University of Liège

School of Engineering and Computer Science

Autonomous navigation of a drone in indoor environments

Master's thesis carried out to obtain the degree of Master of Science in Computer Science Engineering

Author
Maxime Meurisse

Supervisors
Pr. Pierre Geurts
Ir. Christophe Greffe

Academic year 2020-2021

Abstract

University of Liège

School of Engineering and Computer Science

Autonomous navigation of a drone in indoor environments

Maxime Meurisse
Supervised by Pr. Pierre Geurts and Ir. Christophe Greffe

Academic year 2020-2021

Recent research is attempting to develop autonomous navigation algorithms that allow drones to navigate without the supervision of a pilot. Although the obtained results are promising, there are still many difficulties and no perfect solution, for the moment, exists.

This work is a research and development project on autonomous navigation algorithms for a small programmable drone in indoor environments free of dynamic obstacles. Practically, a Tello EDU has been chosen as the reference drone and the corridors of the Montefiore Institute have been considered as the environment. The hypothesis that the drone has access to a simple representation of its environment, in order to plan paths and analyze them, was posed.

First developed in a simulated environment and then adapted to the real world, algorithms working with Deep Learning models to perform image classification and depth estimation, and ArUco markers have been implemented and evaluated. More advanced elements such as battery station management or staircase passage are also addressed. Using a generic controller, developed in the framework of this work, these algorithms can be used on any drone model.

The tests carried out show that the drone can fly a simple path, from a starting point to an objective, in a completely autonomous way. The different methods studied have their strengths and weaknesses, discussed in this work. The main limitation of the developed algorithms is their robustness to errors and unexpected events. Several possible solutions are discussed.

Finally, this work ends by addressing the technological future of such systems, their integration into our modern society as well as their technical and legal limitations and dangers.

***Keywords**: autonomous-drone; indoor-environments; deep-learning; computer-vision.*

Acknowledgments

I would like to warmly thank my supervisors, Professor Pierre Geurts *and Christophe* Greffe*, as well as Michaël* Fonder *and Pascal* Leroy *for their precious help, advice and guidance throughout my work. I address special thanks to Michaël* Fonder *for his valuable advice in the field of drone.*

I also address, once again, particular thanks to Christophe Greffe *for his material support which allowed me to carry out this project.*

Finally, I thank Caroline le Paige *for supporting and motivating me throughout this work carried out in exceptional conditions due to the COVID-19.*

Contents

Chapter 1

Introduction

1.1 Context

Unknown to the public a few years ago, drones are now at the center of many technological innovations. Initially created for military purposes [1, 2], these remotely piloted flying machines are now used in a wide range of applications: photography, journalism, sporting and festive events, rescue missions, delivery of goods, etc. From models measuring several meters to miniature ones barely larger than a hand, these machines tend to facilitate, even replace, human work.

Initially very expensive, drones have now been democratized and are easily accessible. Their popularity is often due to their main advantage: they can be piloted remotely. Unlike ground vehicles (*e.g.* cars, trucks, robots) or aerial ones (*e.g.* airplanes, helicopters), drones can quickly venture into unreachable or dangerous areas without endangering their pilots.

All these applications have a price: an experienced pilot, even a licensed one, is necessary for a correct and safe flight. Technological advances in recent years have attempted to relax this requirement by creating autonomous navigation systems. Initially based on the use of many sensors and extensive data processing, these systems now take advantage of Artificial Intelligence, and more specifically Deep Learning models and intelligent data processing. Autonomous navigation allows the drone to understand its environment and to fly without any human intervention.

Despite the impressive obtained results, autonomous drone navigation remains a difficult subject with no miracle solution for the moment. Indeed, a whole series of problems arise quite quickly: how to understand the environment correctly when it can be vast and open (*e.g.* parks, gardens, cities, roads, countryside), closed and restricted (*e.g.* houses, building corridors) and totally unpredictable (*e.g.* a pedestrian, a car, an unexpected obstacle)? How to navigate in an organized way (*e.g.* a swarm of drones) and in a totally safe way (*e.g.* avoid any accident)?

Autonomous navigation of drones is a current topic in full development in the scientific world. The techniques are difficult to generalize because they are very specific to the application for which they were developed. The processing of large amounts of data

obtained via power-limited on-board hardware makes smooth real-time navigation difficult. These technical challenges are at the heart of the research carried out on the subject nowadays.

1.2 Objective

The objective of this work is to explore modern techniques in the scientific literature and to develop and evaluate autonomous navigation algorithms that allow a small drone to move without human pilot supervision in an indoor environment free of dynamic obstacles.

More specifically, the drone studied is a Tello EDU and the indoor environment considered is the main building of the Montefiore Institute of the University of Liège (*i.e.* a series of corridors). The Tello EDU is a small drone measuring only about 10 cm and having, as main sensor, a monocular RGB front camera.

This work will focus on the implementation of image analysis techniques (using, among others, Deep Learning models) allowing an autonomous navigation with limited capacities and sensors. The tests will first be carried out on a simulator and then tested in the real world.

Note. This project has been realized in parallel with several projects on drones; the final goal being to bring together the different works to obtain a personalized autonomous drone controllable by voice command. Another subject already specifically dealing with the discovery and mapping of an environment, this work will focus on the navigation aspects: determining a path from the starting point to the objective, tackling turns, adjusting without human intervention, etc.

In this work, we therefore consider that the drone already has access to a simple representation of its environment.

1.3 Organization

The rest of this document is organized as follows.

First, in order to better understand the subject and to situate the problem, Chapter 2 reviews the state of the art of drones and autonomous navigation techniques.

Second, the different resources needed for the implementation and testing of autonomous navigation algorithms are set up: Chapter 3 presents the Unreal Engine simulator and the two simulated environments created, Chapter 4 describes in detail the considered drone models and the implemented controllers, and Chapter 5 explains the representation of the environment used by the drone to help navigation.

Third, Chapter 6 focuses on autonomous navigation: a generic autonomous navigation algorithm is first defined. Then, methods to align the drone on a straight trajectory using the vanishing point and different image analysis methods to autonomously guide the drone are implemented: image classification and depth estimation via Deep Learning models, detection of particular key points (*e.g.* turns) via the detection and decoding of QR codes and ArUco markers, staircase passage via these same markers and management

of battery stations located in the environments. Based on the generic algorithm and the different implemented methods, navigation algorithms are created and tested in simulated environments.

Fourth, Chapter 7 takes all the implemented methods and algorithms and tests them in the Montefiore Institute to autonomously control the Tello EDU. The different difficulties brought by the transition to the real world are discussed.

Finally, Chapter 8 addresses the potential future of autonomous drones and Chapter 9 concludes with a summary of the obtained results and ideas for future work on the subject.

1.4 Resources

The various resources of this project are open source.

All the algorithms mentioned in this document are mainly implemented in Python and available on the following GitHub:

https://github.com/meurissemax/autonomous-drone

The latter also contains links to the simulator resources, the simulated environments created and the data sets built.

The main obtained results are illustrated by means of short videos available via the following YouTube playlist:

https://youtube.com/playlist?list=PLJEcTQrQgiVdacuc2HymqLV9RqjaRMNYt

Chapter 2

State of the Art

This chapter is the very first step of this work. It reviews the main concepts of drones and modern techniques for autonomous navigation.

2.1 Drone

A drone or *Unmanned Aerial Vehicle* (UAV) [1] is a vehicle capable of flight (*aircraft* [3]) without a human pilot on board. The vehicle can be either remotely controlled or autonomously piloted.

2.1.1 Terminology

The acronym "UAV" is the most frequently used to define such a flying vehicle, for amateur or professional applications. However, the latter tends to disappear and be replaced by other terms: *drone*, *Remotely Piloted Aircraft System* (RPAS), *Unmanned Air System* (UAS), etc. [4]

The term "drone" is increasingly used in the literature and on the Internet. According to the dictionary *Le Robert*, a drone is a "*small unmanned aircraft on board, remote controlled or programmed*" [5]. In general, the various terms and acronyms share, at least, these common characteristics in their definition.

Although the acronym "RPAS" is the one used with international organizations and other national aviation authorities, the term "drone", being closer to the one used in the literature, will be preferred and used in this work.

2.1.2 Classification

Based on their mass, drones fall into several categories: the *UAV* for drones with a mass greater than 25 kg, the *Small Unmanned Aircraft System* (sUAV) for drones with a mass less than or equal to 25 kg and the *Micro Air Vehicle* (MAV) for very small drones weighing less than 20 g. [4]

2.1.3 Components

The information in this section comes mainly from [6].

The base (or skeleton) of the drone is called the *chassis* (or *frame* by the professionals). This can have several shapes depending on the type of drone: 3 arms for a *tricopter*, 4 arms for a *quadcopter*, 6 arms for a *hexacopter*, etc. In this work, the popular quadcopters will be considered.

The drone is then composed of a *propulsion system* allowing it to take off, fly and land. This includes motors (rotors), propellers, an electronic speed controller and a battery.

Another important component is the *flight controller*. Composed of an integrated circuit with microprocessor and sensors, this system allows the drone to perceive its environment and to be controlled, either remotely (via RC transmitter and receiver) or autonomously.

Finally, although not compulsory, most drones have a camera (either fixed to the front or to a gimbal) allowing them to see their environment and transmit these images to the pilot or to the autonomous navigation system.

2.1.4 Autonomous drone

The information in this section comes mainly from [7] and [8].

An autonomous drone is a drone that can perform a series of tasks without human supervision. Currently, technologies already allow for *partially* autonomous drones: these can perform some tasks autonomously in a lot of situations, but still require the supervision of a pilot, although the latter need only rarely intervene.

To create *fully* autonomous drones, current technologies need to be further improved and developed to meet several challenges, particularly in terms of robustness and safety. The main current challenges come from the environments: how to ensure reliability and safety in a complex environment? Drones must not become a danger to people and must be able to be adopted, without nuisance, in the current regulations.

Artificial Intelligence

With the development of Artificial Intelligence, and more particularly of Machine Learning methods in recent years, the technologies used for drones have progressed considerably.

Indeed, it is now possible to develop models capable of analyzing images in real time, notably for detecting obstacles. The impressive results obtained by these models are a giant step towards the integration of drones in complex environments, populated by unpredictable obstacles, such as urban environments.

Applications

Drones are used for a wide range of applications: surveillance, data collection, monitoring, filming, photography, but also delivery, agriculture, exploration, rescue missions, work tools (for example, for environmental mapping), etc.

Autonomous drones could make it possible to delegate complex tasks, for example monitoring an area during a special event, and thus save time, money and labor. Major manufacturers such as NVIDIA, SenseFly and DJI are actively working on the development of such drones [9–11].

2.2 Autonomous navigation

Autonomous drone navigation is achieved through algorithms that allow the drone to perceive its environment and make decisions based on these perceptions. The scientific literature on the subject is expanding rapidly: a large number of algorithms, increasingly robust to dynamic obstacles and other constraints of complex environments, are emerging.

In general, there are no miracle solutions for autonomous drone navigation. It is a complex task where solutions have to be designed according to the context of use. In an indoor environment, one characteristic is of great importance: the use of GPS data is (almost) impossible. GPS data, when available, does not provide sufficiently accurate data to guide the drone in such a confined environment. Indoor navigation algorithms therefore have to do without GPS data and rely mainly on the drone's other on-board sensors.

The various autonomous indoor navigation algorithms can be grouped into three main categories: techniques using data analysis methods, techniques based essentially on the drone's vision using Deep Learning models and techniques using Reinforcement Learning.

2.2.1 Analysis techniques

Before the development of Machine Learning methods, autonomous navigation techniques were mainly based on the processing of a large quantity of data collected by several sensors, more or less sophisticated.

The work of Foehn et al. [12] focused on the creation of an autonomous drone that had to pass through a series of gates as quickly as possible (in the context of a race). Based on a mathematical model of the drone's dynamics, the Inertial Measurement Unit (IMU), a laser range finder and 4 cameras, they use a *Kalman Filter* to best estimate the state of the drone at any given time and plan its next actions.

The work of Power et al. [13] has focused on the navigation of a swarm of drones without GPS. The main goal is that each drone has an estimation of the position of the other drones. They worked with a *Multi-Target Gaussian Conditional Random Field* (MT-GCRF) model to predict, at each time step and based on their last known position and their path traveled (*dead reckoning*), the position of the drones.

Simultaneous Localization And Mapping (SLAM) is a method frequently used with drones, and robots in general. The principle consists of building a representation of its environment while keeping track of the robot's position. SLAM can be carried out via *odometry* techniques (estimating the displacement of a robot based on a mathematical model) or based on data acquired via sensors. The work of Nemati et al. [14] uses a variant, called *Hector SLAM* [15], with a laser range finder. Brockers et al. [16], on the other hand, uses a variant of SLAM based on visual data, the *vSLAM*. The inertial data collected by the drone and the data obtained via SLAM are combined via an *Extended Kalman Filter* (EKF) [17].

