GE5
2023

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES STRASBOURG

Mid-term Project Report
Mini Drone Crazyflie

Authors:

Arthur BOUILLÉ (arthur.bouille@insa-strasbourg.fr)
Augustin BONNEL (augustin.bonnel@insa-strasbourg.fr)
Leandro PEIXOTO (leandro.peixoto_mattos@insa-strasbourg.fr)

Supervisors:

Renaud KIEFER (renaud.kiefer@insa-strasbourg.fr)
Sylvain DURAND (sylvain.durand@insa-strasbourg.fr)

Contents

1 Introduction **5**

2 Specifications **6**
2.1 Missions . . . 6
2.2 Project components . . . 6
2.3 Work plan . . . 10
2.4 Mindmaps . . . 11

3 Project advancement: Hardware and software integration, and vision development **13**
3.1 Hardware and Software integration . . . 13
3.1.1 Test of the TurtleBot3 Waffle . . . 13
3.1.2 Test of the Crazyflie . . . 13
3.1.3 Simultaneous tests of the CrazyFlie and TurtleBot . . . 14
3.1.4 Motion Capture Room setup optimization . . . 16
3.1.5 CrazyRadio firmware installation . . . 17
3.1.6 Simultaneous tests of the CrazyFlie and TurtleBot with the modified setup . . . 17
3.2 Vision development . . . 17
3.2.1 PiCamera Setup . . . 18
3.2.2 Video Streamer . . . 18
3.2.3 Aruco markers . . . 18

4 Conclusion **22**

5 Bibliography **23**

6 Appendices **24**
6.1 Appendice 1 : "hover_swarm.launch" file . . . 24
6.2 Appendice 2 : Code file for first simultaneous test . . . 26
6.3 Appendice 3 : Code file for second simultaneous test . . . 27
6.4 Appendice 4 : Code file for third simultaneous test . . . 28
6.5 Appendice 5 : "Tutorial: Flashing the crazyradio 2.0 firmware" . . . 30
6.5.1 Building the Crazyradio2.0 Firmware . . . 30
6.5.2 Flashing the Crazyradio2.0 firmware . . . 30

List of Figures

1 Mini drone Crazyflie 2.1 (source: bitcraze.io) . . . 7
2 Crazyradio 2.0 (source: bitcraze.io) . . . 7
3 Crazyflie's Motion Capture deck (source: photo by Léandro PEIXOTO MATTOS) . . . 7
4 Motion Capture System calibration wand (source: photo by Léandro PEIXOTO MATTOS) 8
5 Raspberry PI 4 (source: rapsberrypi.com) . . . 9
6 Horned beasts diagrams . . . 11
7 Octopus diagrams . . . 11
8 Linux computer (source: BOUILLE Arthur) . . . 14
9 First simultaneous test (source: BOUILLE Arthur) . . . 15
10 Second simultaneous test (source: BOUILLE Arthur) . . . 15
11 Linux computer (source: BOUILLE Arthur) . . . 16
12 Latency results of running Crazyswarm on Linux PC (left) and on Virtual Machine (right) 17
13 Aruco marker (source: Aruco marker generator) . . . 19
14 PnP algorithm (source: OpenCV SolvePnP) . . . 20
15 Result of the pose estimation of an Aruco marker (source: Screenshot of a video by Augustin BONNEL) . . . 21

List of Tables

1 Summary table of Crazyflie mini drone specifications 6
2 Summary table of Turtlebot3 Waffle specifications 8
3 Summary table of used commands from Pyton API 9
4 Summary table of the Primary and Constraint Function of the project 12

1 Introduction

The rise of drone technology has opened exciting new prospects in a variety of fields, from aerial surveillance to parcel delivery. Mini-UAVs have attracted considerable interest due to their agility and ability to carry out specific missions efficiently. In this context, our project focuses on exploiting the flight capabilities of a CrazyFlie mini-UAV to perform a complex task: autonomous landing on a mobile platform, in this case, a TurtleBot Waffle.

The main objective of this project is to use computer vision and a specially mounted camera module on the mini drone to achieve precise detection and tracking of the mobile platform. The whole process is to be automated, enabling the mini drone to orientate itself, navigate and make a safe landing on the moving platform.

This report presents a comprehensive overview of our project, covering technical aspects, challenges encountered, problem-solving methods and results obtained. We will also explore the potential applications of this technology, ranging from robotics research to automated logistics and intelligent surveillance.

Beyond the technical feat of this autonomous landing, our project also embraces the exploration of another exciting field: autonomous navigation. Time permitting, we aspire to harness the drone's vision capability to guide the TurtleBot Waffle through a complex maze, demonstrating the versatility of our approach and the possibilities it offers.

In summary, this project highlights how the integration of computer vision and the flight capabilities of a mini drone can lead to an impressive achievement: precise, autonomous landing on a mobile platform. This represents a significant step forward in the field of robotics and automation and offers promising opportunities for future applications. This report is the story of our adventure to discover these exciting horizons.

2 Specifications

In this section, we will present the specifications for our project. It will be composed of the missions, the imposed equipment and the work schedule.
This section will be accompanied by various diagrams for better understanding.

2.1 Missions

To ensure the success of our project, we have divided it into three main phases. Each of these phases is designed to build upon the previous one, facilitating a smooth progression towards our final goal. Here are the details of the different phases of the project:

- *Landing a CrazyFlie drone on a Turtlebot3 Waffle using motion capture:* The first phase focuses on the precise realization of landing a CrazyFlie drone on a Turtlebot3 Waffle in a controlled motion capture environment. This essential step forms the basis for subsequent phases, setting a reliable benchmark for further advances.
- *Landing a CrazyFlie drone on a Turtlebot3 Waffle using vision:* Capitalizing on the successes of the first phase, this step involves the deployment of advanced vision-based techniques for the successful landing of a CrazyFlie drone on a Turtlebot3 Waffle. Using the capabilities of the CrazyFlie AI deck, we aim to improve the accuracy and efficiency of the landing process by integrating vision-based navigation strategies.
- *Guiding a Turtlebot3 out of a maze using CrazyFlie vision:* The final phase marks the culmination of our project, focusing on the integration of complex vision-based algorithms to guide a Turtlebot3 through the intricate pathways of a maze. By exploiting the combined power of CrazyFlie's vision and intelligent navigation systems, we aim to demonstrate our solution's ability to maneuver the Turtlebot3 efficiently and autonomously through challenging environments. This goal testifies to the practical applicability and robustness of our proposed approach.

2.2 Project components

For our project, several pieces of equipment were required. Here is a short description of some of which:

- **Minidrone Crazyflie 2.1:** Crazyflie 2.1 is a minidrone and an open source flying development platform that fits in the palm of the hand. It is produced by Bitcraze and, as an open-source project, source code and hardware schematics are both documented and available. It has a variety of features, some of which are highlighted in the table below:

Weight	27 g
Size (L x W x h)	92x92x29mm
Main microcontroler	STM32F405
Radio and power microcontroler	nRf51822
Sensors	3 axis accelerometer/gyroscope and high precision pressure sensor
Flight time with stock battery	7 minutes
Radio Band	2.4GHz ISM
Bluetooth	BLE support for Android et iOs

Table 1: Summary table of Crazyflie mini drone specifications

The basic version of this drone can be seen on the image below :

Figure 1: Mini drone Crazyflie 2.1 (source: bitcraze.io)

In addition to the default features of Crazyflie, a variety of expansion decks can be attached, both on top and bottom of the drone so extra functionalities can be added. To communicate with it, the Bluetooth can be used with the APP Crazyflie Client available on Android and iOS. However, to send commands from a computer, a Crazyradio is necessary. It's a long-range open USB radio dongle based on the nRF52840 that allows the communication between a computer and the drone. An image of this component can be seen below:

Figure 2: Crazyradio 2.0 (source: bitcraze.io)

- **Motion capture marker deck:**

  The goal of this deck is to facilitate the attachment of infrared reflective markers to the drone. These markers are used for tracking in motion capture systems, as explained in the dedicated topic for this component. This deck contains 35 holes; therefore, different marker configurations can be mounted, and it weighs only 1.6g. A photo of the drone used in the project with this deck and the reflective markers can be seen below:

