose experience and knowledge have been priceless over these six months. I'd also like to thank the PX4 team, especially Ramón Roche, who took the time to try and solve my problems. I also want to thank Pierre Bosser, without whom I wouldn't have been able to achieve my geographic projections. Finally, I would like to thank my family and friends who have supported me throughout this enriching experience.

Contents

1 Video **2**
- 1.1 Improve stream . . . 2
- 1.2 Optical Flow . . . 6
- 1.3 Detect And Avoid (DAA) with Sense Aeronautics . . . 15

2 Simulation **22**
- 2.1 Wind . . . 23
- 2.2 Elevation . . . 25

3 Other tasks **38**
- 3.1 Bug Fixing . . . 38
- 3.2 Unit Testing . . . 39
- 3.3 Vertical geocage boundary . . . 40
- 3.4 Auto-exposure . . . 40
- 3.5 DronePort . . . 41

References **I**

List of Acronyms **III**

Appendices **IV**

List of Figures

1 The Eiger on the landing pad designed for precision landing . . . 2
2 The cloud interface with the front camera streaming . . . 3
3 The simulated video using GStreamer . . . 5
4 The aperture problem; On the right we see the missing orthogonal component . . . 8
5 Two types of Optical Flow from [1] . . . 9
6 Optical Flow visualization on the Eiger using the down camera . . . 13
7 Relation between distance and Optical Flow . . . 14
8 Relation between speed and Optical Flow . . . 14
9 The dynamic interface . . . 17
10 The coordinates used by the video Detect And Avoid (DAA) . . . 18
11 False detection . . . 19
12 Comparison between optical and acoustic Detect And Avoid (DAA) . . . 22
13 The drone flying in Gazebo. . . . 23
14 Gazebo structure and dependencies from [2] . . . 24
15 The drone in the cloud in grey with the wind in white . . . 25
16 Digital Elevation Model (DEM) of the Geneva lake . . . 26
17 What the Application Programming Interface (API) returned in red versus what we asked in green . . . 29
18 A Digital Elevation Model (DEM) with a non standard resolution for Gazebo . . . 30
19 Different resolutions for the same height map . . . 31
20 Different paradigms for height maps from [3] . . . 34
21 A correct Delaunay interpolation from [4] . . . 35
22 Referential used in Gazebo from [5] . . . 37
23 One of the scenarios to test the function. The buffer zone is in red, the geocage in yellow and the pre-geocage in green . . . 39
24 Summary of the altitude frames used to define the max altitude of the geocages. . . . 40
25 Sprint organization in RigiTech . . . IV
26 Another scenario for unit testing . . . IV
27 Another scenario for unit testing . . . V
28 Another scenario for unit testing . . . V

Introduction

Mobile robotics is undergoing a significant boom that has only intensified in recent years. Whether for scientific, defense, leisure, or utility purposes, it is increasingly being used in a wide range of sectors.

RigiTech is a Swiss start-up operating in the market for airborne transport with Unmanned Aircraft Vehicle (UAV)s. Its latest drone, the Eiger, is capable of carrying a load weighing up to three kilograms over a radius of one hundred kilometers in just one hour. This technological feat is made possible by vertical take-off and landing (VTOL) technology, which gives the drone two modes of operation. In quadcopter mode, it takes off vertically with four propellers. Once airborne, it switches from this first mode to fixed-wing mode. The rear propeller takes over and propels the drone, which then flies like a conventional aircraft thanks to its wings that provides the necessary lift. This uses much less battery power than a conventional quadcopter and increases the range tenfold when compared to standard quadcopters.

The Eiger is capable of performing Behind the Visual Line Of Sight (BVLOS) missions, which by definition do not require a pilot or visual control. RigiTech's main market today is the delivery of medical sampled to remote areas. Using the Eiger drone allows to speed the laboratory analysis and reduce the carbon footprint when compared with classical means of transportation such as trucks or cars.

An extremely comprehensive cloud interface has been developed to monitor and program the drone's missions. It allows you to plan the drone's missions, monitor it in flight, access its logs, and simulate its behavior for development purposes.

During my internship, my main task was to improve the simulator. I also had the opportunity to work on all aspects of video. The drone has two cameras, one facing forward and one facing down. When I arrived they were mainly used for streaming and more recently for precision landing using AruCo markers. During my internship, I was able to work on the implementation of optical flow to compensate for drift in the event of Global Navigation Satellite System (GNSS) loss at low altitudes. I also had the opportunity to work hand-in-hand with another start-up, Sense Aeronautics, which is developing a Detect And Avoid (DAA) system using a video stream.

Finally, being in a start-up, I was given less substantial but more urgent tasks. These mostly involved debugging or continuous improvement of existing functionalities. RigiTech follows the logic of development sprints, at least for all software. Each development sprint lasts one month, at the end of which all code improvements are released on a test server. This contains only relatively stable code that has already been tested on a development server. Customers can access the test server and make suggestions for new features, or report bugs. Every three months, after intensive testing and fixes, what is on the test server is moved to the production server for customers. However, this imposes very strict testing protocols on the company, as they are limited in time.

1 Video

The drone is equipped with two high-definition cameras, one facing forward and one facing downward. These cameras allow the drone to do several things. First, it can stream what the cameras see in real-time. This provides real-time visual feedback on what's happening during the mission, even when the drone is BVLOS. The main benefit of the down camera is precision landing. This feature was just being implemented when I arrived. By performing image processing on the down camera and placing a landing target on the ground in the form of AruCo markers, the drone can locate itself in space and correct its position accordingly, right up to the final landing. The drone's precision is truly remarkable: thanks to this technology, it can land with a margin of error of less than 10 centimeters. This allows it to land on a parking lot, for example, which is an impressive feat for a drone with a wingspan of 2.70 meters.

Figure 1: The Eiger on the landing pad designed for precision landing

1.1 Improve stream

When I arrived, the drone's software structure for the onboard computer was divided into several nodes, each representing a particular function of the drone. So it was logical for me to work on the video_node, the part that handled all the video.

To test all the code development and new functionality, simulators were set up. I'll come back to this in more detail in the section dedicated to it, but in this case, they were used to visualize what the drone would see using the cameras connected to the computer where the simulation was running.

First steps with video_node

To get started with video_node, I first had to come up with an efficient method to debug the precision landing feature, which was quite new at the time. To do so, I've developed a function that automatically recorded a video of the drone landing. It was crucial at this time to take into account the computational cost as a Raspberry PI was used as an onboard computer. This feature then allowed me to thoroughly review the performance of the precision landing in detail. Most notably, it allowed to robustify the landing target detection algorithm against false positive detections

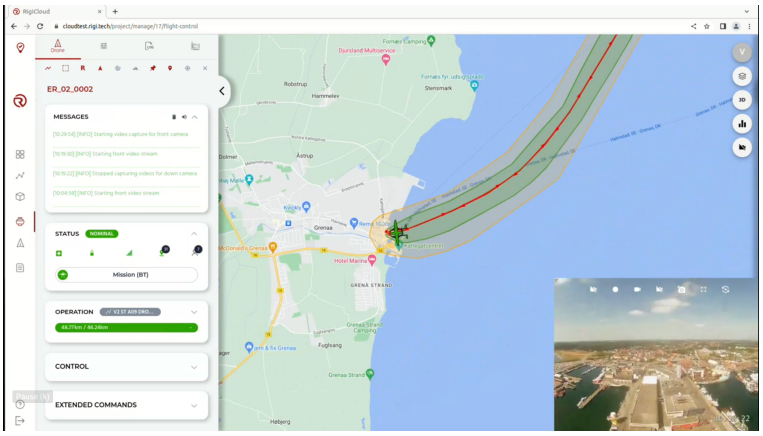


Figure 2: The cloud interface with the front camera streaming

Robot Operating System (ROS) services were already available to start and stop streaming or recording. I was able to reuse the existing structure to start these services at the right time. To determine that moment, I used the way missions are coded. During a mission, the drone follows a set number of waypoints to its final destination. It is able to determine the type of waypoint in advance, up to two waypoints ahead. By starting the recording service as soon as the landing waypoint was detected, it retrieves a video that wasn't too long and met expectations. To keep some flexibility in the functionality, I also implemented a parameter that enables or disables the activation of the recording.

Promotional mode

One of my next tasks was to improve the streaming and recording quality. Mainly for bandwidth reasons, the videos are streamed and recorded in 640x480, which is fine for observation purposes, but has significant potential for improvement. One of the goals was to create high-quality promotional videos by increasing both resolution and frames per second.

Again, by creating a suitable parameter and modifying the video pipeline, I was able to achieve my goal. It did, however, require a bit of code restructuring. I ran into two problems during this task. The first, took me some time to figure out. I wanted to record a video with a resolution of 1080p

and 30 fps. When I ran my tests on my computer's webcam, it refused to show what I wanted. This was because, contrary to my expectations, it didn't support the resolution I requested.

The second problem occurred once the promotional_mode was working properly on my computer. When I tested it on the drone's onboard computer, it couldn't reproduce the expected quality, and the video was jerky and accelerated at times. This was due to the insufficient processing power of the Raspberry PI. In fact, with all of the drone's code to process, the CPU was insufficient for high-definition video processing. To improve performance, I tried lowering the quality required, but to no avail. Changing the way the video was processed, especially the bit rate, didn't help either.

Nevertheless, one of the advances that may have helped was the change in video encoding. By switching from MJPEG to H264, we've gained in performance, both in terms of speed and in terms of the amount of space required to record video. For example, a one-minute video that used to take up 23 Mb of storage now takes up only 4 Mb by switching to H264. MJPEG compresses individual images whereas H264 compresses across frames. In H264 the first frame is compressed by itself and the following frames record changes from the previous frame. This is why it compresses much more and uses less bandwidth than MJPEG.

Simulated video

Still working on improving the use of video, I then turned my attention to video simulation. The goal was to be able to use all the video-related functions in the simulator without having a camera connected. To achieve this, I had to change the way video was processed. Previously, each camera frame was passed through OpenCV [6] methods and functions. In particular, to create a simulated video, I used GStreamer [7].

GStreamer is a library for creating and managing multimedia content. It extends its functionality through a system of plugins that allow it to support different formats or add new features. You can choose your video encoding format or the way it is encapsulated, for example in mp4 or avi. The way the video is compressed by the codec is also manageable. For example, switching from streaming and recording to H264 has greatly improved performance. The H264 codec is actually used in most of video processes around the world.

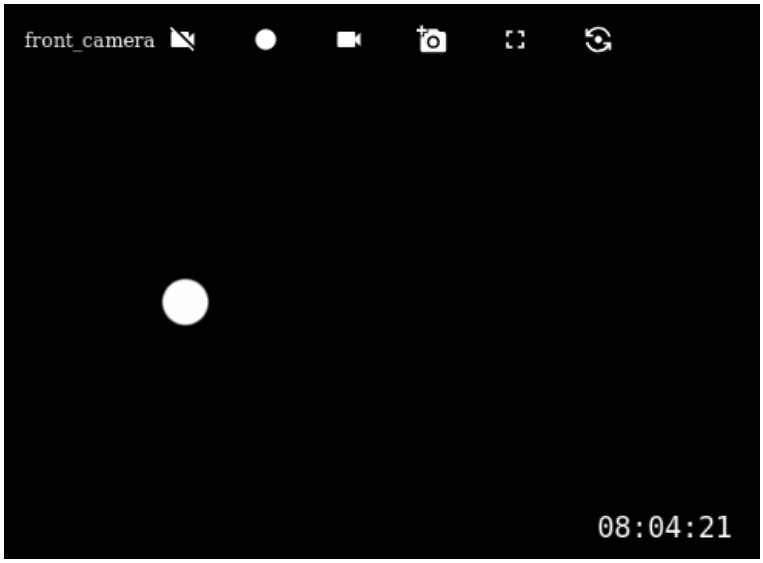


Figure 3: The simulated video using GStreamer

I mainly used it as a data pipeline. With GStreamer, it is possible to input one type of multimedia data and then apply various modifications to it to get the desired output. So, thanks to a GStreamer option to create a simulated video, I was able use it in the cloud. Again, I used a parameter to start this video simulation or not, and adjusted the pipeline according to what was needed. This wasn't easy because there seemed to be two ways to get the same result. I could either change the video capture pipeline or the way the video was encoded after it was written. I started with the latter option without success. Changing the capture pipeline was successful, but it required me to take into account whether or not the video was simulated upstream in the program.

I also took the opportunity to change the recording and streaming behavior of both cameras so that only one camera could be streamed or recorded at a time.

Adaptive Bitrate

In order to improve the quality of the stream without pushing the CPU to its limits, I then turned to the question of bitrate. Bitrate is a measure of the amount of data encoded in a given amount of time. The higher the bitrate, the better the quality, but the greater the bandwidth and computing power required. It is expressed in bits per second. After meticulous testing, I was able to leverage GStreamer to find the optimal bitrate that optimized the quality and the computation required by the processor.

The next step was to create an adaptive bitrate. This is what is implemented in most online video players today. When quality is set to automatic, it will increase or decrease depending on the available bandwidth, allowing the video to have the best possible quality while minimizing the delay required to receive it.