In another register of analysis techniques, the work of Aguilar et al. [18] attempts to approximate drone movements by computing affine transformations between consecutive images of the drone and by tracking the displacement of certain key points.

These analysis techniques usually require several sensors (camera, laser range finder, LIDAR, etc.) that are power hungry and expensive. It is difficult to adopt a real-time navigation solution with small drones in these conditions: the large amount of data is generally difficult to manage and the computation time is too long.

2.2.2 Vision-based techniques

Techniques based on drone vision take advantage of recent advances in image analysis working with Deep Learning models. The latter provide impressive results based solely on images recovered via the drone's camera.

Deep Learning basics

The information in this section comes mainly from [19].

Deep Learning is a branch of Machine Learning. Machine Learning consists of training a model f that can learn to predict some outputs for given data that can be unknown to it. More specifically, the model f is first trained with a large amount of data. Then, it will provide an output $f(x, \theta)$ for an input x where θ are the parameters of the model, determined via its training.

For example, a Machine Learning model can be trained to take as input an image it has never seen before and predict whether the image contains a specific object (*e.g.* a car) or not.

These models need a large amount of data x_i to be trained. When the training data used also contains the output y_i that the model has to predict (for example, a boolean indicating whether the image contains a car or not), we talk about *supervised learning*. In this configuration, the output $f(x_i, \theta)$ of the model can be compared with the expected output y_i. The model f can then be trained via a *loss function* $\mathcal{L}(f(x_i, \theta), y_i)$ that compares the outputs. The smaller the value of this function, the closer the model output is to the true output. Training the model then consists in tuning its parameters θ to minimize the value of $\mathcal{L}$ for all training data (x_i, y_i).

Deep Learning models are complex models inspired by how the human brain works. The models are composed of interconnected *neurons*, distributed in different layers, form-

ing a *neural network*. Their main advantage is that they can handle much larger inputs (*e.g.* image) and learn much more complex functions than simple Machine Learning models.

Deep Learning models can also be trained in a supervised manner. The goal is still to minimize a loss function that compare the output of the network with the true output. They generally need a (very) large amount of data to learn correctly, which sometimes makes them difficult to use.

A popular application of these networks concerns image processing, via *Convolutional Neural Networks* (CNN). Such networks can handle very large inputs, typically an image composed of thousands of pixels, using convolution operations.

Deep Learning-based techniques

As drones generally all have at least one camera as a sensor, the use of CNN for image processing and navigation is very popular. Generally speaking, the different works try to collect and annotate a large amount of data (drone's images) and train in a supervised way a CNN to perform some tasks (predict an action associated to an image, predict a distance, etc).

Amer et al. [20] have collected a large series of images (between 20 000 and 40 000 depending on the environment) on a simulator and have made use of the pre-trained VGG-16 model [21] to extract the *features* from the drone images and then run them through a *Fully Connected Neural Network* (a neural network where each neuron of a layer is connected to all neurons of the next layer) or *Recurrent Neural Network* (a neural network that has a "memory" thanks to feedback connections and can thus handle data in sequence) to infer a command from the drone. The idea is therefore to associate each image of the drone with an action to be carried out. Padhy et al. [22] have also exploited this idea by teaching the drone, via a pre-trained DenseNet161 [23] network, to associate each captured image with a simple action such as "move forward", "rotate left", "rotate right" or "stop".

In the same vein, Lee et al. [24] use several CNNs to analyze the images and detect the various obstacles. They argue that the networks provide very good and resource-efficient obstacle detection compared to methods using sensors such as LIDAR.

An original idea was also proposed by Gandhi et al. [25]: teach the drone how not to fly. More precisely, they collected a large data set of images of the drone just before an impact with an obstacle. Through training using the AlexNet [26] network, they taught the drone to detect dangerous areas where it could crash. Thus, by avoiding all (static) obstacles, the drone is able to follow a simple trajectory autonomously.

Kouris et al. [27] have, for their part, worked with distances inferred via CNNs. More precisely, they collected a large data set of images, cut them vertically into 3 zones (left, central and right) and annotated each zone, via appropriate sensors, with a distance to the nearest feature in that zone. Based on this data, their model learned to predict, for an input image, 3 distances. Working with several distances allows a more accurate perception of the environment and thus a more precise control of the drone.

The work of Wang et al. [28] shows that it is also possible to combine Deep Learning models and advanced sensors. They used an Intel RealSense D435 [29] (a depth camera) to obtain depth maps and combined these results with inferred object detection via the YOLO v3 model [30]. Their system is then able to detect obstacles, calculate the distance of the drone to them and maneuver to avoid them.

The work of Chen et al. [31] is also very interesting. The authors have developed a network, *UAVNet*, capable of detecting obstacles in real time. The focus was on optimization so that the network could be embedded in miniature drones with few resources and still act in real time.

Many other similar works are available on the subject. All of them show that, with current technologies, it is possible to replace expensive and time-consuming sensors with much more efficient Deep Learning models, obtaining nearly similar results. One of the main challenges of these models is the collection of data for proper training. To facilitate this task, work has been done on *Transfer Learning* [32, 33]. The idea is to train a model on synthetic images, for example collected on a simulator, and then use it directly in the real world (or adapting it slightly by a small training with real world images if necessary). This procedure can allow to avoid collecting a lot of images in the real world, which is sometimes difficult or not possible.

2.2.3 Reinforcement Learning

Reinforcement Learning (RL) is a branch of Machine Learning where an agent learns a behavior via interactions with its environment. More precisely, the agent interacts via actions and receives *rewards* for each of its actions. The goal being to maximize its rewards, the agent learns by trials and errors to interact optimally with its environment; it infers, from its experiences, an optimal *policy*.

Deep Reinforcement Learning (DRL) combines Reinforcement Learning and Deep Learning methods. Neural networks are used as an approximator of unknown functions, allowing to handle very large perceptions of the environment (*e.g.* images). DRL techniques have been very successful in recent years, particularly in the field of games [34] (notably the famous example of *AlphaGo* [35]), but also in robotics [36].

Reinforcement Learning has also been explored in the drone domain. By considering the drone as the agent, and discretizing its actions into “move forward”, “turn” and “stop”, Imanberdiyev et al. [37] have developed the TEXPLORE algorithm that allows the drone to find the optimal path, from a starting point to an objective, in an unknown environment by managing its battery level via available charging stations. In the same vein, Pham et al. [38] used a simple *Q-learning* algorithm to guide the drone through an unknown environment.

Based on more complex models and combined with image analysis via CNNs, the works of Walker et al. [39], Wang et al. [40], He et al. [41] and Guerra et al. [42] show that it is possible to teach the drone optimal policy in complex environments.

However, these Reinforcement Learning methods are more complex than simple image analysis methods via Deep Learning. Indeed, the latter are difficult to converge towards

good results and require large computing resources. Moreover, they are difficult to apply to the real world. It seems difficult to train a drone to navigate by trial and error, knowing that each mistake (hitting a wall, for example) is likely to damage the drone.

As with Deep Learning, the use of Transfer Learning has been considered. The idea is to reproduce a similar environment in a simulator so that the drone learns to infer an optimal policy. This policy is then adapted to the real world. We can note the work of Anwar et al. [43] on the subject.

2.3 Application to this work

In this work, a drone with mainly a monocular RGB front camera is used. As it is not equipped with multiple sensors, it is interesting to explore image analysis methods, using classical Computer Vision methods or via Deep Learning models. The idea is to look at the autonomous navigation of a small *low-cost* drone: how capable is a drone of flying alone based on mainly its front camera data?

Multiple ideas coming from papers mentioned earlier has been used: the work of Padhy et al. [22] has been chosen as a basis for this project: the model used by the authors, the navigation algorithm as well as the evaluation metrics have been taken up, used, discussed and improved. The idea of separating the images in several zones in order to multiply the information (Kouris et al. [27]) was also taken up, in particular to guide the drone via a depth estimation. Finally, the idea of Transfer Learning (widely exploited in the work of Anwar et al. [43]) has been tested (however, unlike the work of Anwar et al. [43], Transfer Learning has been tested with the CNNs used and not with Reinforcement Learning algorithms).

Concerning Reinforcement Learning, only a few simple tests have been performed (these are presented in Appendix C). Being a rich and complex family of methods, it is difficult to combine vision-based methods and Reinforcement Learning methods in a single work. The latter are therefore left for future work on the subject.

Chapter 3

Simulated Environment

As drones are expensive and fragile machines with a very limited autonomy, it is common practice to carry out the first tests on a simulator. Before starting any manipulation with drones, a simulator was therefore chosen and two simulated indoor environments were created.

This chapter reviews the main existing simulators, the characteristics of the chosen one and describes the simulated environments created.

3.1 Simulator

A simulator is a tool for reproducing real actions in a virtual environment.

When working with drones, and robots in general, it is essential to carry out preliminary tests so as not to damage the machine or endanger the working environment. In the case of drones, the very limited battery (just a few minutes for MAVs) is an additional problem that can limit real-life testing; hence the importance of a simulator.

For the tests to be realistic, the simulator must be able to model the real world and the drone as faithfully as possible: the forces applied, the possible turbulence, the wind, the movements that are not always precise, but also the brightness of the rooms, the shadows, the blurred images due to the movements, etc.

Although there are many physical simulators, two of them stand out in the drone field: Gazebo and Unreal Engine 4.

3.1.1 Gazebo

Gazebo [44] is an open-source simulator developed by the Open Robotics [45] company. As mentioned on the official website, "*Gazebo offers the ability to accurately and efficiently simulate populations of robots in complex indoor and outdoor environments*" [44].

With its numerous plugins, developed largely by the community, and compatible with the Robot Operating System (ROS) [46], also developed by Open Robotics, Gazebo is a very popular simulator for robots. It is lightweight and available on all platforms.

Gazebo was not created to simulate drones exclusively. It is therefore necessary to manually add and configure all the resources needed. The documentation is also not specifically oriented to the use of drones, making it sometimes difficult to get familiar with the tool.

3.1.2 Unreal Engine 4

Unreal Engine [47] (version 4) is a proprietary game engine developed by the Epic Games [48] company. Initially created for video games, Unreal Engine has now become very popular for a wide range of applications: architecture, 3D modeling, animation and film making, simulations, etc. Its realism and the quality of its rendering have made it one of the most popular game engines on the market.

The disadvantages of Unreal Engine are that it is quite heavy (a high-performance hardware configuration is necessary) and, despite a compatibility announced on all platforms, it is usable correctly only on the Windows operating system.

Being initially a game engine, Unreal Engine is not designed to perform robot simulations. However, being extremely rich in the possibilities it offers, it is possible to set up simulated environments and robots via additional plugins.

AirSim

AirSim [49] is an open-source plugin, developed by the Microsoft company, for Unreal Engine allowing a realistic simulation of cars and drones. It adds all the necessary functionalities to have a simulated environment taking advantage of the very high realism of the game engine.

The great advantage of AirSim is that it has been developed with the aim of simulating drones, and more specifically via Artificial Intelligence systems. An API, developed in Python, is available with all the necessary functionalities to easily set up autonomous and intelligent systems (*e.g.* easy image collection and annotation). A whole series of sensors and images are available and can be used directly (LIDAR, segmented images, depth map), making the constitution of data sets relatively fast and easy.

The plugin has been designed to be as faithful as possible to the real world. Combined with the high level of realism of the Unreal Engine, it is possible to obtain simulations that are extremely faithful to the real behavior of a drone in a real environment.

3.1.3 Comparison

The two simulators, Gazebo and Unreal Engine with AirSim, both offer interesting possibilities for drone simulation. Although still less mature than Gazebo, AirSim is gradually gaining popularity so that it is widely used in the drone field.

Both simulators are compatible with all platforms, although AirSim (with Unreal Engine) is only usable easily (configuration and running) on Windows. However, the drone-simulated oriented aspect and the ready-to-use Python interface of the latter has been preferred, and therefore used, for this project.

Note. All the precise technical characteristics of the machine used to run the simulator as well as the different versions of the programs and languages used in this work are described in Appendix A.

3.2 Environments

The next step is to create simulated environments. It is important that these environments are as realistic as possible and faithful to the real world. Indeed, working mainly with images from the drone, the more realistic the environment, the closer the images will be to real images and the better the algorithms can be adapted to real conditions.

The real reference environment being the main building of the Montefiore Institute at the University of Liège, the objective is to create simulated environments composed of corridors, turns and other characteristic elements (doors, decorations, lamps, etc.). In order to test the efficiency and generalization of the algorithms, it would be ideal to work with at least two different environments.