Figure 3: Crazyflie's Motion Capture deck (source: photo by Léandro PEIXOTO MATTOS)

- **Motion capture system:**

  The motion capture system is used to know the position of objects in the L114 room of INSA. To do so, the room is equipped with four Optitrack PrimeX 13 infrared cameras, one ethernet switch and the Motive software installed on the computer of the room. The cameras are linked and powered by ethernet cables. To be seen, an object has to be equipped with at least three asymmetrically positioned infrared markers, reflecting the infrared light issued by the cameras.
  Besides, the motion capture system needs to be calibrated every 3 to 4 weeks. The first step of calibration is the calibration of the workspace: we use a "Calibration wand" (basically it is a T wand with three infrared markers) and we move this wand all around the workspace, so the software

knows nearly every point of the workspace. The second step of the calibration is the definition of the ground with the "Calibration square", the position of the square defines the origin and X Y Z plans

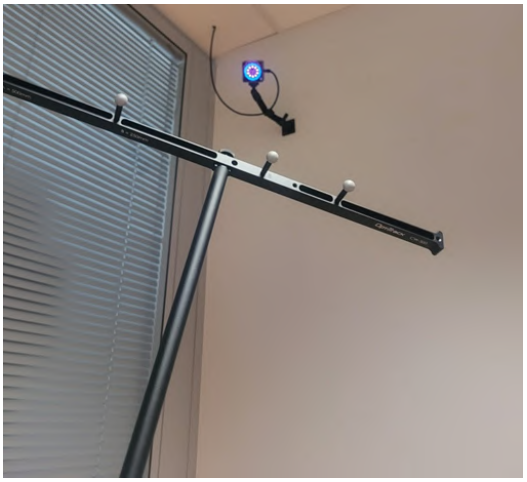

Figure 4: Motion Capture System calibration wand (source: photo by Léandro PEIXOTO MATTOS)

Furthermore, we must define "Rigid objects" in the Motive software, so a group of infrared markers are defined as a single solid object (and we will be able to know it position and orientation). It is possible to change the name and the gravitational center of a defined "Rigid object".

- **TurtleBot Waffle:**

  TurtleBots are standard ROS platform robots for education, research, hobbies and product prototyping. According to Robotis, the aim of the TurtleBot3 is to considerably reduce the size and price of the platform without having to sacrifice functionality and quality, while offering expansion possibilities. For our project, we'd like to use a TurtleBot3 Waffle as a landing pad for a CrazyFlie Mini Drone. Below is an image of a Turtlebot3 Waffle with its infrared markers.

  Here are its technical specifications:

Weight	1.8 kg
Size (L x W x h)	281x306x141mm
Single Board Computer	Raspberry Pi 3
Maximum payload	30 kg
Maximum rotationnal velocity	1.82 rad/s
Operating time with stock battery	2 hours

Table 2: Summary table of Turtlebot3 Waffle specifications

- **Raspberry Pi:**

  Raspberry Pi are affordable, versatile single board computers created by the Raspberry Pi Foundation. They are designed for learning programming and automating electronic projects.

  Raspberry Pi computers feature an ARM processor, USB, HDMI and GPIO ports for connectivity and interaction with other peripherals. They generally run under a Linux operating system, such as Raspbian. These mini computers are used in a variety of applications, including home automation, robotics, servers, education and DIY electronics. Below is an image of a Raspberry Pi 4.

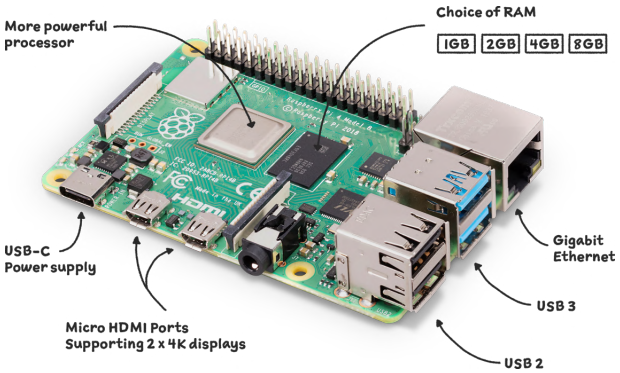

Figure 5: Raspberry PI 4 (source: rapsberrypi.com)

- **ROS and Crazyswarm:**

  ROS, or Robot Operating System, is a set of open-source libraries, tools and development standards designed to simplify the creation, management and control of robots. Unlike a traditional operating system, ROS is not an operating system itself, but rather a robotic middleware that runs on Linux operating systems. ROS offers a flexible framework for robot programming, providing functionalities such as hardware device management, communication between software components (nodes), data visualization and the resolution of common robotics problems, such as mapping and localization. It is widely used in robotics research, industry and education to develop complex, modular robotics applications. ROS facilitates collaboration and code sharing between the global community of developers, making it a valuable tool for the advancement of robotics.

  CrazySwarm is an open-source framework designed specifically for robotics research and the management of swarm drone systems. It enables researchers and developers to create, control and experiment with groups of drones in a coordinated way. CrazySwarm's key features include precise synchronization of drones for simultaneous movement, real-time localization and navigation management, and tools for trajectory planning and simulation. It also offers libraries and interfaces for communication with drones, as well as visualization tools to facilitate development and debugging. CrazySwarm is mainly used in the field of robotics research to explore application scenarios such as surveillance, search and rescue, mapping and more, using swarms of autonomous drones. It fosters collaboration within the robotics research community for the advancement of drone swarm technologies.

  CrazySwarm has a built-in Python API which allows us to code with this language easily. There is a variety of commands

Command	Level	Topic ou service
takeoff()	High	Service - /cf1/takeoff
land()	High	Service - /cf1/land
goto()	High	Service - /cf1/go_to
position()	-	Topic - /cf1/tf
cmdPosition()	Low	Topic - /cf1/cmd_position
notifySetpointsStop()*	-	Service - /cf1/notify_setpoints_stop

Table 3: Summary table of used commands from Pyton API

In a nutshell, commands are divided into two types, high-level, when they refer to a ROS Service, and low-level, when they refer to a ROS Topic. It's always possible to go from high-level commands to low-level commands, however, the contrary, when using commands that control the drone, can only be done after the command "notifySetpointsStop()" is used.

  - **takeoff()**: Execute a takeoff - fly straight up, then hover indefinitely. The parameters are the height and the time of how long until the height is reached.

  - **land()**: Execute a landing - fly straight down. The parameters are the height at which to land and the time of how long until the height is reached.
  - **goto()**: This command has a point in the space as entry. A trajectory from the current point to the desired point is planned and the drone follows this trajectory after. However, this command isn't suited to applications where the streaming of points is quick (<<1 sec). The desired yaw is also a entry parameter.
  - **position()**: This command returns the last Crazyflie position data sent by the motion capture system.
  - **cmdPosition()**: This command have a point in the space as one of the parameters and the desired yaw. The calculation of the actuation on engines is done by the drone's microcontroler. This command have to be send with a minimal frequency of 10 Hz.
  - **notifySetpointsStop()**: This command allows to execute high-level commands after low-level commands are executed. Before, it was not possible due to firmware limitations. The entry parameter is the number of milliseconds that the last streaming setpoint should be followed before reverting to the onboard-determined behavior.

  There are also useful commands to help the temporizing of the system.

- **AI Deck:** The "AI Deck" is a hardware extension for Crazyflie 2.X that offers artificial intelligence (AI) and image processing functionalities. Here are the main features of AI Deck 1.1:

  - GAP8 (ultra-low-power RISC-V MCU): AI Deck is powered by a GAP8 microcontroller (MCU) with 8+1 RISC-V cores. It offers exceptional computing power while maintaining low power consumption, making it suitable for AI and vision tasks.
  - Ultra-low-power monochrome camera: The deck features a 320x320 pixel monochrome camera with low power consumption, ideal for vision and image processing applications.
  - HyperFlash and HyperRAM memory: It features 512 Mbit of HyperFlash memory and 64 Mbit of HyperRAM memory, offering ample space for storing data and AI programs.
  - Interfaces and connectors: It features several ports and connectors for easy expansion and integration, including JTAG pins, buttons, LEDs, UARTs and SPI interfaces.
  - Interfaces and connectors: It features several ports and connectors for easy expansion and integration, including JTAG pins, buttons, LEDs, UARTs and SPI interfaces.
  - Power supply: It works with a 3V to 5V power supply, with a maximum consumption of 300 mA. It can be powered directly from Crazyflie 2.X.
  - Automatic add-on board detection: The deck is equipped with 1-wire memory to automatically detect add-on boards, simplifying configuration.
  - Mechanical compatibility: The deck is designed to be mounted above or below the Crazyflie 2.X, ensuring easy mechanical integration.