The response time between the moment the camera captures the image and the moment the operator receives the video is extremely important. When using video streaming to monitor a drone BVLOS, it is crucial to be able to react to hazards quickly. This can only be done by minimizing the time delay between the instant when the camera image is captured and the moment when the operator receives it. This is why adaptive bitrate is a viable solution. By reducing the quality when the network drops, the stream keeps the latency low. There are several different technical solutions to overcome this problem, as explained in [8] or [9].

Again, I decided to use GStreamer for several reasons. First, it seemed possible to achieve adaptive bitrate with this library. Second, I was beginning to have enough experience with it that I didn't want to jump into something new.

To achieve adaptive bitrate I used GStreamer's target-bitrate command. However, due to plugin compatibility issues, this never worked. Having different configurations between my machine and the video_node docker image did not help either.

In the end, we decided to use a fixed bitrate to suit our needs.

1.2 Optical Flow

Context

Our goal at RigiTech is to have a fully autonomous drone. It must also be able to comply with current regulations, especially with regard to safety standards. In addition to a powerful guidance and control system, this also includes backup systems in case of hardware or software failure. To solve this problem, a failsafe system replaces the onboard computer in case of failure. The failsafe consists of very simple functions designed to be as robust as possible. It is completely electrically isolated from the onboard computer and is powered by a separate battery. Its main purpose is to land the drone in the event of a major malfunction to ensure the safety of potential bystanders.

But before we get to that point, there are other ways to compensate for the loss of a sensor, such as redundancy. This is what optical flow is all about. The goal is to compensate for the loss of GNSS when the drone is in quadcopter mode, i.e. relatively close to the ground. By visually estimating the drone's movements using the optical flow and weighting them with distance measurements to obtain a real displacement, it is possible to continue precision landing without access to position, for example. This technique has already been used for heading estimation in [10] using a phase correlation technique coupled with an Intertial Measurement Unit (IMU). It could also be used as a precision landing technique as in [11].

Introduction to Optical Flow

When using image processing, we have to keep in mind that what we're processing is just a projection of real motion. This leads to an inability to correctly measure real motion in two dimensions. We can only perceive the apparent motion of objects.

These two motions can be identical under certain conditions. For example, a gradual change in the position of the light source illuminating the observed object will introduce apparent motion without real motion. Similarly, in rare cases, there may be a real rotational motion and an apparent translational motion. This is what happens when we observe a worm screw, for example.

Optical flow is the apparent motion of objects, surfaces, and contours in a visual scene caused by the relative motion between an observer and the scene. A common analogy is that of an observer sitting in a moving vehicle. As he observes the landscape, he will notice that the objects he sees are receding relative to him. This is the part of the optic flow that determines speed. What's more, he can estimate how far away he is from the objects in the landscape. For example, a mountain in the background will have a much slower relative motion than an object at the edge of the path, which will pass by extremely quickly, to the point of blurriness.

These relationships between object distance and apparent speed also depend on the viewing angle. In the example above, the optical flow is maximal because the considered objects are at 90° to the actual motion of the observer. If the observer is moving toward the object in question, then on average he will have zero optic flow. However, as the object tends to expand, an optical flow can be observed that expands from the center of the image. This center is called the Focus Of Expansion (FOE). It represents the direction in which the observer is moving, its actual motion.

That's what we're interested in here. We want to use the optical stream to compensate for the drift due to GNSS loss. The goal is to detect this loss and avoid drifting away from the last known position by using the optical stream. To do this, we'll try to minimize it at each point so that we have a zero global vector. Most of the following theoretical background come from [12]

Theoretical Background

We are looking for a vector field that represents the movement of each point such as:

$$\begin{aligned} \sigma(t,\cdot) :&\mathbb{R}^2 \to \mathbb{R}^2 \\ &(x,y) \mapsto (\sigma_1(t,x,y), \sigma_2(t,x,y)) \end{aligned} \tag{1}$$

The basic principle is that the intensity of a point remains constant along its trajectory. To achieve this, we make two assumptions, one that we are interested in small displacements with little change in illumination, and the other that there is no occlusion or transparencies in our image.

If at time t an object is at position (x, y), then at time $t+dt$ it will be at position $(x+dt\sigma_1(t,x,y), y+dt\sigma_2(t,x,y))$.

In this case, we are looking for σ such that :

$$u(t,x,y) = u(t+\Delta t, x+\Delta t\sigma_1(t,x,y), y+\Delta t\sigma_2(t,x,y)) \tag{2}$$

Using a Taylor expansion to first order, we have :

$$u(x+dt\sigma_1(t,x,y), y+dt\sigma_2(t,x,y)) = u(t,x,y)+\Delta t\frac{\partial u}{\partial t}+\Delta t\sigma_1(t,x,y)\frac{\partial u}{\partial x}+\Delta t\sigma_2(t,x,y)\frac{\partial u}{\partial y}+\mathcal{O}(\Delta t^2) \tag{3}$$

This finally gives us the Optical Flow constraint:

$$\frac{\partial u}{\partial t} + \langle \bigtriangledown u, \sigma \rangle = 0 \tag{4}$$

with $\frac{\partial u}{\partial t}$ the variation in image intensity over time, and $\bigtriangledown u$ its spatial variation which is known or at least estimated. Since the time displacement is considered to be between two frames in a row, $\Delta t = 1$.

However, $\sigma = (\sigma_1, \sigma_2)$. So we have an equation for two unknowns, The system is underconstrained and it shows in what is called the aperture problem. Only the vector component in the spatial gradient direction is given. Its perpendicular component is not known, resulting in a loss of information. This can be seen in Figure 4, taken from [13]

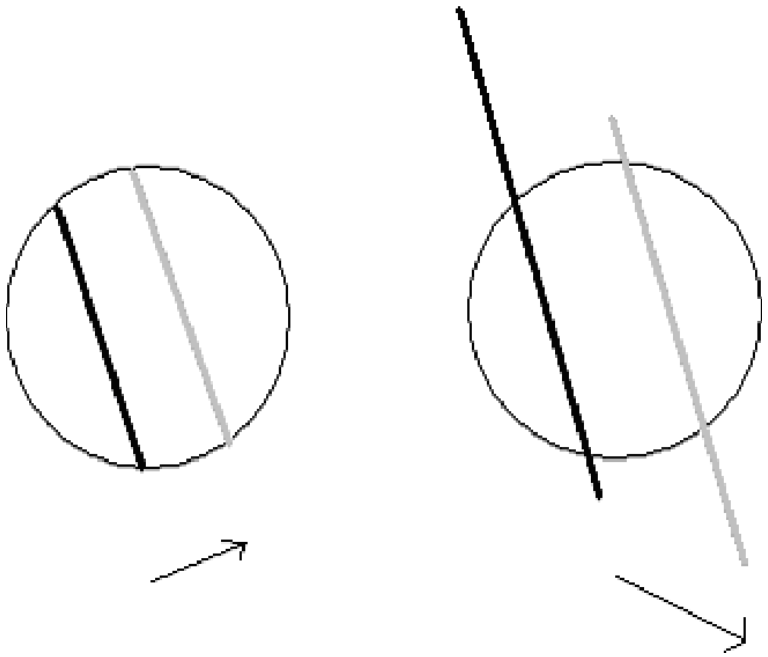
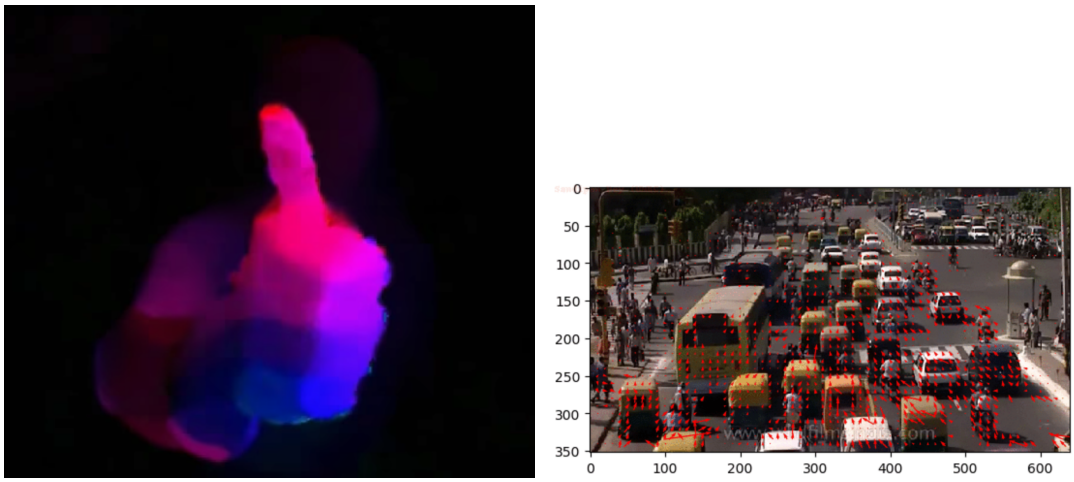


Figure 4: The aperture problem; On the right we see the missing orthogonal component

This issue is more likely to occur when calculating pixel optical flow, resulting in a gradient too small to observe. This problem can be solved by choosing to calculate the optical flow at the corner level.

Horn-Schunck

There are two classical methods for processing optical flow. The first was proposed in 1981 by John Horn and Brian Schunck in [14]. It aims to compute the optical flow for each pixel in the image to obtain a dense optical flow.



(a) Dense Optical Flow **(b)** Sparse Optical Flow

Figure 5: Two types of Optical Flow from [1]

This method is based on several assumptions, some of which have already been mentioned:

1. Brightness Constancy Constraint: The intensity of a pixel does not change significantly between two successive images.
2. Spatial Smoothing: The optical flow varies continuously in space, which implies that neighboring pixels have the same displacement vectors.
3. Temporal regularization: The optical flow changes slowly and uniformly between two successive images.

To respect this last assumption, a smoothness constraint is added to the system. We then have :

$$E_c^2 = \left(\frac{\partial \sigma_1}{\partial x}\right)^2 + \left(\frac{\partial \sigma_1}{\partial y}\right)^2 + \left(\frac{\partial \sigma_2}{\partial x}\right)^2 + \left(\frac{\partial \sigma_2}{\partial y}\right)^2 \tag{5}$$

This introduces an error corresponding to the difference between a vector and its neighbors. To obtain a global error corresponding to a dense optical flow, we sum these errors over the entire image. This gives us an error to minimize of :

$$E^2 = \int\int \frac{\partial u}{\partial t} + \langle \bigtriangledown u, \sigma \rangle + \alpha^2 \left(\left(\frac{\partial \sigma_1}{\partial x}\right)^2 + \left(\frac{\partial \sigma_1}{\partial y}\right)^2 + \left(\frac{\partial \sigma_2}{\partial x}\right)^2 + \left(\frac{\partial \sigma_2}{\partial y}\right)^2 \right) dxdy \tag{6}$$

The first term is the optical flow constraint equation and the second is the smoothness factor multiplied by a weight constant α.

To solve this equation, the Horn-Schunck method recommends using a digital estimate of the Laplacian to obtain two equations for each pixel, one for each direction of the optical flow gradient.

An iterative method is then used to solve this system. Each optical vector component is calculated using the previous iteration of the average of the neighboring vectors.

One of the interesting effects of this method is the propagation of the optical flow calculation. As the iterations progress, the neighbors of the already computed regions are taken into account until all regions of the image have optical flow vectors.

Lucas-Kanade

Another technique for calculating optical flow is the Lucas-Kanade method. It was proposed in [15]. It is implemented in PX4 and will be used in the following. It is based on the tracking of characteristic points. The computed optical flow will be sparse.

The most important assumption for this method is the following. We need to approximate the fact that the optical flow vector at a point will be similar in a Ω neighborhood surrounding that pixel.

The optical flow at (x, y) is then approximated by a least squares method such as:

$$E_u = \sum_{p \in \Omega} W^2(p)(\bigtriangledown I(p) \bullet v + \frac{\partial p}{\partial t}) \tag{7}$$

Where $\bigtriangledown I(p)$ is the spatial gradient and $I_t(p)$ is the temporal gradient of the neighboring pixel p. v is the optic flow vector for pixel (x, y) and $W(p)$ is the weight associated with the neighboring pixels.

We then solve using the method of least squares.

This method is widely used because of its many advantages. By computing a sparse optical flow, you don't need the entire image to estimate the global optical flow. Also, images with large, homogeneous areas don't pose a problem. In fact, with methods similar to Horn-Schunck, iterating to extrapolate the gradient can lead to a false result. What's more, calculating the optical flow only in a defined set of points makes the calculation much faster and less resource-intensive, which is interesting in our case since we're using a Raspberry PI.

Pyramids

Optical flow, and in particular the Lucas-Kanade method, only works correctly in the presence of small movements. However, there is a way around this problem: the pyramid method. The aim

is to create a series of images of progressively lower resolution and then recalculate the optic flow on each image.

By taking the lowest resolution, a large movement is reduced to a much smaller one and can therefore be used by the optical flow. The optic flow is then recalculated on a higher resolution image, taking into account smaller movements, until the original image is reached. The result is an optical flow that takes into account every amplitude of movement.

Harris and Shi-Tomasi corner detector

The basis for the computation of an optical flow is the detection of corners since their intensity changes strongly with the slightest movement. For this purpose, Harris proposed an edge and corner detection algorithm in 1988 in [16].

