Semi-Autonomous Indoor Drones

Bernardo António Bernardino Rocha

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. Alberto Manuel Ramos da Cunha

Examination Committee

Chairperson: Prof. Prof. Alberto Manuel Rodrigues da Silva

Supervisor: Prof. Alberto Manuel Ramos da Cunha

Members of the Committee: Prof. Renato Jorge Caleira Nunes

September 2020

Abstract

The latest advances in Micro Aerial Vehicle (MAV) manufacturing have made these tiny robots very good development tools for both researchers and students. This dissertation aims to provide a system that students from IST, namely in Computer Engineering courses, can easily deploy and use in a laboratory environment to control a MAV, using their own computer and a python API, here described. This system is built on top of the Crazyflie platform and is accompanied by a set of laboratory guides for students to follow during laboratory classes. The topics covered range from manual navigation to mapping, autonomous exploration and drone-to-drone interaction. Experimental results show the system's ability to perform in complex indoor environments.

Keywords

MAV; Crazyflie; Indoor environment; Mapping; Exploration.

Resumo

Os últimos avanços no fabrico de Micro Veículos Aéreos (MAV) têm feito destes pequenos robôs boas ferramentas de desenvolvimento para investigadores e estudantes. Esta dissertação pretende fornecer um sistema que os alunos do IST, nomeadamente em cursos de Engenharia Informática, possam facilmente instalar e utilizar em ambiente de laboratório para controlar um MAV, utilizando o seu próprio computador e uma API em python, aqui descrita. Este sistema é baseado na plataforma Crazyflie e é acompanhado por um conjunto de guias de laboratório para orientar os alunos durante as aulas laboratoriais. Os tópicos abrangidos variam desde navegação manual a mapeamento, exploração autónoma e interação entre vários drones. Os resultados experimentais mostram a capacidade do sistema de operar em ambientes interiores complexos.

Palavras-Chave

MAV; Crazyflie; Ambiente interior; Mapeamento; Exploração.

Table of Contents

Abstract iii

Resumo v

Table of Contents vii

List of Figures ix

List of Tables x

List of Acronyms xii

1. Introduction 1

1.1. Objectives 1

1.2. Requirements 2

1.3. Nomenclature 3

1.4. Document Overview 3

2. Related Work 4

2.1. MAVs in Research and Education 4

2.1.1. SLIM 4

2.1.2. Autonomous Flight in Unknown Indoor Environments 5

2.1.3. Crazyflie 9

2.2. Indoor Localization 10

2.3. Collision Avoidance 15

2.4. Mapping 16

2.4.1. Line extraction 17

2.4.2. Occupancy grid mapping 18

2.5. Exploration 21

2.5.1. Frontier-based exploration 21

2.6. Swarming 23

2.7. Discussion 24

3. System Architecture 25

3.1. Crazyflie 25

3.2. Ground Control Station 28

3.3. Crazyflie API 30

3.4. Discussion 31

4. Laboratory Projects 33
4.1. Laboratory Guide 1 33
4.1.1. Installation 33
4.1.2. Crazyflie API 34
4.1.2.A. Manpilot 34
4.1.3. Demo Flight 35
4.1.4. Assisted Pilot 35
4.2. Laboratory Guide 2 35
4.2.1. Crazyflie API 36
4.2.1.A. Mapping 36
4.2.1.B. Autopilot 38
4.2.2. Autonomous Exploration 39
4.3. Laboratory Guide 3 40
4.3.1. Modifying the Firmware 41
4.3.2. Crazyflie API 42
4.3.2.A. Radar 43
4.3.2.B. DDS 43
4.3.3. Drone Delivery Application 44
5. Evaluation 46
5.1. Mapping 46
5.2. Exploration 49
5.3. Drone-to-drone communication 50
5.4. Discussion 52
6. Conclusion 53
6.1. System Limitations 53
6.2. Future work 54
7. References 56

List of Figures

Figure 2.1 - Multiple use cases of SLIM [2]. (a) An earlier version of the SLIM acting as teaching assistant [31]. (b) The SLIM during experimentation for the DAHV [32]. (c) Victim detection during the camera drones' rescue challenge. (d) Marker detection. (e) Avoidance of a thrown reflective marker. (f) Hula-Hoop visual tracking and passing through. 5

Figure 2.2 - Schematic of the hierarchical sensing, control and planning system [4] 6

Figure 2.3 - Two consecutive laser scans [4]. (a) A rotational error creates a misalignment. (b) Resulting scans after scan-matching. The parts that do not align correspond to sensor errors, occlusion or 3D effects. 7

Figure 2.4 - Maps constructed during autonomous flight [4]. Blue circles indicate goal waypoints clicked by human operator. Red line indicates path traveled based on the vehicle's estimates. (a) Map of MIT stata center, 1st floor. (b) Map of MIT stata center, 3rd floor. (c) Map of MIT stata center, basement....................8

Figure 2.5 - Swarms of Crazyflies flying in a motion capture arena in response to a user's direction [33] 9

Figure 2.6 – Experimental (a) and simulated (b) flying cars [8] 10

Figure 2.7 - Concept of obstacle detection and collision avoidance [13] 16

Figure 2.8 - Comparison between different line extraction algorithms [17] 17

Figure 2.9 - A range finder sensor model for two different measurements [21]. The darkness of each grid cell corresponds to the likelihood of occupancy....................19

Figure 2.10 - (a) Occupancy grid map and (b) architectural floor plan of a large open space [21]. 20

Figure 2.11 - Frontier detection [6]: (a) evidence grid, (b) frontier edge segments, (c) frontier regions 22

Figure 3.1 - Solution architecture 25

Figure 3.2 - Crazyflie hardware architecture [22]....................26

Figure 3.3 - Crazyflie 2.1 drone and expansion decks 28

Figure 4.1 - Sequence diagram for detecting a neighboring drone....................41

Figure 5.1 - Top view of testing room (a) and occupancy grid map generated during manual flight (b)47

Figure 5.2 - Occupancy grid map generated during exploration. Blue dots are goals generated by frontier detection algorithm. 50

List of Tables

Table 1 - Terminology 3

Table 2 - Advantages and disadvantages of different localization techniques [11]...............................12

Table 3 - Summary of different wireless technologies for localization [11] 14

Table 4 - Crazyflie 2.1 specifications [24] 27

Table 5 - Crazyradio PA specifications [24]...29

Table 6 - Occupancy grid mapping performance experimental results 48

Table 7 - Drone-to-Drone collision detection experimental results...51

Table 8 - Cluster-to-Drone collision detection experimental results 52

List of Acronyms

AOA	Angle of Arrival
API	Application Programming Interface
BLE	Bluetooth Low Energy
CRTP	Crazy Real Time Protocol
CSI	Channel State Information
DDS	Drone Delivery Service
DOF	Degrees of Freedom
EEPROM	Electrically Erasable Programmable Read-Only Memory
EKF	Extended Kalman Filter
FANET	Flying Ad-hoc Network
FOV	Field of View
GCS	Ground Control Station
GPS	Global Positioning System
IMU	Inertial Measurement Unit
ITS	Intelligent Transportation System
LDR	Light Dependent Resistor
LED	Light Emitting Diode
LOS	Line of Sight
MAV	Micro Aerial Vehicle
MCU	Microcontroller Unit
NIC	Network Interface Card
OOP	Object-Oriented Programming
PID	Proportional–Integral–Derivative
POA	Phase of Arrival
RC	Radio Control
RF	Radio Frequency
RFID	Radio Frequency Identification Device

RSSI	Received Signal Strength Indicator
RTOF	Return Time of Flight
Rx	Receiver
SLAM	Simultaneous Localization and Mapping
SLIM	Scalable and Lightweight Indoor-navigation MAV
TDOA	Time Difference of Arrival
TOF	Time of Flight
Tx	Transmitter
UAV	Unmanned Aerial Vehicle
URI	Uniform Resource Identifier
UWB	Ultra-Wideband
VLC	Visible Light Communication

1. Introduction

Unmanned aerial vehicles (UAV), commonly known as drones, have been used for a long time in several civilian and military domains, including weather observation, surveillance, search and rescue operations or civil engineering inspections. In recent years, propelled by the latest advances in the field, they have become popular in cinematography, agriculture or transportation of medicines in and out of remote or otherwise inaccessible regions. In fact, as these robots' capabilities keep increasing, they also keep getting smaller and cheaper, and new uses for them continue to be explored. More recently, research has been done towards the development of small human-friendly drones that can fly autonomously in indoor environments. Unlike a regular-sized UAV, a micro aerial vehicle (MAV) can operate in confined and GPS-denied environments, and since it is usually a low-cost solution, it can be easily replaced in case of damage or total loss. Moreover, they can be an excellent development tool for students and researchers, in a big diversity of fields such as embedded systems, robotics or control theory. Because of their small size and weight, it is very safe to use them in a laboratory environment, including near people. More specifically, they make up a new and engaging learning opportunity for university students in the Computer Systems branch, as they would able to modify both software and hardware of the drone and build their own drone-oriented applications during laboratory classes.

1.1. Objectives

The main focus of this work is in providing a full-fledged system that students and researchers from Computer Engineering can easily use and deploy in laboratory environment, such as the facilities at Departamento de Engenharia Informática (DEI) in IST, to develop drone-oriented applications. The system aims to take advantage of the unique and engaging learning opportunities provided by MAVs, specifically in the Computer Systems / Cyber-Physical Systems branch, as they are complex cyber-physical systems, with real-time sensing and control requirements, and at the same time, highly resource-constrained systems, as their small size does not allow powerful sensors, computing units and batteries. To that end, a set of 3 laboratory guides will be produced that will give students the opportunity to tackle these unique challenges, using MAVs, during laboratory classes of a subject in the field. For them to solve the proposed exercises, a high-level API is here developed and documented, which offers a few high-level calls that allow easy control of the drones. Finally, this work also aims to provide a solid base for future students that wish to use these drones for their own projects, or otherwise anyone that wants to extend the system here proposed in any way.

The goal of this project is not to build a drone from scratch and all its components, but to present a complete solution that can be easily deployed and used. Such system should create a safe indoors areal space, where multiple drones can coexist with each other and with people that may be present in the room, while allowing multiple groups of students to independently control their own MAV.

In a narrower scope, the objectives with each laboratory guide are defined as follows:

Laboratory Guide 1:

- A drone as a cyber-physical system
- Introduction to real-time sensing and control
- Introduction to the development environment
- Building the first application: manual flight

Laboratory Guide 2:

- A drone as a highly resource-constrained system
- Management of such resources: battery
- Real-time sensing and control: drone mapping
- Drone as semi-autonomous system: drone exploration

Laboratory Guide 3:

- Interaction between different groups of students
- Multi-robot architecture: Intelligent Transportation system
- Drone-to-drone communication
- Real-time sensing, control and semi-autonomy: Drone delivery

The choice of the appropriate drone platform to work with was a result of the work developed during the Master Project subject [1], as such, this thesis focuses on how that platform together with additional functionality developed were used to fulfill the described goals. For a summarized explanation of why that platform was chosen, refer to section 3.4. It is important to note that the goals here described are addressed mainly as a software challenge rather than focus on enhancing or automating hardware.

1.2. Requirements

To deliver a system that aims to accomplish such goals, some architectural requirements were previously established:

- **Safety**

  Controlling a flying robot inevitably carries some risk, however, measures should be taken to lower that risk as much as possible to avoid damaging the equipment or injuring someone. When designing the API, it is important to consider how much control user should have versus how much automatic safety behavior should be implemented.

- **Deployability**
  In the context of this project, this means that the system should be quick and easy to set up in a classroom/laboratory environment. For instance, it should not be required to have any preinstalled hardware in the room to assist with location strategies.

- **Performance**
  The hardware should deliver good communication responsiveness between PC and MAV, decent battery flight time and good sensor accuracy. Algorithms implemented should be evaluated.

- **Scalability**
  This requirement is specific to the last laboratory setup, where multiple drones will need to communicate with each other simultaneously. It should be tested how many drones the system supports at the same time, without compromising other requirements.

1.3. Nomenclature

This document contains some terminology often used in robotics research specific to aircrafts, such as MAVs. Table 1 is intended to be used as future reference for the reader if needed.

Table 1 - Terminology

Concept	Meaning
Pitch	Rotation around the side-to-side axis
Pose	Combination of position and orientation
Roll	Rotation around the front-to-back axis
Thrust	Upward or forward force produced by propellers
Yaw	Rotation around the vertical axis

1.4. Document Overview

The rest of this document is organized as follows. In Chapter 2, systems involving the use of MAVs in research and education are presented, as well as technologies that helped to understand those systems and in preparing the laboratory guides. In Chapter 3, the solution architecture is described, as well as its main components. Chapter 4 goes over the implementation of everything that went into the laboratory guides. Chapter 5 presents the results of the evaluation done, according to the presented requirements. Finally, Chapter 6 presents some concluding remarks and future work that is thought to be interesting to further explore.

2. Related Work

In this chapter work done in different fields of interest to this thesis will be presented. First, some existing systems that are being used in research and education will be presented. Then, some technologies that help to understand those systems or were necessary for the development of the laboratory guides are discussed. Specifically, techniques related with robot mapping, exploration and swarming will be analyzed. While each one of these topics has an extensive body of research done on their own, this work objective is only to provide an overview of each field. The functionality developed as result of this research should be enough to, on one hand, introduce students to the topic during a laboratory class, and on the other hand, to help researchers that wish to further explore one of these topics, to get started on building drone-oriented applications using the presented system.

2.1. MAVs in Research and Education

In this section, existing systems that are already being used in research and education at university level are presented. Different high-level architectures will be discussed without going too much in-depth about the underlying technologies used. The last sub-section refers to systems built using the same MAV that will be used in this work.

2.1.1. SLIM

SLIM stands for Scalable and Lightweight Indoor-navigation MAV [2]. This project was developed at Graz University of Technology, Austria. It has been in use since 2015, not only in multiple research projects but also lecture courses and student projects. It was even used to promote an indoor drone rescue challenge, where different groups of students had to compete to be the first at reporting back the position of found victims in a maze that was designed to simulate a disaster scenario. Teams were expected to solve problems which included localization, path-planning, navigation, 3D mapping and the detecting the markers that were simulating victim locations. This use case and others are illustrated in Figure 2.1, demonstrating the flexibility of the system.

When designing the SLIM, the goals set by the authors were very similar to the ones of this project. They wanted to enable indoor flight in a flexible setup while providing easiness of use to inexperienced students and, at the same time, making it possible to do experimental flights as safe as possible.

Their MAV selection process was very interesting. They excluded any off-the-shelf solution on the argument that the final solution requires maximum flexibility and accessibility of individual interfaces. The platforms that were analyzed were either too big to be used indoors, did not provide ROS[1] (Robot Operating System [3]) framework support, or did not offer powerful enough on-board sensors for their

[1] ROS framework: https://www.ros.org/. Last Accessed: 9 September 2020

needs. This last case invalidated the drone that will be used in the system presented in this thesis (see section 3.1), which was considered by the authors.

They built the MAV using open hardware components, modeled the motion of the MAV to reflect the dynamics of such a rigid body, integrated a PID controller for controlling the MAV and perform position estimation. There is no interest in showcasing these details, as they are outside of the scope of this work as well as outside of the computer engineering field in general. In addition, these components are already implemented in the firmware of the drone used in this work.

As for software framework, they made use of the Robot Operating System (ROS) [3] because it is open source and has a rich ecosystem of functionality. A set of core functionalities were developed and integrated as individual components in a ROS framework, such as: low-level flight control, localization, high-level flight control (path-planning and exploration), mapping and object detection (using monocular cameras and a RGBD sensor). All computations are done in the drone in external computers that were added to the drone and support UNIX-based operating systems.

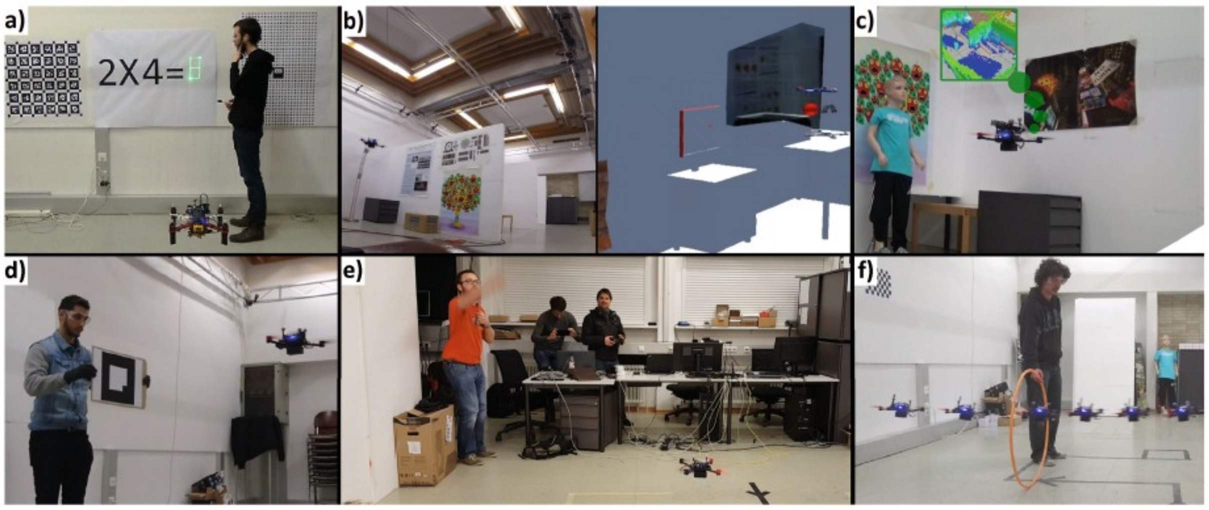


Figure 2.1 - Multiple use cases of SLIM [2]. (a) An earlier version of the SLIM acting as teaching assistant [31]. (b) The SLIM during experimentation for the DAHV [32]. (c) Victim detection during the camera drones' rescue challenge. (d) Marker detection. (e) Avoidance of a thrown reflective marker. (f) Hula-Hoop visual tracking and passing through.

2.1.2. Autonomous Flight in Unknown Indoor Environments

In a work developed by researchers at MIT [4], a quadcopter was equipped with laser rangefinder sensors to autonomously explore and map unstructured and unknown indoor environments. This work is very interesting as they tackle a lot of the problems within the context of this dissertation primarily as a software challenge. To that end, they use only off-the-shelf hardware.

Their architecture relies on a Ground Control Station (GCS) to do the heavy processing and was designed as a combination of asynchronous modules, built upon the CARMEN[2] robot navigation toolkit's software architecture. This architecture is depicted in Figure 2.2. It involves multi-level sensing hierarchy, a high-speed laser scan-matching algorithm, an Extended Kalman Filter (EKF) for data fusion, a high-level SLAM implementation, and an exploration planner. Processes are distinguished

[2] CARMEN: http://carmen.sourceforge.net/. Last Accessed: 9 September 2020

based on the real-time requirements of their respective outputs. At the base level, the on-board IMU and controller (represented in green) create a feedback loop to stabilize the vehicle's pitch and roll. The yellow modules make the real-time sensing and control loop that stabilize the vehicle's pose and avoid obstacles. Finally, the red modules provide the high-level mapping and planning functionalities.

These modules will now be briefly explained, following the flow of estimating a position.