  In summary, the AI Deck 1.1 for Crazyflie is a powerful add-on that brings AI and image processing capabilities to the Crazyflie drone, enabling a variety of advanced applications in robotics, computer vision and artificial intelligence.

2.3 Work plan

In this section, we present the work schedule for this project.

- September →Mid-october :

  - Getting to grips with the CrazyFlie drone and turtlebot, and configuring the ROS master and ROS turtlebot.
  - Installation of Crazyswarm on the RPi 4, so that we now have just one ROS Master in our setup.
  - Testing the 320 by 320 pixel resolution of the AI deck using a Pi camera. Tested recognition of a qrcode or black-and-white image for landing.

- Mid-october →November :
  - AI deck control
  - AI deck flight test
  - Implementation of the landing qrCode recognition algorithm.

2.4 Mindmaps

Horned beast diagrams: This diagram gives us a wealth of information about our project, such as "Who does it serve?", "What does it do?" and "What is its purpose?

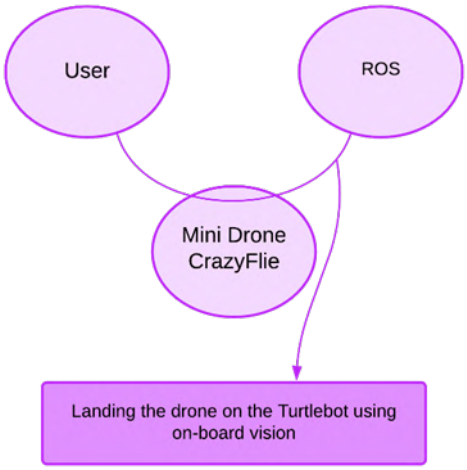

Figure 6: Horned beasts diagrams

Octopus diagram: This diagram shows the links between the various functions in our project, as well as the type of each function ("primary function" and "constraint function").

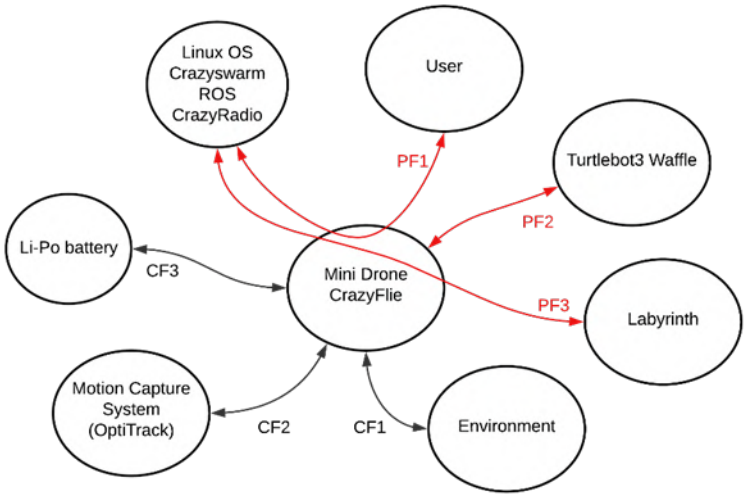

Figure 7: Octopus diagrams

Primary Function	
PF1	Be usable by the user
PF2	Be able to land on a TurtleBot3 Waffle
PF3	Be able to solve a maze and guide the Turtlebot out

Constraint Function	
CF1	Resist falls landings and handling
CF2	Lift infrared markers
CF3	Ensuring sufficient battery autonomy

Table 4: Summary table of the Primary and Constraint Function of the project

3 Project advancement: Hardware and software integration, and vision development

This section represents a key step in our project, highlighting the significant progress made in the integration of hardware and software, as well as the development of the vision function for piloting the CrazyFlie drone and TurtleBot3 Waffle. Our aim is to provide a detailed overview of the processes and technologies involved in these crucial aspects of our project.

We'll start by taking a detailed look at the hardware components used, including the technical specifications of both the CrazyFlie drone and the TurtleBot3 Waffle, as well as the modifications made to ensure seamless integration between the two operating systems. We'll also look at the challenges involved in interfacing and communicating between these two systems, and the solutions adopted to ensure smooth, synchronized piloting.

On the software side, we'll highlight the frameworks and software tools used to program and control the CrazyFlie and TurtleBot3 Waffle. We will discuss the control algorithms and navigation strategies implemented to ensure effective coordination between the two robots, while guaranteeing safety and precision of movement.

In addition, we will present in detail the development of the vision module for the CrazyFlie drone. We will explain the image processing techniques and computer vision algorithms used for the detection, tracking and recognition of relevant elements of the environment, such as the TurtleBot3 Waffle mobile platform and the maze's key characteristics.

In summary, this section will highlight the substantial progress made in hardware and software integration, as well as the development of the vision function, which are essential for the successful accomplishment of our project. We will highlight the importance of these advances in ensuring the consistency, accuracy and reliability of robot operations in complex and dynamic scenarios.

3.1 Hardware and Software integration

3.1.1 Test of the TurtleBot3 Waffle

During the initial tests to trigger the TurtleBot's movement, the Motion Capture room setup had not yet been modified, which meant using a Raspberry Pi as the ROS master to control the TurtleBot. The initial process involved several steps to configure and establish effective communication between the various system components.

The first step was to carefully configure the Raspberry Pi's WiFi networks to ensure that they were connected to the same WiFi network. Next, we had to tell the TurtleBot's Raspberry Pi the IP address of the ROS master to establish a solid, stable connection between the two.

To initiate the tests, we launched the ROS master on the Raspberry Pi, followed by the execution of the bringup command on the TurtleBot. This sequence of actions enabled us to verify that the TurtleBot's topics were correctly active and available from the ROS master, confirming the successful implementation of communication between the two components.

After confirming the availability of the topics and communication between the ROS master and the TurtleBot, we decided to post a message on the TurtleBot's /cmd_vel topic to command it to move in a circle. The following command was successfully sent:

The TurtleBot reacted promptly to the message and did indeed move in a circle, confirming the success of the initial motion tests before the Motion Capture room setup was modified.

3.1.2 Test of the Crazyflie

During the initial tests to fly the CrazyFlie, the Motion Capture room setup had not yet been modified, which meant using an Ubuntu virtual machine on a Windows computer as the ROS master to control the CrazyFlie. In addition, the motion capture system ran on the Windows part of this computer, since OptiTrack's Motive software is Windows-specific. The test procedure involved the application of knowledge acquired by Leandro during his internship at INSA Strasbourg, in particular the use of specific Python code to enable the CrazyFlie to take off and land in the same place.

To perform these tests, we first entered the following command in a terminal on the ROS master (the virtual machine):

Then, in a second terminal, we used the "hello world" Python code to trigger the drone's takeoff and landing. The command executed was as follows:

These steps verified the effectiveness of the Python code and ensured that the CrazyFlie could indeed take off and land in the same place, constituting the first successful flight tests before the Motion Capture room setup was modified.

3.1.3 Simultaneous tests of the CrazyFlie and TurtleBot

After the separated tests with the TurtleBot and the Crazyflie, the next step was to perform tests with them simultaneously. In order to carry out theses experiments, the infrared retroflective markers were added to the TurtleBot structure so the motion capture system could calculate the position of the terrestrial robot. Following the labeling, the "Rigid Body" of the TurtleBot was created in Motive software and it was named "TURTLEBOT2". The other setup that had to be made was inside the CrazySwarm structure, a ROS topic that publishes the position of the TurtleBot had to be created. The following code section was added to the file "hover_swarm.launch"

```
<node pkg="vrpn_client_ros" type="vrpn_client_node" name="vrpn_client_node" output="screen" >
  <rosparam subst_value="true">
    server: $(arg server)
    port: 3883
    frame_id: world
    use_server_time: false
    broadcast_tf: false
    refresh_tracker_frequency: 0.0
    trackers:
    - TURTLEBOT2
  </rosparam>
</node>
```

Figure 8: Linux computer (source: BOUILLE Arthur)

You have access to the complete "hover_swarm.launch" file in appendix 1.