It's based on the aforementioned principle that a corner causes a significant change in intensity when moved. It will therefore move a window $w(x, y)$ on an image, considering the intensity of a point $I(x, y)$ and its intensity after the movement $I(x + u, y + v)$. We then have the change in intensity for a displacement $[u, v]$ such that:

$$E(u, v) = \sum_{x,y} w(x, y)\big[I(x + u, y + v) - I(x, y)\big]^2 \tag{8}$$

We can make a bilinear approximation, which leads us to:

$$E(u, v) \simeq [u, v]M \begin{bmatrix} u \\ v \end{bmatrix} \tag{9}$$

Where $M = \sum_{x,y} w(x, y) \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix}$

We then calculate the eigenvalues of M, which will give us the type of feature detected. If both eigenvalues are low, we've detected nothing, if one eigenvalue is much higher than the other, we're on an edge, and if both eigenvalues are high, we've detected a corner.

To make these cases more visible, we set $R = \det M - k(\operatorname{trace} M)^2$. The Harris algorithm will then find the points for which the value of R is greater than a given threshold and select those that correspond to the local maximum of R.

Existing implementation

For the autopilot part, the drone uses PX4. PX4 is an open-source autopilot system aimed at building low-cost drones for both professional and consumer use. They support a large number of ambitious projects and provide numerous tools and integrations with other services such as ROS, MAVLink, and QGroundControl.

As far as optical flow is concerned, PX4 has an implementation of this functionality. It uses sparse optical flow computed by the Lucas-Kanade method, coupled with pyramids as explained above.

For each frame captured by the camera, a sequence of functions is performed to achieve the desired optical flow:

1. A certain number of noteworthy features are extracted from the entire image. As input to the function, we specify the maximum number of features, and the function returns the active features present in the image. Active features are the notable points in the image that remain present over time. Therefore, we will necessarily have as many or fewer active features as the requested features. Classically, these features are corners detected by the Harris algorithm as explained above.
2. The camera calibration is then used to correct the possible distortion of the image. This distortion can be caused by the camera plane not being parallel to the ground or by a distorted angle of view due to an excessively large FOV. To correct the image, the function takes each point to which a feature corresponds and applies a judiciously chosen transformation. It is implemented in a practical OpenCV function which can be found in [17]
3. Next, all features are averaged along the X and Y axes of the image. The result is a raw mean.
4. The raw mean is then recalculated, taking into account the variance and standard deviation, to obtain a mean with a chosen confidence interval. This allows the result of the optical flow to be taken into account to a greater or lesser extent, depending on the confidence level granted.
5. At the same time, the signal quality is calculated by subtracting the number of active features from the total number of features used.

Visualization

To provide a more accurate representation of what the algorithm has returned, a visualization function has been developed. For each image on which an optic flow has been calculated, this visualization shows the points of interest retained by the algorithm. These are divided into two categories: those used to compute the global optic flow (in green) and those not used because they are outliers (in red). The global vector is also visible, as is the proportion of the image used to calculate the optic flow. The more of the image that is used for the calculation, the more accurate the optic flow will be, as the features are distributed throughout the image.

Figure 6: Optical Flow visualization on the Eiger using the down camera

Determination of the noise

Before taking full advantage of optical flow, it was necessary to determine whether the measurements obtained during position correction were of sufficient amplitude not to be confused with noise. In fact, as explained above, the relative movements of the target become smaller the farther away the camera is from the target.

In our case, the optical stream would be used up to a height of thirty meters. It was therefore necessary to determine the relative movements of the pixels at this height.

To do this, the pinhole camera equations were used. The relationship between height and pixel displacement is given by:

$$x = D * f/Z \tag{10}$$

Where x is the displacement in pixels, D is the displacement in meters, f is the focal length of the camera, and Z is the altitude. We can easily plot the pixel displacement as a function of ground distance.

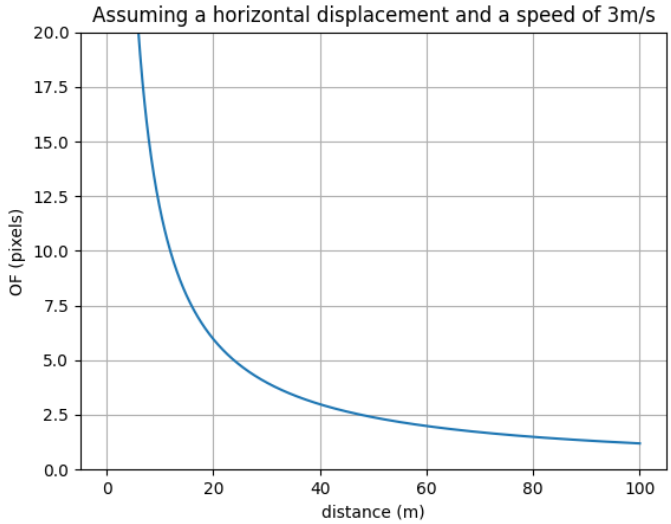


Figure 7: Relation between distance and Optical Flow

It is also possible to plot the optical flow as a function of speed to compare it with the noise. At low speeds, as in our case, the components of the optical flow are not very significant, which can distort the calculations.

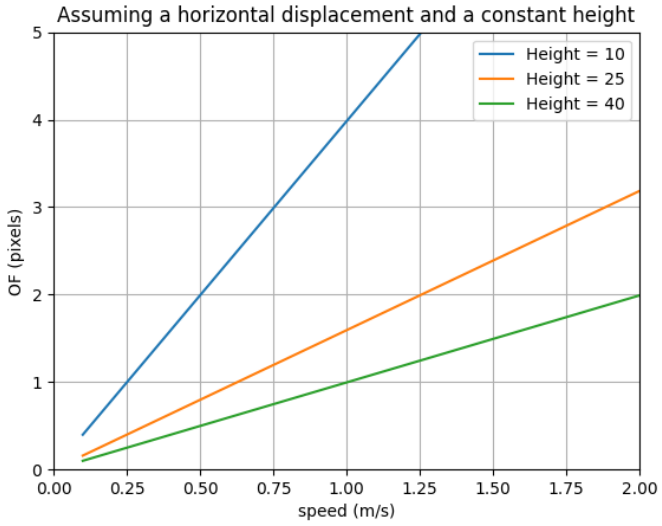


Figure 8: Relation between speed and Optical Flow

Unfortunately, due to conflicting schedules and changing priorities, I didn't have time to analyze the noise from the optical stream while the drone was hovering.

Nevertheless, I did get a head start on all the data processing pipelines to integrate the optical stream into PX4's Extended Kalman Filter (EKF). This means that whoever takes over my job will have a solid base to work from.

1.3 DAA with Sense Aeronautics

Context

The safety of an autonomous drone flight involves several steps. One of them is DAA. The goal is to ensure that if there is an unwanted presence in the drone's path, the drone can make the appropriate decisions to ensure the safety of other users. For the time being, this is only achieved through transponder detection. The drone receives the position of a transponder-equipped aircraft and transmits its own position.

However, the major drawback of this approach is that it assumes that all aircraft flying in the vicinity of the drone have transponders, which is far from being the case. Therefore, alternative and complementary methods are being tested and implemented.

During my internship at RigiTech, another student developed a solution for locating aircraft by sound. He uses a ground station in the take-off and landing zone to monitor the presence of aircraft in the area.

Nevertheless, this does not take into account the BVLOS part of the flight. One solution considered was to use the drone's front camera for distance detection and estimation. I've been working on this implementation.

To do this, I worked with Sense Aeronautics, a Spanish startup that specializes in machine learning aircraft detection. They offer a video processing service that runs on their servers. This was a prerequisite when looking for companies capable of detection. In fact, the onboard Raspberry is not powerful enough to run machine learning algorithms on top of everything else. The Sense Aeronautics solution allows the most demanding calculations to be performed outside of the onboard computer.

Detection in IA

Since all machine learning computations are managed by an outside company, the algorithms used are not open source, so it is not possible to know in detail how they are coded. Sense Aeronautics uses the YoloX-S_640 algorithm, based on the Yolo algorithm. Yolo stands for "You only look once". Yolo's main goal is to detect and locate objects in an image in real-time.

It works according to the following principle:

1. The image is divided into a grid of cells.
2. Each cell predicts several bounding boxes, characterized in particular by confidence scores.

3. Each confidence score is then calculated to determine whether the bounding box contains an object.
4. For each bounding box, YoloX classifies the possible objects by assigning a confidence score to each class of object. The classes with the highest confidence scores are considered the most likely classifications for that box.
5. Redundant boxes are removed. All that remains is to return the detection result.

YoloX is a newer, simpler and more performant version of Yolo.

Implementation and testing

Sense Aeronautics' entire service is divided into two parts. They sell a kit consisting of a camera, an onboard computer, and a ground station that allows image processing to be done locally by integrating it into the drone. This system is completely autonomous from the rest of the drone, so integration is easy and comes only at the price of increased weight and power consumption. Unfortunately, Sense Aeronautics seems to have production problems with these kits, and it was not possible to obtain one despite our repeated requests.

So we had to resort to the second service offered by Sense Aeronautics. This consists of an Application Programming Interface (API). With an Internet browser and the appropriate access rights, you can access their treatment process.

This process is divided into several phases.

1. Source creation: This is where you specify the video stream to use as input. Currently, you can process a video uploaded to YouTube or a stream using the Real Time Streaming Protocol (RTSP) by specifying its URL and resolution.
2. Process creation: A computing process is then created using the previously specified source as input. Once the process is created, it is launched and begins processing the source video stream.
3. Data Retrieval: When a process is launched, two actions become available. There is an option to view the processed video stream in real-time via an URL, modulating the processing time. It is also possible to make a call to the API to retrieve data corresponding to a precise moment in time.

The first problem I encountered when trying to test the system was the incompatibility of the source types with our existing software. In fact, the stream we send is managed by GStreamer and a UDP protocol.

Since setting up a stream using an RTSP protocol was not obvious at first, I started by processing YouTube videos.

What's more, while waiting for access to the API, I searched for drone videos that would be of interest in our case. In fact, to test the reliability of the system, I needed videos of drones flying

at different altitudes and speeds, more or less far from the camera, and in configurations that were difficult for a detection algorithm.

I found what I was looking for in [18]. The goal of the Drone vs. Birds challenge was to develop detection algorithms that could distinguish drones from birds. This was exactly what I needed to test the Sense Aeronautics program.

So I was able to start testing their program, first by using their API via their website. This wasn't very practical, as each step had to be entered by hand. What's more, as I was dealing with relatively short videos, of the order of thirty seconds or so, I had little time to observe what was going on. However, the stream of the video showing whether or not it had been detected was a first step in assessing the quality of the detection. At first glance, the algorithm seemed to be performing well, drones were detected in most cases and the tracking of a single drone was good.

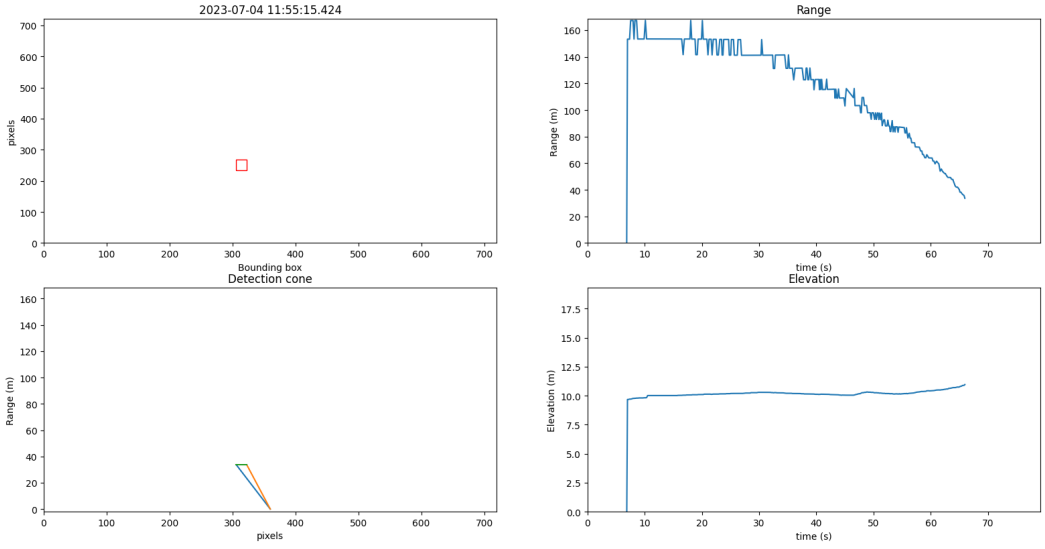


Figure 9: The dynamic interface

To make testing and data processing easier, I started automating the process of calling the API. To do this, I wrote a script using Bash scripting.

Bash is a command-line programming language. It is used by all distributions that run a Linux kernel, for example. You can write a sequence of instructions in a file (a script), which is then executed one after the other. This has the advantage that it is very close to the machine and can be quickly set up and tested for simple applications. For more complicated calculations and structures, it's best to use a more advanced language.

In my bash script that called the API, I also included a way to retrieve the processed stream locally and to start a log of the retrieved data.