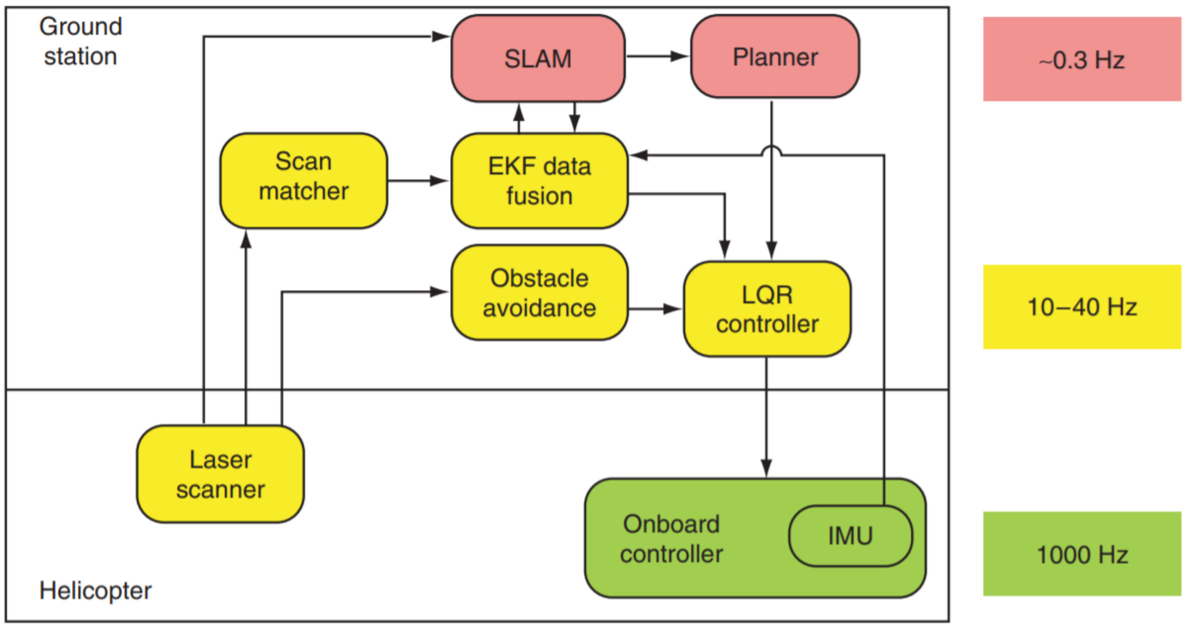


Figure 2.2 - Schematic of the hierarchical sensing, control and planning system [4]

The process of estimating the MAV's motion in the space starts with a standard technique from robotics known as scan-matching. The goal is to find the relative pose (or transform) between the two positions where the laser scans were taken by aligning those scans considering the shapes of their overlapping features, like shown in Figure 2.3.

After the scan matcher outputs the estimated vehicle position (x, y, θ), an EKF is used to fuse those estimates with the acceleration readings from the Inertial Measurement Unit (IMU), which is a combination of accelerometers, gyroscopes, and sometimes magnetometers. The odometry readings drift significantly and are therefore not useful over extended time periods. However they are useful over short time periods at improving estimates of the vehicle's velocities, so it's usually much better to use a combination of both position estimates from the scan matcher and IMU reading, as it results in a more accurate state estimate.

Then, that position estimate is fed to the SLAM algorithm. Simultaneous localization and mapping (SLAM) algorithms build a map of the environment around the vehicle while simultaneously using it to estimate the vehicle's position. Their implementation is based on the GMapping algorithm [5], which produces a 2D occupancy grid map from laser range data. At the same time, the GMapping algorithm periodically sends position corrections to the data fusion EKF to try improving position estimation accuracy even further.

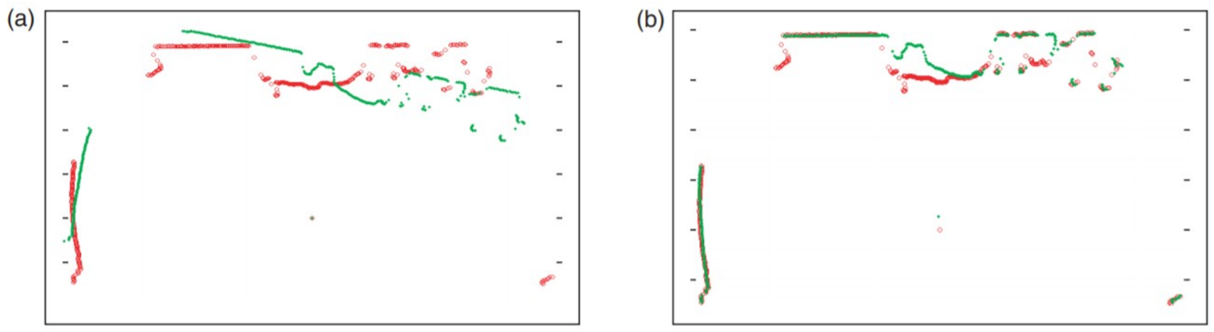
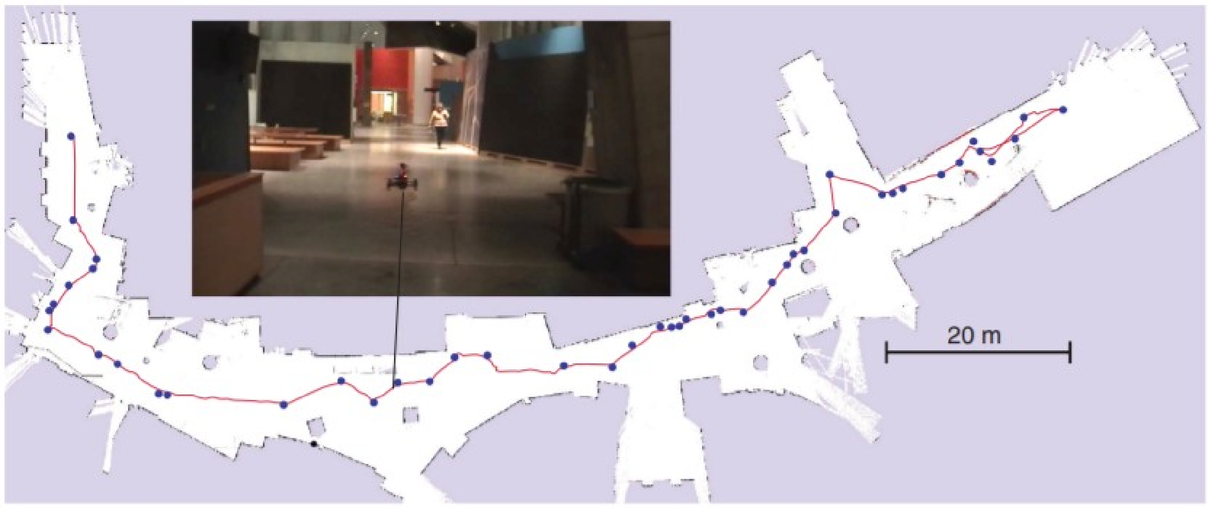


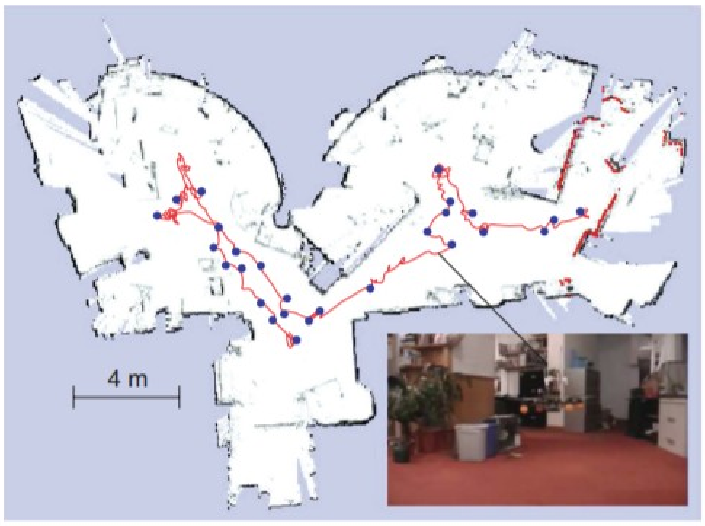
Figure 2.3 - Two consecutive laser scans [4]. (a) A rotational error creates a misalignment. (b) Resulting scans after scan-matching. The parts that do not align correspond to sensor errors, occlusion or 3D effects.

In addition to compute globally consistent state estimates, the map generated by the SLAM algorithm is used to plan actions for the vehicle autonomously. To do this they used a modified definition of frontiers, first proposed in [6], which the author classifies as areas in the map where there is a direct transition between free and unexplored cells. In [4], the authors use a similar method to identify these frontier regions, however, for each frontier they seek to find a pose that maximizes not only the amount of unexplored space that is expected to be observed but also the ability of the MAV to localize itself, since the unknown environment may not contain enough structure for the relative position estimation algorithms to match against. To achieve autonomous exploration of an unknown environment, the planner uses the nearest frontier as its goal. In addition, the frontier extraction modules run fast enough that they can re-generate plans as the vehicle moves through the environment and as the map is updated.

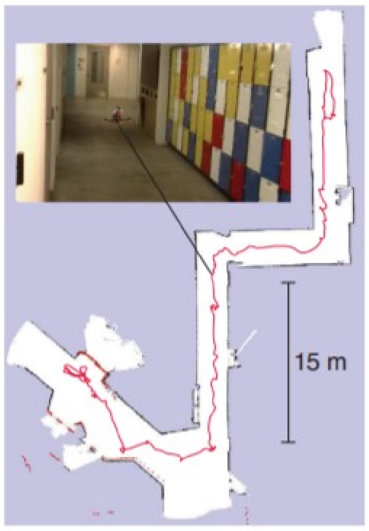
To demonstrate their results, they flew the drone across 3 indoor environments, inside MIT's Stata Center. Firstly, they flew it in the open lobby, an environment with very little obstacles, as seen in Figure 2.4 (a). Here the vehicle was guided by a human operator clicking high-level goals in the map that was being built in real-time, after which the planner decided the best path to the goal. Secondly, in a cluttered laboratory, in Figure 2.4 (b), to test how the SLAM algorithm would perform in an obstacle-rich environment. And finally in a hallway, in Figure 2.4 (c), this time with no human intervention during the complete flight. The resulting maps show that the drone is capable of fully autonomous exploration in unstructured and unknown indoor environments without a prior map.



(a)



(b)



(c)

Figure 2.4 - Maps constructed during autonomous flight [4]. Blue circles indicate goal waypoints clicked by human operator. Red line indicates path traveled based on the vehicle's estimates. (a) Map of MIT stata center, 1st floor. (b) Map of MIT stata center, 3rd floor. (c) Map of MIT stata center, basement.

2.1.3. Crazyflie

Universities around the world have been using this platform for different kinds of research and applications, taking advantage of its off-the-shelf flexibility, extendibility, wide range of tools available and openness.

In Carnegie Mellon University, swarms of Crazyflies were used in multi-robot systems that translate a human operator's intent into dynamically feasible and safe motion plans [7]. Using a motion capture system to provide pose information of the robots, a centralized planner is able to send user instructions consisting of high level information such as robot id's, behavior type (go to a target destination, rotate in a circle), and formation shapes (line, polygon). Once the motion plans are computed, the desired position setpoints are broadcasted to the Crazyflies, resulting in complex and synchronized swarm motions, shown in Figure 2.5.

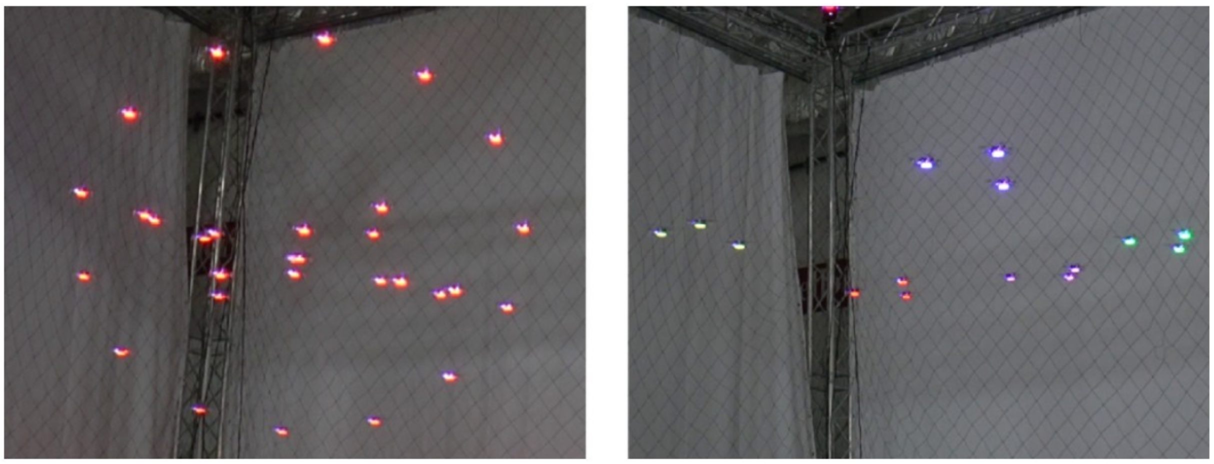
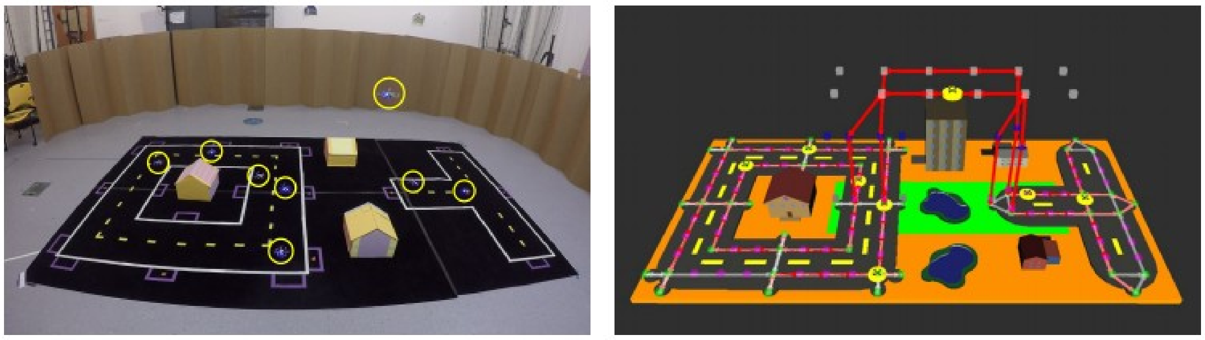


Figure 2.5 - Swarms of Crazyflies flying in a motion capture arena in response to a user's direction [33]

In another work developed by researchers at the Computer Science and Artificial Intelligence Lab at MIT, Crazyflie drones were used to perform multi-robot path planning for a swarm of robots that can both fly and drive [8]. To do this, the drone was fitted with a wheel mechanism on the bottom, as well as a ground controller, that would allow it to behave like a car, and then it would be able to switch to an air controller and start flying. To demonstrate the success of the implemented path-planning algorithms the researchers populate a miniature town with 8 "flying-car" Crazyflies, as show in Figure 2.6.

The Crazyflie platform has also been used to build systems oriented to university level education [9] [10]. In [9], the authors present three distinctive projects. The first one is a freely available Crazyflie Java Client software package, which allows, among other features, flight data acquisition. The main value of this contribution comes mainly from the fact that there is no official Java library provided to control the drone. The second project is the 4FLY Simulator, intended for designing and initial testing of a Crazyflie 2.0 control system with a variety of controller types. It can also run simulations based on data pre-recorded using the first project. Finally, they show how the Crazyflie can be used in the context of vision-based control. Their contributions are directed towards students studying the subjects of dynamical modeling and control, real-time systems, sensors and vision systems, automatic control, embedded control systems or signal processing.

In [10], a flexible swarming architecture of Crazyflies is created. The system is directed towards research and education at Iowa State University and uses a motion capture camera system for localization. Applications developed using this system can range from simple single Crazyflie autonomous flight to more complex gesture controlled multi-robot master-slave scenarios. However this project is integrated in the course of Electrical Engineering and the topics addressed vary from the goals of this work.



(a) (b)

Figure 2.6 – Experimental (a) and simulated (b) flying cars [8]

2.2. Indoor Localization

Most available UAVs rely on GPS for obtaining its location, however other technologies need to be considered as GPS is not available indoors. In a recent survey paper [11], the authors collected, discussed and compared techniques and technologies that are being used in indoor localization problems. This work helped to establish an important baseline, as most of the developed technologies in the field, whether they are intended for drones or not, are based on one of these techniques:

- **Received Signal Strength Indicator (RSSI)**

  This is one of the simplest and most used approaches. The RSS is the actual signal power strength that a receiver (Rx) reads from a transmitter (Tx), usually measured in decibel-milliwatts (dBm) or milliwatts (mW), while the RSSI is a relative measurement of the RSS that has arbitrary units and is defined by the chip vendor. For instance, Atheros Wi-Fi chipset uses RSSI values between 0 and 60. These values can then be used in different signal propagation models, given that either the transmission power or the power at a reference point is known, to estimate an absolute distance. Unless it's being used in proximity-based services, RSS based location will generally require at least three reference nodes for trilateration. Although cost efficient, this approach lacks localization accuracy, especially in indoor environments, where walls and other obstacles can attenuate, cause fluctuations or a multipath effect. While software filters can be used to reduce these effects, it's still very hard to obtain high localization accuracy.

- **Channel State Information (CSI)**

  CSI is similar to the one above but has higher granularity than the RSSI as it can capture both the amplitude and phase responses of a wireless channel in different frequencies and between separate transmitter-receiver antenna pairs, which makes this approach more robust to multipath and indoor noise, as it can register more stable measurements.

- **Fingerprinting/Scene Analysis**

This approach consists of a previous survey to the environment to obtain fingerprints or features of the space. RSSI or CSI measurements are collected during that offline phase, and then real-time measurements are compared with the previously taken and location estimation is calculated. This technique is especially relevant in indoor environments as it should be easy to calculate a reasonably accurate estimation since the scene is not expected to be very large, however a slight change in the environment will require a new offline fingerprint to be taken.

- **Angle of Arrival (AOA)**

This technique uses arrays of antennas at the receiver side that will estimate the angle and time difference at which the signals arrive at each individual antenna of the array. The main advantage of AOA is that the location can be estimated with as low as two antennas in a 2D environment (or three antennas in a 3D environment) but its accuracy deteriorates with increase in the transmitter-receiver distance where a slight error in the angle of arrival calculation is translated into a huge error in the actual location estimation. Furthermore, this requires complex hardware, careful calibration and like RSSI is susceptible to multipath in indoor environments.

- **Time of Flight (TOF)**

In this method, the time it takes for a signal to travel between Tx and Rx is measured and multiplied by the speed of light $c = 3 \times 10^8$ m/sec to provide the distance between them. Like RSSI, there is need for having at least 3 reference nodes for signal triangulation. TOF requires strict synchronization between transmitters and receivers and, depending on the underlying communication protocol, timestamps may be required to be transmitted with the signal. Line of Sight (LOS) is especially important for accurate performance; otherwise, reflected signals will result in greater distance estimates.

- **Time Difference of Arrival (TDOA)**

TDOA exploits the difference in signal propagation times from different transmitters, measured at the receiver, by considering the distance between transmitters. Having multiple transmitters (3 or more) is especially important in order to calculate the exact location of the receiver. Like in TOF, the TDOA estimation accuracy depends on the signal bandwidth, sampling rate at the receiver and the existence of direct LOS between the transmitters and the receiver.