3.2.1 Resources already available

A wide range of assets (3D modeled objects but also complete ready-made environments) are available via the Unreal Engine Marketplace. Unfortunately, the majority of them are not free. The few free resources do not correspond to the desired environment characteristics.

Apart from the Unreal Engine Marketplace, there are few ready-to-use indoor environments for Unreal Engine available for free. Another interesting resource is the PEDRA project [43], which offers a series of simulated indoor environments that match the desired environment characteristics. Unfortunately, these are not editable (either the simulation parameters or the environment itself and its components) and were created with an old version of AirSim. They are therefore difficult to use for this project.

3.2.2 Handcrafted resources

As no resources matching the characteristics of the Montefiore Institute were available, two open-source simulated environments on Unreal Engine has been designed for this project (see Section 1.4 for download links).

The two environments created are called “Indoor Corridor” and “Indoor Staircase”. All the 3D objects used come from free resources available on the Unreal Engine Marketplace.

Indoor Corridor

Indoor Corridor is a simulated environment composed of 5 corridors, 4 turns and 2 crossroads. The extreme corridors have doors and the whole environment is composed of various decorations (plants, frames, chairs, armchairs). Lamps are displayed on each wall. Their intensity has been adjusted so that their effect on the environment is realistic: the

corridors are fairly well lit, but some areas, such as the corners, are a little less so. The floor has a wooden floor texture to be faithful to the Montefiore Institute.

A series of images illustrating the main elements of the environment is shown in Figure 3.1.

Figure 3.1: Images of the Indoor Corridor simulated environment.

A video illustrating the environment in more detail is available via the following link: https://youtu.be/cz1zRka21UY.

Indoor Staircase

Indoor Staircase is an environment strongly inspired by Indoor Corridor. The basic shape is identical, but all the textures and decorative objects are different. The central corridor has also been replaced by a staircase leading to a floor consisting of a single corridor.

This environment will be used to test the generalization of the algorithms and to try to teach the drone to autonomously fly over staircases.

A series of images illustrating the main elements of the environment is shown in Figure 3.2.

Figure 3.2: Images of the Indoor Staircase simulated environment.

A video illustrating the environment in more detail is available via the following link: https://youtu.be/muzKT-dCab4.

Aerial views

In order to give a better idea of the structures of each of the simulated environments, aerial views are presented in Figure 3.3.

(a) Indoor Corridor (b) Indoor Staircase

Figure 3.3: Aerial views of the simulated environments.

Chapter 4

Controllers

With the environments defined, it is now necessary to look at the programmable control system of the drones. Indeed, before any manipulation and creation of algorithms, it is important to define all the actions that can be performed by the drones.

This chapter describes the flight controllers of the drones considered and presents the generic control interface, compatible with any model of drone, created for this work.

4.1 Drone models

In this project, two drone models are considered: the simulated drone and the real one.

4.1.1 Simulated drone

The default simulated drone model in AirSim is a Parrot AR Drone 2.0 [50] (Figure 4.1).

Figure 4.1: Illustration of the Parrot AR Drone 2.0. [51]

This drone is a quadcopter equipped with a high-definition front camera and is remotely controlled via Wi-Fi. It is fully programmable and has been designed mainly for experimenting with control algorithms. The main characteristics of this drone are presented in Table 4.1.

Physical characteristics	Weight	380 g
	Dimensions (width × height)	45.1 cm × 45.1 cm
	Sensors	Gyroscope, accelerometer, magnetometer, pressure sensor, ultrasound sensor
Performances	Maximum flight distance	50 m
	Maximum flight speed	5 m s^{-1}
	Maximum flight time	12 min
	Maximum flight height	50 m
Front camera	Resolution	HD, 720p, 30 FPS
	Field of view	92°

Table 4.1: Main characteristics of the Parrot AR Drone 2.0. [50, 52]

It should be noted that, as the drone is simulated, these characteristics are not all rigorously applied in AirSim. Indeed, it is possible to manually configure the size, the maximum flight speed, the resolution and the field of view of the camera, etc.

4.1.2 Real drone

The real drone model is a Tello EDU [53] (Figure 4.2), manufactured by the Ryze Robotics company and equipped with flight technologies from the DJI company.

Figure 4.2: Illustration of the Tello EDU. [54]

This drone is also a quadcopter equipped with a high-definition front camera and is remotely controlled via Wi-Fi. Also being fully programmable, this drone is mainly used to experiment with programmed flight. The main characteristics of this drone are presented in Table 4.2.

Physical characteristics	Weight	87 g
	Dimensions (width × height)	9.8 cm × 9.2 cm
	Sensors	Range finder, barometer, LED
Performances	Maximum flight distance	100 m
	Maximum flight speed	8 m s^{-1}
	Maximum flight time	13 min
	Maximum flight speed	30 m
Front camera	Resolution	HD, 720p, 30 FPS
	Field of view	82.6°

Table 4.2: Main characteristics of the Tello EDU. [53]

4.1.3 Comparison

The Parrot AR Drone 2.0 and the Tello EDU are two drones that can be controlled remotely and in a programmable and automatic manner. Although both are designed for experimentation, the AR Drone 2.0 is aimed at a more experienced public, whereas the Tello EDU is aimed at a younger, less experienced public. The latter comes with two mobile applications that allow for fun learning about programming.

Both drones are equipped with a fixed high-definition front camera. Their perception of the environment is therefore essentially limited to what is in front of them. This information is important and must be taken into account because it implies that the drones move sideways or backwards "blindly".

The AR Drone 2.0 is larger than the Tello EDU. To have the most accurate simulation possible, the simulated drone has been resized in AirSim to match the size of the Tello EDU. Being sufficiently similar, the other characteristics were left unchanged (including the field of view which, despite a difference of almost 10°, has no real impact on the obtained results between simulator and real world).

It should be noted that, as both drones are controlled by Wi-Fi, no data processing operations are embedded and performed directly on the drone; everything is done on the computer. This implies that, for this work, the processing time of the algorithms is not critical since they benefit from the power of a computer. However, it is important to bear in mind that, in most real cases, the algorithms will have to run directly on the drone and therefore be as light and optimized as possible.

4.2 Flight controller

A drone consists of a series of sensors and actuators. The purpose of a flight controller is to bring the drone to a desired state using its actuators based on its current state as perceived by its sensors.

4.2.1 Simulated drone

The drone simulated via AirSim is equipped with a flight controller called "Simple Flight" [55]. This controller has been designed to make life easier for users. It has its own interface and can be used on both simulated and real models, making it easy to adapt algorithms.

Simple Flight can take as input (state of the drone) angular rates, angular levels, speeds or position. Internally, the controller consists of a series of PID controllers to generate an output signal for the actuators.

4.2.2 Real drone

The Tello EDU is equipped with DJI's control technology and digital image stabilization. Not being explicit about their technologies, it is difficult to say how the controller exactly works.

An important characteristic of this controller is that, although it is very good in general, it is not extremely accurate in terms of distance. For example, when the drone has to move from 2 m forward, it will move correctly along a straight path (in reality, there is a very small deviation, but completely negligible for short distances) but will travel $2 + \varepsilon$ m where ε is random noise (the latter is approximately quantified in Section 4.3.2).

This characteristic has a very important implication for the realization of control algorithms: it is not possible to rely entirely on the drone's controllers to move in a fully autonomous and safe manner. It is not, for example, sufficient to tell the drone to "move 10 m forward, turn 90° clockwise and move 2 m forward" because there is no guarantee that the distances will be scrupulously respected, which could lead the drone to hit an obstacle or a wall.

4.2.3 Utilization

This work focuses on the realization of high-level autonomous flight algorithms; the drone and their controllers will therefore be used as is. These controllers are sufficiently powerful to achieve high-level control. Indeed, they allow a correct stabilization and displacement with a few inaccuracies.

However, it is important to keep in mind that the flight controller is a very important part of a drone and that its correct implementation, in order to create a drone, is crucial.

4.3 High level control interface

Although each drone can perform the same main actions (take-off, landing, moving), each has its own control interface, usually designed by the manufacturer. To avoid creating navigation algorithms specific to a particular interface, a new high-level control interface

has been implemented. The latter defines general control methods and is an additional layer that can be used on top of the manufacturer's control interface.

4.3.1 Architecture

The architecture of the high-level control interface as well as the controllers designed for each drone are shown in Figure 4.3.

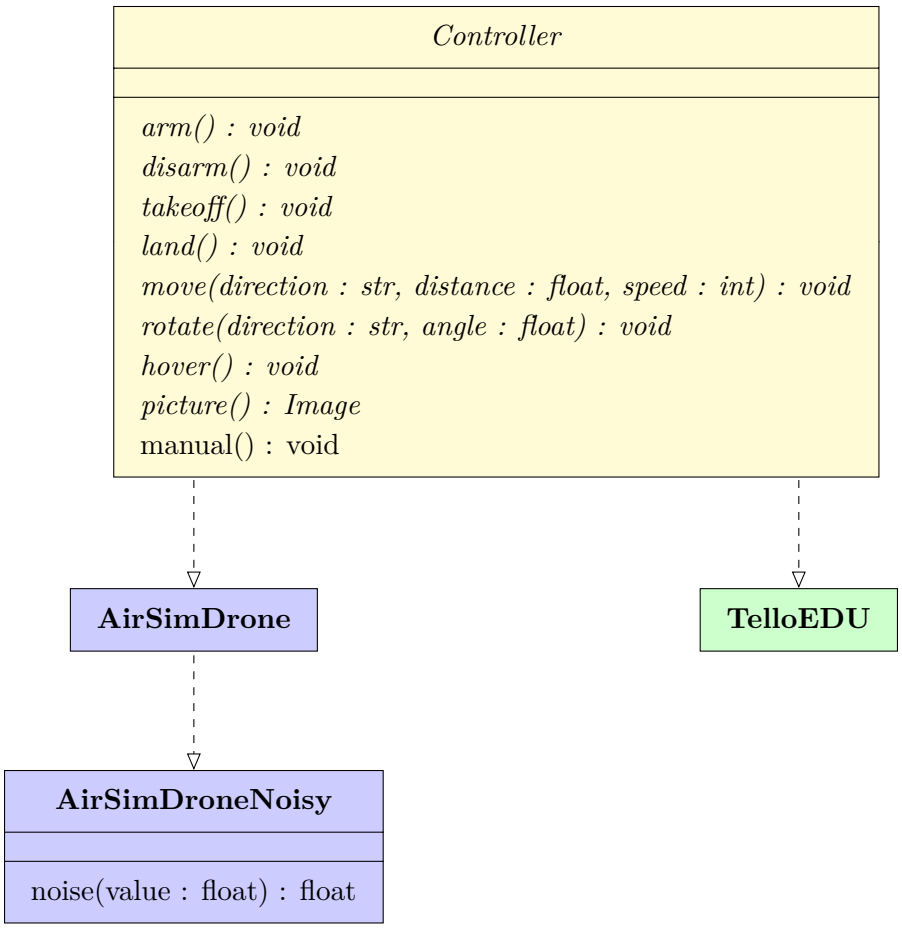


Figure 4.3: Architecture of the high-level control interface and controllers of each drone considered in this work.

The high-level control interface (`Controller`) is an abstract class that must be implemented by each drone used. In this work, a controller was implemented for the simulated drone (`AirSimDrone`) and another for the real drone (`TelloEDU`). The simulated drone being too precise in its movements, another controller adding noise on displacements was also created (`AirSimDroneNoisy`).

The common control commands of the interface are described in Table 4.3.

Command	Description
`arm`	Arms the drone. Must be called before any other command.
`disarm`	Disarms the drone. Must be called after the last command.
`takeoff`	Takes the drone into the air. The altitude depends on the manufacturer flight controller.
`land`	Land the drone.
`move`	Moves the drone in a chosen direction, a chosen distance at a chosen speed.
`rotate`	Turns the drone in a chosen direction at a chosen angle.
`hover`	Make the drone stay in place.
`picture`	Take a picture.
`manual`	Enter the manual control mode of the drone (using the the keyboard of the computer).

Table 4.3: Description of the high-level control interface commands.

The main advantage of working with a similar interface for the simulated and the real drone is that their use is completely interchangeable. An algorithm that moves the simulated drone can be applied directly to the real drone without requiring any changes to the code. In other words, the high-level control of the drone in the different algorithms is totally agnostic of the drone considered.

In order to demonstrate the use of this interface, a simple test was carried out: a series of pre-defined commands were executed by the simulated and the real drone based on the exact same algorithm, requiring no code modification. A small video of this test is available via the following link: `https://youtu.be/1kVLT63IXUE`.

The following sections explained how each control interface has been designed using the manufacturer control interface.