This data retrieval was not practical. In fact, as mentioned earlier, one data point corresponds to one API call. During our discussions, Sense Aeronautics assured me that they were working on a continuous data stream, but this apparently didn't happen before I left RigiTech. So I had to code a while loop that made API calls as long as the video was being processed. The downside of this method, besides the fact that it can lead to server problems, is its relative slowness. I had a data capture frequency of between five and ten hertz.

The next step was to determine how the data would be processed. For each data point, several quantities were recorded.

1. id : the identifier of the detected drone. If the algorithm lost the drone and then found it again, the id will be different.
2. Bearing: The angle of the drone to the north.
3. Azimuth: Cylindrical coordinates.
4. Elevation: Cylindrical coordinates.
5. Range: The distance between the drone and the camera.
6. Recommendation: A boolean indicating whether the detected drone should be avoided or not.
7. Bounding Box: The coordinates of the bounding box around the detected drone.
8. Confidence: The confidence in the detection.
9. Timestamp: The UTC time the drone was detected. This was added at my request, but is only accurate to the second.

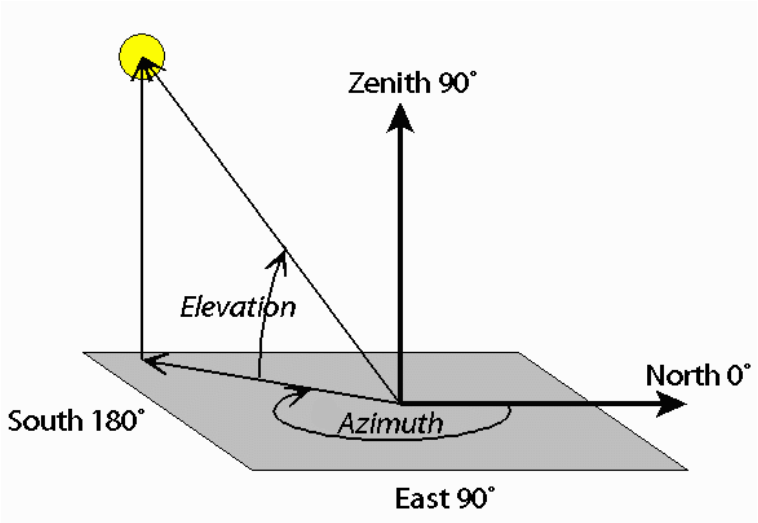


Figure 10: The coordinates used by the video DAA

A confidence is calculated on bearing, azimuth, elevation and range. Like the recommendation or the global confidence, I did not know how they were calculated. So I didn't take them into account when I started processing the data.

I then developed a kind of graphical interface in Python to dynamically visualize the drone's position and the data of interest. This visualization was almost in real-time, modulo the video processing time. When sending a YouTube video for processing, there was a delay of about five seconds. As a comparison, my final system with a RTSP stream and ROS publisher had a fifteen seconds delay.

In the course of these tests, I became aware of a rather annoying issue. The false positive rate was extremely high. I noticed this problem on one of the Drove vs. Birds challenge videos. On this video the false positive rate reached 67%, which is extremely high. As highlighted by Sense Aeronautics, this video did not represents a normal situation that could have happened to the drone. In fact, the camera was fixed to the ground and detected fixed objects such as a lamppost.

Using video from the drone's front camera reduces the false positive rate, but does not eliminate it. Some man-made structures are still detected as drones. So are waves or boats in the ocean.

This is a serious issue for RigiTech because one of the drone's missions is to drop packages on offshore wind turbines. If waves and wind turbines are detected as drones to be avoided, then this method is not viable.

(a) Boat **(b)** Wind turbine

Figure 11: False detection

The goal of this task was to perform a demonstration to prove the interest of this method. We have therefore decided to ignore this problem for the time being and have notified Sense Aeronautics to work on it.

Integration to the existing system

After my tests, the goal was to be able to integrate it into existing code infrastructures. There was already a module that could send information from a Raspberry PI to the cloud. By sending the right information in the right format to the output of my program, the position of the detected drone would appear on the cloud.

To do this, I had to perform a number of steps:

1. Create an RTSP stream
2. Transmit it to Sense Aeronautics
3. Integrate the whole thing into a Docker
4. Create a publisher so the received data can be sent to the cloud

So the first step was to be able to stream video from a camera to the Sense Aeronautics servers. To achieve this goal, I had to comply with their standards. As mentioned above, they only accept YouTube videos or RTSP streams as input. Since the first option was obviously not valid, I had to set up an RTSP stream from a Raspberry PI.

Some cameras have this option built in. All you have to do is connect to their address and set a password to access the stream from anywhere. Our goal was to use the drone's front camera, which doesn't have this option. However, Sense Aeronautics took into account the specifics of the drone and calibrated its algorithm accordingly. To send an RTSP stream from a Lambda camera to a Raspberry PI, I had to use two things.

The first is Libcamera. It's an open-source library that provides tools for everything related to managing camera streams on embedded systems running Linux. You can use it to control any kind of camera and connect it to other processes that use video streams.

This is the case with Mediamtx, a tool that can be used to create a real-time multimedia server that follows numerous protocols, including RTSP. The combination of these two tools, after some laborious installation and configuration, allowed me to obtain a stream via RTSP in real-time, visible from any device connected to the local Wifi network.

But I needed to retrieve this stream from a global address so that the Sense Aeronautics servers could also access and process the stream. To do this, there's a technique that's not very recommended from a security perspective called port forwarding. This involves opening a port on a network router to the public. Anything coming through that port will be redirected to a network address, for example. This allows an application outside the network to access an internal service inside the private network.

In our case, we would have given Sense Aeronautics a global RTSP address, which would then be redirected to the internal address where the stream is actually running.

Port forwarding only works if the router is configured to give static IP addresses so that the external process can always connect in the same way. At RigiTech, however, the router is configured to provide dynamic IP addresses. These will therefore change over time, rendering port forwarding obsolete.

To solve this problem, we can set up a dynamic Domain Name System. A DNS is what converts Internet addresses used by humans into IP addresses that machines can understand. To set up

a dynamic DNS, you need to use an external service provider that redirects a certain number of dynamic domain name addresses to a static address.

I've tried to set up this whole system for a long time without success.

In the end, the cloud has ports dedicated to communicating with the outside world. By using port forwarding to point to my RTSP stream, it was visible to the outside world.

So I had a real-time stream that could be processed by Sense Aeronautics and send data back to me. All I had to do was integrate it into the existing structure.

Docker and ROS

The existing structure is a ROS node running on an Edge Node. The Edge Node is a drone port manufactured by RigiTech. It provides internet access to the drones and at the same time sends information to the cloud. That's what we're interested in here. On the Raspberry of an Edge Node, you'll find all the code infrastructure needed to send information to the cloud. It's a ROS node, but it has the peculiarity of sending what it publishes to Redis, a database capable of high-performance real-time work.

Like all code developed by RigiTech, this bridge between the edge node and the cloud takes place in a Docker. A Docker is an application that is isolated from the system on which it is deployed. This ensures the reproducibility of working and development environments. A Docker image contains everything you need to build an application, including dependencies, libraries, and code. All you have to do is deploy it on another machine to find exactly what you need in the first place.

So I had to integrate all my dependencies into Docker to run Libcamera and Mediamtx. Then, in the code that would run inside the Docker image, I added a publisher and a ROS subscriber to be able to transmit the data I was processing. An existing function took care of passing the ROS information to Redis.

Demo day and post processing

In order to coordinate with the other student doing acoustic DAA, I had to make some improvements and adjustments to my data processing. For example, in order to have a similar position on the cloud, I had to return positions in degrees. But I only had access to azimuth, elevation, and bearing. I also had range. So I used the following formula to find a position for the detected drone, knowing the camera coordinates Lat_c and Lon_c.

$$\begin{aligned} Lat_f &= \arcsin(\sin(Lat_c) * cos(range/R) + \cos(Lat_c) * \sin(range/R) * \cos(azimuth)) \\ Lon_f &= Lon_c + \arctan2(\sin(azimuth) * \sin(range/R) * \cos(Lat_f), \cos(range/R) - \sin(Lat_f)^2) \end{aligned} \tag{11}$$

Note that the angles are in radians, which also gives us Lat_f and Lon_f in radians.

So I had a video stream that gave me the geographic position of the detected drone on the cloud.

At the very end of my course, we went out for some real-life tests. These consisted of a drone, manually controlled by a pilot, making passes both in the field of view of my camera and in the listening range of the microphones. It was a pleasant surprise to see that the results were more or less the same. Despite a long delay of about fifteen seconds on my part, and an approximate calibration of both the camera and the placement of the microphones, the detected position of the drone, particularly at altitude, was relatively similar and at least comparable.

Due to lack of time, we were unable to process the data to a higher quality, but this remains promising for the future of the Detect and Avoid project.

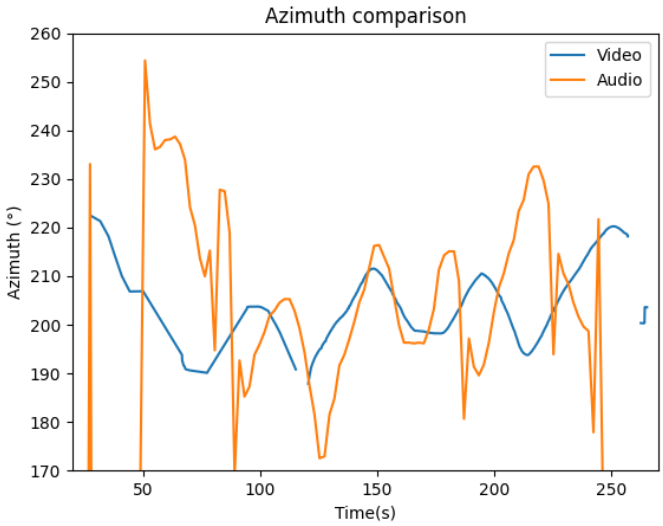


Figure 12: Comparison between optical and acoustic DAA

2 Simulation

Simulation is an essential step in R&D. In the past, prototypes had to be built at great expense before they could be tested in the field. This was a long and expensive process. The development of computer simulation tools has eliminated both of these disadvantages. Whether for testing situations that are difficult to reproduce in a real test, or for extensive testing of a wide range of scenarios, simulation is a practical and widely used tool. Its main drawback lies in its ability to accurately simulate the real world. If simulation tests differ from what would happen under similar conditions in the real world, then simulation is meaningless and irrelevant. However, an extremely sophisticated simulation is not necessarily necessary, as it depends largely on the tests that are being run on it. At RigiTech, simulation is an important part of every development process. It's even possible to simulate a real drone in the cloud. All its parameters and characteristics are taken into account, so that a simulation flight in the cloud is similar to a real flight.

There is still room for improvement, especially when it comes to the influence of the outside world on the drone. That's one of the areas I'm working on.

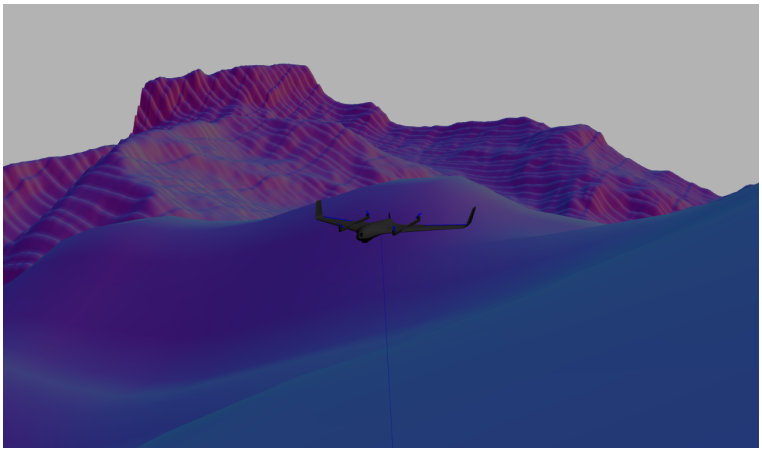


Figure 13: The drone flying in Gazebo.

2.1 Wind

PX4 and Gazebo

The entire physical simulation of the drone is managed by PX4. More specifically, PX4 integrates and uses a version of Gazebo. Gazebo is an open-source 3D simulator that is widely used in robotics. It allows you to create complete environments for simulating robots, taking into account their actuators, motors, interactions with the environment and the physics of the world.

There are two ways to create a world and a robot in Gazebo. The first and easiest way is to use the simulator's graphical interface. You can add simple shapes, set the simulator's physics and import user or community created models. It is much less easy to connect this creation to something else.

It is also possible to program in a different way with Gazebo. It's a simulator based on a set of files that can be modified at will. Worlds and robots can be created in Simulation Description Format (SDF) files. These use a system of tags to specify everything from geometry and physics to objects, sensors and simulation environment parameters.

Traditionally, a simulation is divided into two or three main files. The robot is described in the Unified Robotics Description Format (URDF), a standard originally developed for ROS. The world is described in a *.world* file, which is an overlay of SDF. A configuration file can also be written to set simulation options, physical properties and object and robot behavior. This is usually in *.yaml*.