- **Return Time of Flight (RTOF)**

Measures the round-trip (transmitter-receiver-transmitter) signal propagation time to estimate the distance between Tx and Rx. The main advantage when comparing with TOF is that only relatively moderate clock synchronization between the Tx and the Rx is required. Estimation accuracy is affected by the same factors as TOF which in this case is more severe since the signal is transmitted and received twice. Another disadvantage is the time that it takes for the receiver to process the signal and send a response, which often depends on the electronics of the receiver.

- **Phase of Arrival (POA)**

POA based approaches use the phase or phase difference of carrier signal to estimate the distance between Tx and Rx. It is common to assume that the waves transmitted are of pure sinusoidal form, having same frequency and zero phase offset. For range estimation, algorithms used for TOF can be used, or TDOA based algorithms if the phase difference between two transmitted signals is used. This can be used in conjunction with RSSI, TOF, TDOA to improve the localization accuracy and enhance the performance of the system.

Table 2, from this work, provides a summary of the presented techniques for indoor localization and discusses the advantages and disadvantages of these techniques.

Table 2 - Advantages and disadvantages of different localization techniques [11]

Technique	Advantages	Disadvantages
RSSI	Easy to implement, cost efficient, can be used with several technologies	Prone to multipath fading and environmental noise, lower localization accuracy, can require fingerprinting
CSI	More robust to multipath and indoor noise	It is not easily available on off-the-shelf NICs
AOA	Can provide high localization accuracy, does not require any fingerprinting	Might require directional antennas and complex hardware, requires comparatively complex algorithms and performance deteriorates with increase in distance between the transmitter and receiver
TOF	Provides high localization accuracy, does not require any fingerprinting	Requires time synchronization between the transmitters and receivers, might require time stamps and multiple antennas at the transmitter and receiver. LOS is mandatory for accurate performance
TDOA	Does not require any fingerprinting, does not require clock synchronization among the device and reference nodes	Requires clock synchronization among the reference nodes, might require time stamps, requires larger bandwidth
RTOF	Does not require any fingerprinting, can provide high localization accuracy	Requires clock synchronization, processing delay can affect performance in short ranger measurements
POA	Can be used in conjunction with RSS, TOA, TDOA to improve the overall localization accuracy	Degraded performance in the absence of LOS
Fingerprinting	Fairly easy to use	New fingerprints are required even when there is a minor variation in the space

Next will be presented some technologies that enable the techniques mentioned above and have been used to provide indoor localization services. Most of them are known communication protocols.

- **Wi-Fi**

This is one of the most widely studied localization technologies. Because existing Wi-Fi access points can also be used as reference nodes for signal collection, basic localization systems that can achieve reasonable localization accuracy can be built without the need for additional infrastructure. The RSSI, CSI, TOF and AOA techniques can be used together with Wi-Fi.

- **Bluetooth**

The Bluetooth Low Energy (BLE) version has been used with different localization techniques such as RSSI, AOA, and TOF, but most of the existing solutions rely on RSSI based inputs as RSSI based systems are less complex. Recently, BLE based protocols have shown going more towards context aware proximity-based services.

- **Zigbee**

Zigbee is concerned with physical layers for low cost, low data rate and energy efficient personal networks as it’s mainly used in wireless sensor networks. This is not readily available on majority of the user devices though, especially commercial drones.

- **Radio Frequency Identification Device (RFID)**

It is primarily intended for transferring data using electromagnetic energy from a transmitter to any radio frequency compatible circuit. The system consists of a reader that communicates with an RFID tag, using a predefined RF and protocol, known to both the reader and tag a priori. There are two types of RFID systems:

    - **Passive RFID:** very short communication range (1-2m) that makes this unsuitable for indoor localization. Can be used for proximity-based services and user access control.
    - **Active RFID:** These tags periodically transmit their ID to the reader and can do it from hundreds of meters. Could be used for tracking a drone but it cannot achieve submeter accuracy.

- **Ultra-Wideband (UWB)**

Ultra-short pulses with time period of less than 1 nanosecond are transmitted over a large bandwidth in the frequency range from 3.1 to 10.6 GHz using a very low duty cycle, which results in reduced power consumption. This is interesting for indoor localization because it is immune to interference from other signals (due to its drastically different signal type and radio spectrum) and the signal can penetrate a variety of materials, including walls. The slow progress in the UWB standard development has limited its use in consumer products though.

- **Visible Light**

Visible Light Communication (VLC) uses visible light between 400 and 800 THz, emitted primarily by Light Emitting Diodes (LEDs) as transmitters, and light sensors as receivers. AOA is considered the most accurate localization technique.

- **Acoustic Signal**

Here the receivers are microphones, and the transmitters are sound emitters. The traditional method used for acoustic-based localization has been the transmission of modulated acoustic signals, containing timestamps which are used by the microphone sensors for TOF estimation. The extra hardware required, and the noise caused by sound pollution (which is in this case can come not only from the environment but also from the drone itself) makes this not practical for localization.

- **Ultrasound**

TOF measurements of ultrasound signals (>20 KHz) and sound velocity are used to calculate the distance between a transmitter and a receiver node. It offers high indoor localization accuracy and can track multiple mobile nodes at the same time with high energy efficiency. Since the sound velocity varies with humidity and temperature, the system should adapt for these changes. It is also mentioned that a permanent source of noise may degrade the system performance severely.

The maximum range, throughput, power consumption, advantages and disadvantages of using these technologies for localization are summarized in Table 3.

Table 3 - Summary of different wireless technologies for localization [11]

Technology	Maximum Range	Maximum Throughput	Power Consumption	Advantages	Disadvantages
IEEE 802.11 n 802.11 ac 802.11 ad	250 m outdoor 35 m indoor couple of meters	600 Mbps 1.3 Gbps 4.6 Mbps	Moderate Moderate Moderate	Widely available, high accuracy, does not require complex extra hardware	Prone to noise, requires complex processing algorithms
UWB	10-20 m	460 Mbps	Moderate	Immune to interference, provides high accuracy	Shorter range, requires extra hardware on different user devices, high cost
Acoustics	Couple of meters		Low-Moderate	Can be used for proprietary applications, can provide high accuracy	Affected by sound pollution, requires extra anchor points or hardware
RFID	200 m	1.67 Gbps	Low	Consumes low power, has wide range	Localization accuracy is low
Bluetooth	100 m	24 Mbps	Low	High throughput, reception range, low energy consumption	Low localization accuracy, prone to noise
Ultrasound	Couple-tens of meters	30 Mbps	Low-Moderate	Comparatively less absorption	High dependence on sensor placement
Visible Light	1.4 km	10 Gbps	Relatively higher	Wide-scale availability, potential to provide high accuracy, multipath-free	Comparatively higher power consumption, range is affected by obstacles, primarily requires LOS

2.3. Collision Avoidance

One of the core challenges of this work is to ensure single drone robustness. In order to guarantee a reasonable amount of autonomy and safety, a robot must be able to detect obstacles and avoid collisions. Even in a system where continuous communication with a ground station is required, fault tolerance mechanisms can be implemented in case of disconnection, so that the robot can remain autonomous. This level of independence also facilitates the scalability of the system, as each drone that is part of a cooperative mission can individually sense its surroundings and act accordingly.

To perform this sensing, popular approaches make use of either laser-ranging sensors [4], ultrasonic sensors [12] [13] or cameras [14] [15] [16].

- **Laser-Ranging Sensors**

  A laser-ranging sensor or laser rangefinder emits a laser beam to determine the distance to an obstacle. These sensors usually operate on the TOF principle, discussed earlier in section 2.1.2, where a laser pulse is sent and eventually reflected on the first obstacle encountered. The pulse is then received by a matching sensor placed close to the emitter, and the time it took to travel is measured. Depending on the sensor, maximum range can reach up to a few kilometers (20-25 km) with most common sensors getting millimeter accuracy.

- **Ultrasonic Sensors**

  These sensors work similarly to the ones just described, but the pulse is an ultrasonic wave instead and the distance is calculated considering the speed of sound (343 m/s in dry air at 20ºC). Since this velocity will vary as humidity or temperature changes, dedicated on-board sensors can help improve the accuracy of the calculated distance by taking these changes into account. As demonstrated here [13], unlike optical sensors like the presented above, ultrasonic sensors can detect obstacles through smoke or steam, but can struggle at detecting soft surfaces like foamed material and clothes.

- **Cameras**

  There is also a lot of work done towards computer vision algorithms, where the main sensor used is a camera. Most of the systems are based on the concept of depth from focus to locate static and moving obstacles with adequate accuracy for obstacle avoidance but while some do all the image processing in real-time others focus on generating maps of collision-free waypoints, which later can generate navigable maps. All the approaches require some level of computational power that, most of the times, must be done by a ground station.

In this paper [13], the authors propose an approach using ultrasonic sensors intended to support autonomous flight. Their concept is divided into two main modules, independent of each other: One for Obstacle Detection and one for Collision Avoidance.

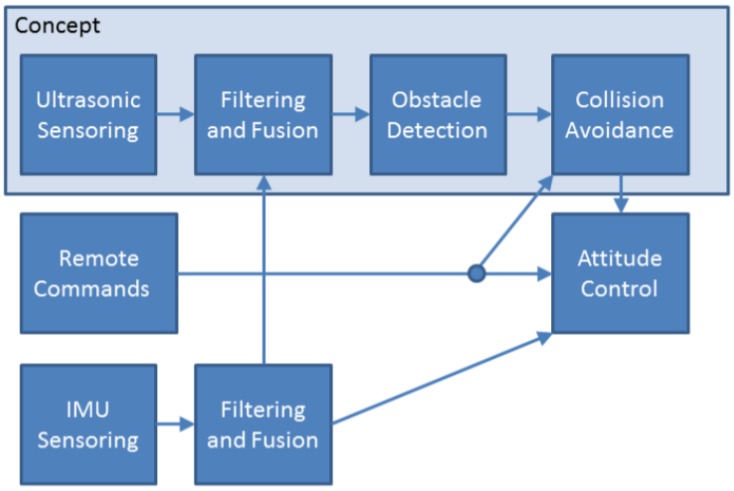


Figure 2.7 - Concept of obstacle detection and collision avoidance [13]

Figure 2.7 abstracts the system as a set of various subsystems, which is presented by this paper. The ultrasonic raw data is filtered and fused with the IMU data. Then the obstacle detection module looks at this data and passes the results to the collision avoidance module that will enable a controlled flight. Remote data from a computer or an RC (Radio Control) controller for activating and deactivating the system as well as sending steering commands are also fed to the module.

In the obstacle detection module, it is worth mentioning the use of redundant sensors to increase detection resolution and sensor data reliability, which translated into 12 ultrasonic sensors for a total 360º FOV, where each sensor shares at least half its FOV with adjacent sensors. As for the collision avoidance module, its first job it's to divide the area around the quadcopter into three zones, depending on the measured distance: Far or safe zone, close zone and dangerous zone. The behavior of the module is then described by the authors as a state machine: Initially, and if the autonomous collision avoidance is off, the quadcopter is in state 0. If the autonomous mode is activated, the quadcopter can switch between state 1, 2 or 3 depending on the measured distance to a nearby object. State 1 (safe zone) is active, if there is no obstacle nearby. If an obstacle is detected in the close zone, state 2 (close area) is activated and the corresponding pitch or roll angle towards the obstacle is limited depending on the measured distance reducing the speed of approach. In the dangerous zone, state 3 is activated and the distance to the obstacle is controlled using a PID controller, preventing a further approach to the obstacle. Hence, such a state machine is necessary for every direction.

2.4. Mapping

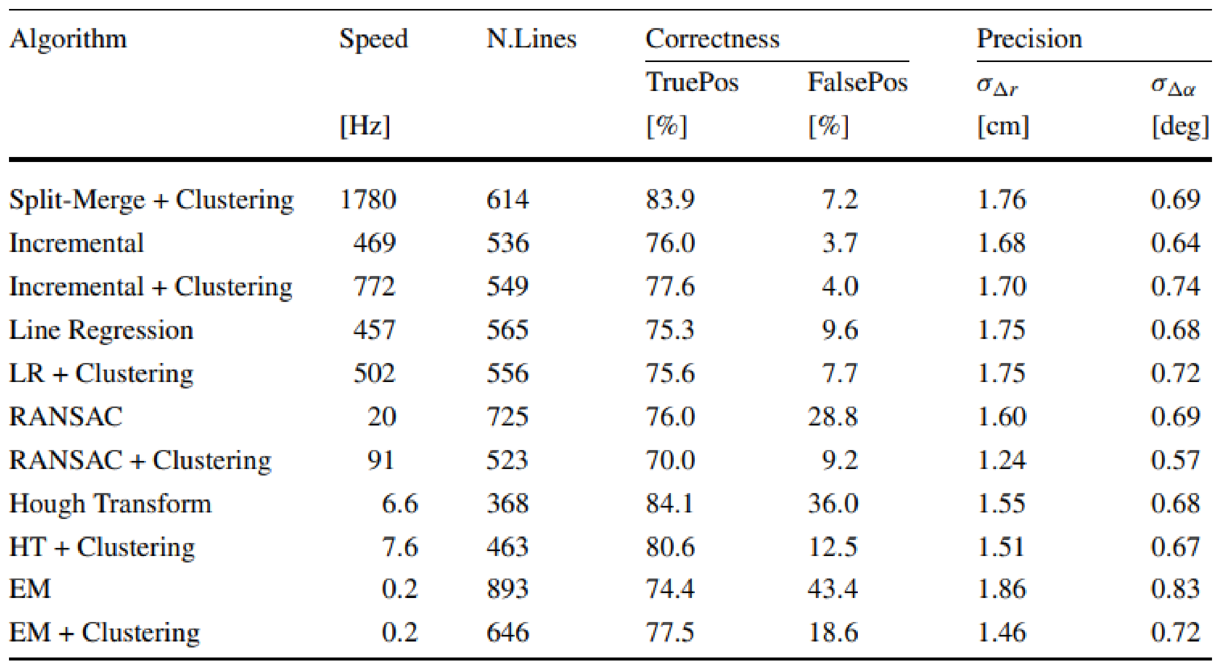
Mapping is concerned with the problem of building a representation of the physical world, using information gathered from a robot's sensors, and is one of the core competencies of truly autonomous robots. This work is specifically concerned with indoor mapping, which generally results in a map which looks like a floor plan. Students will be asked to build a floor plan in laboratory guide 2, so this functionality needs to be provided by the API. Different types of algorithms that allow implementing this functionality will now be presented.

2.4.1. Line extraction

Line extraction algorithms try to obtain line segments from raw sensor measurements, which can then be used to construct a map of the environment. The measurements create data points in the 3D (or 2D) world, which are then used to estimate line segments.

There are several types of line extraction algorithms. To learn what were the most promising ones to use in this work, a survey paper [17] that evaluates six popular algorithms used in mobile robotics was analyzed. The algorithms are given previously collected 2D laser scans of two office environments and are evaluated using the following criteria: complexity, speed, correctness and precision. A condensed version (does not include complexity comparison) of their results are shown in Figure 2.8 . The presented results are from one of the environments, that has a map size of 80 m x 50 m. To collect the measures, a ground robot was used, equipped with two laser sensors. From the 11 candidates shown on the figure, 6 of them are the selected algorithms combined with a simple cluster algorithm that filters largely noised measurement points and coarsely divides a raw scan into clusters of continuous scan points. The other five candidates are the basic version of the algorithms.

For the system to be developed in this work, the priority should be that the algorithm needs to be fast enough to be run in real-time as the drone is flying, even if it means giving up on high precision. Based on those two metrics the Split-Merge algorithm is the clear winner, as it is the fastest of all the candidates and even with very good precision and correctness metrics when comparing to the other candidates. Therefore, this algorithm will now be discussed in detail.



Algorithm	Speed	N.Lines	Correctness		Precision	
			TruePos	FalsePos	$\sigma_{\Delta r}$	$\sigma_{\Delta \alpha}$
	[Hz]		[%]	[%]	[cm]	[deg]
Split-Merge + Clustering	1780	614	83.9	7.2	1.76	0.69
Incremental	469	536	76.0	3.7	1.68	0.64
Incremental + Clustering	772	549	77.6	4.0	1.70	0.74
Line Regression	457	565	75.3	9.6	1.75	0.68
LR + Clustering	502	556	75.6	7.7	1.75	0.72
RANSAC	20	725	76.0	28.8	1.60	0.69
RANSAC + Clustering	91	523	70.0	9.2	1.24	0.57
Hough Transform	6.6	368	84.1	36.0	1.55	0.68
HT + Clustering	7.6	463	80.6	12.5	1.51	0.67
EM	0.2	893	74.4	43.4	1.86	0.83
EM + Clustering	0.2	646	77.5	18.6	1.46	0.72

Figure 2.8 - Comparison between different line extraction algorithms [17]

Split-and-Merge

Split-and-Merge is one of the most popular line extraction algorithms and originates from the computer vision field [18]. It has been widely studied and used in robotics research [19]. It is, for the most part, a recursive procedure of fitting a segment in a dataset and splitting that segment into two. After that, two segments may be merged if they are close / collinear enough. This is better described in Algorithm 1. In the implementation used in the comparison study [17], the authors used a least-squares method for line fitting (step 2). Another common option is to simply connect the first and last points of the set [19]. In that case, the algorithm is usually referred to as *Iterative-End-Point-Fit* instead. However, the promising performance of the algorithm still needed to be tested on a real-time application.

Algorithm 1 - Split-and-Merge

```
1. Initial: set $\boldsymbol{S}$ contains all points
2. Fit a line to points in set $\boldsymbol{S}$
3. Find point $\boldsymbol{P}$ with maximum distance $\boldsymbol{d_p}$ to the line
4. If $\boldsymbol{d_p}$ is greater than a threshold, split $\boldsymbol{S}$ at $\boldsymbol{P}$ into $\boldsymbol{S_1}$ and $\boldsymbol{S_2}$ and, for each set, repeat from 2.
5. When all sets (segments) in $\boldsymbol{S}$ have been checked, if two consecutive segments are close /
   collinear enough, fit a line in the common set and find a point with maximum distance $\boldsymbol{d}$ to the line
6. If $\boldsymbol{d}$ is less than a threshold, merge both segments
```

2.4.2. Occupancy grid mapping

Occupancy grid mapping algorithms are a common solution in robotics for the problem of generating maps from noisy sensor measurements, when the robot pose is known. They were first proposed in [20] and are explained in detail in Chapter 9 of this book [21]. The basic idea is to represent the map as an evenly spaced grid of binary random variables, each representing the presence (or absence) of an obstacle at that location in the environment. The most common maps are 2D floor plan maps, which describe a 2D slice of the 3D world. It is very important to consider the coarseness of the grid, as it generally represents a tradeoff between accuracy (when the grid is finer) and computational efficiency (when the grid is coarser).

The goal of any occupancy grid algorithm is to compute the approximate posterior probability for these random variables, given the data:

$$p(m \mid z_{1:t}, x_{1:t}) \tag{2.1}$$

where m is the map, $z_{1:t}$ is the set of measurements from time 1 to t, and $x_{1:t}$ is the set of robot poses from time 1 to t. However, if a map contains, for example, 10,000 grid cells, then the number of possible maps that can be represented by this gridding is $2^{10{,}000}$. Thus, calculating a posterior probability for all maps is infeasible. Instead, the standard approach is to break the problem down into smaller problems of estimating

$$p(m_i \mid z_{1:t}, x_{1:t}) \tag{2.2}$$

where m_i is a grid cell with index i. This decomposition is convenient but does not allow representing dependencies among neighboring cells. Instead, the posterior over maps is approximated as the product of its marginals:

$$p(m \mid z_{1:t}, x_{1:t}) = \prod_i p(m_i \mid z_{1:t}, x_{1:t}) \tag{2.3}$$

In order to represent the probability of each grid cell being occupied, the log-odds representation is used. The probability is easily recovered by normalizing the representation.