4.3.2 Simulated drone

The AirSim plugin already provides a high-level control interface. This has a large set of commands that are only valid in simulation (*e.g.* teleport the drone to a specific position) and is specific to the AirSim simulated drone. In order to have interchangeable commands in the algorithms, the high-level control interface in Figure 4.3 has been implemented as an additional layer to the AirSim interface (`AirSimDrone` and `AirSimDroneNoisy`), using only methods that could be applied to a real drone.

Noisy version

As the control of the simulated drone is too perfect (*e.g.* when we ask the drone to move 2 m forward, it will move almost exactly 2 m forward, which is not really realistic), a noisy version was implemented. The noisy controller (`AirSimDroneNoisy`) is an additional layer

to the simulated drone controller (`AirSimDrone`). It is responsible for calling the simulated drone's commands by adding a noise ε to the move and rotate command values such that

$$\varepsilon \sim \mathcal{N}\left(\frac{v}{15}, \frac{v}{50}\right) \tag{4.1}$$

where v is the initial value sent to the command.

This noise corresponds approximately to the noise on the displacements of the Tello EDU, which was approximated experimentally. For this purpose, several trajectories of fixed length were performed (*e.g.* move 2 m forward). Each time the drone completes a trajectory, its position is noted and the deviation from its objective is calculated (manually). Based on 50 samples, a noise model was approximated.

It is clear that 50 samples are not sufficient to approximate a faithful model of the noise. However, the chosen model seems realistic enough to be used.

4.3.3 Real drone

When the Tello EDU is switched on, it creates a Wi-Fi network. The drone is controlled by connecting a computer to this network and sending data packets via communication sockets. The contents of the packets are described in the user guide supplied with the Tello EDU [56].

The high-level control interface in Figure 4.3 has been implemented as an additional layer to the Tello EDU interface. This is responsible for initiating sockets and sending corresponding messages when commands are called. An additional socket is reserved to retrieve the video stream.

It should be noted that there is a delay of about 0.5 s between the moment where a command is sent and the moment where the drone executes it but also between the images received from the drone and the reality.

Chapter 5

Environment Representation

In this work, it is assumed that the drone has access to a simple representation of its environment. Therefore, the next step, before the implementation of navigation algorithms, is the creation of such a representation that can be exploited by the drone.

This chapter presents the simple representation designed for the environments, its characteristics, its use and the information that can be derived from it.

5.1 Usefulness of a representation

Any navigation system requires, in one way or another, knowledge of its environment. This knowledge can be built up entirely on the fly (*e.g.* using SLAM with the appropriate sensors [57, 58]) or based on a priori knowledge (*e.g.* using a predefined map of the environment).

In this project, the second option is chosen: a priori knowledge of the environment, in the form of a map, is available. Although this assumption may seem simplistic, it is in fact quite realistic for two reasons.

1. As this work is part of several works on drones, the problem of mapping an environment is already addressed separately from this work. It is therefore plausible to assume that the results of that other work would be available for the navigation algorithms of this work.
2. Without a priori knowledge of its environment, the drone can hardly learn to move from any starting point to any objective since it can not plan a path in advance. For example, when the drone comes to a turn, it will have no idea whether to turn left or right. It is possible to hardcode this information (via proper training of a Deep Learning model, for example), but the drone will only be limited to one possible path.

A knowledge of the environment therefore allows the drone to navigate in the environment without being restricted to certain pre-defined fixed paths : it can predict a path from any starting point to any objective. The drone can also extract information from this path in advance: the number of turns and crossroads, their directions, the number

of staircases to be crossed, the approximate distances to be covered, the potential other difficulties present (*e.g.* a door to be crossed), but also keep an approximate trace of its position in the environment.

5.2 Representation

The environment must be represented in a way that can be used by the drone and the navigation algorithms. In order to optimize the computing time, the representation must be as light as possible and contain only the essential information useful for autonomous navigation (*i.e.* the global structure of the environment, not every details it contains).

A known means in the field of robots to represent an environment is an *occupancy grid*.

5.2.1 Occupancy grid

An occupancy grid [59] is a grid, which can be represented via a matrix, discretizing the environment and where each location is marked with a value, belonging to the interval $[0, 1]$, indicating the probability of presence of an obstacle.

Within the framework of this work, a particular case of occupancy grid will be used: a *binary occupancy grid*. In this case, the matrix representing the grid contains only two values: 0 to indicate a free position and 1 to indicate an occupied position (*e.g.* a wall).

This representation implies a discretization of the environment. The smaller the step size chosen, the more accurate the representation will be, but it will be memory intensive and therefore slow to manipulate. As mentioned in Chapter 4, the drone considered is not capable of extremely precise movement; it is therefore not necessary to have a representation accurate to the centimeter. As a good compromise between accuracy and simplicity, a 1 m step precision was chosen.

Note. With an occupancy grid and a step precision of 1 m, it may be difficult to represent precisely complex shapes (a circle, for example). In this project, only simple indoor environments (with simple shapes) are considered, so this is not really a problem.

Two-dimensional representation

The binary occupancy grid is a two-dimensional representation of the environment. Although the drone moves in 3 dimensions, this representation, in the context of this work, is sufficient.

Indeed, the environment considered (the corridors of the Montefiore Institute) is relatively simple. The drone will only very rarely have to modify its altitude (only for staircases, see Section 5.3.5 for their management), and never in the context of path planning. It is not necessary to store the height of the environment in its representation; this would only make it more cumbersome and increase the processing time of the algorithms for nothing really useful.

5.2.2 Implementation

The implementation of the binary occupancy grid has been done in such a way that it can be easily created manually and easily exported and modified. In practice, the representation is stored in a text file (`.txt`) and is built based on several symbols explained in Table 5.1.

Symbol	Function
`.`	a free position
`#`	an occupied position
`N, S, W, E`	the starting position of the drone, additionally representing its orientation (North, South, West, East)
`B`	a battery station
`*`	the objective to reach
`+`	staircases that go up
`-`	staircases that go down

Table 5.1: Symbols used to represent characteristics of the representation of an environment.

An example of a very simple environment representation is given in Figure 5.1.

```
1  ##########
2  #E.....#*#
3  #......#.#
4  #......#.#
5  #..#####.#
6  #..#.....#
7  #..#.....#
8  #........#
9  ##########
```

Figure 5.1: Representation of a very simple environment in `.txt` format.

The representation in Figure 5.1, when interpreted, results in a visual map shown in Figure 5.2.

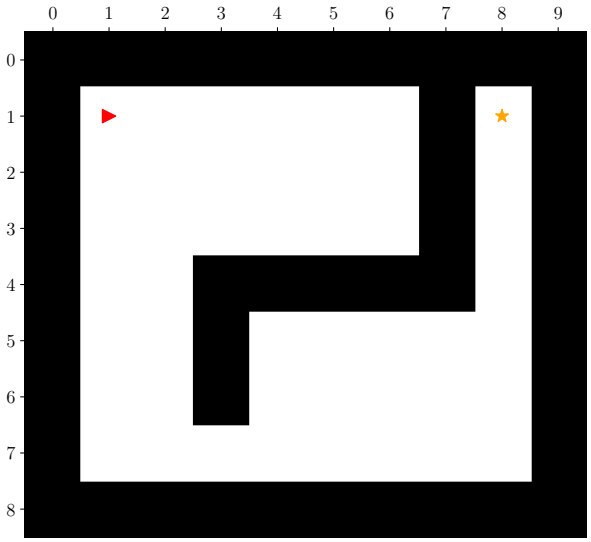


Figure 5.2: Visual representation of the simple environment described in Figure 5.1 where the red arrow represents the position and orientation of the drone and the yellow star represents its objective. Each free and occupied positions are shown, respectively, in white and black.

The other symbols not used in this example (battery station, staircases) are used to add more complex features which will be explained in Section 5.3.

Note. Since the binary occupancy grid is represented by a matrix, it is mandatory that the environment is represented by a grid of $N \times M$ symbols. If the environment we want to represent is not rectangular in shape, the empty areas can be filled with non-free position symbols.

5.3 Features

The representation of the environment has several features that help the drone to navigate autonomously.

5.3.1 Path planning

Shortest path planning is one of the most important features. With a representation of its environment, the drone is able to move from any start point to any objective, provided that it can plan a path between these two points.

Very popular for its simplicity and efficiency, the A^* path planning algorithm [60, 61] was chosen. This algorithm allows the planning of a shortest path between a start node and an end node in a graph. For each node, the algorithm computes a cost defined as

$$f = g + h \tag{5.1}$$

where g is the distance between the considered node and the start node and h is an estimation of the distance, via a heuristic, between the considered node and the end node.

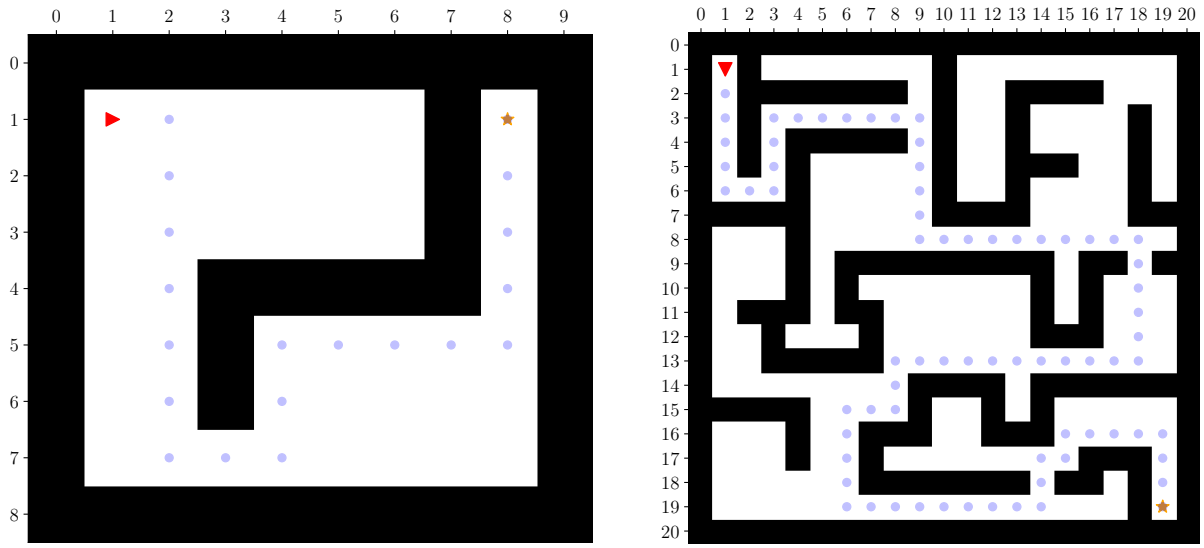
The different nodes are then, during the iterations, considered or not by the algorithm according to their costs, the goal being to minimize the total cost of the path.

Considering that each location of the binary occupancy grid is a node of an undirected graph and that each node has 4 neighbors (top, bottom, left and right), the distance between a node and one of its neighbors (used to compute g) is set to 1 and the heuristic h chosen is the *Euclidean norm* between two nodes, *i.e.* for two nodes n_1 and n_2,

$$h(n_1, n_2) = \sqrt{(n_{2_x} - n_{1_x})^2 + (n_{2_y} - n_{1_y})^2} \tag{5.2}$$

where $\circ_x$ and $\circ_y$ are, respectively, the line and column of a node $\circ$ in the matrix.

The algorithm was tested on the simple environment 5.1 but also on a complex one. The obtained paths are presented in Figure 5.3.



(a) Simple environment

(b) Complex environment

Figure 5.3: Examples of shortest paths (in blue dots) determined, based on the environment representation, via the A* path planning algorithm.

In both cases, the algorithm correctly found the shortest path from the starting point to the objective. The execution times[1] of the algorithm were, in both cases, less than 1 ms. Since the path is calculated before the drone flies, its execution time is not critical. However, from a perspective where the path could be re-calculated in flight, a fast execution time is nevertheless interesting.

Note. In this work, diagonal movements have not been considered. Indeed, the environments considered being corridors, the drone will never really be required to move diagonally.

[1] All execution times mentioned in this work were obtained on the same machine which is described in Appendix A.

5.3.2 Key points extraction

For the purpose of autonomous navigation, it is important that the drone is able to extract information from the path it has to follow. For example, the drone must be able to know whether it should turn left or right at the first crossroads it encounters, whether the staircases are going up or down, etc.

All this information can be extracted based on the environment representation and the path the drone has to follow. This information is called *key points*. In this work, the different key points of an environment considered are turns, crossroads and staircases.

When the drone has determined the shortest path to its objective, it extracts key points. These are tuples consisting of a position and an action, *e.g.* the key point `((1, 1), 'right')` means that there is a key point at position $(1, 1)$ and that it is a turn (or crossroad) where the drone must turn right.