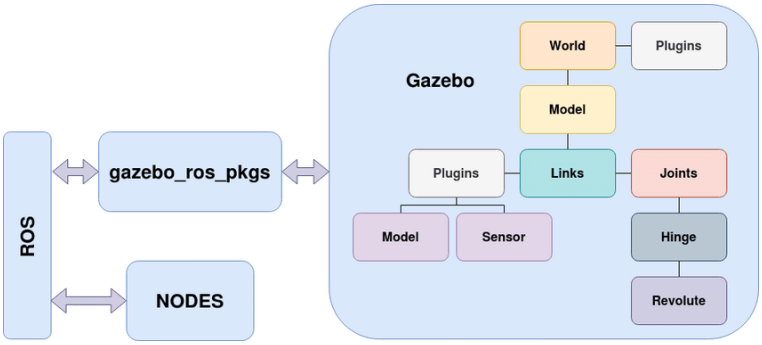


Figure 14: Gazebo structure and dependencies from [2]

All these files, called in the right place, are easy to interface with other systems used in robotics. It's very common to run Gazebo inside a ROS node. Gazebo has an internal messaging system similar to ROS.

PX4 makes interfacing easy. All you have to do is call up the model you want to simulate and the world you want to run it in, and all the connections are made so that the drone can also be simulated via PX4. We then have a complete model capable of reproducing with good fidelity what would happen in reality.

To integrate all this into the RigiTech structure, the whole simulation part is integrated into a Docker.

Task realization

My first contact with simulation under Gazebo and PX4 at RigiTech was to carry out a fairly simple task. Gazebo allows you to simulate wind using a plugin. I'll come back to this particular structure later. When I started, the wind had a constant direction and speed. My aim was to make it controllable when the simulator was started.

To start this I had to find and write my first Bash script. In fact, to set up the simulator and get feedback from the cloud at the same time, I had to include my calculations and modifications when creating the Docker image. As I did later for my Detect and Avoid integration, I placed myself in the entry point of the docker file to directly modify the world invoked by the simulation.

To simulate wind in Gazebo, the plugin only needs two pieces of information: wind speed and direction along the X and Y axes. It is also possible to set other parameters such as gust strength and frequency, but this was not of interest in our case.

In order to run the simulation with the correct parameters, I had to call up the correct world and change these values within the file. This can be done using the *sed* command in bash, which looks for a text pattern in a file and replaces it with another.

All that remained was to implement the necessary calculations to switch from one angle to two projections along the X and Y axes. It's a simple projection calculation, but it requires the use

of sines and cosines. This was the main difficulty of the task, as it's not easy to do complex calculations like this in Bash.

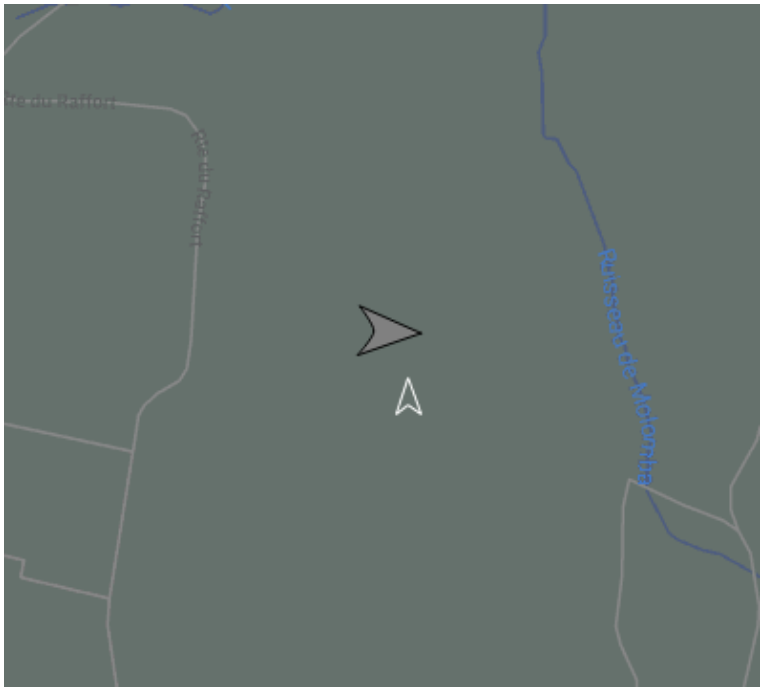


Figure 15: The drone in the cloud in grey with the wind in white

After a bit of research and discovery, I learned of the existence of a library for this type of calculation in bash [19]. This saved me a lot of trouble, as one of the recommended methods for calculating trigonometric functions was to use integer series.

After implementation and testing, the wind worked fine! By the time I left, another intern specializing in the frontend was implementing this function to access and change these parameters from the cloud.

2.2 Elevation

After this warm-up, the real work began. During the mission tests, the pilots had encountered a problem with the drone's reactions. If the departure and arrival altitudes weren't the same, the drone wouldn't react well. It would crash in mid-flight if the arrival altitude was higher than the departure altitude, and sink to the ground if the departure altitude was lower.

There was a simple reason for this. The world in Gazebo is basically represented by an infinite plane on which the drone starts. This is the only point of support it has in the world, so it's bound to collide with it at some point. To avoid this issue, I had to implement a Digital Elevation Model (DEM) in the world to solve it. A great introduction to DEM and the way to store and use them could be found in [20].

As with the Gazebo wind implementation, I had to do my manipulations on the simulator before starting it. The DEM had to be centred on the drone's position and the model had to be adapted to the drone's maximum range. We couldn't fly without a DEM, but it couldn't be too large for performance and memory reasons. The strategy I used to implement a DEM was as follows:

1. Retrieve the DEM for the region of interest
2. Create a world file to use with the DEM
3. Position the DEM correctly in relation to the world coordinates
4. Launch simulation with world and drone

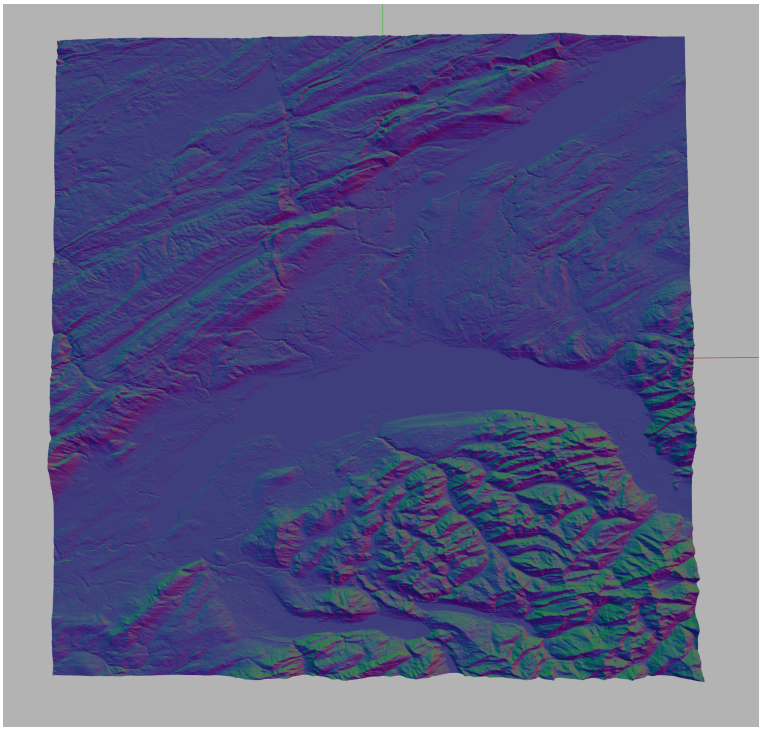


Figure 16: DEM of the Geneva lake

Fetch the DEM

For various functions, including this one, the entire world modeled by Copernicus in DEM is uploaded to RigiTech's servers. An extract can be downloaded via a special API request. By specifying a pair of coordinates in degrees and size in kilometers, we get the requested DEM file. This is a Geotiff file, which assigns latitude, longitude, and altitude to each pixel. The coordinate system is available, as is other information that I'll return to later.

However, after learning how to make requests to an API, I realized that the file returned was not exactly what I had requested. This is because, by default, the Copernicus model is divided into tiles

whose corners are geographical coordinates in half-degree increments. If we take a quadrilateral with a side of one degree, we'll have four tiles.

What the API returned were the tiles contained in the requested area, rounded up. So the requested point and area were included in the returned file, but the center of that file did not match the requested point.

By default, Gazebo centers DEM files around the world and makes the drone appear in the center as well. It was therefore convenient for me that this central point should be the correct one. I then had two options. I could either download the Geotiff file and then resize it to my liking, or solve the problem upstream and download the correct file directly.

Although the whole part of controlling what was returned by the API calls was handled very differently from the code I was used to. It was Javascript, coded asynchronously so that nothing would block the code. As the only person dealing with this part of the code was extremely busy, I started by coding something that would take the result of the API call as input and re-encode the result to get the expected format.

To work efficiently with Geotiff files, I used the Geospatial Data Abstraction Library (GDAL). This is a very powerful open-source library for manipulating geographic data. In particular, it allows conversion between formats, image transformation and access to the metadata of Geotiff files. To visualize the resulting files, I first used online Geotiff viewers. However, they offered limited functionality and didn't gave me the opportunity to check what I wanted. I then used QGis, a specialized software package that allowed me to validate the sizes of the files I was resizing. Without this software, it wasn't an

To get the desired square, I had to find its corner coordinates, taking into account the center, whose position I knew. There's a function that takes as input a pair of coordinates Lon_M/Lat_M, a distance D and an angle b, and returns the corresponding pair of coordinates. We then have :

$$\begin{aligned} Lat_F &= \arcsin(\sin(Lat_M)) * \cos(d/R) + \cos(Lat_M) * sin(d/R) * \cos(b) \\ Lon_F &= Lat_M + \arctan2(\sin(b) * sin(d/R) * \cos(Lat_M), \cos(d/R) - \sin(Lat_M) * sin(Lat_F)) \end{aligned} \tag{12}$$

Using $d = D * \sqrt{2}/2$ you get the sides of the square of the desired size. All angles and coordinates are given in radians. R is the radius of the Earth, approximated to be 6371000 m.

In order to run tests quickly, I started by implementing these equations in my Bash script after downloading the global Geotiff file. This allowed me to perfect my code and get what I wanted by modifying a Python file locally, which was much faster than the API code. In fact, to test my changes in Javascript, I had to commit each time to a git and run a pipeline that built me the adapted image. For a long time, I didn't have a Javascript environment on my PC because the installation process was so cumbersome. However, I couldn't be satisfied with my local function. In fact, every time I called the API, I was downloading an extremely large map, too large to keep just a small part of it. This meant that my process took far too long, for no justifiable reason.

I worked on the Javascript code to implement these equations upstream in the data processing stack. The latter had the particularity of using only asynchronous functions, which gave me quite a lot of trouble. In fact, I learned to code with synchronous functions, where calling one function after another results in one response after another. This is not the case with asynchronous functions, which are executed simultaneously, depending on which one takes the longest and uses thread priority and management.

I spent a lot of time trying to get this part to work in Javascript. I had functions that fetched variables that I thought had been defined earlier, but hadn't been because the previous part of the function hadn't run yet. This would, for example, result in Geotiff files of the correct size and position, but with zero elevation everywhere.

To control this kind of behavior you need to use the await function and promises. The use of await interrupts the execution of the function, freeing a thread, until the expectation conditions are met. This is done by resolving a promise. A promise is a representation of a value that may be available now, in the future, or in the past. Once the promise has been fulfilled, execution of the function can resume. It is also possible to rely on the non-resolution of a promise to keep a function running.

After a lot of experimentation, installing a proper Javascript development environment and help from the cloud expert at RigiTech, I managed to get something working. In the end, I didn't use the above equations directly. Instead, I used a function from the GDAL async library, [21] which had the advantage of working directly in asynchronous mode. However, it used the same equations underlying it.

So I had a well-placed file of the right size as soon as I called the API. The next step was to insert the card into Gazebo and place it correctly.

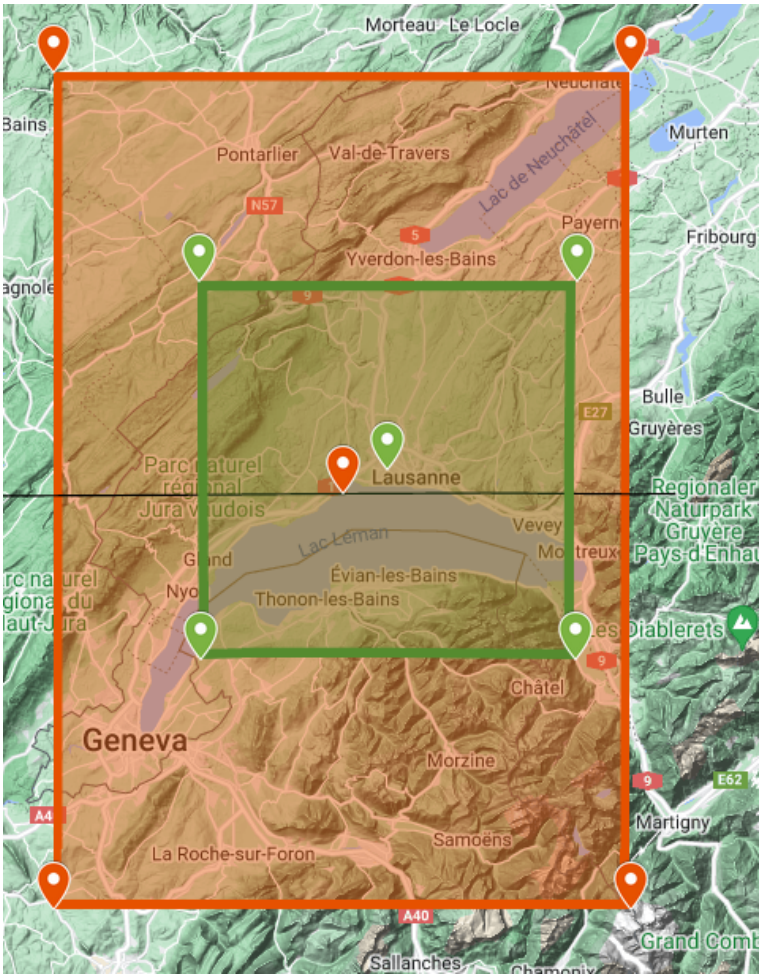


Figure 17: What the API returned in red versus what we asked in green

DEM related stuff

To integrate a height map, Gazebo provides a structure similar to that used to introduce 3D objects into the world. This involves specifying the position and size of two entities: the visual of the object and its collision model.