In a simplistic implementation of this algorithm that is described in the book, it goes through all grid cells and updates those that fall into the sensor cone of the measurement. For those where it does, it updates the occupancy probability of occupied and free cells, according to a model specific to the range finder sensor used. This model assigns to cells within the sensor cone whose range is close to the measured range an occupancy value of $log_{occupied}$ and to cells within the sensor cone whose range is less than the measured range an occupancy value of log_{free}. This is visually represented in Figure 2.9. In Figure 2.10 an occupancy grid map and its respective floor plan are represented. The map was generated using the algorithm and sensor model presented. The grey-level in the occupancy map indicates the posterior of occupancy, which corresponds to an "unknown" state. The darker a grid cell is, the more likely it is to be occupied. While occupancy maps are inherently probabilistic, they tend to quickly converge to estimates that are close to the two extreme posteriors, 1 and 0, which results in a map not taking too long to be generated (this one was gathered in a few minutes). In comparison between the learned map and the floor plan, the occupancy grid map shows all major structural elements and obstacles as they were visible at the height of the laser.

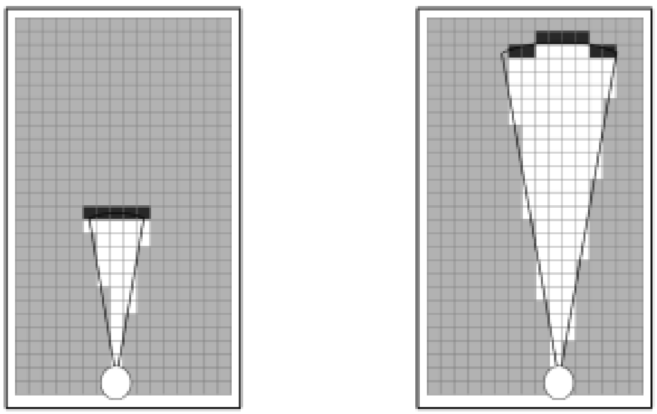


Figure 2.9 - A range finder sensor model for two different measurements [21]. The darkness of each grid cell corresponds to the likelihood of occupancy.

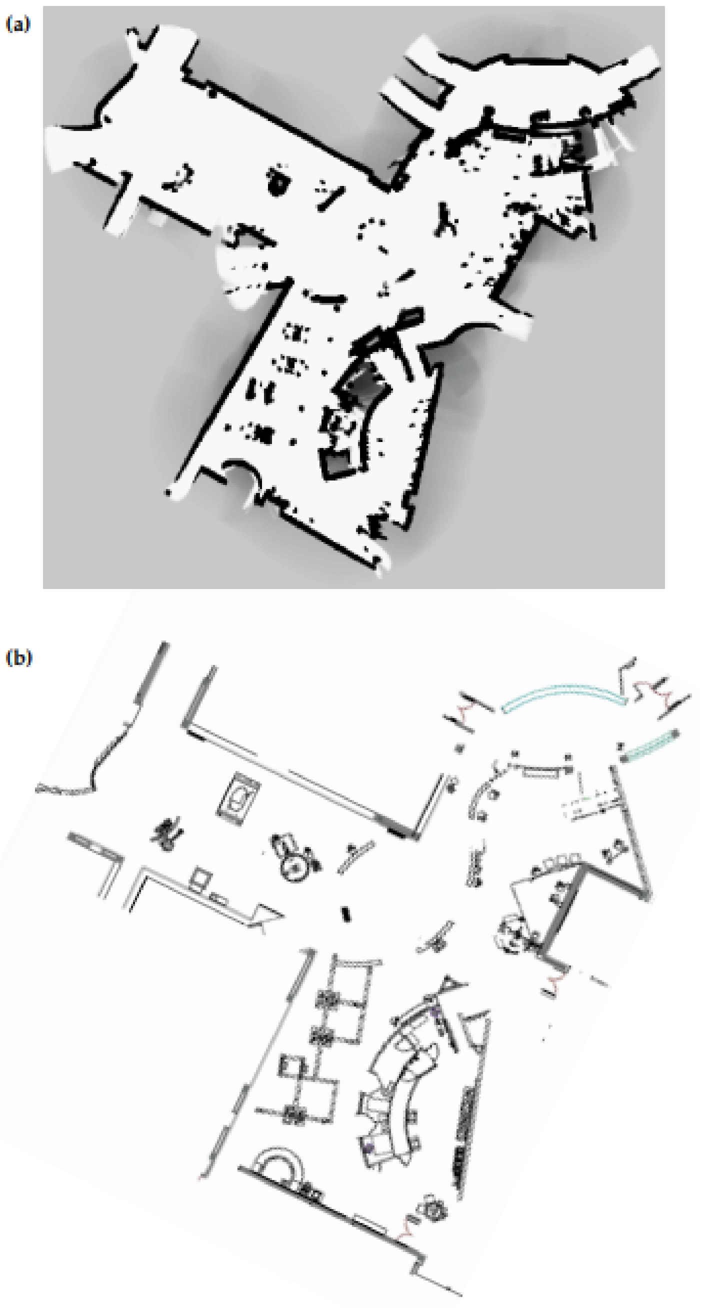

(a)
(b)

Figure 2.10 - (a) Occupancy grid map and (b) architectural floor plan of a large open space [21].

2.5. Exploration

Exploration is closely related to the mapping problem, previously described in 2.4. It refers to a robot ability to autonomously decide where it needs to go in order to maximize information gain of the environment. The robot can also be provided with in-map goals to where it needs to navigate, depending on its current mission. This functionality should be provided by the API. It will be especially useful during laboratory guide 2, where students will need to program autonomous behavior in their drone.

2.5.1. Frontier-based exploration

Frontier-based exploration is a very common approach to robot exploration. It is a simple strategy developed for exploring complex environments such as indoor offices or laboratories. It was first proposed in [6] and is based on the concept of frontiers, which the author identifies as "regions on the border between space known to be open and unexplored space". The robot is an iterative process of detecting a frontier, moving there so it can see the new unexplored space, and add the new information to the current map. The updated map will have more discovered space and frontiers will be pushed back to a more distant location, where the robot will now navigate to. With each new iteration the robot's knowledge of the world increases until eventually all the accessible space in the world is explored.

To represent the world, the same author uses evidence grids, also known as occupancy grids, described in 2.4.2. After the grid has been constructed, each cell is classified by comparing its occupancy probability to the prior probability previously assigned to all cells and is placed into one of three categories:

- **open**: occupancy probability < prior probability
- **unknown**: occupancy probability = prior probability
- **occupied**: occupancy probability > prior probability

A value of 0.5 for prior probability was used in the experiments described in this paper. The process of detecting frontiers in the occupancy grid described by the author goes as follows: "Any open cell adjacent to an unknown cell is labeled a frontier edge cell. Adjacent edge cells are grouped into frontier regions. Any frontier region above a certain minimum size (roughly the size of the robot) is considered a frontier". Finally, in order for the robot to have goal to navigate to, the centroids of each region are calculated. The closest centroid to the robot position is the assumed goal. The various steps of this process can be observed in Figure 2.11. Cells representing open space are represented by whitespace. Cells representing occupied space are represented by black circles. Cells representing unknown territory are represented by small dots. In Figure 2.11 (a), the evidence grid built by a robot in a hallway adjacent to two open doors can be seen. Figure 2.11 (b) shows the frontier edge segments detected in that grid. Finally, Figure 2.11 (c) shows the regions that are larger than the minimum frontier size. The centroid of each region is marked by crosshairs. Frontier 0 and frontier 1 correspond to open doorways, while frontier 2 is the unexplored hallway. While navigating, there is a path planner that attempts to take the shortest obstacle-free path to the cell that contains the goal, using a depth-first search on the grid. The robot also implements reactive collision avoidance. As the author points out, in a dynamic environment where people may be walking by, the robot cannot trust solely on its map, since fast changes will most likely not update the map in time. When the robot reaches a goal, it performs a 360º sensor sweep in order to add as much information as possible. If at some point during navigation the robot is unable to make progress towards its destination for a certain amount of time, it will determine that the destination in inaccessible and add it to the list of inaccessible frontiers.

In the experiments conducted in this paper, a ground robot was used, equipped with a laser rangefinder, sixteen sonar sensors and sixteen infrared sensors. Only the forward-facing sensors are used to build the map, while all sensors are used to perform collision avoidance. All the computation for frontier-based exploration is offloaded to an external workstation, which communicates with the robot via a radio ethernet. In one of the experiments, done in a cluttered laboratory environment, the robot took half an hour to map a 13,7 m x 7,6 m room.

This is a promising approach to autonomous exploration and mapping, however, implementations using air vehicles that do not support the amount of sensing payload used in this experiment should be tested. It is also unfeasible for students to take half an hour to map a room during laboratory classes, even if laboratories are not as big as the one in the experiments.

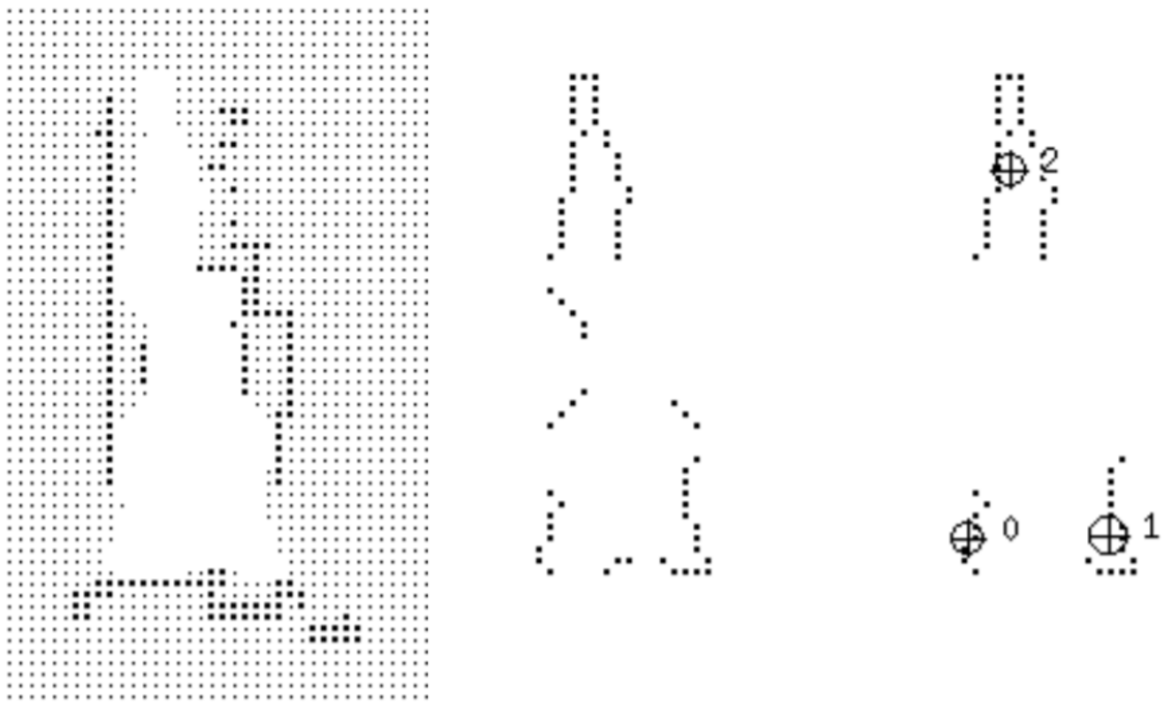


Figure 2.11 - Frontier detection [6]: (a) evidence grid, (b) frontier edge segments, (c) frontier regions

2.6. Swarming

Swarming refers to a robot ability to work together with other robots to achieve a common goal. An individual unit must be cheap, with low resource demand and high energy efficiency, in order to allow the swarm to be scalable. This mechanism is inspired in biological behaviors observed in the nature. When designing swarm-based system architectures, different approaches can be taken into consideration. The most common architectures make use of either a central ground station that is coordinating all the drones or a communication network that links all the drones in the swarm.

- **Ground Control Station (GCS) based architecture**

This infrastructure-based architecture consists of a ground control station that receives telemetry information from all drones in the swarm and sends commands back to each robot individually. In some cases, the GCS communicates back to individual drones in real time, sending commands to the flight controllers on board of each UAV. In other cases, the drone firmware is pre-programmed, and the GCS is simply used to monitor the system. These UAV swarms are considered to be semi-autonomous as they still require direction from a central control to complete an assigned operation. Some commonly used and readily available GCS software that contain basic infrastructure-based swarm capabilities are Ardupilot[3] and QGroundControl[4]. One advantage of this type of architecture is that computations can be conducted in real time in the ground control station via a higher performance computer. The main disadvantage of this architecture is its dependency upon the ground station, with no redundancy. In the event of an attack or failure to any operation of the GCS, the operability of the entire swarm is compromised. Additionally, due to the payload weight limits of small drones, the hardware necessary to establish reliable communication channel with an infrastructure may be limited.

- **Flying ad-hoc network (FANET) architecture**

In this architecture the main way of communicating is drone-to-drone, but at least one of them must be connected to a ground base station or satellite. In a wireless ad-hoc network no routers or access points are needed. Instead, nodes are dynamically assigned and reassigned based on dynamic routing algorithms. Direct communication between UAVs forces distributed decision making and provides built in redundancy as the entire swarm is not dependent on the decision coming from the GCS. Although this approach increases the range from which a drone can operate from the GCS, it relies on the on-board hardware that, as mentioned already, will be limited.

In another paper, a researcher from Czech Technical University introduces a different approach for stabilization and navigation of MAV swarms along a predefined path through an environment with obstacles [10]. The architecture excludes any drone-to-drone communication. Instead, the relative localization system uses monocular cameras carried by all drones and simple visual patterns attached on all MAVs for estimation of their mutual positions. This setup provides information on close proximity of each MAV in a similar way as it is done in swarms of animals in nature, which is pointed out by the

[3] Ardupilot: https://ardupilot.org/. Last Accessed: 12 September 2020
[4] QGroundControl: http://qgroundcontrol.com/. Last Accessed: 12 September 2020

author to be important since swarming behavior seen in nature is very similar to the behavior that is required by MAV swarm applications.

This approach is interesting because it refers to a 3rd architecture, where the drones not only are not integrated in a communication network, but also do not rely on ground control station for coordination. Drones are completely unaware of others and only sense each other when they need to (if they are close or about to collide). There are generally few cases where this approach is reliable enough to be considered. Although this offers the greatest degree of autonomy possible to the MAVs, in real applications the need for drones to communicate some events to each other, directly or indirectly, cannot be discarded. For example, if a small swarm has a mission and they need to divide to conqueror (to perform different tasks, or the same task in different rooms, etc.) it's mandatory that all the drones are informed of their co-worker's progress. Besides that, LOS is required so that the drone camera can see other drones' markers.

2.7. Discussion

All the presented system involving MAVs in research and education were very useful for this work, in some way or another. Just like the SLIM [2] system, this work also aims to enable a big diversity of use cases to be implemented, namely through the laboratory guides. In [4], a system to fly autonomously in complex and unknown environments using a GCS architecture was looked at. This work was important to help understand the lower-level modules necessary to produce a somewhat accurate position estimate. In the system presented in next chapter, these modules are provided off-the-shelf in the drone's default firmware build. Finally, some research and education systems involving the Crazyflie were also presented. They showcase the platform's potential to run swarming architectures and build high-level applications. All in all, these systems helped to converge into the topics to be addressed in laboratory classes, such as Mapping, Exploration and Swarming (later simplified into drone-to-drone communication).

Most of the presented techniques and technologies for indoor localization assume one or more transmitters that communicate a signal to one or more receivers. Most of the time, this implies that this set of transmitters (or receivers) that need to be previously installed in the room, which does not promote easy deployability. While this does not mean that a mobile robot system would not benefit from that kind of setup, especially in a scenario where multiple drones need to be aware of each other, a more device-centered approach needs to be taken, where most of the hardware required is in the robot. In doing so, there is a tradeoff between localization accuracy and deployability,

Finally, some research is done on topics addressed in the laboratory guides, with the goal to provide and high-level implementation through the developed API. In particular, the occupancy grid mapping and frontier-based exploration algorithms were implemented, as mapping and exploration were not provided off-the-shelf by the Crazyflie platform.

3. System Architecture

In this chapter, the system architecture of this project is presented. This is the system that will be used by each group at laboratory classes. It comprehends a Crazyflie 2.1 quadcopter, a ground control station and a python[5] API. Each component will be described over the next sections. In the last section some design choices are discussed, including the choice of the Crazyflie platform.

The Crazyflie is a small and versatile MAV developed with research and education purposes in mind. *Bitcraze AB* [22], the company that develops and manufactures the Crazyflie, also maintains a wide ecosystem of expansion decks, clients and development tools that enable rapid development, flexibility and ease of use. In addition, all their projects are open source with extensive documentation available. The system that will be used in laboratory context has three main components, the Crazyflie drone, a corresponding ground control station, which is a computer enabled by a Crazyradio PA USB dongle, and a python API, that is extending the functionality of the python library *cflib* [23], made available by *Bitcraze*. It provides a full-fledged solution that can be easily deployed in a laboratory and allows students to focus on software development while having to respect the constraints and challenges adjacent to controlling a flying robot. The architecture is depicted in Figure 3.1.

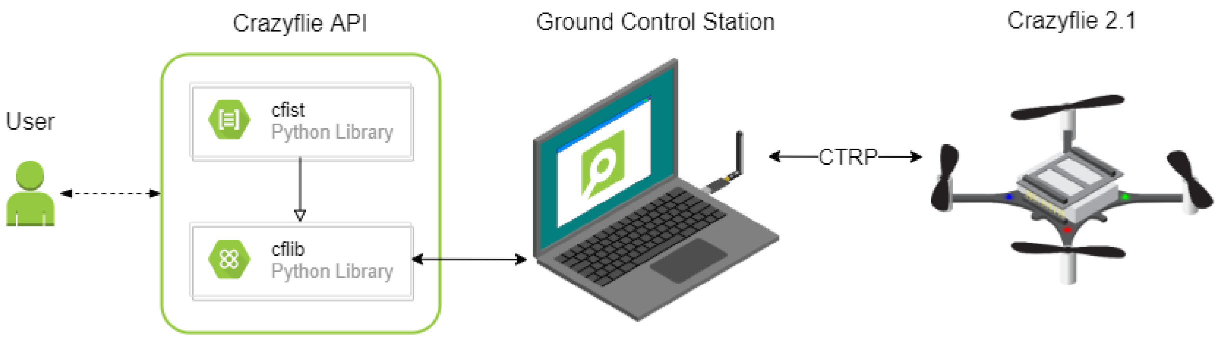


Figure 3.1 - Solution architecture

3.1. Crazyflie

At the time of writing this thesis, the Crazyflie 2.1 (in Figure 3.3) was the latest iteration of the drone and that was the version used in this system. Its board contains an EEPROM memory for storing configuration parameters and a 10-DOF IMU with accelerometer, gyroscope, magnetometer and a high precision pressure sensor. The MAV is also equipped with low latency/long-range radio and Bluetooth LE, which gives the user the option of either downloading the official app and use a mobile device as a controller or, in combination with the Crazyradio PA, flying with a game controller. This is the fastest way to start flying right out of the box, but it's not how students will be controlling it, since the main goal is to make it as autonomous as safely possible. The firmware of the drone is written in C and can be easily modified and flashed over the radio. The drone weighs only 27g and it's so small

[5] Python: https://www.python.org/. Last Accessed: 9 September 2020

that it fits in the palm of a hand. Despite its size it is designed to be durable, as it will, in most cases, remain intact in minor crashes and break at the cheapest components, like propellers and motor mounts, in the event of a major accident (as verified during the development of this system). These characteristics make it ideal for flying indoors. Although small, the four 7mm coreless DC-motors in the Crazyflie grant it a maximum take-off weight of 42g, which enables it to carry multiple expansion decks for extra capabilities in sensing, positioning or visualization. There is an extensive range of decks available, but the platform is also designed to make it easy to design and add custom decks, enabling the user to use sensors and other devices on the platform. From the expansion interface the user can access buses such as UART, I2C and SPI as well as PWM, analog in/out and GPIO. In the system presented in this work, two expansion decks will be used. They will be described later in this section.