Staircases, and the associated actions, are extracted simply on the basis of knowledge of their position in the environment (given in the `.txt` file by the symbols `+` and `-`, see Table 5.1). Turns and crossroads, on the other hand, are extracted by applying a simple algorithm, on each point that composes the path, that checks the number of free neighbors. If a point has at least two non-aligned free neighbors, it is considered as a key point.

Note. One must pay attention that this simple algorithm only works with environment representations where each corridor has a width of one discretized step (for example, the environment shown in Figure 5.5). For more complex environments, such as those presented in Figure 5.3, the algorithm must be adapted, else it will consider that each point in an "open" area is a key point.

The actions associated with each turn are then deduced on the basis of its position, the position of the next point in the path and the orientation of the drone. This can be `left`, `right` but also `forward` if the drone has, for example, to continue straight forward at a crossroad.

By applying this key point extraction procedure to the complex environment of Figure 5.3b, we obtain, as the first key point of the list, `((6, 1), 'left')`. Indeed, the key point that the drone will meet will be a turn to the left (from its point of view) and located at the position $(6, 1)$.

It is important to note that the key points composing a path are extracted in the order in which the drones will encounter them.

5.3.3 Drone position

The position and orientation of the drone in the environment representation (red arrow) can be updated based on the actions performed and its current orientation.

When the drone performs the action `forward`, its position is updated in the environment according to its orientation. For example, in the representation 5.3a, the drone's new position would be $(1, 2)$ and in the representation 5.3b, its new position would be $(2, 1)$.

When the drone performs the action `left` or `right` (turn left or right), its orientation is updated according to its current orientation. For example, in Figure 5.3a, the action `right` would orient the drone towards the South. In Figure 5.3b, this same action would orient the drone towards the West.

It is therefore possible to estimate the position of the drone in the environment solely on the basis of the actions it performs. It should be noted that the update of the position has to be adapted according to the precision of the environment and the movement of the drone. For example, if the precision is 1 m and the drone moves in steps of 50 cm, its position has to be updated in the environment only one action over two.

It is also possible to update the position of the drone in the environment directly on a coordinate basis. This can be used when the drone detects a key point. Knowing the position of the key points of the environment (see Section 5.3.2), when the drone detects one, it can be moved to the exact corresponding position in the environment representation.

5.3.4 Battery management

The battery is a critical issue for drones. Indeed, in small models, such as the Tello EDU, the battery only lasts a few minutes (see Table 4.2).

In this work, the battery was not considered a major problem because the paths taken by the drone are very short. However, in order to improve the drone's complete autonomy navigation system, it is interesting to take the battery into account when planning paths.

Another work, always carried out within the framework of various works on drones mentioned before, consists in designing charging stations for the drone that work by induction. We could therefore imagine a scenario where these charging stations would be present in the environment. These can be represented via the appropriate symbol (the symbol `B`, see Table 5.1) when creating an environment representation.

The idea is to check the battery level of the drone before planning a path. Depending on the battery level, the drone can directly reach its objective or go first to a battery station. The adapted path planning algorithm in described in Algorithm 1.

Algorithm 1 Path planning with battery stations.

```
1  function PATH(start, end)
2      Let d be the distance to end
3      Let max_d be the maximal distance that the drone can travel
4
5      if max_d ≥ d then
6          return A_STAR(start, end)
7      else
8          Let b be the farthest reachable battery station
9
10         if b has already been visited then
11             return empty path
12
13         return A_STAR(start, b) + PATH(b, end)
```

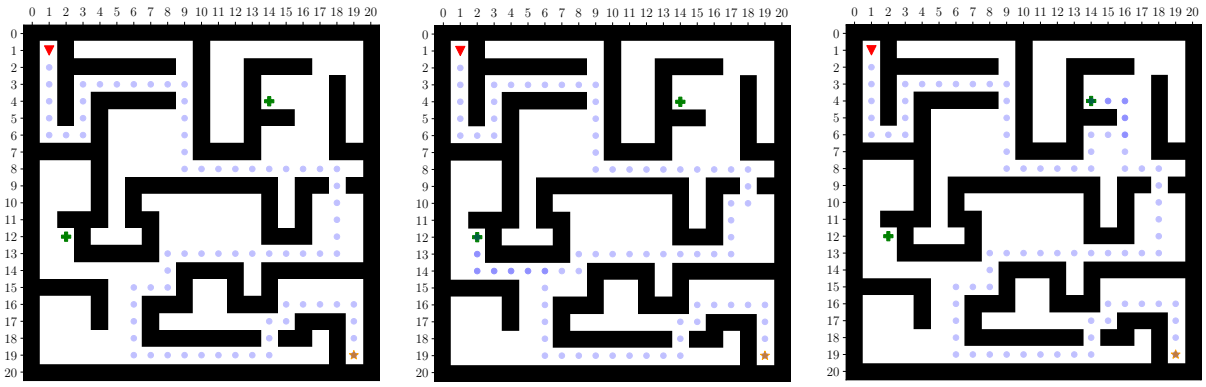
The maximum distance the drone can fly (line 3 of the Algorithm) can be obtained via its battery level and a simple model. For example, if we know that, with a fully charged battery, the drone can fly 10 min and that it flies at $1\,\mathrm{m\,s^{-1}}$, then it could fly 600 m. This model is, of course, a very simple approximation but can be sufficient, if we take a safety margin, to plan a correct path.

To get the farthest reachable station (line 8 of the Algorithm), the station with the higher distance to the drone but still lower than the maximal distance the drone can travel is chosen.

If the drone has already visited a battery station b, the algorithm returns an empty path and stops (line 10 of the Algorithm). Indeed, if the drone has to return to a battery station already visited, it means that it can never find a path that can lead it to the objective with sufficient battery level.

This algorithm is greedy: the drone always tries to reach its objective first if its battery allows it, else it tries to go to the farthest battery station in order to recharge as late as possible and arrive at its objective with the maximum amount of battery. It does not always provide the optimal path, especially if there are battery stations that are far from the drone *and* the objective. In the context of the simple tests realized in this work, it is sufficient, but it could be improved, using for example dynamic programming, in a future work.

The Algorithm 1 was applied to the complex environment with, respectively, a battery level of 100%, 50% and 20%. The results are presented in Figure 5.4 where the green crosses represent a battery station.



(a) Battery level: 100% (b) Battery level: 50% (c) Battery level: 20%

Figure 5.4: Examples of paths determined via the Algorithm 1 for different battery levels.

In the case of a full battery, it can be seen that the drone can go directly to its objective. When the battery is charged to 50%, the drone can go to the farthest battery station. This allows it to arrive at its objective at a high battery level. With 20% battery, the drone first passes through the nearest battery station before going to its objective.

5.3.5 Staircases and floors management

The last feature is the management of staircases and floors. Indeed, an environment can contains several floors linked with staircases.

Staircases are considered as key points of the environment and can be represented using the appropriate symbols (`+` and `-`, see Table 5.1).

The representation of an environment being in two dimensions, it is not possible to manage several floors in a single representation. The solution adopted in this work is to represent each floor via a different representation. When a staircase is reached by the drone, the corresponding representation of the next floor is loaded into memory and replaces the current representation used. A new path is determined and the drone can continue its navigation.

5.4 Application to this work

Environment representations have been made for each of the environments considered in this work.

5.4.1 Indoor Corridor

Based on the aerial view of the environment presented in Figure 3.3, a representation of the Indoor Corridor environment has been realized in Figure 5.5.

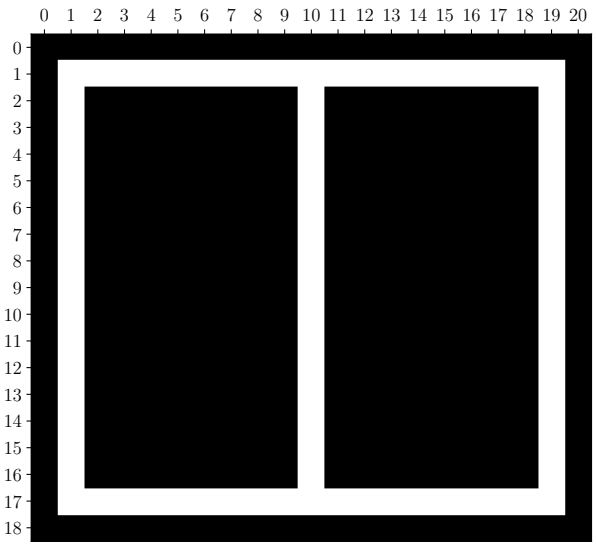
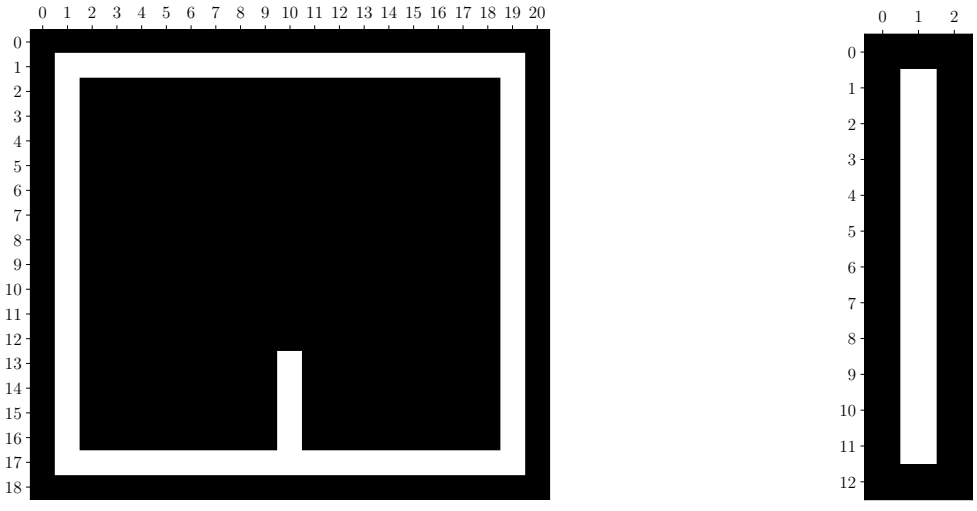


Figure 5.5: Visual representation of the Indoor Corridor simulated environment.

A small video illustrating a simple usage of this environment representation is available via the following link: https://youtu.be/fyyEQhqTZp8.

5.4.2 Indoor Staircase

Also on the basis of the aerial view of the environment presented in Figure 3.3, a simple representation was made. As the Indoor Staircase environment has a staircase, and therefore a floor, two representations were made: floor 0 and floor 1 (Figure 5.6).



(a) Floor 0

(b) Floor 1

Figure 5.6: Visual representation of the Indoor Staircase simulated environment.

5.4.3 Montefiore Institute

On the basis of small plans available at the entrance of the building, a representation of the Montefiore Institute (floor -1) has been made (Figure 5.7).

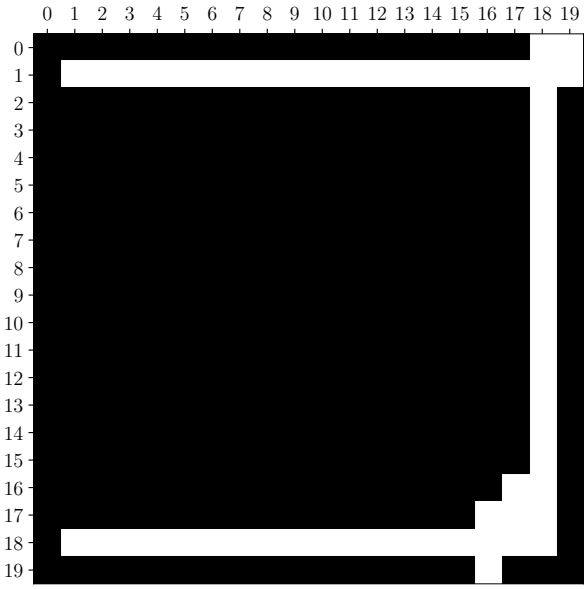


Figure 5.7: Visual representation of floor -1 of the Montefiore Institute of the University of Liège.

Chapter 6

Autonomous Navigation

With all the necessary tools for creating navigation systems in place (environments, their representations and controllers), the next major step in the work is to create and implement autonomous navigation methods.

This chapter presents the different methods studied and the navigation algorithms designed. The various tests carried out were done in simulated environments. Real-world testing is discussed in Chapter 7.

6.1 Generic navigation algorithm

In order to navigate the drone autonomously, a navigation algorithm must be defined. For this, the main steps of a navigation from a starting point to an objective must be clearly defined.

As the drone has a simple representation of its environment (Chapter 5), its navigation essentially consists of detecting the key points that compose it. In other words, the drone must be able to detect, in an autonomous way, turns, crossroads and staircases of an environment.