We would like to remind that executing the launch file sets up all the ROS network, including ROS Master and all topics to command the Crazyflie Drone and read its position. Essentially, two ROS networks were created on these tests, the first one consisted of a RaspiberryPi and the TurtleBot, with the Raspiberry being the ROS Master of the network and where the commands to control the TurtleBot's speed were written. The second network contained the Crazyflie drone and the computer with Motive and Crazyswarm (on a virtual machine). The ROS Master of this network is inside the PCs and drone position data, drone topics control and TurtleBot position data were available on this second network through ROS topics ans ROS services. Finally, we were able to start the tests with both robots.

The first performed test consisted in taking off the drone to an height of 50cm and then reading the TurtleBot position through the topic "". After this, the Crazyflie is commanded to go to the same position (x, y) of the TurtleBot while still at the height of 50cm. When the drone is at the same position, it lands on the TurtleBot. After some seconds landed on the robot, the drone takes off again and returns to its initial taking off position. During this test, we noticed that the drone was not landing on the center of the TurtleBot. This happened because the markers on objects on the motion capture system have to be asymmetrical so the orientation of the body can be correctly calculated. As a consequence, the gravity center automatically calculate by the software was not on the middle of the robot body. However, this was manually adjusted by easily dragging the gravity center of the Rigid Body on Motive Software. After this adjustment, we could see that the drone landed closely to the robot body center.

Following the tests sequence, the second experiment consisted in commanding the speed of the TurtleBot so it would move in circles and the drone following the ground position (x, y) of the robot while still on a 50cm height. Of course, the drone first takes off from a defined position and after following the robot for 20 seconds, it lands on its initial take off position. With this experiment, we reached to some conclusions:

- Initially, the drone was posed in a far away position to the TurtleBot and, when testing with this condition, the drone crashed to ground. Differently from the first test where "goTo()" command was used, the position command used on this second test is "cmdPosition()", because it is more appropriated to quickly calls, and the tracking requests a high frequency position command. However, the calculation is made onboard the drone and the trajectory is not calculated (as opposed to "goTo()", which plans a trajectory), therefore, posing the drone far away may cause a great error signal on the controllers and the signal command can be saturated. The solution for this was simply posing the drone closer to the TurtleBot at the start.

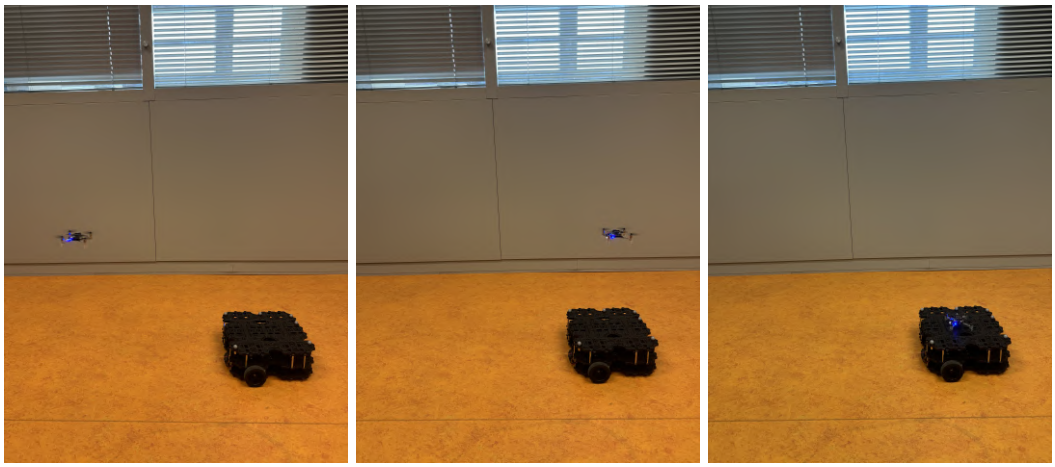

Figure 9: First simultaneous test (source: BOUILLE Arthur)

You have access to the complete code file of this test in appendix 2.

- When following the robot circle, we noticed that the drone was a little bit late in relation to the robot. This is normal because between getting TurtleBot's position, sending it to the Crazyflie and effectively moving the drone to the position, there is a delay time during which the TurtleBot continues to move. Hence, the Crazyflie is always behind the TurtleBot. This result made us change our third test, because the initial idea was to land the drone on the turtlebot while it was moving, but it was no longer possible after this result was attested.

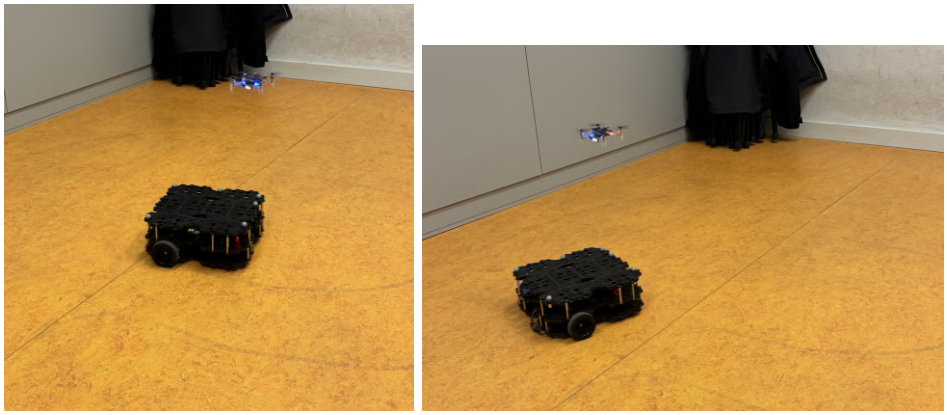

Figure 10: Second simultaneous test (source: BOUILLE Arthur)

You have access to the complete code file of this test in appendix 3.

- The last conclusion becomes from the command "notifySetpointsStop()". As on table 3, this command is used to communicate to the program to stop sending and reading low-level commands and that it can follow high-level commands again. Nevertheless, when this command is used, the

drone stays a little time without receiving commands, and this results that it falls some centimeters before receiving a new command. It does not affect the final result, but it is a phenomenon that can be seen on this test.

As said before, the initial idea of the third test was to land the drone on the moving robot. However, this was changed after the second experiment. Thus, the performed test was to the drone take off and follow the TurtleBot trajectory indefinitely. Then, at some point, the speed command on the TurtleBot is stopped and, once the robot is not moving, the drone lands on the robot. In order to measure when the robot is idle, we have used the euclidean distance between two consecutive points. While the robot moves, the euclidean distance between two position points is about 0.17m and this distance is about 0.0005m when the robot is not moving. Therefore, this was the metrics used to verify if the robot is moving or not. After landing the Crazyflie on the TurtleBot, the script is over. This was the final test performed with both robots as of now. You have access to the complete code file of this test in appendix 4.

3.1.4 Motion Capture Room setup optimization

Adjustments to our initial Motion Capture L114 room setup were essential to reduce latency and improve consistency of robot control. The previous system, split between a Windows computer with a virtual machine to control the CrazyFlie, and a Raspberry Pi to control the TurtleBot3 Waffle, presented limitations in terms of communication and latency. To solve these problems, we made significant modifications to improve the efficiency and synchronization of the system as a whole.

Migration of the CrazyFlie control system to a computer natively equipped with Linux (Ubuntu) eliminated the latency problems associated with using a virtual machine. In addition, by also transferring control of the TurtleBot3 Waffle to the Linux computer, we were able to unify control of both systems, allowing consistent, synchronized management in a single software environment.