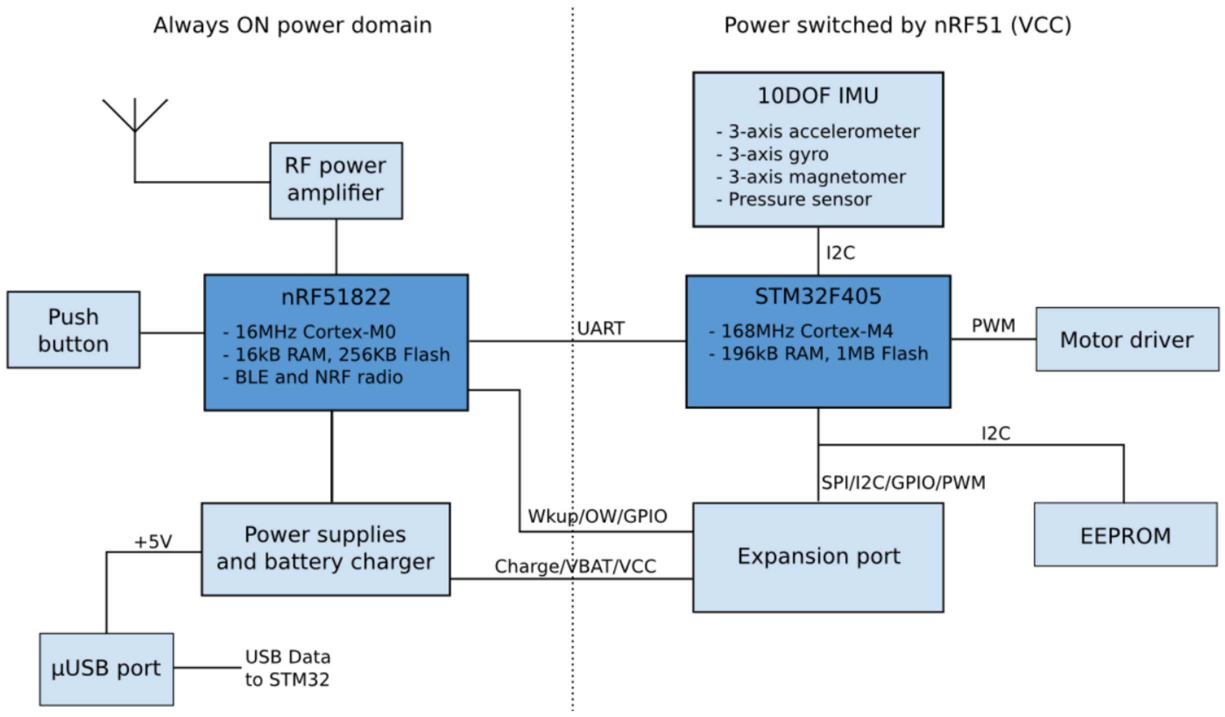


Figure 3.2 - Crazyflie hardware architecture [22]

The Crazyflie 2.1 features dual-MCU architecture, as depicted in Figure 3.2. The first microcontroller, the NRF51, handles radio communication and power management, specifically: ON/OFF logic, enabling power to the rest of the system (STM32, sensors and expansion board), battery charging management and voltage measurement, master radio bootloader, radio and BLE communication and detect installed expansion boards. The second microcontroller, the STM32F405, handles all the heavy work, including sensor reading, motor control, flight control, telemetry (including the battery voltage) and everything that is user developed in the firmware.

Table 4 - Crazyflie 2.1 specifications [24]

Mechanical Specifications	
Take-off weight	27 g
Max recommended payload weight	15 g
Size (W x H x D)	92 x 92 x 29 mm
Radio Specifications	
2.4GHz ISM band radio	
Increased range with 20 dBm radio amplifier, 1 km range LOS with Crazyradio PA (environmentally dependent)	
BLE support with iOS and Android clients available	
Dual antenna support with both on board chip antenna and U.FL connector	
Microcontrollers	
STM32F405 main application MCU (Cortex-M4, 168MHz, 192kb SRAM, 1Mb flash)	
nRF51822 radio and power management MCU (Cortex-M0, 32Mhz, 16kb SRAM, 128kb flash)	
Micro USB connector	
On-board LiPo charger with 100mA, 500mA and 980mA modes available	
Full speed USB device interface	
Partial USB OTG capability (USB OTG present but no 5V output)	
8KB EEPROM	
IMU	
3 axis accelerometer / gyroscope (BMI088)	
High precision pressure sensor (BMP388)	
Battery	
Flight time	7 minutes
Charging time	40 minutes
Expansion Connectors	
VCC (3.0V, max 100mA)	
GND	
VCOM (unregulated VBAT or VUSB, max 1A)	
VUSB (both for input and output)	
I2C (400kHz)	
SPI	
2 x UART	
4 x GPIO/CS for SPI	
1 x wire bus for expansion identification	
2 x GPIO connected to nRF51	

All the specifications presented so far are related to a Crazyflie 2.1 drone, without any expansion decks attached. These specifications and others are collected in Table 4. As already mention, since this work cares deeply about autonomy, the drone will need two decks attached that will help achieve a considerable degree of autonomy. The first, the flow deck v2, is attached on the bottom of the drone and gives the ability of performing relative localization. The deck achieves this using two sensors. First, a PMW3901 optical flow sensor that measures movements relative to the floor. Internally, it uses a low-resolution camera and predictive algorithms that try to detect motion of surfaces. The second sensor is a VL53L1x TOF sensor that measures the distance to the ground with high precision. This is a laser-ranging sensor that can accurately measure distances up to 4 m with a ranging frequency up to 50Hz. The flow deck gives much more control over the drone, as it not only can now be pre-programmed to fly specific distances in any direction but also greatly improves overall flying stability. It plays a key role in pose estimation. Usually, pose estimation using only odometry sensors (dead reckoning) is too unreliable to be considered, as the relative position will drift too much when considering only accelerometer or gyroscope sensors. The flow deck greatly improves this estimation process and allows dead reckoning to be considered in systems that require location updates, but do not need highly accurate estimations, like this one. The second deck it is called Multi-ranger. It is attached to the top of the drone and adds the ability to detect obstacles around the Crazyflie. It contains 5 VL53L1x TOF sensors (the same as in the flow deck) that will measure distances in the directions front, back, left, right and up. This deck is essential to perform collision avoidance or work on environment-aware problems like mapping a room. With these two decks, the Crazyflie can now be an interactive autonomous platform. This setup (disassembled) can be seen in Figure 3.3.

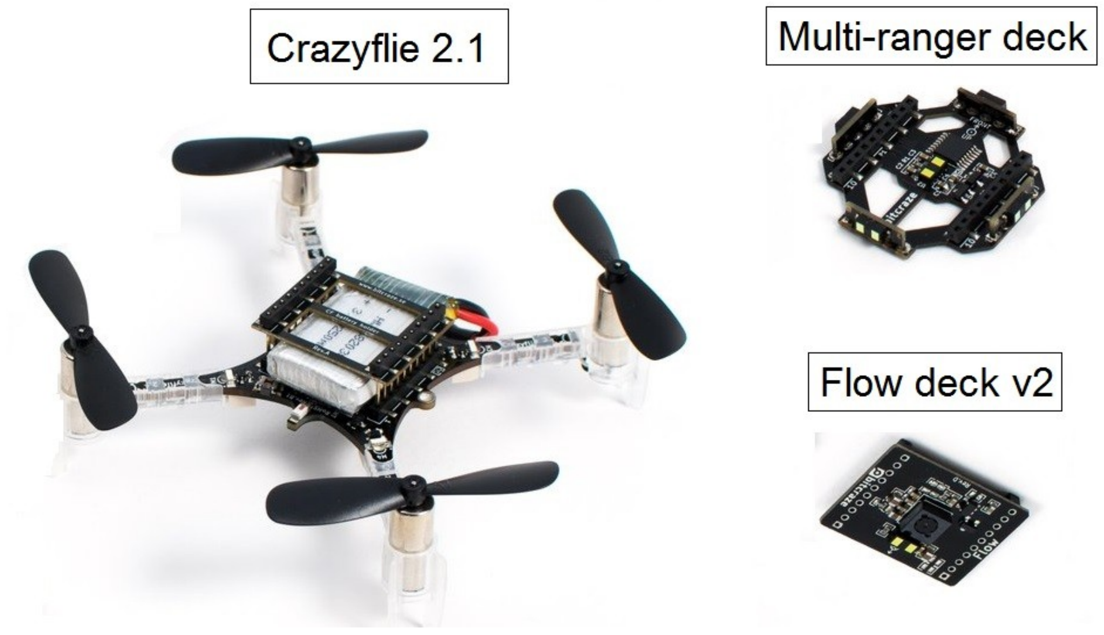


Figure 3.3 - Crazyflie 2.1 drone and expansion decks

3.2. Ground Control Station

A ground control station (GCS) is a land-based infrastructure that has the necessary hardware and software for human control of UAVs. It acts as a middleman, in the sense that all commands to the drone and all readings from it, go through the GCS, to the human pilot. In the context of this work, a GCS will be a laboratory computer or a student laptop, plugged with a USB dongle called Crazyradio PA. This dongle relays the CTRP (Crazy Real Time Protocol) from the python library to and from the

Crazyflie. The CTRP is a high-level communication protocol developed by *Bitcraze* to send and receive data in either direction, but in most cases the communication will be driven from the host, the CGS. In this protocol, each packet has a 1-byte header and can carry up to 31 bytes data payload. The header field has the following layout: Bits 0-1 define the channel, which identifies the sub-task/functionality. Bits 2-3 define the link, which is reserved for future use and bits 4-7 define the port, which identifies the functionality or task that is associated with the message. For instance, there is a port for sending control set-points, one for accessing non-volatile memory, one for packets related with localization, etc. CRTP is designed to be state-less, so there's no handshaking procedure that is needed. Any command can be sent at any time, but for some logging/parameter/memory commands the TOC (table of contents, in the drone firmware, that contains variables that a user might want to read and/or write) needs to be downloaded in order for the host to be able to send the correct information. For this reason, the python side is responsible for downloading the log/param/mem TOC at connect, in order to be able to use all the functionality.

As for the Crazyradio PA hardware, it is based on the nRF24LU1+ from Nordic Semiconductor. It features a 20dBm power amplifier, LNA and comes pre-programmed with Crazyflie compatible firmware. The power amplifier (PA) boosts the range up to 1 km LOS together with the Crazyflie. The Crazyradio itself had its firmware written from scratch and is also open source and fully modifiable, for other systems that may require longer range than Wi-Fi and doesn't have the same requirements for bandwidth. See electrical and radio specifications in Table 5.

Table 5 - Crazyradio PA specifications [24]

Based on nRF24LU1+ chip
8051 MCU at up to 16MHz with 32Kb flash and 2Kb SRAM
2.4GHz ISM band radio
USB device peripheral
125 radio channels
2Mbps, 1Mbps and 250Kps communication data-rate
Sends and receives data packets of up to 32 bytes payload
Automatically handles addresses and packet ack
Hardware SPI and UART
Compatible with “Enhanced ShockBurst” protocol from Nordic Semiconductor
Radio specifications
20dBm output power (100mW)
Low Noise Amplifier (LNA)
RP-SMA connector
Hardware support for PPM input
Up to 13V input power
GND
PPM
SPI/UART

It is also important to mention that, when flying using the python API, the Crazyflie relies on constant communication with the GCS. The GCS is the one running the application and has the obligation to continuously send control commands to keep the drone flying. The drone will autonomously kill its motors if it stops detecting the radio signal from the Crazyradio PA. Such behavior is implemented by default in the firmware, for safety reasons, and it's why the drone is not considered “fully autonomous” as it requires constant communication with the host. In order to grant higher degree of autonomy, one could remove the GCS and python API entirely from the architecture and do all the programming in firmware, however there are two big disadvantages that invalidate this option. First, because of usability. The point of this work is trying to simplify a complex system into various problems that an IT student could solve in a class environment. Programming directly in the drone’s firmware would require much deeper knowledge of how the firmware is structured and working with a programming language that is less user-friendly than python, C. And second, because of performance reasons. Having a GCS always ready allows the drone to offload computing power when performing heavy tasks, which is especially relevant in such a small and low-cost device that, consequently, has limited computational power, storage and energy.

3.3. Crazyflie API

The Crazyflie API is the gateway intended for users to interact with the Crazyflie, which assumes they have some basic knowledge of the python programming language. *Bitcraze* maintains a python library called *cflib* [23], which is the main connection point for programs and scripts to communicate with the drone. The version used in this system is 0.1.11. This library contains all the core functionality needed to implement a simple semi-autonomous mission:

- How to connect to a Crazyflie using an URI that identifies a communication link.
- Setting up logging configurations that will request the drone firmware to send specific variables at a predefined time interval (in ms) to the GCS, like a reading from a sensor.
- Read and set parameters on the drone (they differ from the logging as the variable is not changed by the Crazyflie but by the client and is not read periodically).
- Sending control set-point commands.

However, the laboratory guides will naturally start to introduce slightly more complex problems over time. Sometimes, those problems may require knowledge that is outside of what is intended for computer engineering students to know. This was the case with the problem of building a floor plant, a 2D representation of the room where the drone is flying, introduced in laboratory guide 2. Other times it was just necessary to enforce some common rules that all groups should obey and agree, mainly for safety reasons. This was particularly important in laboratory guide 3, where drones from multiple groups share the same airspace and must navigate in the same airway, as a functional, yet very basic, intelligent transportation system (ITS). These two factors were the main motivation to develop *cfist*, a python library that extends the functionality of the *cflib*, with everything that students would need to solve the exercises proposed in the laboratory guides, which includes:

- Building a floor plan, using an occupancy grid mapping algorithm.
- Running and developing a custom autopilot, using frontier-based exploration.
- Detecting other Crazyflies, using drone-to-drone communication.
- Traffic Management.

This functionality is explained in detail in chapter 4, as well as how is it intended to be used. As for the *cflib*, is still the only way of communicating with the GCS, which means that students are still expected to use all functionality made available from that library. Together, these libraries make up the API that students will use and learn from to build their first drone applications.

A case could be made about using already existing software frameworks instead, such as ROS (it supports Crazyflie integration[6]), which was used in multiple projects presented in Related Work. However, this would require specific knowledge of the framework, which students are not expected to have in a course from Computer Engineering. On the other hand, python is commonly taught in introductory courses in Computer Science and Computer Engineering. To that end, only minimal python knowledge and the provided educational resources are necessary to start building drone-oriented applications using the presented system. It also has the added benefit of providing a more customizable implementation that anyone can adjust to their specific needs.

3.4. Discussion

The choice of the Crazyflie platform was a result of the work developed during the Master Project subject [1]. Nevertheless, is still worth to point out what makes this system such a good option to learn and develop a drone application.

Firstly, unlike most commercial drones, it offers a lot of control to the user as an off-the-shelf solution. The platform touches all the components of the architecture here presented. Whether it's the drone firmware, the Crazyradio firmware or the *cflib*, everything is open source, well documented and easily modifiable. The same goes for the hardware, since there is an extensive number of sensors and boards that you can add, whether they are from *Bitcraze* or not. Assembling the drones was quick and easy, and the installation process is well documented with a step-by-step guide that goes through how to setup their client and update the latest firmware of the Crazyflie, which really allowed focusing on the problems presented in the next chapter purely as a software challenge.

Secondly, how safe and easy it is to fly this drone indoors. In particular, with the flow deck v2 attached, the drone offers very good performance and control in a very small and durable package. The Crazyflie was designed to break at the cheapest components, which are already included as spare parts. This feature should not be overlooked. It was verified during the development of this work that a simple shadow on the floor can easily fool the flow deck predictive algorithm, and make the drone simply go forward until it crashes. Things like this can be avoided (all laboratory guides contain some safety warnings), but chances are that someone, at some point, will crash a drone. When this

[6] Crazyflie ROS Package: http://wiki.ros.org/crazyflie. Last Accessed: 13 September 2020

happens, it is important to recall that they will withstand most crashes, with only a simple replace of propellers being necessary.

Finally, the low-cost system that is offered, which promotes system scalability. The architecture here presented can be easily replicated by adding more drones as there are groups of students in the classroom.

Swarming architectures were initially considered to be implemented in laboratory guide 3, to promote group cooperation in a collective task. This was later concluded to be unpractical, because a single group, that would be taking too long to complete his share of the task, could delay the task for the whole class. Instead, a different approach was taken, where students implement their own independent system, but it needs to be able to detect drones from other systems. This is explained in detail in section 4.3. Arguably, this may not even be considered swarming because there is no cooperative and collective behavior. Nevertheless, it is important to note that the system here presented can indeed be used to control a swarm of Crazyflies at the same time. This kind of setup has some limitations that someone who is looking to use it should be aware. These and other limitations are described later in section 6.1.

4. Laboratory Projects

In this chapter the implementation of the laboratory projects is described. This includes the guides that were written, the *cfist* library developed and the modified firmware that was flashed to all the drones. The chapter is structured chronologically. The first section includes the first guide along with the developed functionality necessary for that guide, and so on for the remaining guides.

Each guide is planned for a 90 min laboratory class and all follow a similar structure, which includes at least these sections:

- **Goal**
  This tells right away to the students what they will be doing and gives a quick idea of what will be needed to accomplish that task.
- **Crazyflie API**
  This is a briefing of the functionality they will need to use or implement to accomplish the task. For each functionality, it is always included a description of what it does and references to usage examples and to the respective implementation in the API.
- **Safety Warnings**
  There will always be some risk involved. This section includes some practical measures that students must take, before and during flight, to lower that risk and avoid injure themselves and the equipment.
- **Exercise**
  A description of the task to be developed divided into up to 3 smaller exercises.

4.1. Laboratory Guide 1

In this first laboratory guide [25], the goal is to introduce students to the Crazyflie development environment. This includes the installation process of everything needed for all the guides, the run of a demonstration script and the development of their first flight script.

4.1.1. Installation

This is a step-by-step guide on how to install the Crazyflie python API. Students will start by downloading a crazyflie-api folder that includes all the source code, from *cfist* and *cflib*, as well as a “setup.py” script that will allow them to install the API via *pip*[7], along with their dependencies.

The second step is instructions to install python 3 for the most popular operating systems.