Based on this, an algorithm can be defined. First, the drone must calculate the shortest path from the starting point to the objective, on the basis of its representation of the environment, and extract the corresponding list of key points. Then the drone can take off. As long as the drone has not reached its objective, it takes a picture of its environment and analyzes it to try to detect whether it is a key point or not. Based on this, the drone performs an action and updates its position in its representation of the environment. When the drone has arrived at its objective, it lands.

When the drone moves, it is likely to deviate slightly from its trajectory. It is therefore interesting to also add a system allowing the drone to align itself correctly before continuing its navigation.

This generic algorithm is presented in Algorithm 2.

Algorithm 2 Generic algorithm for autonomous drone navigation.

```
1  function NAVIGATE(start, end)
2      Let env be the representation of the environment
3      Let drone be the controller of the drone
4
5      path ← ENV.PATH(start, end)                          Compute the shortest path
6      keypoints ← ENV.KEYPOINTS(path)                      Extract key points from the path
7
8      DRONE.TAKEOFF()                                      Take off the drone
9
10     while drone has not reach its objective do
11         picture ← DRONE.PICTURE()                        Take a picture
12         type ← ANALYZE(picture)                          Analyze the image
13
14         if type is a key point then
15             action ← GET_ACTION(keypoints)               Get appropriate action
16         else
17             ALIGN(picture, drone)                        Align the drone
18             action ← move forward
19
20         DRONE.EXECUTE(action)                            Execute the action
21         ENV.UPDATE(action)                               Update position of the drone in environment
22
23     DRONE.LAND()                                         Land the drone
```

The actions related to the drone (take-off, image capture, displacement and landing) are performed via the controller (Chapter 4). The actions related to the environment (determination of the shortest path, extraction of key points, updating of the drone's position) are carried out via the representation of the environment (Chapter 5).

As the position of the drone in the environment representation is updated each time it moves, in order to determine whether the drone has reached its objective (line 10 of Algorithm 2), the position of the drone in the environment representation is simply compared to the position of the objective.

Note. This method to check if the objective is reached may seem highly inaccurate, but it actually provides good results. Although the position of the drone is approximated as it moves, it is updated accurately each time a key point is detected (Section 5.3.3). Unless the objective is very far from a key point, the position reached by the drone is always very close to its intended objective. However, it would be possible, in a future work, to improve the landing position of the drone by using, for example, distinct landmarks to detect in the environment.

Determining the action to be taken when a key point has been detected (line 15 of Algorithm 2) is done simply by consulting the list of key points (and their associated actions) extracted from the path. The first key point detected will give rise to the first

action in the list, the second key point to the second action in the list, etc.

Only the actions ALIGN and ANALYZE need to be determined in order to define a functional algorithm. The rest of this chapter therefore presents the architecture adopted to implement this generic algorithm, the methods used to implement the alignment and analysis actions and the various results obtained.

6.2 Architecture

The different navigation algorithms all share a common structure (Algorithm 2), only the methods ALIGN and ANALYZE are different. In order to be able to carry out multiple algorithms while keeping a comprehensible and easily maintainable solution, a particular architecture has been adopted.

Each of the methods (ALIGN and ANALYZE) is implemented via a *module*. A module is therefore simply a small algorithm that allows a specific task to be carried out on the basis of information provided by the drone (*e.g.* a picture).

The generic navigation algorithm is implemented via an abstract class (`NavAlgorithm`). Each algorithm created inherits from this class and is composed of one or more modules allowing the implementation of the different methods (alignment and analysis).

The general architecture, with some generic examples, is shown in Figure 6.1.

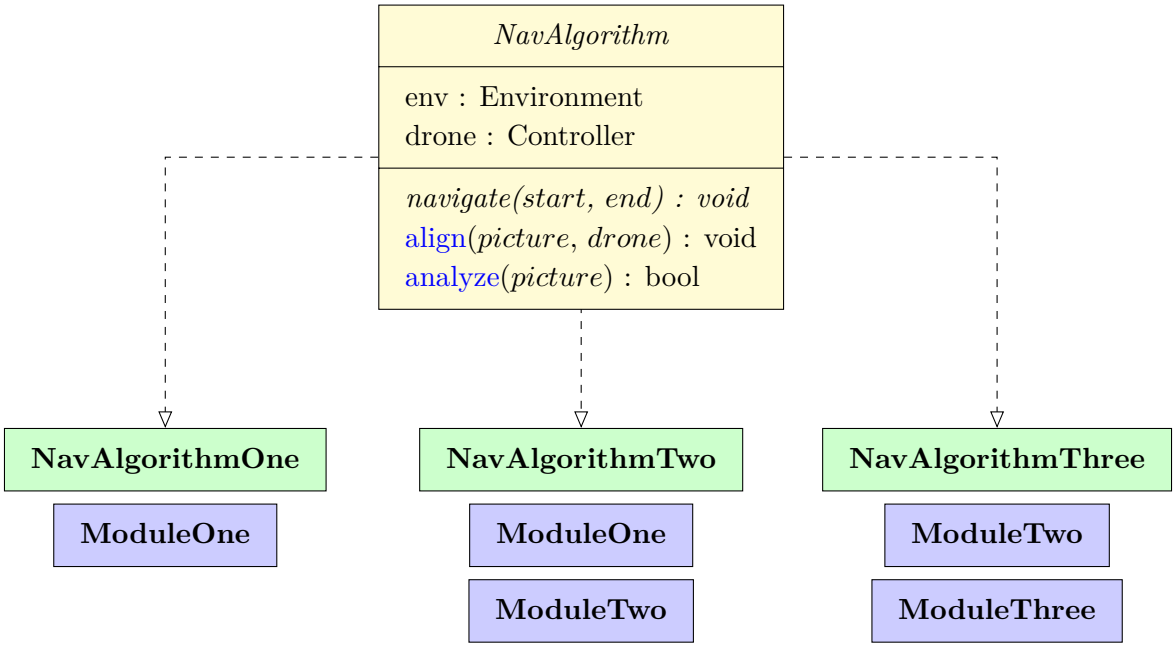


Figure 6.1: General architecture adopted to design multiple navigation algorithms.

Such an architecture has the advantage of allowing the very easy creation of several algorithms by simply combining one or more modules while avoiding code redundancy in the different implementations.

6.3 Alignment

This section concerns the creation of modules used to align the drone (ALIGN).

In a perspective view of an environment (for example, a picture), the lines parallel to each other no longer appear parallel but all converge at a point called the *vanishing point* (Figure 6.2).

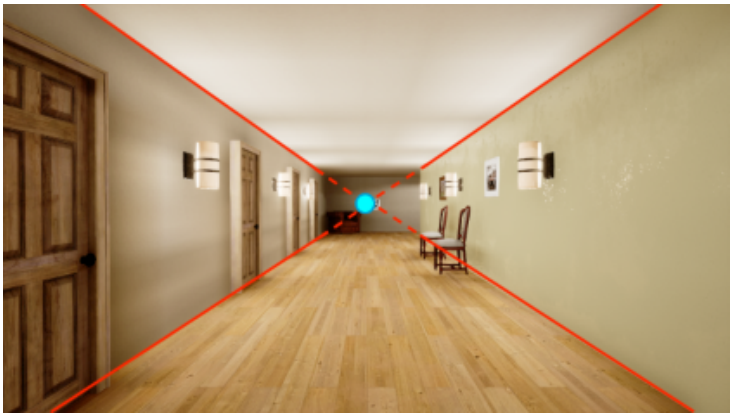


Figure 6.2: Example of a vanishing point (in blue) in a corridor image. The parallel lines (in the real world) are represented in red.

The vanishing point is frequently used when moving a robot. The idea is to make sure that this point is always in the center of the image so that the robot follows a straight trajectory, facing the horizon.

The generic alignment procedure used in this work is described in Algorithm 3, with δ being the maximum allowed deviation from the center of the image.

Algorithm 3 Generic procedure for aligning the drone using the vanishing point.

```
1  function ALIGN(picture, drone)
2      Let δ be a tolerance.
3
4      cx, cy ← CENTER(picture)                    Get center position of the image
5      vx, vy ← VANISHING_POINT(picture)           Get vanishing point
6
7      while vx < cx − δ or vx > cx + δ do
8          if vx < cx − δ then
9              DRONE.ROTATE(left)
10
11         if vx > cx + δ then
12             DRONE.ROTATE(right)
13
14         picture ← DRONE.PICTURE()               Take a new picture
15         vx, vy ← VANISHING_POINT(picture)       Get new vanishing point
```

This algorithm first computes the positions of the center point of the image and the vanishing point. It then compares these positions, according to a tolerance δ, to determine if the drone has to rotate left, rotate right or do nothing (because it is sufficiently well aligned). While the drone is not aligned, it continues to take picture and analyzes it. No vertical correction is computed because, as explained in Section 5.2.1, the drone only rarely moves up or down (only for staircases) and, practically, it successfully keeps its altitude without (significant) deviation.

The tolerance δ can be set dynamically according to the dimensions of the image and the method used. The different adjustment actions are performed via the drone controller. In this work, small rotations (left or right) of 5° have been used.

Note. When the detected vanishing point seems completely wrong (*e.g.* if the detected vanishing point is in a corner of the image), the alignment procedure is not applied in order not to lead the drone into an inconsistent position.

To have a functional alignment procedure, the detection of the vanishing point has to be defined (VANISHING_POINT in Algorithm 3). Several vanishing point detection methods have been tested: two methods working with line detection and one method working with a Deep Learning model.

6.3.1 Line detection

Since the vanishing point is at the intersection of the parallel lines (in the real world), a first method consists in detecting these lines and compute their intersection.

The detection of the lines of an image is a well-known application in Computer Vision. Although many methods exist, it is certainly not an easy problem: the results are very sensitive to the quality of the image, variations in brightness, colors, etc. It is then difficult to design an algorithm robust to all possible variations.

In this work, the drone moves in corridors, we can therefore make some simplifying assumptions: in the vast majority of images, only one vanishing point will be present and the vanishing point will generally be the intersection of 4 lines representing the boundaries between the walls, the floor and the ceiling.

Detection via classical methods

The first method implemented uses classical Computer Vision techniques to detect edges and lines. This method will be called `VPClassic` and is described below.

1. A bilateral filter is applied to the image. This filter replaces the value of each pixel with a weighted average of the intensities of neighboring pixels [62]. The resulting image preserves its shapes and contours but is smoother; thus reducing noise.
2. A Canny filter [63] is then applied to extract the edges of the image. This filter first applies a Gaussian filter to remove noise and then computes gradient, using convolution masks, to get the intensity of each point. It finally applies some post processing operations to only keep points that belong to edges.

3. Based on the previously extracted edges, the lines are calculated using a probabilistic Hough transform [64]. The resulting lines are represented by segments with given x and y endpoints.

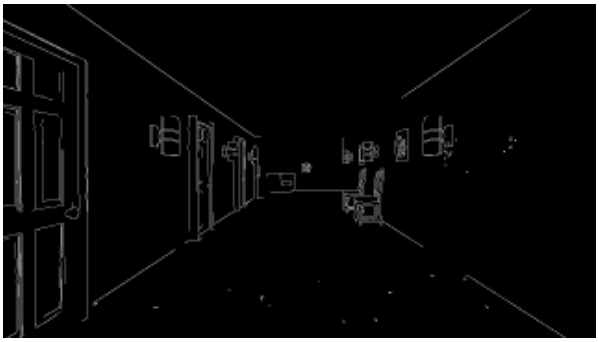
4. The resulting lines are filtered, based on the x coordinates of their endpoints, to remove all those that are vertical (or nearly vertical).

5. Finally, the vanishing point is calculated based on the intersections of the remaining lines. However, it is (very) possible that more lines than necessary are presented, including lines that are not needed for the vanishing point calculation, which may lead to an incorrect result. Furthermore, as the detection is not perfect, not all lines may intersect at a specific point.

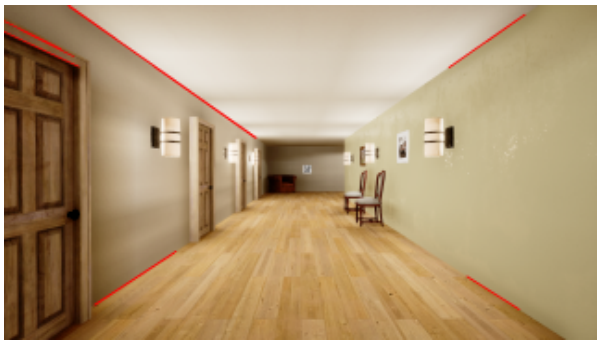
   To overcome this problem, a method inspired by [65] was used: the image is separated into a grid and the intersections contained in each cell of the grid are counted. The cell with the most intersections is considered to be the one containing the vanishing point. This method does not give the exact coordinates of the vanishing point, but an area, more or less precise depending on the size of the grid, in which it is located. The coordinates of the vanishing point can be approximated by taking the coordinates of the center point of the cell.