Gazebo works this way for a simple reason. Collision models are generally very simplified compared to visuals. They require a lot of computation, which is not necessarily desirable in a simulation to maintain fluidity. For example, the collision model of a car could be approximated by a rectangular parallelepiped.

This is not the case with a DEM. In fact, it's desirable for the collision model to be as close to reality as possible, which inevitably leads to greater complexity.

So I knew quite easily how to integrate a height map into a Gazebo world. For the choice of the size, this corresponds to the real dimensions, if not specified. I also had to take into account the

height of the desired point when creating the DEM.

By default, Gazebo places the center of the DEM at the origin. The drone, when it appears, must also be positioned at the origin so that PX4 uses the correct elevation and doesn't create an offset. This requires moving the center of the elevation map to the origin and therefore knowing its elevation.

I had two ways to do this. Either I used known positions of the drone in the real world to reproduce them in the simulation, or I used another call to the cloud API that returned the altitude of a point whose coordinates were sent to it. This way I had a DEM that seemed to be well placed in the world.

That's when I noticed a problem that wasn't happening all the time. As my tests progressed, my process became more and more automated. At first, I used external software to make my API calls and manually integrated the resulting DEM into a version of Gazebo running standalone on my computer. It was a process that I mastered every step of, but it had the disadvantage of being very slow.

By automating everything with a Bash script, I was able to run my tests much faster. However, some of my API calls returned a DEM which was like being cut in half. One half was normal, while the other half had no elevation.

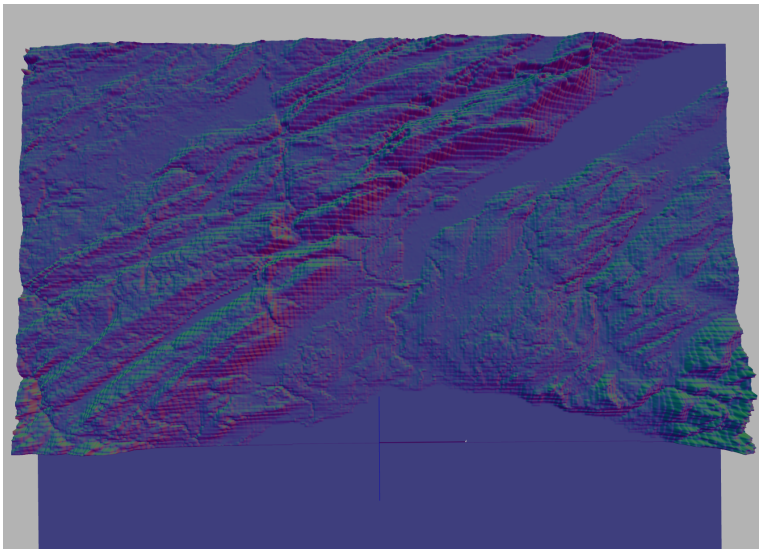


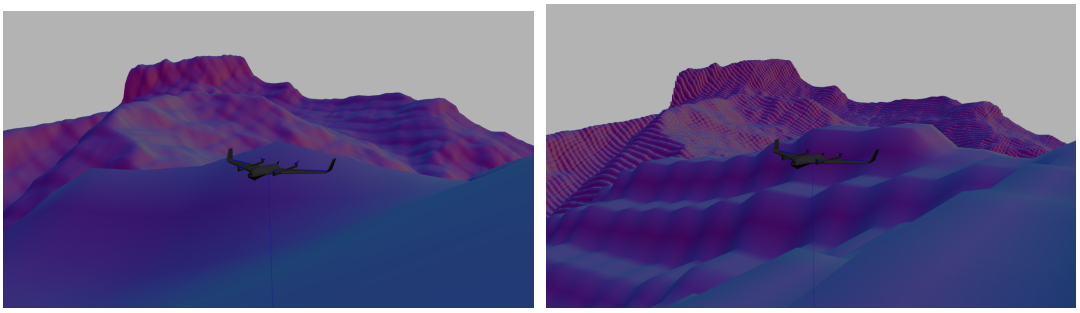
Figure 18: A DEM with a non standard resolution for Gazebo

After much research, it seems that Gazebo can only handle height maps of a certain resolution. The DEM must be square and have a resolution of $2^N + 1$ to be displayed correctly. Once these considerations were taken into account, I got a height map that displayed correctly every time. Before that, I had already changed the resolution, but only to maintain the width/height ratio

of the base image. Without reducing the resolution or the size of the base image, loading under Gazebo was very slow at best and impossible at worst.

So for my tests I drastically reduced the resolution and size of the height map to have something fast to test. The slow loading of the DEM was not a problem during the final integration into PX4's Gazebo. In fact, the Gazebo was started without a graphical interface, which greatly reduced the loading time. It was therefore in my interest to choose the highest possible resolution to get a simulated world as close to reality as possible. That's why I added a part to my height map generation bash that allows me to choose the highest possible resolution without exceeding the initial one, always respecting the $2^N + 1$ constraint.

I also took the opportunity to do some CPU testing. By running a Docker image, it's quite easy to observe the percentage of CPU it uses. Changing the resolution when the GUI was not present had no noticeable effect on CPU usage. I could therefore increase the resolution as much as possible without affecting the speed of the simulation, which in our case should be as close to real-time as possible.



(a) The extrapolation is too high, there is some data losses

(b) A high resolution with a Pixel=Area paradigm

Figure 19: Different resolutions for the same height map

Task realization

So I was at a point where I had mastered the map I was generating, both in terms of size and resolution. It was also integrated into Gazebo. All that remained was to place it correctly in the world.

As explained above, Gazebo places the center of the DEM at the origin of the world by default. I had already more or less set the position along the Z axis. However, I needed to add a margin of about two meters. By using the exact elevation value at the origin, the drone tended to appear in the DEM, which caused collision problems and made it move quite violently. This offset was not constant and I had to change it with a trial and error method until finding the good value. It couldn't be too small to avoid collision problems, nor too large, as the drone would then tend to fall away from its initial position and flip over. This behavior was very strange, I'll explain it in details a bit later.

I also had to make sure that the X and Y positioning was correct. The drone was always slightly out of position. To do this, I had to correlate Gazebo's reference frame and coordinates with those of the PX4, which are the most commonly used in the real world.

The main problem was Gazebo's handling of movement. To move a DEM, you have to specify a distance in meters along the three axes of a conventional Cartesian coordinate system. This is easy for the vertical Z axis, since elevation is generally expressed in meters. For the other axes, however, I had to convert from a geographic projection in degrees to a Cartesian projection.

To do this, I used Harvesine's formula. This gives the distance in meters between two geographic coordinates, written as :

$$\text{Haversin}(\theta) = \sin^2\left(\frac{\Delta\text{lat}}{2}\right) + \cos(\text{lat}_1) \cdot \cos(\text{lat}_2) \cdot \sin^2\left(\frac{\Delta\text{long}}{2}\right) \tag{13}$$

Where :

- Δlat and Δlon is respectively the difference in latitude and longitude between the two points in radians.
- lat_1 and lat_2 are the latitudes of the two points in radians.

Once the Harvesine has been calculated, the distance between the two points can be deduced using the following equation:

$$d = 2 \cdot R \cdot \text{atan2}\left(\sqrt{\text{Haversin}(\theta)}, \sqrt{1 - \text{Haversin}(\theta)}\right) \tag{14}$$

Knowing the actual position of the drone and its desired position, it is then possible to find the distance between the two and provide it to Gazebo. In order to have data that can be used to move the map, though, this distance needs to be broken down along the X and Y axes. I did this using a Python library that specializes in projections.

To do so, I had to specify my input and output coordinate systems and transform my start and end points into the input system. After the transformation, I was able to find the displacement I was looking for by calculating the difference between the two coordinates along the X and Y axes.

I now had all the information I needed to achieve my goal - all I had to do was test it on the cloud simulators. That's when the problems began.

Projection Problems: Resizing

To test my modifications, all I had to do was call up the correct image that integrated Gazebo when I launched the simulator. I then had a view of the simulated drone placed on the map in the cloud. Visually, everything was fine. The drone seemed to be in the right place and at the right

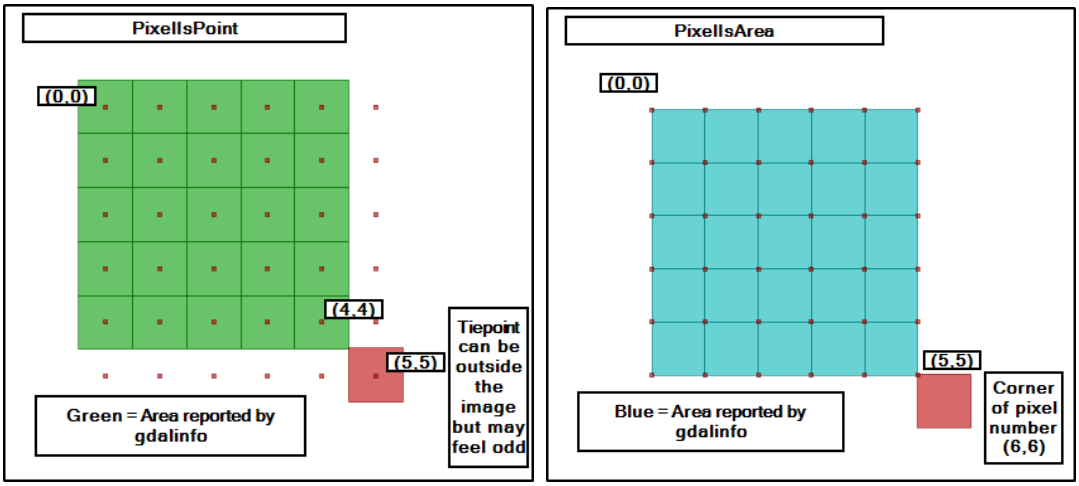
altitude. However, when I launched a mission to check that it was working properly, everything changed.

In fact, position tests are performed before a mission is launched to ensure accurate take-off and correct initialization of parameters, especially GPS. These tests allowed me to determine that the drone was not in the correct position. The error ranged from three to fifteen meters.

For internal use, this wasn't a big problem. All you have to do is increase the tolerance of the position tests at the beginning to get something functional with a small error. However, from the customer's point of view, it wasn't possible to tolerate this error. In fact, some of our customers had to launch a drone from a rooftop. Visually, it would have appeared next to it, which could have caused confusion. So I had to figure out the source of the issue.

I started by changing the way I was resizing the DEM. In fact, using a GDAL function could gradually accumulate errors as the elevation map was transformed. For comparison, I used two different resizing methods. The first took a DEM corresponding to what was needed as input and resized it, keeping all geo-referenced pixels. This was the method I implemented initially. The second method took a larger DEM as input and cropped it to the correct dimensions without resizing it. I specified the dimensions and coordinates of a square of pixels extracted from the global elevation map. This avoided the resizing problems I hadn't solved, but introduced others. In fact, the choice of pixels on which to base the square was not obvious and led to positioning errors.

In terms of pixels, a DEM can be encoded in two different ways. It can associate an elevation with a point or with a region of the image, which will then have a uniform elevation [3]. The main difference between the two approaches lies in the way the calculations involving pixels are handled. When they are considered as discrete points, the value associated with them corresponds to the center of the pixel. This keeps the data accurate when resizing. What changes is the distance between pixels. Locally, the height remains the same. An interpolation is calculated between each pixel, which introduces inaccuracy. This can be controlled to some extent by changing the interpolation method.



(a) Pixel=Point **(b)** Pixel=Area

Figure 20: Different paradigms for height maps from [3]

Considering that a pixel corresponds to an area on the DEM, the inaccuracy increases. In fact, in this paradigm, when GDAL reduces the resolution, the pixels are aggregated to provide a new value for the pixel resulting from the operation. The initial value of a pixel, when considered as a surface, is one of its vertices. When the DEM is resized, this value is lost in the calculation, and fine details may be lost.

Redimensioning could therefore pose a problem. I tested three different approaches:

1. Resize the DEM by treating pixels as local points
2. Resize the DEM by treating pixels as areas
3. Crop a larger DEM to the right size.

Unfortunately, these methods didn't produce any conclusive results. So I approached the problem from a different angle.