Figure 11: Linux computer (source: BOUILLE Arthur)

Despite these changes, we have maintained the Motion Capture system on the Windows computer, as the Motion Capture software is only compatible with this operating system. This decision was taken to ensure the continued operation of the Motion Capture system without introducing new compatibility or latency problems.

Thanks to these adjustments, we have considerably reduced communication and latency problems, ensuring smoother and more efficient coordination between the various system components. These improvements optimized overall system performance and contributed significantly to the successful progress of our project. It's possible to see at the images below that the message "Latency too high!" is not show

by the system while running Crazyswarm with the Linux PC. We may see too that the latency of the Motive is more than 10 times lower at Linux PC:

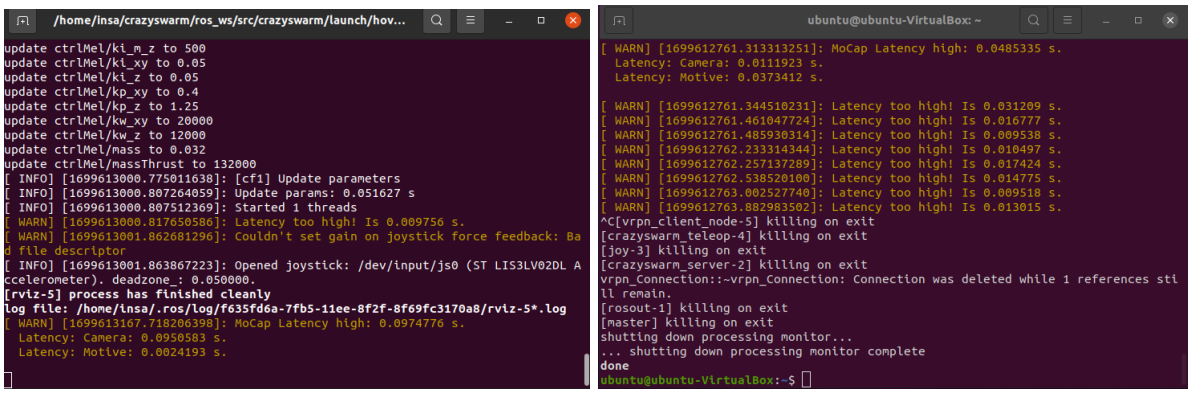

Figure 12: Latency results of running Crazyswarm on Linux PC (left) and on Virtual Machine (right)

3.1.5 CrazyRadio firmware installation

Setting up the firmware for CrazyRadio on the new Linux computer was a complex task, full of unexpected challenges. One of the main obstacles we encountered was the fact that we had two different versions of CrazyRadios: a CrazyRadio PA (version 1.0) and a CrazyRadio 2.0 (version 2.0). When we made the transition to the new Linux computer, we intended to use CrazyRadio 2.0 to ensure compatibility with the system, but we soon discovered that each version of CrazyRadio required its own firmware to function correctly.

During our first attempts to set up CrazyRadio 2.0 on the Linux computer, we were confronted with a major problem: firmware incompatibility between CrazyRadio versions. The firmware of CrazyRadio PA (version 1.0) was not compatible with CrazyRadio 2.0 (version 2.0), making it impossible to use this latter. To solve this problem, we undertook extensive research, which eventually led us to the official Bitcraze documentation. A tutorial was available, describing how to download and flash the CrazyRadio 2.0 firmware. However, we quickly noticed that this guide was not accessible to everyone, as some information was not explicitly provided. By carefully interpreting these instructions, we were able to successfully install the correct firmware for CrazyRadio 2.0, making it compatible with our new Linux computer. The next step was to flash the firmware to the CrazyRadio.

After correctly flashing the firmware on CrazyRadio 2.0, we were able to resolve the incompatibility obstacle, enabling us to use the latter effectively on the Linux computer. This crucial step was a prerequisite for ensuring optimal communication with CrazyFlie from our new control environment. Finally, to facilitate future operations, we have written a detailed tutorial in the appendix 5 for installing firmware 2.0 on the CrazyRadio 2.0, available to our classmates and teachers to facilitate firmware flashing on CrazyRadio 2.0s.

3.1.6 Simultaneous tests of the CrazyFlie and TurtleBot with the modified setup

This part consists of the same tests we carried out before the modifications to the Motion Capture room setup. These were a landing and take-off test from the turtlebot and a test of precise tracking of the turtlebot by the CrazyFlie. Note that these tests were also successful with our new setup.

3.2 Vision development

In this section, we will discuss the vision part of the project. As describe above, the drone will be equipped with a camera deck (the AI deck) and will have to land on the Turtlebot using a recognition algortihm. However, as the deck is very expensive, we have decided to perform some tests before buying it. Those tests are made with a Raspberry PI and a PiCamera2. The following sections present the procedure and results of these tests.

3.2.1 PiCamera Setup

We have chose to use a Raspberry Pi to perform tests, because it is easy to mimic the behaviour and characteristics of the AI deck camera. Moreover, these items are readily available and well documented.

Items used during tests:

- Raspberry Pi 4 Model B with 2 GB RAM
- PiCamera V2.1
- Python 3
- OpenCV 4.8

First we plugged the PiCamera into the Raspberry dedicated connector. Then after an update and a reboot of the Raspberry PI, we can test the camera wiring with the command :

```
$ vcgencmd get_camera
```

Then we can test the Camera using this command:

```
$ libcamera-hello
```

This command should open a preview of a video stream. If it does, then the camera is up and ready. For further operations we are going to use the python API of the PiCamera to write Python codes. These codes are also going to use the OpenCV library and so perform visions algorithms on the camera's video stream.

3.2.2 Video Streamer

The first step was to stream live video from the camera onto a preview. But we also need to tweak the PiCamera configuration to mimic the AI Deck camera behaviour. The video returned by the camera must therefore be in black and white and have a resolution of 320 pixels by 320 pixels.

Here is the code for this first programm :

```
import cv2 #OpenCV importation
from picamera2 import Picamera2

IMG_DIMS = (320,320) #wanted image resolution

picam2 = Picamera2()
config = picam2.create_preview_configuration()
config['main']['size'] = IMG_DIMS #Configuration of the camera following the wanted
                                  resolution
config['main']['format'] = "YUV420"
picam2.configure(config)
picam2.start()

cv2.startWindowThread() #start of a OpenCV preview windows

while True:
    im = picam2.capture_array()
    im_grey = im[:IMG_DIMS[1], :IMG_DIMS[0]] #we only take the black and white
                                               layer of the image

    cv2.imshow("Camera", im_grey)
    cv2.waitKey(1)
```

So we have an image with the same characteristics as the one normally produced by the AI deck. We can move on to the next step : recognition and pose estimation of an Aruco marker.

3.2.3 Aruco markers

This section is divided in two steps, first we want to detect an aruco marker and then to estimate his pose. An Aruco marker looks like this :

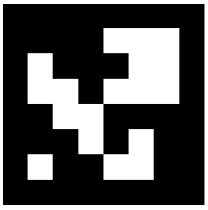

Figure 13: Aruco marker (source: Aruco marker generator)

Aruco markers is composed of white pixels placed on a black background, the position and number of pixels indicated the id of the aruco. They have a defined size (from 4x4 to 7x7).

We want to detect and estimate the position of an aruco marker placed on the robot in order to get rid of the motion capture system on the latter.

This first code is only designed to detect an aruco and draw the id of the aruco on the preview, it's an introduction to markers handling with OpenCV :

```
import cv2
from picamera2 import Picamera2
import cv2.aruco as aruco

IMG_DIMS = (320,320)

#Same camera configutation as besfore
picam2 = Picamera2()
config = picam2.create_preview_configuration()
config['main']['size'] = IMG_DIMS
config['main']['format'] = "YUV420"
picam2.configure(config)
picam2.start()

cv2.startWindowThread()

while True:
    im = picam2.capture_array()
    im_grey = im[:IMG_DIMS[1], :IMG_DIMS[0]]

    #We define the parameters for the type of marker we want to
    aruco_dict = aruco.getPredefinedDictionary(aruco.DICT_6X6_1000)
    arucoParameters = aruco.DetectorParameters()
    detector = aruco.ArucoDetector(aruco_dict, arucoParameters)

    #Use of the build in OpenCV aruco detection
    markerCorners, markerIds, rejectedCandidates = detector.detectMarkers(im_grey)

    #drawing of the aruco features on the preview
    im_grey_markers = aruco.drawDetectedMarkers(im_grey, markerCorners, markerIds)

    cv2.imshow("Camera", im_grey_markers)
    cv2.waitKey(1)
```

Having successfully completed simple detection, we now move on to estimating the position of the marker. To do so we are going to use the PnP (Perspective N Points) algorithm, which gives the rotation and translation matrices between the camera frame and the real-world frame. And as a consequence an estimation of the position of an object regarding to the camera position.