An example of the results obtained with this method is shown in Figure 6.3. The determined vanishing point is located in the green cell in Figure 6.3d.

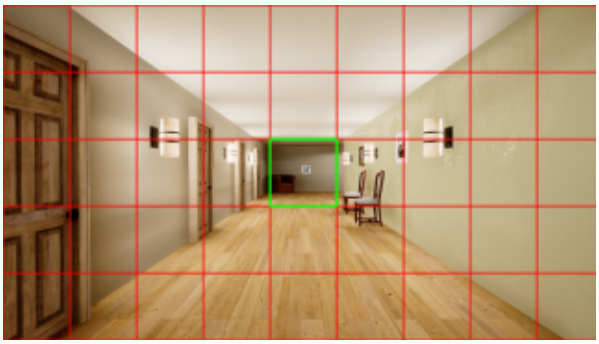
(a) Bilateral filter



(b) Canny filter



(c) Filtered Hough transform



(d) Extracted vanishing point

Figure 6.3: Example of results obtained via the `VPClassic` vanishing point detection method.

Note. The lengths of the detected lines could have been taken into account to get more information, for example the distance to the end of the corridor (the smaller the lines, the closer to the end). However, the algorithm does not always detect the whole line (see Figure 6.3, only small parts of the lines were detected), making this procedure very unstable and thus difficult to use in practice.

Detection via edgelets and RANSAC

The detection of the vanishing point being a common problem, the scientific literature is rich in various methods. Another interesting one working with *edgelets* (a triplet, composed of a location, a direction and a strength, representing the features of an edge) and *RANSAC* has been implemented, inspired from the work of Chaudhury et al. [66] and adapted from the implementation of [67]. The main idea is to no longer work directly with line detection but with edgelets and to use RANSAC, robust to noise and outliers, to vote for the best candidate at the vanishing point.

This second method will be called `VPEdgelets`. An example of result is shown in Figure 6.4. The obtained vanishing point is represented by a red dot.

Figure 6.4: Example of obtained result via the `VPEdgelets` vanishing point detection method.

Evaluation

Although the obtained results with these two methods seem to be very good on the examples, it is interesting to see how well they really perform. For this purpose, two series of 50 images (one per simulated environment) have been collected (by manually controlling the drone in the environments). These 100 images are such that the vanishing point is at the most varied locations possible. Some examples are given in Figure 6.5.

Figure 6.5: Examples of images collected for the evaluation of vanishing point detection methods.

Each of the images was annotated with its vanishing point using an annotation tool developed for this work. The tool allows, with a simple mouse click, to select the location of the vanishing point on the image and to save its x and y coordinates.

To evaluate the methods, the coordinates of the vanishing points detected were compared to the manually annotated vanishing points. As manual annotation is not necessarily the most accurate, the evaluation was done by simply comparing whether the coordinates found by the method lie within a zone, of fixed radius (set to one tenth of the width of the image in this work), around the annotated vanishing point. The mean execution time per image of each method, in seconds, was also evaluated. The obtained results are presented in Table 6.1.

Method	**Score** (over 100)	**Mean execution time** (s)
`VPClassic`	93	0.04
`VPEdgelets`	79	0.31

Table 6.1: Results obtained with the evaluation of the vanishing point detection methods.

The obtained results show that the `VPClassic` method obtains a better score in terms of detection and a much better mean execution time per image. For these reasons, this method will be kept to create a first module implementing the ALIGN method (with the deviation δ set to one twenty-fifth of the image width). A small example of utilization of this module is shown in a video available via the following link: `https://youtu.be/oSyHeMTe4y8`.

6.3.2 Deep Learning

Inspired by the work of Chang et al. [68], a Deep Learning model has been used to predict the vanishing point of an image. More precisely, the image is discretized into a grid of

a given size and the goal is to determine in which cell the vanishing point is located. The problem then becomes a classification problem: the Deep Learning model infers a probability for each cell indicating the chance that the vanishing point is located there. The coordinates of the center point of the cell with the highest probability is then selected as the vanishing point location.

The more cells the grid contains, the more accurate the vanishing point location will be, but the harder the classification will be (because the number of cells represent the number of possible classes). In this work, being a good compromise between precision, ease and speed, a grid of 5×5 was chosen (the width and height of each cell are the width and height of the image divided by 5).

Data set

A data set of 1052 images (of size 320×180 pixels) with annotations has been created. The images are as varied as possible so that the network learns to detect vanishing points at various positions.

The images were collected by manually controlling the drone, and automatically taking pictures at a fixed interval of 0.5 s, in the environment and slightly varying its orientation from time to time in a random fashion. An automatic solution to collect the images was considered but not retained (more details in Section 6.4.1).

The vanishing point of each of the images was first manually annotated using the previously mentioned tool. The image was then discretized into a grid and the cell containing the vanishing point was marked. More formally, the annotation associated with an image i is a binary vector y_i containing 24 zeros and 1 one indicating the cell containing the vanishing point. The data set was then separated into a training set of 736 images and a testing set of 316 images.

Model

Being already implemented for classification purposes (see later in Section 6.4.1), the model used is a pre-trained version of DenseNet-161 [21]. The original output layer has been replaced by three convolutional layers as well as a linear layer and a Softmax activation function. The detailed architecture can be found in Appendix B.

Training

The model was trained via a *Mean Square Error* (MSE) loss function (because this loss was also already implemented for classification purposes, see Section 6.4.1) and an *Adam* [69] optimizer (with a learning rate of 0.001) during 30 epochs with batches of 8 images.

In order to improve the robustness of the model, *data augmentation* was performed. This process consists in applying a random transformation to the image given as input to the network so that the latter is not specific to the characteristics of the images (mainly, its colors). In this work, 5 transformations were implemented: no modification, blur, edge enhance, smooth and color jitter (randomly change the brightness, contrast, saturation and hue).

The evolution of the loss function during training is shown in Figure 6.6.

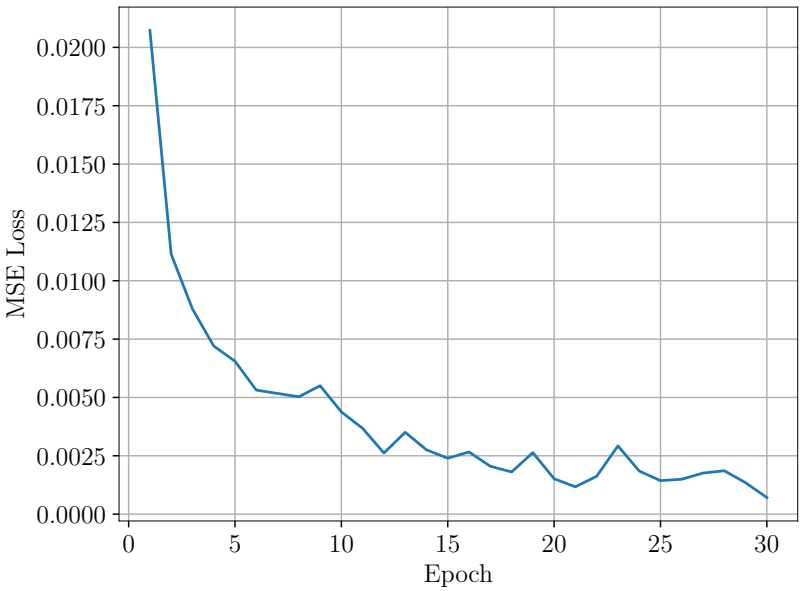


Figure 6.6: Evolution of the MSE loss function during the training of the DenseNet-161 model for the vanishing point detection problem.

We observe that the loss function well decreases over the epochs and converges toward 0, which is expected for a good model (see Section 2.2.2). In order to be sure that this was not a lucky case, several trainings were performed: all gave similar results (the same verification is performed for other trainings mentioned in this report).

Note. As the network is only trained on images from one or two different environments, it is likely that it slightly overfits these environments (be too specific). This is actually not a big problem as the drone performs its test flights in these environments. However, a *validation set* could have been used to properly quantify the overfitting.

Evaluation

Being a classification problem, a *confusion matrix* can be computed. The latter is a matrix that measures the quality of a classification.

To better understand what is a confusion matrix, let's imagine a binary classification problem: each image can be of specified class (for example, a "car") or not. Each line of the matrix corresponds to a real class (the class associated to each image of the testing set, in this example, "car" or "not car") and each column to a predicted class (the same classes, but predicted by the model). The cell located at line i and column j represents the number of elements whose real class is i and whose predicted class by the model is j. The different cells (four in total) have specific names:

- True Positive (TP): an image whose real class is "car" and predicted class is also "car";

- True Negative (TN): an image whose real class is "not car" and predicted class is also "not car";
- False Positive (FP): an image whose real class is "not car" but predicted class is "car";
- False Negative (FN): an image whose real class is "car" but predicted class is "not car".

Based on these measures, two metrics, called *precision* and *recall* [70], can be obtained via

$$precision = \frac{TP}{TP + FP}, \quad recall = \frac{TP}{TP + FN}$$

The precision represents how many of the positively classified were relevant and the recall represents how good the model is at detecting the positives. Is is important to use at least two metrics because a single metric can be easily fooled if the model predicts, for example, always the same class for all input images.

These metrics, popular to evaluate a classification problem, were used to evaluate the model.

As the classification problem has multiple classes and is totally unbalanced (there are not the same number of images of each class in the data set; in other words, there are not the same number of images for each possible cell of the vanishing point), versions of the metrics weighted according to the occurrence of each class were used (it calculates metrics for each class, and find their average weighted by the number of true instances for each class). The closer these metrics are both to 1, the better the classification.

The model obtains, on the testing set, a precision of 0.96 and a recall of 0.94. The mean inference time of the model is 0.24 s. These results seem to show a very good performance of the network. To illustrate them, some practical tests were carried out (Figure 6.7).

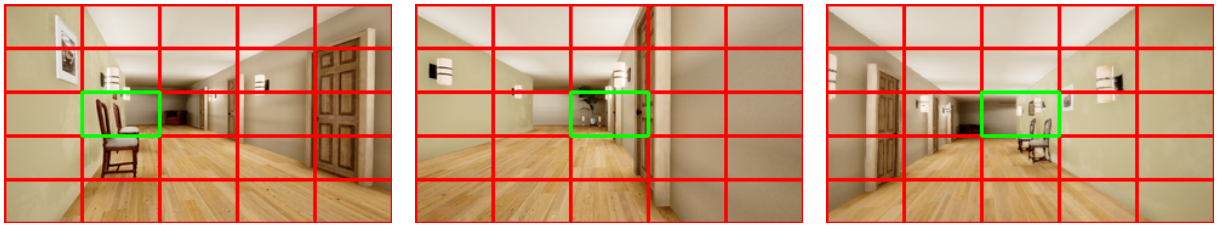


Figure 6.7: Examples of obtained results using the trained DenseNet-161 model for vanishing point detection.

Although not extremely accurate (this is impossible given the chosen size of the grid), the obtained results are very convincing. They seem to be sufficient to allow a correct alignment of the drone. Moreover, the trained model has also been evaluated on the 100 images used to evaluate the `VPClassic` and `VPEdgelets` methods (these images are not coming from the training set). It obtains a score of 100 vanishing points detected over 100, which is a very impressive result showing the robustness and efficiency of a Deep Learning model.

For these reasons, this method, that will be called `VPDeepLearning`, will be kept to create a second module implementing the ALIGN method (with the deviation δ fixed at the width of a cell).

6.4 Key points detection

This section concerns the creation of modules used to analyze images and detect key points (ANALYZE).

After the creation of modules allowing the alignment step of the drone, this work continues by focusing on the most important step of the navigation algorithm (Algorithm 2): the detection of key points (turns, crossroads and staircases) based on images captured by the drone.

For this, several methods were tested. Inspired by the different works using Deep Learning models (Chapter 2) to process images, a method for classifying images via a Deep Learning model was studied. Deep Learning models were also evaluated to perform depth estimation. Finally, the use of markers, popular in the field of robotics, was tested.

6.4.1 Image classification

Inspired by the work of Padhy et al. [22], this method allows the detection of key points of an environment based on a classification of images captured by the drone via a Deep Learning model.

In their work, the authors used the DenseNet-161 network and trained it to classify each image of a defined path in a specific action ("move forward", "turn left", etc.). This implies that a training of the model forces the drone to always follow the same trajectory.