Projection problems: Spherical Coordinates in Gazebo

It turns out that Gazebo can integrate spherical coordinates directly into the world. This is specified when the world file is initialized and allows objects to be placed in the world by geo-referencing them. This seemed like a straightforward solution to my problem. So I tried to place the elevation map at the exact coordinates needed by the drone. But then I noticed a curious behavior. No matter what coordinates I entered, the drone always placed itself at the coordinates corresponding to the upper left corner of the DEM. What's more, physically it was always at the origin of Gazebo, i.e. at the center of the elevation map. After much research, this was due to a bug in Gazebo when integrating a DEM into a world. Its coordinates overwrote the spherical

coordinates defined in the world as explained here [22] by the specialist of Gazebo in the PX4 company.

To solve this problem, I tried to stop using height maps directly by transforming them into another object. I had two options:

1. Transform the DEM into a 3D object and import it into Gazebo as such.
2. Remove the geo-referencing from the original Geotiff, while keeping the elevation.

The first method had the advantage of eliminating all external coordinate systems in favor of Gazebo's own. What's more, a 3D model can be easily integrated into a world and is, in fact, its basic component.

To transform a DEM into a 3D model, I had to take each of its points and then perform a Delaunay triangulation on them. Delaunay triangulation consists of dividing a set of points into non-overlapping triangles with certain properties. These are as follows:

1. The length of the sides of the triangles is minimal.
2. The angles of the triangles are minimum.
3. The circumscribing circle of each triangle contains no points.
4. The transformation is unique except for a particular configuration.

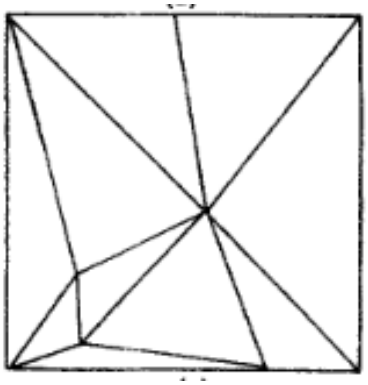


Figure 21: A correct Delaunay interpolation from [4]

It is therefore possible to reconstruct coherent surfaces on a three-dimensional point cloud. However, this method is computationally expensive for a point cloud with more than 500 points [4]. For the surfaces I needed to process, my computer was not powerful enough to perform the transformation. Despite its advantages, this was not a viable method for implementing a DEM in Gazebo.

That left me with the second method. Using GDAL, I was able to convert my Geotiff to a png image that behaved like a DEM, while removing all geo-referencing. This method had the advantage of

being extremely fast, and I was hoping that Gazebo's spherical coordinates would override and replace those of the elevation map. Unfortunately, this was not the case. The drone was positioned at zero latitude and zero longitude. In the end, Gazebo behaved exactly the same as [22].

Trying to position the DEM using spherical coordinates was therefore not possible. So I looked for a different solution.

Projection problems: Moving the DEM

I then set out to revise the way I was moving the height map, assuming that my method of calculation was wrong. So I immersed myself in the theory of geodesy.

First, I calculated my displacements by switching from the WGS84 world reference frame (EPSG 4326) to a Cartesian reference frame in EPSG 3857. These two projections correspond respectively to a classical reference frame expressed in latitude/longitude and to a plane that has been projected according to the Mercator projection.

I did the same when I switched from WGS84 to UTM. In fact, this projection is quite similar to Mercator's, but takes into account the distortion introduced by the latter. The Mercator projection, traditionally used for cartography, is highly distorted in certain places. The most common example is Greenland, which appears much larger on a map than it actually is.

By dividing the earth into several smaller zones, the UTM projection reduces the distortions inherent in the Mercator projection. To implement this method, I had to automate the search for the number corresponding to the UTM zone in which I wanted to make the projection. I used the following method.

1. Take the longitude in decimal degrees and add 180.
2. Divide by six.
3. Round up to the nearest whole number.

But that didn't solve my problem. I still had a discrepancy between the actual position of the drone and the desired position. This tended to increase with the size of the DEM requested. Finally, with the help of one of the hydrography professors at ENSTA Bretagne, I again changed my projection to a local tangent plane. This is indeed the projection used by Gazebo within a world. The unclear documentation had caused me to miss this essential information.

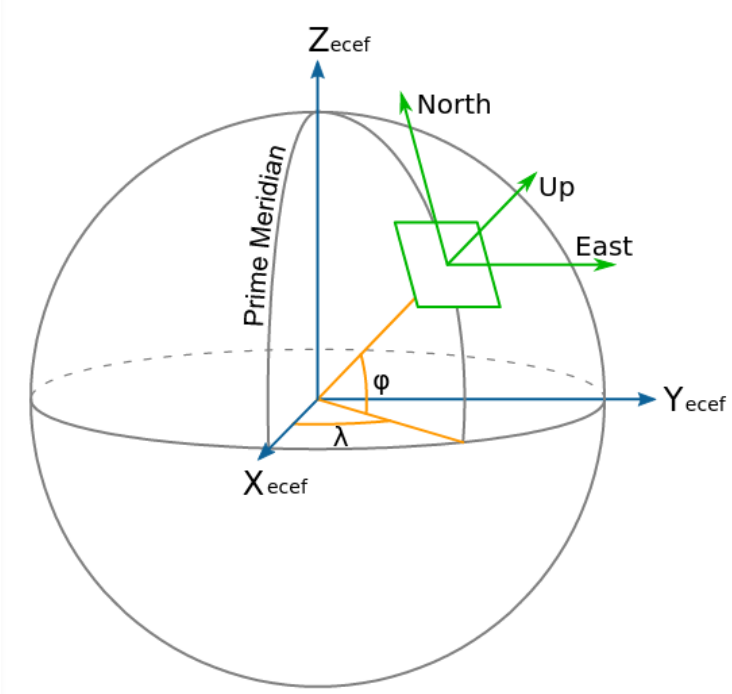


Figure 22: Referential used in Gazebo from [5]

To calculate the displacement in this projection, I used the equations from [23]. The problem raised by [24] was also very useful. I was now sure that I was using the correct projections to calculate the displacement between the drone's actual position and its desired one.

It was also at this point that I understood why the Z offset was not constant. As explained earlier, when I changed test locations, I had to change this value quite significantly.

Again, this was due to a Gazebo bug. When a height map is moved, only the visual moves. The collision model remains static, centered on the world origin [25]. To solve this problem, I set out to develop a plugin that would synchronize the movement of the visual and the collision.

A Gazebo plugin is a software module that can be loaded into a URDF or world file to change its properties. They can be used to add a further level of customization to the world, for example by changing its physical properties. So I tried to develop the missing functionality. However, despite my best efforts, I was never able to move the collision model.

In desperation, having run out of ideas to solve my problem, I turned to the Gazebo and PX4 Internet communities. By explaining my problem in their forums, I kept the hope alive that a solution or a new idea would be suggested to me. I even participated in a call with the CEO of PX4 and some of the engineers in charge of development. They referred me to [22], which I unfortunately already knew. We tried to revive the issue, without much success.

After spending a lot of time on it, I stopped working on it. I sincerely hope that someone will take

over my work and finish it. There are still a few things to improve.

To consider for the future

The most obvious clue is a new version of Gazebo. In fact, the new version of PX4 integrates a different version of Gazebo. With any luck, the bugs that were present in the version I was working on will have disappeared. Nevertheless, it's a lot of work for RigiTech to change the autopilot version, because it requires checking all existing functions. We have to make sure that the existing features work in the same way and that the new features are compatible with the drone.

I was also interested in the case that motivated the whole implementation of elevation in Gazebo, i.e. taking off from a building. A file obviously does not contain buildings. It only takes the ground elevation into account. One possible way to implement this feature would have been to have the drone take off from a plane elevated to the height of the building. However, the positional problems remain.

3 Other tasks

During my internship at RigiTech, I also had the opportunity to work on tasks that took much less time than the ones mentioned above. From the point of view of a startup working in successive sprints, there are always small side tasks, whether it's the quick improvement of an already existing feature or bug fixing.

3.1 Bug Fixing

I had to fix a couple of bugs that came up mainly during test flights. I didn't deal with them at the very beginning of my stage, but I was able to take on more responsibility as I gained experience. That's why the bugs I was able to fix were all in the drone's vision and camera. There were two main ones.

Precision landing not triggered

The first was precision landing, more specifically, not being triggered when performing an emergency maneuver. The drone has several of these functions so that it can have a defined behavior in the event of a problem. In this case, the feature in question was the one that allowed the drone to fly directly to a rally point. Rally points are places on the map where the drone can land in the event of a problem. There will be one or more per mission. The default rally point is the drone's launch point. This is the point that could be used in case of an emergency landing, e.g. due to the triggering of an ACD program.

However, when an SRP was triggered, precision landing was not activated. This was because the waypoints were not updated in the return mission. Once this was fixed, everything worked as

expected.

Crash of video_node

The second bug I had to deal with was more of a testing nature. When calling the ROS service in the video node, it had a strong tendency to close unexpectedly due to a wrong function call order. This had already been fixed by the time I picked up the task. My goal was to run some services in a loop for a while to see if the bug had been fixed. It had been fixed.

3.2 Unit Testing

Another task I was able to take on was writing unit tests for a function. This function checked whether a point was in a geocage by returning true or false.

Geocages are physical areas in which the drone can fly according to regulations. There are three interlocking zones that correspond to certain safety requirements. There is the pre-geocage, which is where the drone should fly at all times. It is contained within the geocage, which the drone must not cross under any circumstances. Finally, the buffer zone is the area where the drone can land in the event of a parachute opening. It corresponds to the maximum drift of the drone when its parachute opens.

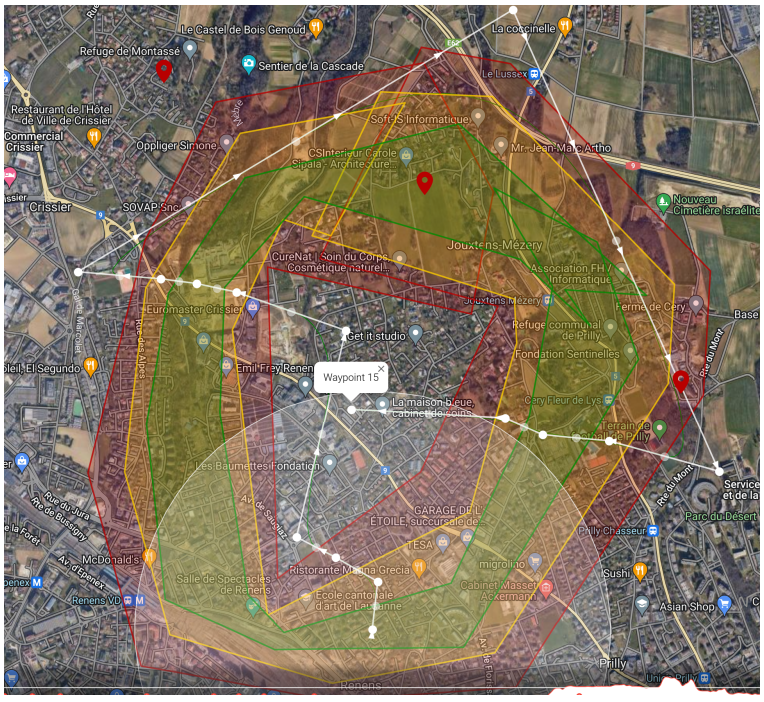


Figure 23: One of the scenarios to test the function. The buffer zone is in red, the geocage in yellow and the pre-geocage in green

In order to carry out this task, I created scenarios such as in Figure 23 in which I tried to find corner cases for the function that checks whether a point belongs to a geocoding type or not. I then ran unit tests for each point, giving them the expected value and automatically checking that the function to be tested returned the same thing.

3.3 Vertical geocage boundary

While working on the geocages, I also had the opportunity to delve a little deeper into the workings of PX4 by adding a previously unused parameter. This parameter controls the maximum height of the geocage that the drone cannot exceed. Previously, this parameter was only used in the onboard computer. If the onboard computer failed, this safety feature was lost. So I took it upon myself to assign the correct value to this parameter when starting a mission. This allowed me to understand the PX4 messaging system and how it interacted with MAVLink. I also had to pay attention to the altitude reference frames, which were not the same between the older parameter and the new one as shown in Figure 24.

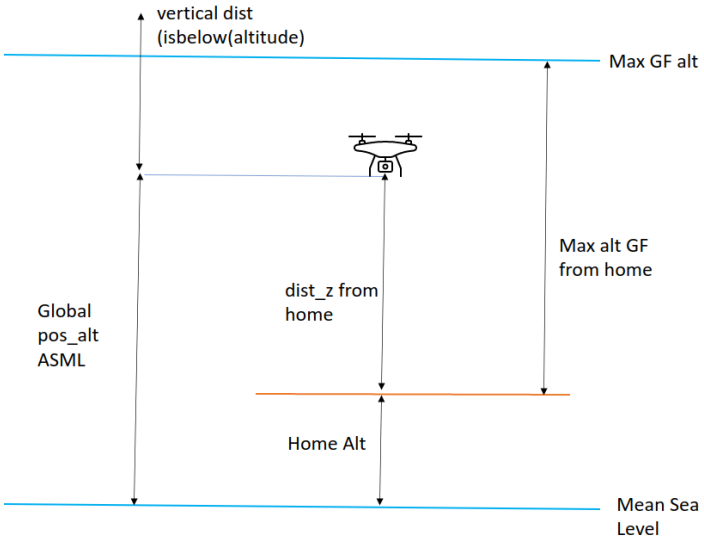


Figure 24: Summary of the altitude frames used to define the max altitude of the geocages.