After that, instructions on how to set up a virtual python environment with *venv*[8] are given. Virtual environments are highly recommended to use when developing python applications. Essentially, it

[7] Pip package installer: https://pypi.org/project/pip/. Last Accessed: 9 September
[8] venv docs: https://docs.python.org/3/library/venv.html. Last Accessed: 9 September 2020

allows creating a “virtual” isolated python installation and installing packages into that virtual installation, instead of installing globally in the system. Students will install the exact same versions of all the dependencies (including the *cflib*) used during the development of the API. This decision was made to ensure that this API will work fine in future years without being broken by one of its dependencies making major changes in its libraries. However, this creates a big restriction in the system since a project (A) might require version 1.0 of a particular library, while another project (B) requires version 2.0 of the same library, and only one version can be installed in the system. To avoid these dependency conflicts, students are highly encouraged to create a virtual environment in their project directory. The step is still optional, since a student that doesn’t use python outside of this project will have no problem installing the project system-wide.

The next step it’s just the *pip* command to install everything, which it’s the same whether they use *venv* or not.

Next, it is necessary to enable support for the Crazyradio dongle. Once more, instructions for multiple operating systems are presented.

Finally, the system is set for this and future laboratory guides.

4.1.2. Crazyflie API

In this laboratory guide, the main functionality is provided by the *cflib*. Namely, how to connect to the drone, how to send control commands and how to get data from it, like sensor readings from the multi-ranger. Since this was not implemented for this thesis, this functionality will not be explained in detail. Interested readers should refer to the library documentation in [23].

The *cfist* provides one module used in this laboratory though, called *Manpilot*, which will now be described.

4.1.2.A. Manpilot

This module, which stands for Manual Pilot, has the class *KeyboardPilot*, which was created to make it easy to control a Crazyflie using the keyboard of the computer, so that students don’t need to import some 3rd party library themselves, which would most likely result in a lot of repeated code to create the key maps. Instead, the API has a pre-defined key map that students can extend by adding their own callbacks that will be called at a key press that they choose. It is still their responsibility to send the actual command. The *KeyboardPilot* holds a data structure that contains the command ready to be sent but it won’t send it directly. This was intentionally implemented for two main reasons. First, for safety purposes. It is a general implementation practice in the API to never send a command directly to the Crazyflie to avoid that an inexperienced user that is calling a method that he is not sure what it does to compromise the safety of the equipment and people by sending the drone to unpredicted locations. Because of this, users have the responsibility to send all commands themselves. Secondly, it makes it much easier when later, in the laboratory guide 2, they need to send commands that are generated by different sources: *KeyboardPilot* and *Autopilot*. Another recurring practice used a lot in both the *cfist* and *cflib* is the use of context managers. The main idea is that when the user instantiates a class like the *KeyboardPilot* with the *with* keyword, some initialization code can automatically run, as well as clean up code that could be run when the context is exited. In the case of the *KeyboardPilot*, it used to start and stop the observer that is listening for keyboard input.

4.1.3. Demo Flight

Before students start to dabble into the exercise, they will run a demo where they can "push" the Crazyflie around with their hand, as it tries to keep away from anything that comes close to it. This is mainly for the students to see the drone in action and get comfortable using it, get excited about all its possible applications but also learn what all the LED's in the drone mean. The demo fully showcases the drone capabilities, such as its general performance and responsiveness, the Multi-ranger performing collision avoidance and the flow deck providing high flight stability.

4.1.4. Assisted Pilot

Since the installation process and running the demo have already consumed a lot of the class time, it was important that this exercise was simple, mainly to consolidate everything that they have just learned.

The goal of this exercise is to make a "smart" Manual pilot, that is, manually piloting the drone but it will stop itself if the user tries to fly it against an obstacle. To accomplish this, students basically need to merge the *KeyboardPilot* usage example with the demo they just ran. This exercise is also a great follow up to the next laboratory, where they will be switching between manual navigation and autonomous navigation.

4.2. Laboratory Guide 2

This laboratory guide [26] focus on three components: Logging, Mapping and Autopilot. The *log* module is provided by the *cflib* and is one of the main features provided by this library. It is the standard way of reading values from the Crazyflie firmware. In this laboratory guide it will be used to monitor the drone resources and make informed decisions based on that information. This will be briefly explained in the next subsection. The *Mapping* module of the *cfist* will be used by students to build a 2D representation of the room while flying the drone. It is very easy to use; with only a couple lines of code a map can be created and update itself passively when it needs to. The *Autopilot* module, also from the *cfist* library, gives the drone the ability of understanding where it needs to go, solely based on its perception of the world and a pre-programmed algorithm. It will output a command that can be redirected to the Crazyflie instead of using the command from the *KeyboardPilot* as in the previous guide. For the students, this is where most of the work will be focused on.

4.2.1. Crazyflie API

The *log* module allows setting up logging configurations. These are used for getting variables at specified intervals (in ms) from the firmware of the drone to the GCS. Once the log configuration is added to the firmware, the Crazyflie will automatically send back the data at every period. The python side should implement a callback that will be called every time data from that configuration is received. This functionality was already being used to obtain the readings from the multi-ranger, for instance. In this lab, students are told to set up a logging configuration to monitor the battery level, and later, land the drone automatically if they receive critical low values. The two modules implemented in the *cfist* will now be explained.

4.2.1.A. Mapping

This module has two main responsibilities. The implementation of an occupancy grid mapping algorithm, based on an implementation[9] from Chapter 9 of “Probabilistic Robotics” [21], and the drawing functionality of that map, using a 3rd party library.

First, a *Grid* object is initialized by the user, preferably as a context manager, so that the drone can start to automatically update the map with new pose estimates and measurements from the multi-ranger. Due to limitations on the total size of variables allowed in a single log configuration, the pose and the measurements were split into two distinct log configurations, both being sent at 100 ms intervals. This means that when the map is going to update, which is triggered by the reception of new measurements, the pose used in calculations can be up to 100 ms delayed from the real pose the drone was when it sent those measures. However, the drone cannot fly fast enough for that discrepancy to be meaningful.

Just like any occupancy grid algorithm, the goal is to estimate the posterior probability over maps just like described by equation (2.1). The occupancy grid map here implemented is a 2D floor plan. Although the drone operates in a 3D world, it will be flown at a constant height, making the resulting map a 2D slice of the world. If the user of this API wishes to change height during flight, when building the map, there are methods available in the API to clear and reset the map. However, such behavior will not be asked to the students in any laboratory guide.

At its core, the map is represented as a matrix, where each cell contains the log-odds representation of being occupied or not. This representation is accumulative, meaning each time the drone detects the same cell as being occupied the map will represent a higher degree of certainty. The same goes for cells detected as free. There is also another important data structure that stores the (x, y) coordinates that correspond to each cell in the matrix, which is needed to be able to classify a cell as being free or occupied.

As already mentioned, the map will be automatically updated whenever the user receives new multi-ranger measurements from the Crazyflie. With each update, a series of important steps are taken. First, the measurements are pre-processed before being used by the algorithm. The purpose of

[9] Occupancy grid mapping example: https://gist.github.com/superjax/33151f018407244cb61402e094099c1d. Last Accessed: 9 September 2020

this step is to take out measurements that do not belong to the 2D slice representation. Because of the unstable nature of a flying robot, even flying at constant height, depending on the pitch and roll of the drone when the measurements were taken, the obstacle detected could be placed too high or too low of the desired height. In that case those measurements are simply removed. Next, a check to see if the map needs to be expanded is run. This is necessary to ensure that, regardless of where the drone flies, the user can always see the whole room in the map. Next, two auxiliary matrixes are created. One of which contains the angle from the drone to each cell, and another which contains the distance to each cell. Then, for each measure taken from the multi-ranger, it is calculated which cells are at the end of the sensor FOV. Those are considered occupied while the ones in the path to the Crazyflie position are assumed free. To make this calculation, the algorithm considers the sensor FOV with a cone shape, rather than a straight line. Although the multi-ranger returns a single value for each of the four directions, each of the four sensors has also a predefined FOV of 20º. What happens is that sensor returns the shortest measure that it detects in that 20º angle. This makes it weird for the algorithm because we cannot give it the same 20º FOV otherwise it would be assuming too much about the environment that it may or may not have detected, but, on the other hand, updating only the one cell that we know for sure is occupied would take an infeasible amount of time to have the whole room drawn, since the drone would have to fly by every single cell that makes up an obstacle. Because of this, a low FOV value, like 5º, should be used. This is demonstrated later in chapter 5. Finally, the log probabilities are normalized before the matrix is used to draw the map. The normalization will result in the following classification of cells:

- **Occupied:** 1 ≥ occupancy probability > 0.5
- **Unknown:** occupancy probability = 0.5
- **Free:** 0 ≤ occupancy probability < 0.5

The draw functionality is implemented using the *matplotlib*[10] library. This includes the grid image for the map representation and a drone-like figure that will be transformed with each pose update. Using this library will impose a constraint on the user though, as it needs to block the main thread while drawing. This means that whenever students need to send commands to the Crazyflie while using this module, they will need to do it in a separate thread. Fortunately, doing this in python is quite easy. Furthermore, the usage example provided with the API already has this functionality implemented.

Before implementing this algorithm though, a different approach was tried, using a split-and-merge algorithm, as presented in 2.4.1. However, early runs demonstrated that the algorithm was not fast enough to perform in a real-time application, which was critical for this laboratory. Since this algorithm was the fastest of the line extraction algorithms (shown by the results in 2.4.1), a different approach was necessary and occupancy grid mapping proved to be the best solution for this task.

Although the drone is performing localization while the API is mapping, this implementation should not be mistaken with SLAM, as the measures that the drone takes and draws on the map are not being used to further improve position estimation, these two modules have to be looked at separately.

[10] Matplotlib: https://matplotlib.org/. Last Accessed: 1 September 2020

4.2.1.B. Autopilot

The main implementation idea is simple. Each *Autopilot* object needs to have a method *run*, which receives multi-ranger measurements from the Crazyflie and outputs a command ready to be forwarded to the Crazyflie, like the *KeyboardPilot* does. Usually, in Object-oriented programming (OOP), an abstract class is used to enforce these rules in all its subclasses. By default, python does not provide abstract classes. However, it comes with a module which provides the base for defining Abstract Base classes, which is called, *ABC*[11]. While this does not offer the same functionality as in a programming language like *Java*, it is still useful to enforce every implementation of the abstract class *Autopilot*, whether from the user or from within the API, to implement an abstract method *run* which receives a parameter called *measurements*. Unfortunately, parameter types and return types cannot be enforced at the time of implementing this functionality. The base class also offers a simple method to verify if all measurements are being received, since the measurement could return the special value *None*, in case the measurement is too far off the sensor maximum distance or if the sensor is malfunctioning.

One of the predefined subclasses of the *Autopilot* is the *Follower*. Its name comes from the algorithm that it implements, which is a simple wall-follower algorithm. With each call of the *run* method, it will return the command that allows the drone to keep following, at a safe distance, the obstacle that is currently detecting at its left (usually it will be a wall). It will also adjust its velocity dynamically according to how close it is to the obstacle. Just like in the *KeyboardPilot*, the command that is returned contains velocities (in m/s), along the x and y axis, and yaw rate (in degrees/s).

In the module, there is also another type of *Autopilot* called *Explorer*. Its goal will be to conduct a frontier-based exploration, as proposed in [6]. This means that the drone will have the ability to generate an in-map goal where it will try to fly to. This is the central piece of this Autopilot. In order to do this the *Explorer* needs access to the *Grid* object, so it knows which cells are unknown, occupied or free. In the process of generating the goal, the concept of frontiers proposed by Yamauchi plays a key role. Identifying regions of the map which make up the boundary of free space and unknown space is the first step of the goal generating algorithm, better described in Algorithm 2.

Algorithm 2 - generate_goal

```
1. Initial: get normalized map $\boldsymbol{M}$
2. Find all frontier cells in $\boldsymbol{M}$, that are open and adjacent to a unknown cell and put them in $\boldsymbol{F}$
3. If $\boldsymbol{F}$ is empty, end, otherwise continue
4. Group adjacent cells of $\boldsymbol{F}$ into regions $\boldsymbol{R}$. Regions smaller than a threshold are discarded
5. If $\boldsymbol{R}$ is empty, end, otherwise continue
6. For each region $\boldsymbol{r}$ in $\boldsymbol{R}$, generate a corresponding centroid $\boldsymbol{c}_{x,y}$ and put it in $\boldsymbol{C}$
7. Closest centroid $\boldsymbol{c}_{x,y}$ to the last known pose is the goal returned
```

[11] ABC docs: https://docs.python.org/3/library/abc.html. Last Accessed: 9 September 2020

There are also other useful methods that allow, for example, calculating the distance to the current goal or the yaw needed to be facing the goal. One particular method is missing implementation though: the *run* function. This is because this class is intended for students to extend, so that they can use the methods just mentioned to implement their own *run* behavior. More details about this exercise will be given in the next subsection.

4.2.2. Autonomous Exploration

This is the exercise proposed for the second laboratory guide. It is divided into 3 smaller exercises. They focus on Logging, Mapping and Autopilot respectively.

The first exercise asks students to set up their own log configuration to periodically read battery updates from the drone. This is something done multiple times during the implementation of the API to get, for example, measurement and pose updates. As such, there are a lot usage and implementation examples available, some of which are pointed out to students in the guide. To complete this exercise successfully the drone can simply be sitting on a desk or on the ground and so students are told not to take-off, for two reasons. First, because it's safer and if there is a possibility of avoiding risk it should be taken. And second because it will save battery for the most challenging exercises coming next. This task is expected to take 10-15 min for students to complete it.

In the second exercise students will use the *mapping* module while they manually control the drone with the previous *KeyboardPilot*. This is quite simple to do, so far it can be accomplished by just combining the two respective usage examples. The purpose of this is for students to see how the map building works, namely how responsive and customizable it is, before using it together with the *Autopilot* in the next exercise. Additionally, they are asked to incorporate the functionality developed last exercise, so they can trigger an automatic landing of the drone if the battery gets critically low. This showcases one way of granting the drone a little bit more of autonomy, even if it is being manually piloted. The task is expected to take 15-20 min.

The final exercise will be the most challenging and the one that requires the most coding. Students are expected to extend the *Explorer* autopilot and implement their own control algorithm as they follow a series of steps that denotes the behavior that is intended. They will make use of the functions available to the *Explorer* that will, in the end, dictate the path the drone travels until it discovers the whole room. There is still a lot of room for students to implement according to their personal preference as different possible solutions will come out of this exercise, some more efficient than others. For instance, after a goal is generated, the drone will need to travel there. The drone can be simply told to go to those coordinates. However, since no collision avoidance is being done it can easily go wrong. On the other hand, depending on the previous knowledge of the student, shortest path algorithms can be implemented, to take advantage of the already created grid, which knows which nodes are already visited or not. This is of course a more sophisticated solution for the interested student of the field, which is outside of the scope of this work and is not necessary to successfully complete the task. Nevertheless, it was important to give the students that freedom of implementation. For this reason, the duration of this task can vary a lot, but it is estimated to take between 25-35 min.

4.3. Laboratory Guide 3

In this lab [27], each group will be simulating a Drone Delivery Application. Delivery Services are one of the most promising applications for drones and here students will have the opportunity to implement their own system, while being part of a larger intelligent transportation system (ITS), where every group will have to follow basic traffic control rules, like travelling in a common “air highway”, also known as airway. This is of course a simplistic version of the system, which doesn’t have to deal with some of the biggest challenges that real-world systems that are currently being developed (like Amazon Prime Air[12]) have, such as having to fly in very complex environments, like crowded cities, or compliance with the local laws. This is also a simulation because it will be done indoors, in a controlled environment with permanent human monitoring. The behavior of picking up and dropping off a package will also be simulated, although a system that could physically lift a light package using the Crazyflie would be an interesting extension to this project.

When designing such a system, where multiple Crazyflies will be flying at the same time and sharing the same airspace, there is a need to enforce some common traffic rules to help reduce the probability of traffic congestions and collisions between drones, much like common traffic rules in the road greatly reduce car accidents for every driver. One possibility to implement such system would be to link all GCSs in a distributed network, so they could all agree on a common protocol that all drones must follow in order to participate in this ITS. This would also allow every group to easily ask other GCSs for their respective drone’s position, to prevent drone collisions. However, there is a constraint imposed by the hardware involved, as for the time of writing this thesis, the Crazyradio PA does not allow such communication network to be implemented while communicating with the respective Crazyflie as well. Instead, a different approach was taken. On one hand a semi-automated detection mechanism was developed, which allows students to ask their drone if they “see” other drones nearby. This is necessary because since the Crazyflie is such a small drone, the multi-ranger sensors can easily miss other nearby drones. They can then act depending on the estimated distance to the detected drone, as well as who is being detected. There are two sides to the implementation of this functionality. The *Radar* module of the API (described in 4.3.2.A), which exposes the functionality just mentioned to the user, and the firmware of the drone, which had to be modified so that it could start broadcasting messages that other drones could receive and send to its respective GCS. Figure 4.1 illustrates how these parts interact with each other in a typical use case, when the user tries to detect neighboring drones. Each component will be described over the next subsections. On the other hand, there is still the need to enforce the common rules that will decrease the risk of two drones even getting close to each other. This is achieved using the *DDS* module. *DDS* stands for Drone Delivery Service, and besides providing the route that a drone should take when it is travelling to a destination, it also provides basic logistics, like methods to manage locations available for delivery that every application might need. This is explained in detail in 4.3.2.B.

[12] Amazon Prime Air: https://amazon.com/primeair. Last Accessed: 12 September 2020

Globally, this results in an ITS where each group's system is unaware of the other systems until the moment their drones need to avoid each other, since they still need to operate in the same environment. On one hand this allows systems implemented in different ways to be able to live together. On the other hand, by not using a distributed GCS network, the responsibility of enforcing common rules and check for drone collisions falls upon each individual GCS, which is susceptible to implementation error from the students. To decrease this risk, some measures were taken:

- Simply by using the *Radar* module as a context manager, the Crazyflie will automatically start to broadcast location messages, to ensure that no one is penalized by a group that forgets to start disclosing its location;
- The first exercise of this laboratory guide verifies that everyone's Crazyflie can be detected by all groups, before starting to build their application;
- The common behavior that everyone should agree is predefined by the API (in modules *Radar* and *DDS*);
- When implementing recommended behavior, students are provided with pseudocode.