Here is an explanatory diagram :

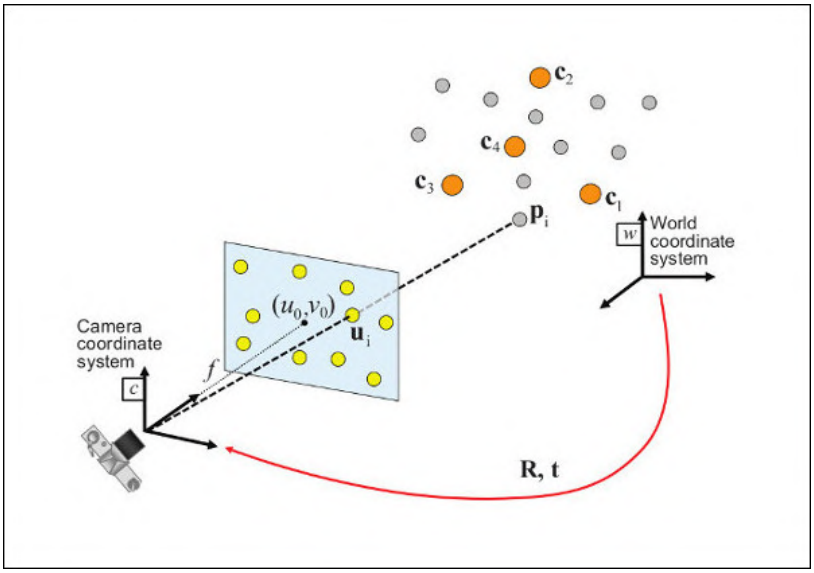

Figure 14: PnP algorithm (source: OpenCV SolvePnP)

Please find below the code of the Aruco marker pose estimation :

```
#We begin with the same configuration as before
import cv2
from picamera2 import Picamera2
import numpy as np

IMG_DIMS = (320,320)

picam2 = Picamera2()
config = picam2.create_preview_configuration()
config['main']['size'] = IMG_DIMS
config['main']['format'] = "YUV420"
picam2.configure(config)
picam2.start()

dict_aruco = cv2.aruco.getPredefinedDictionary(cv2.aruco.DICT_6X6_1000)
parameters = cv2.aruco.DetectorParameters()
detector = cv2.aruco.ArucoDetector(dict_aruco, parameters)

#To perform the PnP algorithm we need the internal camera parameters and the camera
                                                distortion matrix
mtx = np.array([[313.89376405, 0, 158.04145609], [0, 316.64900574, 117.57279469], [
                                                0, 0, 1]])
dist = np.array([[ 0.04853669, -0.1917236, -0.00088241, -0.00245449, -0.21968433]])

cv2.startWindowThread()

while True:

    im = picam2.capture_array()
    im_grey = im[:IMG_DIMS[1], :IMG_DIMS[0]]

    markerCorners, markerIds, rejectedCandidates = detector.detectMarkers(im_grey)

    #If marker are detectec :
    if not markerIds is None:

        #The custom function of the solve PnP algorithm is detailed after
        rvecs, tvecs, trash = my_estimatePoseSingleMarkers(markerCorners, 5.3, mtx,
                                                dist)

        for idx in range(len(markerIds)):

            cv2.drawFrameAxes(im_grey,mtx,dist,rvecs[idx],tvecs[idx],5)
            print('marker id:%d, pos_x = %f,pos_y = %f, pos_z = %f' % (markerIds[
```

```
                                                    idx],tvecs[idx][0],tvecs[
                                                    idx][1],tvecs[idx][2]))

        cv2.aruco.drawDetectedMarkers(im_grey, markerCorners, markerIds)

        cv2.imshow("Camera", im_grey)
        cv2.waitKey(1)
```

As written in the code, we need the camera's intrinsic parameters and its distortion matrix. Intrinsic matrix parameters are composed of precise camera characteristics, such as focal length, and can be found in the camera documentation. But the distortion matrix is not given, so it has to be estimated with a calibration tool. This tool is a Python code downloadable on GitHub.

The estimation pose function is detailed below :

```
    def my_estimatePoseSingleMarkers(corners, marker_size, mtx, distortion):

        marker_points = np.array([[-marker_size / 2, marker_size / 2, 0],
                                  [marker_size / 2, marker_size / 2, 0],
                                  [marker_size / 2, -marker_size / 2, 0],
                                  [-marker_size / 2, -marker_size / 2, 0]], dtype=np.
                                                                        float32)
        trash = []
        rvecs = []
        tvecs = []

        for c in corners:
            nada, R, t = cv2.solvePnP(marker_points, c, mtx, distortion, False, cv2.
                                                          SOLVEPNP_IPPE_SQUARE)
            rvecs.append(R)
            tvecs.append(t)
            trash.append(nada)

        return rvecs, tvecs, trash
```

As you can see on the image below, this algorithm is working, but it would certainly benefit from improvements, such as better marker tracking, to enhance its robustness :

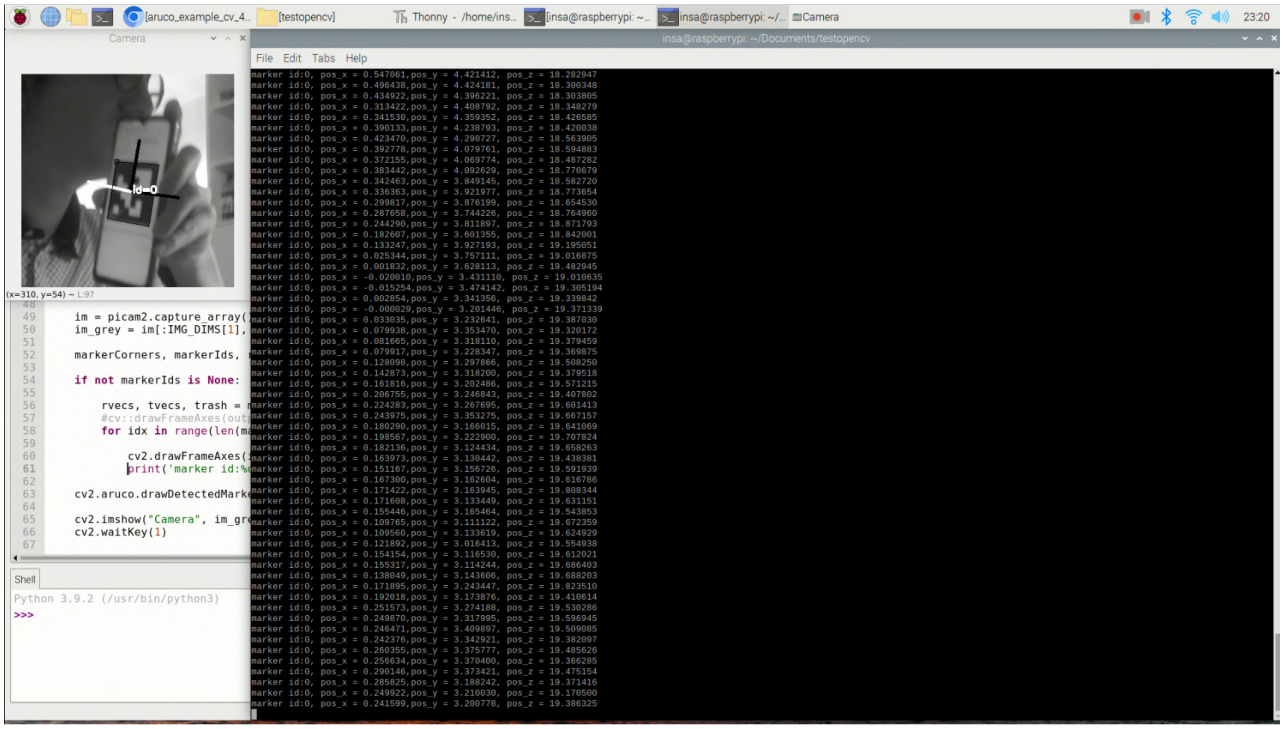

Figure 15: Result of the pose estimation of an Aruco marker (source: Screenshot of a video by Augustin BONNEL)

4 Conclusion

On this mid term report, we were able to show all the progress that has been made from the beginning of the project until the delivery date of this document. Despite a lot of technical difficulties that we faced involving the WiFi password, the setup and installation of TurtleBot libraries on the RaspiberryPi's, we reached to the expected results according to the schedule and our comprehension about Crazyswarm, ROS and vision algorithms is substantially greater if compared to the beginning of the assignment.