3.4 Auto-exposure

In relation to all the video features, I had to set up an auto exposure parameter. This was useful for the precision landing. In fact, depending on the brightness, the ARUCO markers on the landing pad were detected more or less well. By activating the auto-exposure parameter, the detection was more consistent depending on the weather conditions. To activate this parameter, I had to insert a line in a bash file at the right place, which got the parameter value with sed and set it to the desired value.

3.5 DronePort

The last task I had to complete was different to anything else I'd done on the course, as it had a much stronger hardware connection. One of RigiTech's R&D projects is the development of a DronePort that can automatically load and unload parcels. I was responsible for controlling the motor that pushes the parcel into the drone or, conversely, unloads it smoothly. To do this I had to use a motor shield on a Raspberry. I also had to integrate two inductive proximity sensors into my system.

The advantage of this project was that I was given a lot of autonomy. I started from scratch and had to produce working code for a demonstration. My code also had to be reusable by someone else in the future. So I created a whole class to make my functions intuitive to use. The demonstration went very well.

Conclusion

During my internship at RigiTech I had the opportunity to work on two main aspects of drone development. Firstly, I got to work on all aspects of the video. Whether it was theory or implementing new features, this gave me a great start to my internship. Completing the Detect and Avoid project was a very satisfying end to my internship. Despite the difficulties and setbacks, the fact that we ended up with a coherent result was a pleasant surprise. The fact that I was working with a foreign company with which I had only occasional contact was very formative. In a way, this contrasted with the easy collaboration between my friend who was developing an acoustic ACD solution, the RigiTech staff and myself. Whether it was solving the network problems, integrating the code into the cloud, or taking us out into the field at odd hours to do some testing, there was always someone there to support me.

The failure of my second major project, involving the integration of a 3D model into the simulation, was something I found very hard to let go of. I spent an enormous amount of time on this project, probably too much, before exhausting all the available solutions one by one. However, I learnt a lot about a less common area of robotics. It also earned me the status of Gazebo expert within RigiTech, given the level of complexity I reached during my research. It also made me aware of any weaknesses I might have in managing and organizing my time, so I could work on them.

The side tasks were welcome and helped me to try out new things quickly. Overall, I learned a lot during this internship. Working closely with the software gave me an overview of much of the drone's software operation. What's more, RigiTech's development of cloud tools is very advanced and a radical change from what I learned at school. Finally, I'd like to pay tribute to the excellent team management I've had the opportunity to be part of. When working in this company, it's easy to feel part of a whole that is moving in the same direction: the development, from scratch to production, of a high quality drone.

References

[1] C. en Lin, "Introduction to motion estimation with optical flow," 2023. https://nanonets.com/blog/optical-flow/ [Accessed: (13/08/23)].

[2] V. Tinoco, B. Malheiro, and M. Silva, "Design, modeling, and simulation of a wing sail land yacht," *Applied Sciences*, vol. 11, p. 2760, 03 2021.

[3] jratike80, "Deal correctly with geotiff pixel-is-point vs pixel-is-area," 2020. https://github.com/opengeospatial/ogcapi-coverages/issues/92 [Accessed: (12/06/23)].

[4] D. T. Lee and B. J. Schachter, "Two algorithms for constructing a delaunay triangulation," *International Journal of Computer & Information Sciences*, vol. 9, pp. 219–242, Jun 1980.

[5] Gazebo, "Spherical coordinates," 2023. https://gazebosim.org/api/gazebo/6.0/spherical_coordinates.html [Accessed: (13/08/23)].

[6] "Opencv," 2023. https://opencv.org/ [Accessed: (24/08/23)].

[7] "Gstreamer," 2023. https://gstreamer.freedesktop.org/ [Accessed: (24/08/23)].

[8] A. Domi, "Low latency adaptive video encoding," in *Computer and information sciences*, vol. 113, 05 2019.

[9] Y. Shuai, M. Gorius, and T. Herfet, "Low-latency dynamic adaptive video streaming," in *2014 IEEE International Symposium on Broadband Multimedia Systems and Broadcasting*, pp. 1–6, 06 2014.

[10] J. Stowers, A. Bainbridge-Smith, M. Hayes, and S. Mills, "Optical flow for heading estimation of a quadrotor helicopter," *International Journal of Micro Air Vehicles*, vol. 1, pp. 229–239, Dec 2009.

[11] L. Rosa, T. Hamel, R. Mahony, and C. Samson, "Optical-flow based strategies for landing vtol uavs in cluttered environments," *IFAC Proceedings Volumes*, vol. 47, no. 3, pp. 3176–3183, 2014. 19th IFAC World Congress.

[12] P. Maurel, "Analyse de séquences d'images, flot optique," 2023. https://perso.univ-rennes1.fr/pierre.maurel/Prepa_Agreg/CM/agreg06_flot_optique_presentation.pdf [Accessed: (13/08/23)].

[13] P. J. O'donovan, "Optical flow : Techniques and applications," 2005.

[14] B. K. Horn and B. G. Schunck, "Determining optical flow," *Artificial Intelligence*, vol. 17, no. 1, pp. 185–203, 1981.

[15] B. Lucas and T. Kanade, *An Iterative Image Registration Technique with an Application to Stereo Vision (IJCAI)*, vol. 81. Apr 1981.

[16] C. G. Harris and M. J. Stephens, "A combined corner and edge detector," in *Alvey Vision Conference*, 1988.

[17] OpenCV, "Documentation for the undistortpoints() function.," 2023. https://docs.opencv.org/3.4/da/d54/group__imgproc__transform.html#ga55c716492470bfe86b0ee9bf3a1f0f7e [Accessed: (13/08/23)].

[18] D. Workshop on Small-Drone Surveillance and C. Techniques, "Drone-vs-bird detection challenge," 2022. https://github.com/wosdetc/challenge [Accessed: (12/06/23)].

[19] S. Tips!, "How to do advanced math calculation using bc?," 2020. https://www.shell-tips.com/linux/how-to-use-bc/ [Accessed: (12/06/23)].

[20] R. Trent, "The secret life of geotiffs," 2023. https://blogs.loc.gov/maps/2023/05/the-secret-life-of-geotiffs/ [Accessed: (12/06/23)].

[21] M. Momtchev, "gdal-async 3.7.0," 2022. https://mmomtchev.github.io/node-gdal-async/ [Accessed: (12/06/23)].

[22] Jaeyoung-Lim, "Dem origin overrides spherical coordinate flags defined in .world," 2020. https://github.com/gazebosim/gazebo-classic/issues/2884 [Accessed: (10/08/23)].

[23] P. S. A. Society, "Conversion of geodetic coordinates to the local tangent plane," *Portland State Aerospace Society*, 2007.

[24] swissknight, "Reproject wgs84 raster file to topocentric ltp-enu using gdal with a custom proj string," 2021. https://gis.stackexchange.com/questions/384512/reproject-wgs84-raster-file-to-topocentric-ltp-enu-using-gdal-with-a-custom-proj [Accessed: (12/06/23)].

[25] O. Robotics, "Setting position on heightmap creates offset between collision and visual," 2023. https://github.com/gazebosim/gazebo-classic/issues/868 [Accessed: (12/06/23)].

Acronyms

API Application Programming Interface

BVLOS Behind the Visual Line Of Sight

DAA Detect And Avoid

DEM Digital Elevation Model

EKF Extended Kalman Filter

FOE Focus Of Expansion

GDAL Geospatial Data Abstraction Library

GNSS Global Navigation Satellite System

IMU Intertial Measurement Unit

ROS Robot Operating System

RTSP Real Time Streaming Protocol

SDF Simulation Description Format

UAV Unmanned Aircraft Vehicle

URDF Unified Robotics Description Format

Appendices

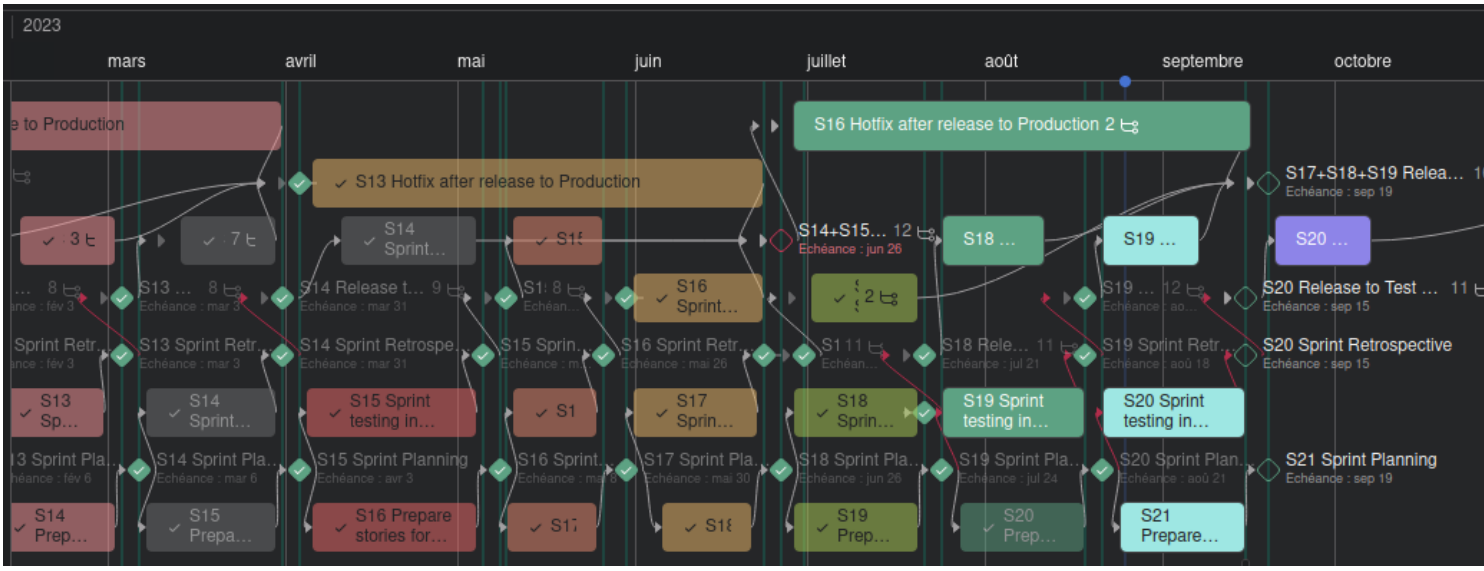


Figure 25: Sprint organization in RigiTech

Figure 26: Another scenario for unit testing

Figure 27: Another scenario for unit testing

Figure 28: Another scenario for unit testing

FICHE D'APPRECIATION DE STAGE

A renseigner et à viser par le tuteur entreprise puis faire retour sous aurion rubrique « mise à jour de PFE »

Organisme	Rigi Technologies SA
Dates du stage	01.03.2023 – 31.08.2023
NOM, Prénom du stagiaire	WANCTIN Antoine

	Cocher les cases appropriées					
	F (échec)	E (insuffisant)	D (passable)	C (assez bien à bien)	B (bien à très bien)	A (remarquable)
Critères d'intégration – Savoir être						
Adaptabilité						x
Disponibilité						x
Culture de l'entreprise						x
Puissance de travail					x	
Qualité d'expression					x	
Conduite du projet						
Identification des tâches						x
Organisation/répartition des tâches dans le temps					x	
Respect des délais des livrables demandés						x
Force de proposition					x	
Éventuellement : travail en équipe						x
Rapport de stage						
Forme (présentation, style…)						x
Fond (exactitude)						x
Exploitabilité par l'organisme						x

Appréciation de la formation ENSTA Bretagne						
Les compétences scientifiques et techniques répondent à mes attendus						x
Les compétences méthodologiques répondent à mes attendus					x	
Sur quels sujets a-t-il fallu former le stagiaire avant qu'il ne soit autonome ?	Gestion du temps de travail, certaines specificités de git					
Quelles seraient les compétences ou les contenus de formation à renforcer ?	Gestion personnelle de taches					

Appréciation générale

Antoine a été un très bon stagiaire et a su s'adapter aux différentes tâches que nous lui avons assignées.
Il est devenu notre expert résident sur une partie du code de l'entreprise (simulateur) et s'est intégré au sein de l'entreprise. Les deux grands thèmes confiés à Antoine étaient complexes et variés, mais Antoine les a brillamment réalisés.
Bien que son travail et son intégration dans l'équipe soient tous deux adéquats, nous n'avons actuellement pas d'ouverture de poste au sein de l'entreprise pour son profil, mais nous le recommandons à quiconque voudrait l'embaucher. Nous lui souhaitons beaucoup de succès dans ses futurs projets.

Si vous disposiez d'un poste correspondant au profil du stagiaire, souhaiteriez-vous lui proposer ? ✗ NON

NOM, Prénom du tuteur entreprise : DELAFONTAINE Victor **Date : 17.08.2023**

Fonction : Ingenieur software et communication **Signature :**