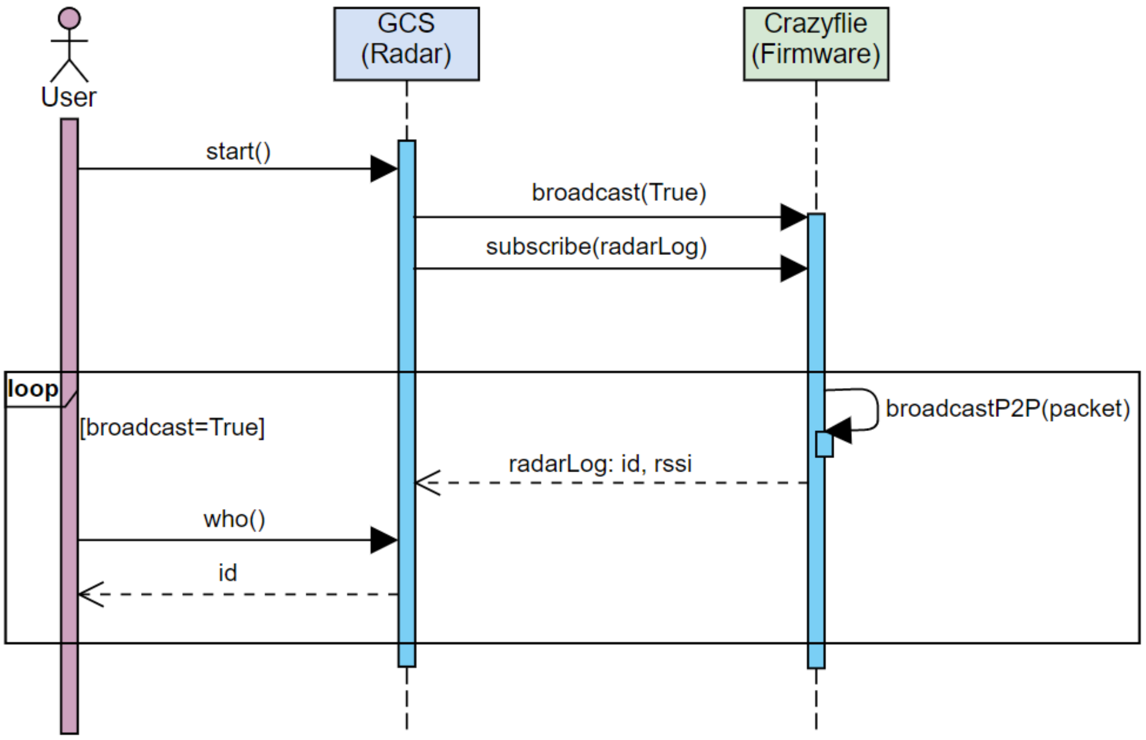


Figure 4.1 - Sequence diagram for detecting a neighboring drone

4.3.1. Modifying the Firmware

In order to have the *Radar* module delivering the intended functionality to the user, the Crazyflie needs to be prepared to start disclosing its location to others as well as reporting if it is detecting anyone. This is something that is not available in the default build of the Crazyflie (v.2020.06) [28]. However, *Bitcraze* provides a Peer to Peer API that helps implementing this functionality in the firmware. At the time of writing this thesis, peer to peer communication on the Crazyflie is still in development but it's already possible to send P2P messages in broadcast mode. After defining an entry-point for the application, a callback is registered that will be called every time a P2P packet is received. Whenever a message is received, two variables are updated; one that stores the packet data, which will contain the ID of the transmitter Crazyflie, and one that stores the RSSI value of that message, which, as discussed in 2.1.2, can be used to make a rough estimate of the distance that the radio signal travelled. These are the variables that the *Radar* module will be able to pull using the *Log* module. When detecting multiple Crazyflie at the same time, these variables will be overridden every time a new drone is detected. However, they will be read by the *Radar* every 100 ms, which means that all drones should be detected by the API quickly enough. It should be noted that, when detecting a large enough number of Crazyflies, it is expected that some detections start to be skipped, and so there is a limit to the number of Crazyflies that a drone can detect at the same time when using the *Radar* module. As demonstrated in the Evaluation chapter, it was only possible to test with a set of 4 Crazyflies detecting each other at the same time, which has proven to be safe. This design also has the problem of the firmware continuing to report the same variables when no one is being detected anymore. To avoid this, a timeout mechanism is implemented that will override these variables with empty values, in case the Crazyflie is 3 seconds without receiving any P2P message.

After registering the callback, we can go to the typical control loop, where there is a *boolean* flag that can only be set by the *Radar* module. This flag is what the Crazyflie calls a parameter. The *Parameter* module works very similarly to the *Log* module, explained in 4.2.1. Instead of retrieving variables at a predefined interval of time, it is possible to change a variable in the firmware from within the python API, using the parameter framework provided by *cflib*. In the firmware, while running the main control loop, if this flag was set by the host, a packet is broadcasted via radio using the method *radiolinkSendP2PPacketBroadcast*. This packet contains the ID of the drone sending the message, which is the last number of the radio address of that drone.

Building and flashing this firmware is a one-time operation that must be done before putting a new Crazyflie in use for laboratory classes, as well as changing the drone's radio address so that every drone in the lab has a unique ID. Such operations should not be by students, but by an admin user instead. To help doing this an admin guide [29] is provided (in README.md file) together with the code to be uploaded.

4.3.2. Crazyflie API

The two modules of the API that students will need in this laboratory guide were already briefly presented. The *Radar* exposes methods that return information about a single drone that its being detected at a particular moment in time, while the *DDS* provides a common interface for every drone delivery application that will be developed by each group to use. The two *cfist* modules will now be explained in detail.

4.3.2.A. Radar

This module provides a simple interface so that user can know if their drone is currently detecting other Crazyflies. The name comes from the fact that we are estimating distance using radio signals. As such, it can be seen as a poor man's implementation of radar in the true sense. As for previous modules of this package, it is intended to be used as a context manager. In this case, when the user creates a *Radar* object it will trigger two operations. First, it will set the set the flag in the firmware that allows the Crazyflie to start broadcasting P2P messages. And second, it will start logging the last detected ID and RSSI values of the Crazyflie. This means that by using the API as provided it is not possible to detect others without letting yourself be detected. This is purposely enforced for safety reasons. The user can then call methods to get this ID, proximity estimation, done based on the RSSI received, and to ask if their drone has priority over the currently detected drone. This a mechanism implemented to resolve conflicts when two drones are in dangerous proximity to each other. Instead of the two drones just stopping any motion and waiting for the human pilot to resolve, they can ask the module which of them has priority over the other, so that the drone with less priority lets the other fly by. The intended behavior is that this drone would land, without disconnecting from its GCS, and wait for clearance, this is, wait until its radar no longer "sees" a drone. Currently, a drone has lower priority over another if its ID is greater. This is the simplest implementation possible because that's all that is needed to avoid traffic congestion. Although this makes it a bit unfair to the group with the greatest ID in the room, unless the laboratory room is very small, these conflicts are expected to happen rarely.

As will be shown in Evaluation chapter, the disadvantages of using RSSI to estimate distances, presented in 2.1.2, will be evident. Localization accuracy is low and is very susceptible to environmental noise. Due to the localization of the radio antennas in the drone, even the drone pose can make the proximity estimate vary. However, as far as it was possible to ascertain, there is no other way for drones to detect each other without relying on external infrastructures, like the GCS network architecture already mentioned. This is also much safer than relying exclusively on the multi-ranger alone for collision detection.

4.3.2.B. DDS

The drone delivery service (DDS) provides a common interface for students to use while implementing their applications. It is essentially a logistics manager, which can provide live information like the status of an on-going deliver, delivery location, etc. One of the most important methods is called *travel* and it returns the command that students should forward to the Crazyflie in order to be flying in the predefined airway. This airway is like an air corridor that defines a space in the room where all drones should preferably fly. As previously discussed, this helps reduce the probability of collisions between drones, since all drones will be flying in the same direction in the airway, but at the same time, will also delimit areas where people can safely walk, without interfering with the system. Implementation wise, this effect is achieved with a simple *Autopilot* object, which will be the same for everyone. When the user calls the method *travel,* the call will be redirected to the autopilot *run* method. The default airway is defined by *Follower* autopilot, which will define a trajectory near the perimeter of the room, but it can be changed using the library methods. The module also allows managing locations which are known to the application. These locations are points in the Cartesian coordinate system, labeled with a name. They are relative the drone’s take-off position which has the special location (0, 0) called “Home”. There are a small set of predefined locations that can be used, if the user doesn’t need custom locations. The existence of these locations also allows the implementation of some easy-to-use primitives like *distance_to_local* or *yaw_to_local* that students can use to build their application. To be able to implement these methods, regular pose updates from the Crazyflie are required. Like has been done in the past, these updates will be required automatically, when creating context. It is also possible to get the current delivery status which is very useful when designing the execution flow of the application.

4.3.3. Drone Delivery Application

As mentioned already, the goal of the exercise is to simulate an automated package delivery from a point A to a point B, while coexisting with similar systems that will be doing the same. Because of the extension of the task at hand, it was divided into 3 exercises.

The goal of the first exercise is just to make sure that all groups understand the basic usage of the *Radar* module, so that when everyone is flying, there is no “stealth” drone, i.e. not sending P2P messages that others can detect. Like the first exercise of the second laboratory guide, this can be done without taking off the ground. The code required to complete this is very similar to the provided usage example. However, since this requires that all groups wait for each other so that everyone can be verified, this can probably take up to 15-25 min.

The second exercise is about the drone-to-drone collision detection mechanism, which will run before each command in the application is sent to the Crazyflie. This is done using the *Radar* module exclusively. This is the behavior that students are told to implement:

a. Check if the drone detects anyone nearby with higher priority;
b. If it does not, proceed to sending setpoint commands;
c. Otherwise, if it is in close proximity, reduce the cruising velocity to 0.1 m/s;
d. Otherwise, if it is in dangerous proximity, land the drone, without disconnecting, and wait for clearance until no drone is close. Then take-off and proceed.

When everyone implements this behavior the chances of drone collisions will be greatly reduced, however, it is prone to a faulty implementation from the students. The degree to which something should be done automatically or manually implemented by students was always something greatly considered during this project, which is a natural challenge of designing an API. Ultimately, this choice allows for different uses of the *Radar* module, which could all be considered equally safe. This specific behavior was chosen because of its simplicity. However, implementing this behavior automatically not only would go against the general rule of not directly sending setpoints to the Crazyflie through the API (as previously mentioned, only users should do that), but also remove an excellent learning opportunity, as this feature is such a core and unique challenge in a system like this. Finally, it is important to consider a scenario where all these safety measures fail. Because there is so many variables at play, collisions and crashes will always be a possibility. When this happens, it is important to recall that the drones are designed to withstand minor accidents without damaging themselves. Unless a group is waiting for another, students will have no opportunity to test this until the next exercises, which means that this should not take more than 15-20 min.

In final exercise, students will implement the main logic for their delivery's application. It should consist on the following route: pick up a package in a first location, drop it off in a second location and head back home to the take-off position and land. Here's the behavior they're told to implement:

a. Get a pick up and a drop off location, for instance, using *the assign_delivery* method from the API;
b. Include the Radar safety check, developed last exercise, before sending any control command to the Crazyflie;
c. Fly in the airway predefined by the API by calling the *travel* method;
d. While traveling, leaving the airway is only allowed when they are close enough to a location;
e. Define the pickup and drop off behavior, like a simple decrease and increase of altitude.

Adding locations that are too far from the airway may result in the drone wondering indefinitely trying to find them. The threshold distance to leave the airway should be kept large enough so they can be detected, but small enough to force everyone using the airway while they are flying. This will require the most coding and test runs. Depending on the student might take between 25-35 min.

5. Evaluation

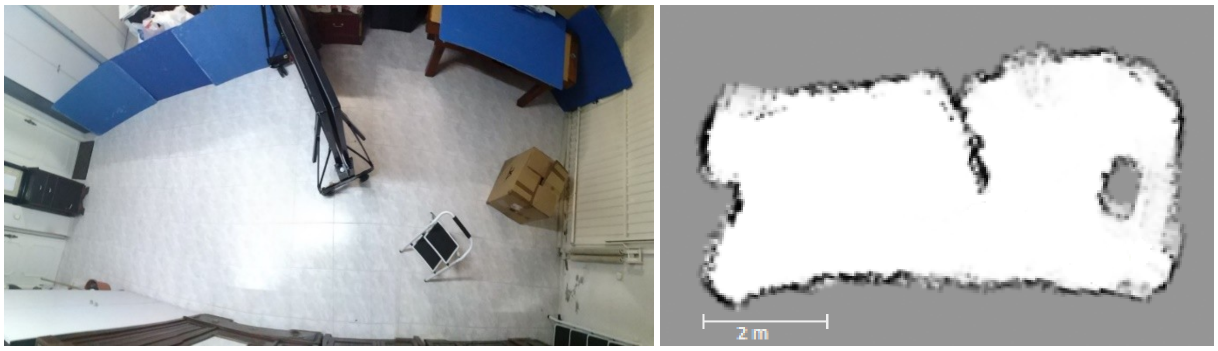
This chapter describes the experiments conducted to evaluate and validate the suggested system. Core features of the system will be tested, and it will be discussed how they help to meet the requirements presented in section 1.2. All tests were performed in an Asus K550J laptop, with an Intel Core i7-4710HQ CPU @2.50GHz and 8,00 GB of RAM. As for software versions, the *cflib* library is v.0.1.11, *cfist* is v.0.0.1 and the firmware is a modified version of v.2020.06.

5.1. Mapping

The performance of the occupancy grid mapping algorithm, which implementation was described in 4.2.1.A, will now be tested. First, the quality of the generated map will be evaluated by manually flying a Crazyflie drone in an indoor environment and comparing it with the ground-truth top view of this room. And second, it will be tested how two of the algorithm parameters can vary the resulting quality of the map as well as how fast it can be generated. These parameters are the cell size of the grid (which define the grid resolution) and the sensor FOV, which as previously explained in 4.2.1.A, will determine approximately how many cells should be assumed occupied when a sensor detects an obstacle.

The testing environment can be seen in Figure 5.1 (a), from where it's possible to distinguish two different zones. A wide-open obstacle-free space (on the left) and a smaller cluttered space (on the right). This is to evaluate how the complexity of the environment affects the quality of the generated map. The space is approximately 6,5 m x 2,5 m. All the experiments described will have the Crazyflie flying at 0,2 m/s.

From the first experiment, the map that resulted from manually flying a Crazyflie around the room for 2'30" minutes is presented in Figure 5.1 (b). A side-by-side comparison shows that the map is fairly accurate, as it is possible to distinguish the general shape of the room. One thing that stands out however, it's how the small stairs, as well as the table legs, are completed ignored. This is because, at the height that the drone was flying, the obstacles are so thin that even if a few cells register an obstacle a few times, most of the time the obstacle is dismissed and so the cell is considered more likely to be free than occupied. As will be demonstrated in the next experiment, it is possible to improve this by increasing the grid resolution (the cell size used in this flight was 6 cm). It is also possible to see some irregularities in the zones where there are gaps instead of continuous walls and closets. In addition, it was possible to see clear performance differences while navigating in the two zones of the room, namely, the clustered zone was mapped much faster because of its smaller size.



(a) (b)

Figure 5.1 - Top view of testing room (a) and occupancy grid map generated during manual flight (b)

In the second experiment, to evaluate how the cell size and FOV would vary the quality of the resulting map, multiple test runs were executed where only one of these parameters was changed. The cell size values tested were 0,02 m, 0,05 m and 0,08 m and FOV values were 2.0º, 5.0º and 10º. These values were chosen because of previous experiments that had demonstrated before that cell sizes inferior to 0.02 would result in maps that *matplotlib* would struggle to update in real-time, due to the huge number of cells that compose the map. On the other end of the spectrum, cell sizes bigger than 0,08 m and FOV values bigger than 10º result in maps too inaccurate to be considered and accentuate noise measurements too much. With each test run is also presented the total number of cells of the resulting maps, the time it took to complete the flight, and the quality evaluation of the map, which was further divided into accuracy and coverage. The last refers to the percentage of terrain that was explored after the flight. Because of the probabilistic nature of occupancy grid mapping algorithms, the longer the Crazyflies is flying the more certain it will be about the occupancy probability of the cells it sees, which, assuming perfect pose estimation, would translate to a more accurate map. For this reason, to ensure accurate test trials, the time taken during each run will be approximately the same.

In the context that this system is intended to be used, is not practical for a map to take a really long time to be constructed just to get it more accurate, as long as the whole room is covered. Taking a long time to map a room will consume too much time from the practical class as well as consuming precious battery time with the drone. Also, there is no real benefit in having a map with very high accuracy versus a map with medium accuracy, as long as students can immediately see that the map they are looking at gives a real representation of the room. This experiment also hopes to get the optimal parameters that better represent this tradeoff, so they can be provided as the "default".

The behavior followed during all test runs was the following:

a. Take-off;
b. Go around the room, without stopping;
c. Land in starting position.

The results are shown in Table 6.

By analyzing the table, the first conclusions we can draw are that, generally speaking, map accuracy increases with finer grids and coverage speed increases with a bigger FOV. This is to be expected, as a smaller cell size will produce a more realistic map and a wider FOV will assume a lot more cells to be occupied, so it will be faster to generate. Secondly, it is also possible to see that, generally speaking, there is a tradeoff between accuracy and coverage. This is because on one hand, smaller cells mean more cells that the drone will need to "see" and on another hand, a wider FOV means that the probability of assuming wrongly occupied cells greatly increases. Third, it is possible to see that using a FOV value as big as 10º always make the map too inaccurate, regardless of cell size. This may indicate that a lower max value should have been chosen. Finally, we can conclude that the optimal parameters to express the quality requirements that were stated before are a FOV of around 5.0º and a cell size that can vary between 0.05 m and 0.08 m. If someone who is looking to extend this system needs a high accuracy representation, it is recommended to decrease the cell size, while maintain a similar FOV value. This will of course require more time to fully map the same area.

There are also other conclusions worth mentioning that cannot be directly interpreted from the table. The first one is that although in theory, map accuracy should improve with time, in practice this was not always the case. This is justified by the fact the MAV uses dead reckoning for position estimation, which will drift over time. When the drone flies over some already visited cells with the new drifted position, the same walls and obstacles will be overlapped. In this experiment this was especially visible when the drone comes back from the cluttered zone. To help prevent this from happening, the floor should contain rich texture patterns and the room should be well-lit. This will improve the performance of the internal algorithm being run in the flow deck v2 (described in 3.1) at tracking the floor and predicting motion.

In addition, unlike the map presented in Figure 5.1 (b), when using cell size of 0,02 m it was possible to see a representation of the table legs and the small stairs in the cluttered zone, although it was a very inaccurate representation that would slowly fade away as the Crazyflie would fly by the same place. Just like in Figure 5.1 (b), regardless of FOV, the map was much faster to build in the cluttered zone.

Table 6 - Occupancy grid mapping performance experimental results

Test Run	**Cell size (m)**	**FOV (º)**	**Total of Cells**	**Time**	**Quality**	
					Accuracy	**Coverage**
1	0,02	2.0	237 220	1'36"	High	Low
2	0,02	5.0	256 280	1'46"	High	Medium
3	0,02	10.0	248 811	1'34"	Low	High
4	0,05	2.0	37 490	1'33"	Medium	Low
5	0,05	5.0	41 001	1'41"	Medium	High
6	0,05	10.0	38 014	1'37"	Low	High
7	0,08	2.0	14 746	1'37"	Low	Low
8	0,08	5.0	15 288	1'37"	Medium	High
9	0,08	10.0	15 476	1'30"	Low	High

5.2. Exploration

Now, the performance of the frontier detection algorithm will be tested. Its implementation was described in 4.2.1.B. This is the algorithm used by students in laboratory guide 2 to calculate a goal in exercise 3 of autonomous exploration. To evaluate the quality of the generated goals, the drone will be flying in the same testing environment with the following behavior:

a. Take-off;
b. Do a 360º scan, to maximize information gained;
c. Run the frontier detection algorithm, which will generate an in-map goal;
d. If a goal is returned, manually fly there and repeat from first step;
e. Otherwise, the map is considered fully discovered. Land.

Figure 5.2 shows the generated occupancy grid map. Generated goals are marked as blue dots and numbered by order of appearance, defining the path traveled. Mark "0" is the take-off position. The presented map was generated in 1'54'' time with a grid resolution of 5 cm per cell and FOV parameter at 4.0º. When analyzing the generated map, there are two metrics here defined to evaluate the algorithm performance. The quantity of goals, in the sense that the ideal map would contain the minimum amount of goals possible that allow to fully discover a room. And the quality of those goals, in the sense that goals should be generated in a position that offers as much information gain as possible. As for quantity, the amount of goals generated was very good. Because of the sensor maximum range is capped at 3 m (for more accuracy), each new goal would be ideally at approximately that distance, to ensure the minimum goals generated and fastest map generation. This is possible to verify by looking at the estimated distance between a goal and the next. As for quality, there are some goals better than others. Goal 1 would ideally be place at the center of the wide zone, however it is place very near the take-off position. This can be attributed to the fact that the scan that was done in position 0 gained very little information because of its cornered position, so the drone didn't have enough information to decide the best position to scan the zone. Goals number 3 and 4 are particularly interesting. After the scan done at 2, the algorithm saw the obstacle in the middle of the cluttered zone and defined two frontier regions, one for each side of the box. Because the goals are in such a good position there was no need to have more than two goals to learn all the map of the cluttered zone.