It is important to remark that all steps of setup, installation and configuration are being documented in different tutorials so it will be more simple to reach to the same point we are and all the experiments easily can be replicated.

For the rest of the project, our focus will be on learning and making use of the camera from the AI Deck to detect the TurtleBot and landing on it. One time this task is accomplished, we will proceed to the maze resolution problem using the drone camera to guide the robot out of a maze. With all work that has been done on vision using the PiCamera, we are confident on the continuation of the project.

5 Bibliography

- Tutoriel ROS Noetic, Guillaume Hansen et Alexandre Thouvenin, 2023
- How to Control a Real TurtleBot with ROS through a Remote Raspberry Pi as ROS Master and with an OptiTrack Motion Capture System, Sylvain Durand, 2023
- Robot Operating System, Olivier Stasse
- Official documentation for Crazyradio 2.0 firmware
- Python API for the PiCamera2
- Official OpenCV python documentation
- Wikipedia of the PnP algorithm

6 Appendices

6.1 Appendice 1 : "hover_swarm.launch" file

In this appendix, you will find the code for the "hover_swarm.launch" file.

```
<?xml version="1.0"?>
<launch>
  <arg name="joy_dev" default="/dev/input/js0" />

  <rosparam command="load" file="$(find crazyswarm)/launch/crazyflieTypes.yaml" />
  <rosparam command="load" file="$(find crazyswarm)/launch/crazyflies.yaml" />

  <node pkg="crazyswarm" type="crazyswarm_server" name="crazyswarm_server" output="
                                                  screen" >
    <rosparam>
      # Logging configuration (Use enable_logging to actually enable logging)
      genericLogTopics: ["log1"]
      genericLogTopicFrequencies: [10]
      genericLogTopic_log1_Variables: ["stateEstimate.x", "ctrltarget.x"]
      # firmware parameters for all drones (use crazyflieTypes.yaml to set per type
                                                    , or
      # allCrazyflies.yaml to set per drone)
      firmwareParams:
        commander:
          enHighLevel: 1
        stabilizer:
          estimator: 2 # 1: complementary, 2: kalman
          controller: 2 # 1: PID, 2: mellinger
        ring:
          effect: 16 # 6: double spinner, 7: solid color, 16: packetRate
          solidBlue: 255 # if set to solid color
          solidGreen: 0 # if set to solid color
          solidRed: 0 # if set to solid color
          headlightEnable: 0
        locSrv:
          extPosStdDev: 1e-3
          extQuatStdDev: 0.5e-1
        kalman:
          resetEstimation: 1
      # tracking
      motion_capture_type: "optitrack" # one of none,vicon,optitrack,
                                                  optitrack_closed_source,qualisys
                                                  ,vrpn
      object_tracking_type: "motionCapture" # one of motionCapture,libobjecttracker
      send_position_only: False # set to False to send position+orientation; set to
                                                   True to send position only
      motion_capture_host_name: "192.168.0.253"
      # motion_capture_interface_ip: "" # optional for optitrack with multiple
                                                  interfaces
      save_point_clouds: "/dev/null" # set to a valid path to log mocap point cloud
                                                   binary file.
      print_latency: False
      write_csvs: False
      force_no_cache: False
      enable_parameters: True
      enable_logging: False
      enable_logging_pose: True
    </rosparam>
  </node>

  <node name="joy" pkg="joy" type="joy_node" output="screen">
    <param name="dev" value="$(arg joy_dev)" />
  </node>

  <node pkg="crazyswarm" type="crazyswarm_teleop" name="crazyswarm_teleop" output="
                                                  screen">
    <param name="csv_file" value="$(find crazyswarm)/launch/figure8_smooth.csv" />
    <param name="timescale" value="0.8" />
  </node>

  <!--<node name="rviz" pkg="rviz" type="rviz" args="-d $(find crazyswarm)/launch/
```

```
                                                test.rviz"/>

    <node pkg="rqt_plot" type="rqt_plot" name="rqt_plot_x" args="/cf2/log1/values[0]"
                                                />
    <node pkg="rqt_plot" type="rqt_plot" name="rqt_plot_roll" args="/cf1/log1/values[
                                                2] /cf1/log1/values[3]"/>-->

    <arg name="server" default="192.168.0.253"/>

    <node pkg="vrpn_client_ros" type="vrpn_client_node" name="vrpn_client_node"
                                                output="screen" >
      <rosparam subst_value="true">
        server: $(arg server)
        port: 3883
        frame_id: world
        use_server_time: false
        broadcast_tf: false
        refresh_tracker_frequency: 0.0
        trackers:
        - TURTLEBOT2
      </rosparam>
    </node>

  </launch>
```

6.2 Appendice 2 : Code file for first simultaneous test

In this appendix, you will find the code file for the first simultaneous test.

```
from pycrazyswarm import Crazyswarm
import rospy
from geometry_msgs.msg import PoseStamped
import numpy as np

TAKEOFF_DURATION = 2.5
HOVER_DURATION = 2.0

turtle_pose = PoseStamped()

def turtle_pose_callback(msg):
    global turtle_pose
    turtle_pose = msg

def main():
    swarm = Crazyswarm()
    timeHelper = swarm.timeHelper
    cf = swarm.allcfs.crazyflies[0]

    inital_time = timeHelper.time()

    sub = rospy.Subscriber('/vrpn_client_node/TURTLEBOT2/pose', PoseStamped,
                                                callback = turtle_pose_callback)

    while(timeHelper.time()-inital_time < 0.2):
        print('x= ', turtle_pose.pose.position.x, 'y= ', turtle_pose.pose.position.
                                                    y)
        timeHelper.sleepForRate(10)

    pos_x = turtle_pose.pose.position.x
    pos_y = turtle_pose.pose.position.y

    print('x= ', pos_x, 'y= ', pos_y)

    cf.takeoff(targetHeight=0.5, duration=TAKEOFF_DURATION)
    timeHelper.sleep(TAKEOFF_DURATION+HOVER_DURATION)

    position_initial = cf.position()
    goal = np.array([pos_x, pos_y, position_initial[2]])
    cf.goTo(goal, yaw = 0.0, duration = TAKEOFF_DURATION)
    timeHelper.sleep(TAKEOFF_DURATION + HOVER_DURATION)

    cf.land(targetHeight=0.16, duration=TAKEOFF_DURATION)
    timeHelper.sleep(TAKEOFF_DURATION+5)

    cf.takeoff(targetHeight=0.5, duration=TAKEOFF_DURATION)
    timeHelper.sleep(TAKEOFF_DURATION+HOVER_DURATION)

    goal = np.array([position_initial[0], position_initial[1], position_initial[2]]
                                                )
    cf.goTo(goal, yaw = 0.0, duration = TAKEOFF_DURATION)
    timeHelper.sleep(TAKEOFF_DURATION + HOVER_DURATION)

    cf.land(targetHeight=0.04, duration=TAKEOFF_DURATION)
    timeHelper.sleep(TAKEOFF_DURATION)
  # /vrpn_client_node/TURTLEBOT2/pose
  # geometry_msgs/PoseStamped


if __name__ == "__main__":

    main()
```

6.3 Appendice 3 : Code file for second simultaneous test

In this appendix, you will find the code file for the second simultaneous test.

```
from pycrazyswarm import Crazyswarm
import rospy
from geometry_msgs.msg import PoseStamped
import numpy as np

TAKEOFF_DURATION = 2.5
HOVER_DURATION = 2.0

turtle_pose = PoseStamped()

def turtle_pose_callback(msg):
    global turtle_pose
    turtle_pose = msg

def main():
    swarm = Crazyswarm()
    timeHelper = swarm.timeHelper
    cf = swarm.allcfs.crazyflies[0]

    sub = rospy.Subscriber('/vrpn_client_node/TURTLEBOT2/pose', PoseStamped,
                                                    callback = turtle_pose_callback)


    cf.takeoff(targetHeight=0.5, duration=TAKEOFF_DURATION)
    timeHelper.sleep(TAKEOFF_DURATION+HOVER_DURATION)

    position_initial = cf.position()

    pos_x = turtle_pose.pose.position.x
    pos_y = turtle_pose.pose.position.y

    inital_time = timeHelper.time()

    while(timeHelper.time()-inital_time < 20):
        pos_x = turtle_pose.pose.position.x
        pos_y = turtle_pose.pose.position.y
        print("x: ", pos_x, "y: ", pos_y)
        goal = np.array([pos_x, pos_y, position_initial[2]])
        cf.cmdPosition(goal)
        timeHelper.sleepForRate(10)

    cf.notifySetpointsStop(remainValidMillisecs = 100)

    goal = np.array([position_initial[0], position_initial[1], position_initial[2]]
                                                    )
    cf.goTo(goal, yaw = 0.0, duration = TAKEOFF_DURATION)
    timeHelper.sleep(TAKEOFF_DURATION + HOVER_DURATION)

    cf.land(targetHeight=0.04, duration=TAKEOFF_DURATION)
    timeHelper.sleep(TAKEOFF_DURATION)
  # /vrpn_client_node/TURTLEBOT2/pose
  # geometry_msgs/PoseStamped


if __name__ == "__main__":

    main()
```

6.4 Appendice 4 : Code file for third simultaneous test

In this appendix, you will find the code file for the third simultaneous test.

```
from pycrazyswarm import Crazyswarm
import rospy
from geometry_msgs.msg import PoseStamped
import numpy as np
import math

TAKEOFF_DURATION = 2.5
HOVER_DURATION = 2.0

turtle_pose = PoseStamped()

def turtle_pose_callback(msg):
    global turtle_pose
    turtle_pose = msg

def main():
    swarm = Crazyswarm()
    timeHelper = swarm.timeHelper
    cf = swarm.allcfs.crazyflies[0]

    sub = rospy.Subscriber('/vrpn_client_node/TURTLEBOT2/pose', PoseStamped,
                                                    callback = turtle_pose_callback)


    cf.takeoff(targetHeight=0.5, duration=TAKEOFF_DURATION)
    timeHelper.sleep(TAKEOFF_DURATION+HOVER_DURATION)

    position_initial = cf.position()

    pos_x = turtle_pose.pose.position.x
    pos_y = turtle_pose.pose.position.y

    inital_time = timeHelper.time()

    pos_x_bf = 1000
    pos_y_bf = 1000

    pos_x = turtle_pose.pose.position.x
    pos_y = turtle_pose.pose.position.y

    timeHelper.sleep(1)

    while(math.sqrt((pos_x-pos_x_bf)**2+(pos_y-pos_y_bf)**2) > 0.0001):
        pos_x_bf = pos_x
        pos_y_bf = pos_y
        pos_x = turtle_pose.pose.position.x
        pos_y = turtle_pose.pose.position.y
        goal = np.array([pos_x, pos_y, position_initial[2]])
        cf.cmdPosition(goal)
        timeHelper.sleepForRate(10)

    pos_x = turtle_pose.pose.position.x
    pos_y = turtle_pose.pose.position.y
    #q_x = turtle_pose.pose.orientation.x
    #q_y = turtle_pose.pose.orientation.y
    #q_z = turtle_pose.pose.orientation.z
    #q_w = turtle_pose.pose.orientation.w
    #t3 = 2.0*(q_w*q_z+q_x*q_y)
    #t4 = 1.0-2.0*(q_y*q_y+q_z*q_z)
    #yaw_z = math.atan2(t3,t4)

    cf.notifySetpointsStop(remainValidMillisecs = 100)

    goal = np.array([pos_x, pos_y, position_initial[2]])
    cf.goTo(goal, yaw = 0.0, duration= 1.0)
    timeHelper.sleep(1.0)

    #goal = np.array([position_initial[0], position_initial[1], position_initial[2
                                                    ]])
```

```
        #cf.goTo(goal, yaw = 0.0, duration = TAKEOFF_DURATION)
        #timeHelper.sleep(TAKEOFF_DURATION + HOVER_DURATION)

        cf.land(targetHeight=0.16, duration=TAKEOFF_DURATION)
        timeHelper.sleep(TAKEOFF_DURATION)
    # /vrpn_client_node/TURTLEBOT2/pose
    # geometry_msgs/PoseStamped


if __name__ == "__main__":

    main()
```

6.5 Appendice 5 : "Tutorial: Flashing the crazyradio 2.0 firmware"

To use a crazyradio 2.0, crazyradio2.0 firmware is required. As the instructions on the Bitcraze website seemed rather obscure, we decided to supplement them with our own personal experience. This is therefore a supplement to the tutorial available here.

FIRST : clone the git repository associated with the tutorial.

6.5.1 Building the Crazyradio2.0 Firmware

To build the firmware we have decided to use the "Natively installed tools" method. So you need to install Zephyr RTOS, as explained in the tutorial. Following tips outline its installation steps.

- Install the necessary tools/libraries :

```
$ sudo apt install python3-pip cmake curl ninja-build
$ pip install west
```

  Those tools are necessary to access Zephyr official repository and installing Zephyr.

- Install Zephyr in you local folder of your local directory of your Ubuntu version. (Follow the corresponding commands in the tutorial). This tool is general so it will be installed as a normal tool in your Ubuntu, so it is normal to see no difference in your project directory (directory where you want to build and flash the firmware)

- After installing Zephyr, make sure to be in your project directory ! (If not, move there using cd command). Then pull and and install Zephyr's tools onto your project directory using :

```
$ tools/build/fetch_dependencies
$ pip install -r zephyr/scripts/requirements.txt
```

  This time you should see modification made by Zephyr in your project directory.

When Zephyr is installed you are now ready to build the crazyradio2.0 firmware. Simply follow the tutorial :

```
$ west build -b bitcraze_crazyradio_2 -- -DCONFIG_LEGACY_USB_PROTOCOL=y
```

The option at the end of the command enable you to use the Crazyradio2.0 as a CrazyRadio1.0, it's way better for compatibility.

Normally you should have a built firmware in your directory, don't try to flash the firmware with this command written at the end of the build part of the tutorial :

```
$ west flash
```

We think that this command is written in the wrong section as we are still in the "Build" part of the tutorial, plus you need a JTAG cable to flash with this command, it's restrictive and it's possible to flash the firmware without it.

6.5.2 Flashing the Crazyradio2.0 firmware

Go to the "flashing" section of the tutorial. As written on it, flashing simply consists into drag and dropping a .uf2 file in the Crazyradio USB folder (appears when you plug the Crazyradio2.0 to your computer in bootloader mode).

Before doing that you can ensure that you have correctly build the firmware by searching this file (the .uf2 file) in the result of the building operation directory. Contrary to what is written in the tutorial the file ends with .uf2 and not .UF2. If you want to check rapidly you can just verify the presence of the file by typing the following command :

```
$ find /home/insa/crazyswarm/crazyradio2-firmware -name *.uf2 | wc -l
```

The output of this command is the number of files that end with .uf2 in the specified directory (this directory may vary depending on where you have decided to clone and build the git project). So if the command output is one, your build is correct !

Then follow the command line option for flashing it's much easier than searching for the .uf2 file.

After that your Crazyradio2.0 should be ready to make fly your Crazyflie drone, congratulations !