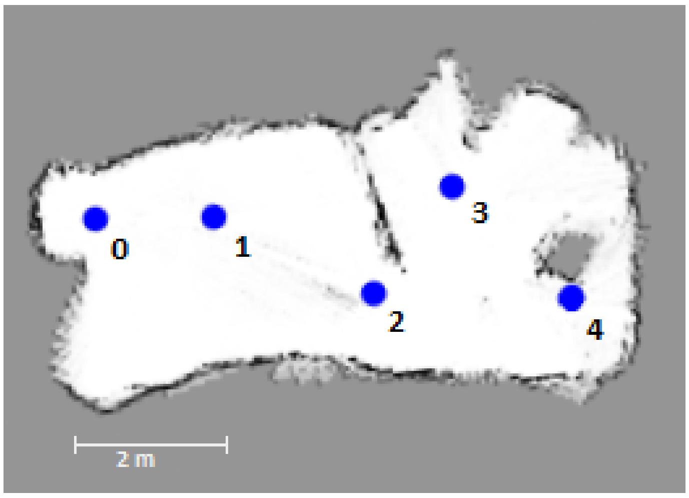


Figure 5.2 - Occupancy grid map generated during exploration. Blue dots are goals generated by frontier detection algorithm.

5.3. Drone-to-drone communication

In this section, multiple requirements are being validated. In a first experiment, only the performance of the RSSI distance estimation will be evaluated. Then the experiment is scaled, to test how many drones can be detected at the same time without compromising other requirements. Safety is also being validated, by evaluating how many collisions is the system able to avoid.

The first experiment consists on having a Crazyflie (A) hovering still, while another drone (B) slowly flies towards it at 0,2 m/s, starting from 3 meters away. When one of the drones detects it is in dangerous proximity with another drone, they both land and the distance between them is measured. Proximity is considered dangerous if the RSSI value returned is bigger than –48 dBm. This value was obtained by trial and error and its validity should also be considered when analyzing these results. This experiment is repeated 8 times to account for RSSI noise. Because the floor in this environment is very reflective, the drones will be flown higher than usual, at 0,5 m, to try to mitigate some noise in the signal propagation. The results are shown in Table 7.

As it is possible to conclude, the accuracy of the distance estimation is quite poor, with values ranging from 23 cm to 197 cm in only 8 trials. This can be justified not only by the natural lack of accuracy of the technology in indoor environments, namely because of the multipath effect, but also because no proper signal propagation model is being used, namely one that makes use of software filters that help to counteract this effect. This is something that should be investigated in future extensions of this work. Nevertheless, at this velocity with the reference RSSI of –48 dBm, it was still possible to automatically prevent the collision in all trials. To this end, no value greater than –48 should be used, as the signal could easily be dismissed. On the other hand, values smaller than –48 dBm can quickly start to be detected in the whole room. This means that the system should not be too conservative either or it will quickly become unusable as a proximity detector. Being hovering still or in motion doesn't appear to have any effect on accuracy.

Table 7 - Drone-to-Drone collision detection experimental results

Test Run	**Distance from A to B (in cm)**	**Detected by**
1	156	A
2	90	B
3	195	B
4	54	A
5	170	B
6	197	A
7	58	A
8	23	B
Mean	117.88	-
Standard Deviation	65.02	-

As previously explained in 4.3.2.A, the current implementation of the Radar only allows a single drone to be detected every 100 ms, which is expected to cause some issues when detecting multiple drones at the same time. This next experiment intends to measure if that compromises the safety of the system. It is important to note that, in practical laboratorial context, encounters between 3 or 4 drones are expected to rarely occur, since most of the time each drone will be flying in the airway predefined by the API.

The experiment is very similar to the one just described, but instead of only one drone (A) hovering, there will be a cluster of 3 drones. The behavior is still the same. Another drone (B) will approach and whoever detects each other first (between A and B) makes both land. The goal is to measure how difficult it is for A to detect B, since it must process the messages coming from the other two drones of the cluster too. The cluster was about 1 m apart in a triangle formation. Since there were only 4 Crazyflie drones available this was the maximum it was possible to scale up the experiment. Results are shown in Table 8.

Contrary to what was expected, detection accuracy not only didn't decrease, it improved a little. The mean shows that Crazyflie B was generally being detected sooner than in the previous experiment. Not only that, but measures taken were also more consistent, as shown by the standard deviation. This may be explained by the signal propagation from the other two drones from the cluster interfering with the reflected weaker waves, coming from the multipath effect, which ultimately results into only the stronger direct signal being received. The fact that the drone A, that's being "shielded" by two other drones, is consistently the first to detect the collision, supports this hypothesis. This would require further testing to be proved, as well as expertise that is not part of this work. As for the previous experiment, all collisions were automatically prevented without the need for manual intervention.

Table 8 - Cluster-to-Drone collision detection experimental results

Test Run	**Distance from A to B (in cm)**	**Detected by**
1	174	A
2	73	A
3	187	A
4	35	B
5	188	A
6	165	A
7	81	A
8	126	A
Mean	128.63	-
Standard Deviation	55.28	-

5.4. Discussion

Overall, the evaluation done offered very positive results that satisfy the proposed goals.

Although deployability was not explicitly evaluated, it is worth mentioning that the system is very easy to deploy. In terms of hardware it comprehends only the Crazyflie drone and the Crazyradio PA dongle that should be assigned to each group of students. The installation process can take a while, but the step-by-step guide offered in laboratory guide 1 was created to speed up that process. Also, students are given enough time to perform the installation and after that, everything is set for all upcoming guides and no extra configurations are needed.

Due to the global health crisis, COVID-19 pandemic, that occurred during most of the development period, live tests with users were not possible to conduct, in order to assess difficulties in apprehending and solving the proposed exercises, as well as obtaining more realistic time estimation. Therefore, they should be further evaluated in the future, before being used in laboratory classes. For instance, a teacher that wants to use this system in its class should evaluate the guide not only to further ensure overall correctness, but also to make any necessary modification to account for specific needs.

6. Conclusion

In this dissertation, a system to help students and researchers implement drone applications is suggested. The system is built on top of the Crazyflie platform and exposes a python API so that users can easily interact with the MAV. A set of 3 laboratory guides were also developed [25] [26] [27], that aim to help students from IST to use this system over the course of 3 laboratory classes. In addition, an admin guide [29] has also been created to help an admin user to prepare a newly bought Crazyflie to be used in the suggested system.

Firstly, topics that were thought to be of interest for students to explore in laboratorial classes were explored. Specifically, techniques related with robot mapping, exploration and swarming were researched.

Secondly, the system's architecture was presented, as well as all its components. Here was also described what parts of system are provided directly by the Crazyflie platform, and what parts are extended through this work.

Thirdly, the components that were implemented during this work are explained in detail, namely, the laboratory guides, the Crazyflie python API and the modified version of the Crazyflie firmware.

Finally, the system core features were evaluated. In particular, the quality of the generated occupancy grid maps was tested, as well as how the performance of the algorithm would change when certain parameters were tinkered. Then, the performance of the frontier exploration algorithm was tested, in particular, the quality of the generated goals. Finally, it was also tested the scalability and safety of the system, in a scenario where multiple Crazyflie drones share the same airspace.

Results obtained are overall very positive and show that the goals that were presented in 1.1 were successfully achieved. This work also serves as proof-of-concept that validates the choice of the Crazyflie platform [1] as an option for research and education, namely in the fields here addressed. Because of the broad scope of this work, it was only possible to "scratch the surface" at each one of the topics covered. To this end, the modular implementation of each main feature in this work hopes to greatly facilitate future extensions that can be made in a particular field of interest.

6.1. System Limitations

Just like any architecture, this system has some known limitations. Some come from the Crazyflie platform itself and others from the implementation described in chapter 4.

As explained in 4.1.1, *venv* is used to prevent python dependency conflicts in the installing system. This is important to prevent that one of the dependencies of the *cfist* library (including *cflib)* breaks the project by someone making major changes in a new version release, for instance. This is crucial to ensure this work's validity over the years. However, it also creates a huge limitation in the system. As was said before, *Bitcraze* regularly maintains their libraries and firmware code. Whether it is in fixing existing bugs or delivering new features, by freezing the version of the *cflib* (v. 0.1.11) and the base firmware (without Radar support, v. 2020.06) the system will lose all those benefits. For example, just a couple of days before the delivery of this thesis, a new firmware version (v. 2020.09) was released, that promised to greatly improve overall flying stability when using the flow deck. Unfortunately, there was not enough time to test it and conduct a new evaluation and so it was not possible to make the upgrade. Readers that want to extend the proposed system in any way are encouraged to use the latest versions of *cflib* and the firmware instead, and test the system themselves to ensure its validity, knowing that compatibility with the presented system is not assured.

Regarding this system's ability to fit a swarming architecture, there are some limitations that may be useful for a user who is looking to implement such a system. Using a single Crazyradio PA dongle and the *cflib* library (specifically, the *Swarm* module) provided by this system, it is only possible to control up to 3 or 4 Crazyflies at the same time. When surpassing that threshold, the packet loss will start to become problematic. However, there are external projects that allow to overcome this limitation, such as Crazyswarm [30], that allow controlling more than 15 Crazyflie drones with a single Crazyradio, by using more efficient communication schemes.

To communicate with the drone, the system presented in this thesis relies on the Crazyradio PA dongle, connected to a PC. However, the Crazyflie platform also offers the possibility of communicating via BLE which is intended to be used with mobile phone apps. A reader interested in implementing such application should know that, using the *cflib* and firmware versions used in this system, the Crazyflie drone cannot communicate concurrently via Crazyradio PA and BLE, as there will be a small amount of packet loss that increases with the number of Crazyflies added. In addition, it is also not possible to use the Radar module developed for the system here presented, as it requires BLE to be disabled in the NRF chip (see the admin guide [29] for instructions on how to do this), in order to use P2P communication between drones. Therefore, using a Crazyflie that was previously used as part of this system requires the BLE to be activated again.

6.2. Future work

The system here suggested was designed to be easy to extend, either by adding new functionality to the existing modules, or by adding new modules. During the writing of this dissertation some potentially interesting suggestions were left that aim to either fix an existing problem or simply add a new feature. These and other suggestions are listed below:

- **Radar**

To improve the Radar performance, a signal propagation model for estimating distance from RSSI radio signal could be used, along with various filtering techniques.

- **Drone Delivery**

It would be very interesting to add a mechanical extension to the drone that would allow him to physically grab a light package and drop it off in another location using method calls in the API, instead of simulating this behavior, like it's done in laboratory guide 3.

- **Mobile Signal Triangulation**

In theory, 3 drones could be used as reference nodes to triangulate the radio signal and allow a 4th drone to estimate its position, creating, this way, a mobile triangulation solution. In practice, although such system would be possible and interesting to explore, it has a lot of constraints that need to be accounted for. For instance, each Crazyflie already estimates its own position through their own odometry, which would hardly improve since the position of the antennas (the 3 other Crazyflies) would also being estimated and the error would have been accumulated. This setup would be useful though, in a scenario where the antenna that we want to estimate position doesn't have any position estimation available. Another possible scenario would have been having the 3 reference Crazyflies not flying, but just acting as cellular antennas in a corner of the room. Then after triangulating the signal and estimating position, that estimation could be fused with the position from the EKF that the Crazyflie produces. It would even be possible to dynamically change nodes, between who is the antenna and who is being located.

- **Path Planning**

Most path planning algorithms are usually tested in simulation tools and programs, instead of real robots. This system provides an easy way for developers and researchers to test their algorithms in a real flying robot. The system even provides, through the Mapping module, the drawing functionality and the occupancy grid mapping code which is commonly used to implement these algorithms.

- **Mobile Phone Applications**

Mobile phones provide a very interesting way of interfacing with a drone. For instance, one could use the Mapping module developed in this work to create point-and-go in-map goals, or use the phone gyroscope to create an intuitive controller. It is important to keep in mind adjacent limitations, related with using BLE in the Crazyflie, presented in the previous section.

- **Hardware Extensibility**

Adding extra sensors to the Crazyflie is a feature that holds great potential for future projects. For instance, an early idea was related to adding simple temperature and LDR sensors to the drone to make a mobile weather station that could be programmed to go take reading at specific places and predefined times of the day.

7. References

[1] B. Rocha, "Semi-Autonomous Indoor Drones," Master's Project Report, Instituto Superior Técnico, Lisbon, 2019.

[2] W. A. Isop and F. Fraundorfer, "SLIM - A Scalable and Lightweight Indoor-Navigation MAV as Research and Education Platform," in *Robotics in Education*, 2020, p. 182–195.

[3] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler and A. Ng, "ROS: an open-source robot operating system," *ICRA Workshop on Open Source Software,* vol. 3, 2009.

[4] A. Bachrach, R. He and N. Roy, "Autonomous Flight in Unknown Indoor Environments," *International Journal of Micro Air Vehicles,* vol. 1, no. 4, pp. 217-228, 2009.

[5] G. Grisetti, C. Stachniss and W. Burgard, "Improved Techniques for Grid Mapping With Rao-Blackwellized Particle Filters," *IEEE Transactions on Robotics,* vol. 23, no. 1, pp. 34-46, 2007.

[6] B. Yamauchi, "A frontier-based approach for autonomous exploration," in *Proceedings 1997 IEEE International Symposium on Computational Intelligence in Robotics and Automation CIRA'97. 'Towards New Computational Principles for Robotics and Automation'*, Monterey, CA, USA, 1997.

[7] E. A. Cappo, A. Desai and N. Michael, "Robust Coordinated Aerial Deployments for Theatrical Applications Given Online User Interaction via Behavior Composition," in *DARS*, 2016.

[8] B. Araki, J. Strang, S. Pohorecky, C. Qiu, T. Naegeli and D. Rus, "Multi-robot Path Planning for a Swarm of Robots that Can Both Fly," in *2017 IEEE International Conference on Robotics and Automation (ICRA)*, Singapore, 2017.

[9] W. Giernacki, M. Skwierczyński, W. Witwicki, P. Wroński and P. Kozierski, "Crazyflie 2.0 quadrotor as a platform for research and education in robotics and control engineering," in *2017 22nd International Conference on Methods and Models in Automation and Robotics (MMAR)*, Miedzyzdroje, Poland, 2017.

[10] J. Noronha, "Development of a swarm control platform for educational and research applications," Master's Thesis, Iowa State University, 2016.

[11] F. Zafari, A. Gkelias and K. K. Leung, "A Survey of Indoor Localization Systems and Technologies," *IEEE Communications Surveys & Tutorials,* vol. 21, no. 3, pp. 2568-2599, 2019.

[12] J. S. G. Guerrero, A. F. C. González, J. I. H. Vega and L. A. N. Tovar, "Instrumentation of an Array of Ultrasonic Sensors and Data Processing for Unmanned Aerial Vehicle (UAV) for Teaching the Application of the Kalman Filter," *Procedia Computer Science,* vol. 75, pp. 375-380, 2015.

[13] N. Gageik, T. Müller and S. Montenegro, "Obstacle detection and collision avoidance using ultrasonic distance sensors for an autonomous," *Proc. UAVveek Workshop Contrib.,* 2012.

[14] I. R. Nourbakhsh, D. Andre, C. Tomasi and M. R. Genesereth, "Mobile robot obstacle avoidance via depth from focus," *Robotics and Autonomous Systems,* vol. 22, no. 2, pp. 151-158, 1997.

[15] T. Mori and S. Scherer, "First results in detecting and avoiding frontal obstacles from a monocular camera for micro unmanned aerial vehicles," in *2013 IEEE International Conference on Robotics and Automation*, Karlsruhe, 2013.

[16] F. Fraundorfer, L. Heng, D. Honegger, G. H. Lee, L. Meier, P. Tanskanen and M. Pollefeys, "Vision-based autonomous mapping and exploration using a quadrotor MAV," in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Vilamoura, 2012.

[17] V. Nguyen, S. Gächter, A. Martinelli, N. Tomatis and R. Siegwart, "A comparison of line extraction algorithms using 2D range," in *Auton Robot*, 2007.

[18] T. Pavlidis and S. L. Horowitz, "Segmentation of Plane Curves," *IEEE Transactions on Computers,* Vols. C-23, no. 8, pp. 860-870, 1974.

[19] G. A. Borges and M. -J. Aldon, "A Split-and-Merge Segmentation Algorithm for Line Extraction," in *Proceedings 15th International Conference on Pattern Recognition. ICPR-2000*, Barcelona, Spain, 2000.

[20] H. Moravec and A. Elfes, "High resolution maps from wide angle sonar," in *Proceedings 1985 IEEE International Conference on Robotics and Automation*, St. Louis, MO, USA, 1985.

[21] S. Thrun, W. Burgard and D. Fox, Probabilistic Robotics, The MIT Press, 2005.

[22] Bitcraze AB, "Bitcraze Home Page," [Online]. Available: https://www.bitcraze.io/. [Accessed 12 September 2020].

[23] Bitcraze AB, "cflib 0.1.11 docs," [Online]. Available: https://www.bitcraze.io/documentation/repository/crazyflie-lib-python/0.1.11/. [Accessed 9 September 2020].

[24] Bitcraze AB, "Bitcraze Store," [Online]. Available: https://store.bitcraze.io/. [Accessed 12 September 2020].

[25] B. Rocha, "Lab Guide 1: Crazyflie - Manual Navigation," Laboratory Guide, Instituto Superior Técnico, Lisbon, 2020.

[26] B. Rocha, "Lab Guide 2: Crazyflie – Autonomous Navigation and Mapping," Laboratory Guide, Instituto Superior Técnico, Lisbon, 2020.

[27] B. Rocha, "Lab Guide 3: Crazyflie – Drone Delivery," Laboratory Guide, Instituto Superior Técnico, Lisbon, 2020.

[28] Bitcraze AB, "Crazyflie firmware v.2020.06 docs," [Online]. Available: https://www.bitcraze.io/documentation/repository/crazyflie-firmware/2020.06/. [Accessed 9 September 2020].

[29] B. Rocha, "Getting a Crazyflie 2.1 ready for Lab Classes (Admin Guide)," Instituto Superior Técnico, Lisbon, 2020.

[30] J. A. Preiss, W. Honig, G. S. Sukhatme and N. Ayanian, "Crazyswarm: A Large Nano-Quadcopter Swarm," in *IEEE International Conference on Robotics and Automation*, 2017.

[31] W. A. Isop, J. Pestana, G. Ermacora and F. Fraundorfer, "Micro Aerial Projector - stabilizing projected images of an airborne robotics projection platform," in *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2016.

[32] O. Erat, W. A. Isop, D. Kalkofen and D. Schmalstieg, "Drone-Augmented Human Vision: Exocentric Control for Drones Exploring Hidden Areas," *IEEE Transactions on Visualization and Computer Graphics,* vol. 24, no. 4, pp. 1437 - 1446, 2018.

[33] Bitcraze AB, "Towards Persistent, Adaptive Multi-robot Systems," 26 June 2017. [Online]. Available: https://www.bitcraze.io/2017/06/towards-persistent-adaptive-multi-robot-systems/. [Accessed 10 September 2020].