In this work, this idea has been taken up again but in a generalized way. Instead of associating a precise action to each image, the model performs a simple binary classification ("key point" or "not key point").

Data sets

To create data sets for training the model, the drone was manually piloted in the simulated environments by capturing images at regular intervals (every 0.1 s for this task). Each set of images collected was manually annotated: for example, when the drone is piloted in a corridor, all images are annotated as being "not key point"; when the drone is piloted in a turn, crossroad or staircases, the images are annotated as "key point".

Note. An automatic image collection and annotation solution was considered: a path is planned, the different areas are segmented into "key point" and "not key point" and the images are automatically annotated based on the position of the drone at the time of capture. However, this solution is not applicable to the real world since it is not possible to accurately retrieve the position of the drone in the environment. As the manual annotation of the images is fast (since the images of the same class follow each other in the capture order), it has been kept to create the data set. However, it has the

disadvantage of making the data collection process difficult to reproduce. It would be interesting, for future work, to design an automatic image collection method that can work in a simulator *and* in the real world.

The quantities of images collected for the classes "key point" and "not key point" are necessarily very unbalanced. Indeed, an environment such as indoor corridors contains more straight lines than turns or crossroads. In order to reduce the gap between the classes, the "not key point" images were filtered, keeping only one out of three, and the key point images were augmented via a horizontal flip. It should be noted that each image was resized to 320×180 and that the α transparency channel was removed.

This resulted in 1409 images (927 "not key point" and 482 "key point") for the Indoor Corridor environment and 1489 images (842 "not key point" and 647 "key point") for the Indoor Staircase environment. Each data set was then separated into a training set (70% of the images, containing the same proportion of images of each class) and a testing set (the remaining, 30% of the images).

A small sample of the images collected is shown in Figure 6.8.

Figure 6.8: Example of images constituting the data sets for training the Deep Learning model. The first line contains images annotated as "not key point". The second and third lines contain images annotated as "key point".

Models, training and evaluation

The model used is a pre-trained version of DenseNet-161 with the last layer replaced by several convolution layers, a dense layer and a Softmax activation function. The detailed architecture can be found in Appendix B.

As with the vanishing point classification problem, the model was trained via a Mean Square Error (MSE) loss function and an Adam optimizer (with a learning rate of 0.001) during 30 epochs via batches of 8 images. Data augmentation, similar to that of the vanishing point, was performed. For the following results, the model has been trained on the Indoor Corridor data set.

The evolution of the loss function is shown in Figure 6.9.

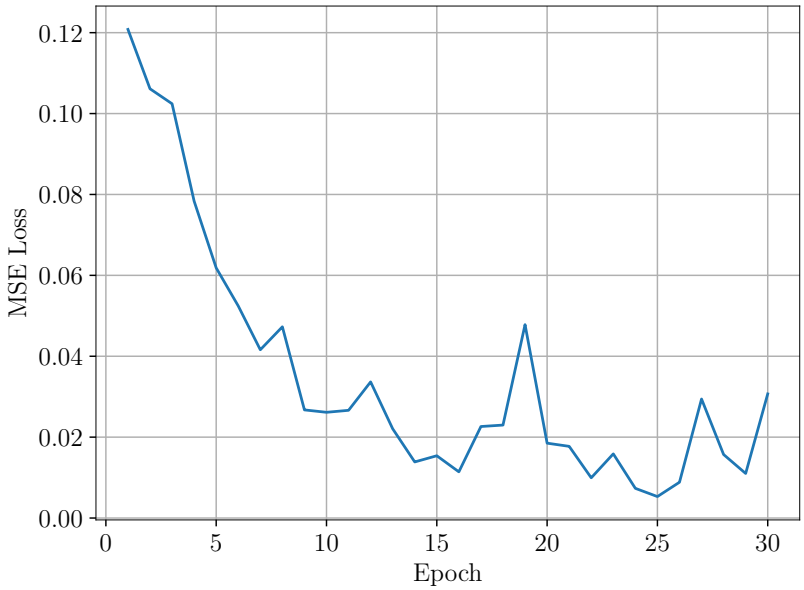


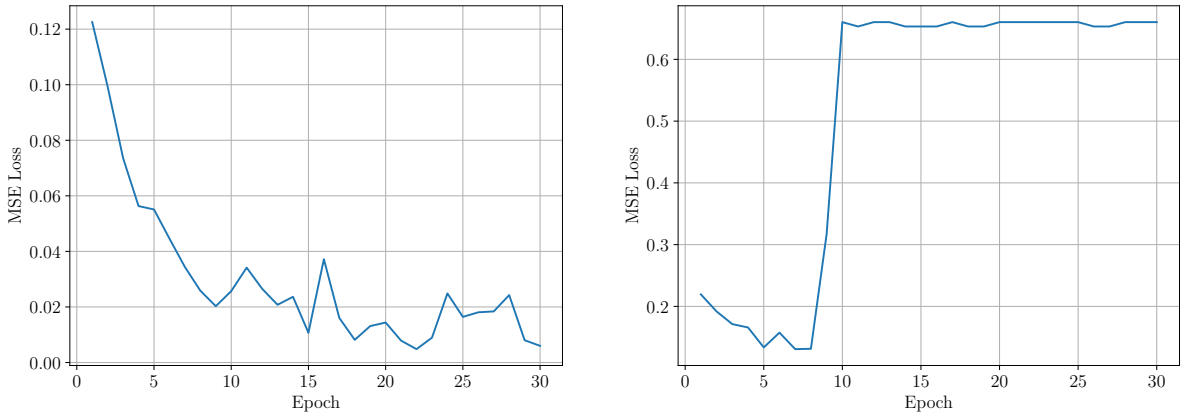
Figure 6.9: Evolution of the MSE loss function during the training of the DenseNet-161 model for the image classification problem.

We observe that the MSE loss function decreases well over the epochs and seems to converge towards 0.

Being a classification problem (unbalanced), the precision and recall metrics (weighted according to the occurrence of elements in each class) were, once again, used for the evaluation. The DenseNet-161 model obtained a precision of 1 and a recall of 0.99. The mean processing time of an image is 0.24 s.

The obtained results show that the model seems to be excellent in its classification task. However, a model such as DenseNet-161 seems very heavy for a classification task containing only 2 classes. To overcome this, lighter models have been implemented. More precisely, DenseNet-121 and a simple convolutional network created manually, which we will call "SmallConvNet", have been implemented. Their precise architectures are explained in Appendix B.

These networks were trained in exactly the same way as DenseNet-161. The evolution of their loss function is shown in Figure 6.10.



(a) DenseNet-121 (b) SmallConvNet

Figure 6.10: Evolution of the MSE loss function during the training of the DenseNet-121 and SmallConvNet models for the image classification problem.

It can be seen that DenseNet-121 seems to converge to 0 while SmallConvNet does not converge at all to 0. More precisely, we can observe that the loss function of SmallConvNet increases after 10 epochs. This can be explained by a learning rate too high making the model moving too far from a local minimum of the loss function. Different values for the parameters (including learning rate decay over time) have been tested but no real better results have been obtained.

The models were also evaluated via the precision and recall metrics, in the same way as DenseNet-161. The obtained results, as well as their mean processing time per image, are shown in Table 6.2.

Model	**Precision**	**Recall**	**Mean inference time** (s)
DenseNet-121	0.95	0.89	0.21
SmallConvNet	0.33	0.34	0.03

Table 6.2: Evaluation of the DenseNet-121 and SmallConvNet models for the image classification problem.

In view of the obtained results, despite a much faster inference time, the SmallConvNet network does not provide sufficiently qualitative results to be used in practice. This may be explained by its architecture which is certainly too simple, compared to DenseNet models, to learn correctly.

The DenseNet-121 model, on the other hand, seems to provide good results but is nevertheless inferior to those of DenseNet-161. The time saving compared to the latter is also low. For these reasons, the DenseNet-161 model has been preferred for the creation of a first module, which we will call `Vision`, allowing the implementation of the ANALYZE method.

Generalization

It is interesting to test how well a model trained on one simulated environment can generalize to another unknown simulated environment.

To this end, the DenseNet-161 model was trained, in the same way as before, on the two simulated environments, Indoor Corridor and Indoor Staircase. The model trained on Indoor Corridor was tested on Indoor Staircase, and vice-versa. The obtained results are presented in Table 6.3.

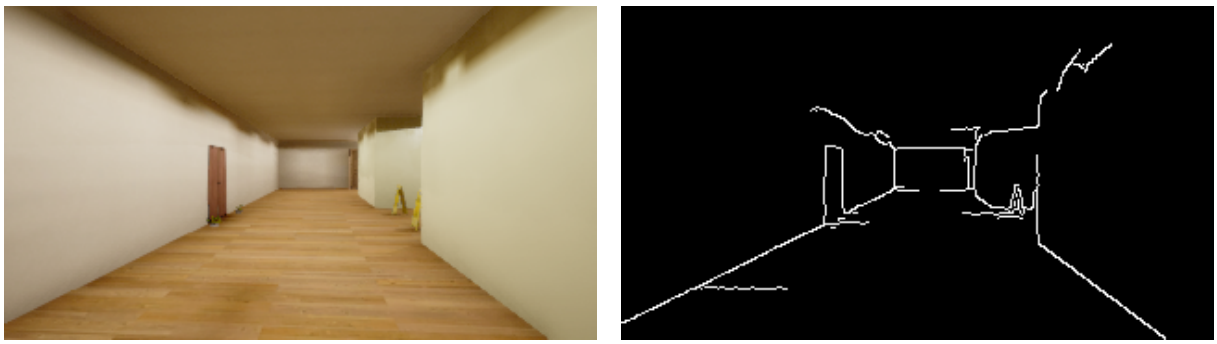
Training	**Evaluation**	**Precision**	**Recall**
Indoor Corridor	Indoor Staircase	0.80	0.69
Indoor Staircase	Indoor Corridor	0.96	0.82

Table 6.3: Evaluation of the generalizability across environments of the DenseNet-161 model for the image classification problem.

The obtained results show that the model trained on Indoor Corridor does not generalize very well to Indoor Staircase. The results degrade too much for the model to be used in practice. This can be explained by the fact that the Indoor Staircase environment contains staircases. The model trained on the Indoor Corridor has never had the opportunity to see staircases during training and therefore most likely provides poor results for them.

The model trained on Indoor Staircase, on the other hand, seems to generalize better on Indoor Corridor. Nevertheless, the results deteriorate a little, making it risky to use the model in practice to navigate the drone.

In an attempt to improve the generalization of the models, they were trained on the edges of the images and not on the images directly. This allows the model to work with the shape of the environment and to abstract from the elements that are very specific to an environment (mainly, the colors and textures). Edges were extracted from each of the training images using the Canny algorithm. An example of training image is shown in Figure 6.11.



(a) Original image (b) Extracted edges

Figure 6.11: Example of an image whose edges were extracted for training the DenseNet-161 model for the image classification problem.

The models were trained under the same conditions as before. It should be noted that no data augmentation was performed since this one, working mainly on the colors of the image, has no meaning on the edges. The new obtained results are shown in Table 6.4.

Training	**Evaluation**	**Precision**	**Recall**
Indoor Corridor	Indoor Staircase	0.98	0.79
Indoor Staircase	Indoor Corridor	0.99	0.84

Table 6.4: Evaluation of the DenseNet-161 model, trained with edges, on an unknown environment.

The obtained results show a slight improvement for the model trained on Indoor Staircase and a good improvement for the model trained on Indoor Corridor. Training the models on the edges of the images seems to be a good idea to improve the generalization of the models. A good generalization of a Deep Learning model can be a real asset to do *Transfer Learning* when the algorithms are tested on the real world. This aspect will be discussed and evaluated in Chapter 7.

6.4.2 Depth estimation

Depth estimation is a Computer Vision problem consisting in estimating the depth information of a scene based on one (or several) image(s). This can be exploited to get very useful information: distance to different elements in the scene, obstacle detection, etc. In this work, depth estimation has been used to detect if an image is a key point or not (more details later).

Depth estimation can be done on the basis of a *monocular* image (typically, an image captured by a single camera, for example the front camera of a drone) via Deep Learning models. The data to be collected to train such models is not easy to obtain. Indeed, the images must be annotated with depth information, which is only possible with adequate sensors (*e.g.* LIDAR). As the Tello EDU does not have such a sensor, a pre-trained model was used.

Model and results

The model used is MiDaS [71]. This model allows the inference of an *(inverse) relative depth map* of a monocular image. A relative depth map provides depth information of each element *relative* to other elements in the scene (but does not provide directly any real distance to camera).

The pre-trained version from PyTorch [72] was used. The latter was trained on several data sets, making the model robust to a wide variety of scenes. The model was tested on several images of the simulated environments. Some of the obtained results are shown in Figure 6.12.

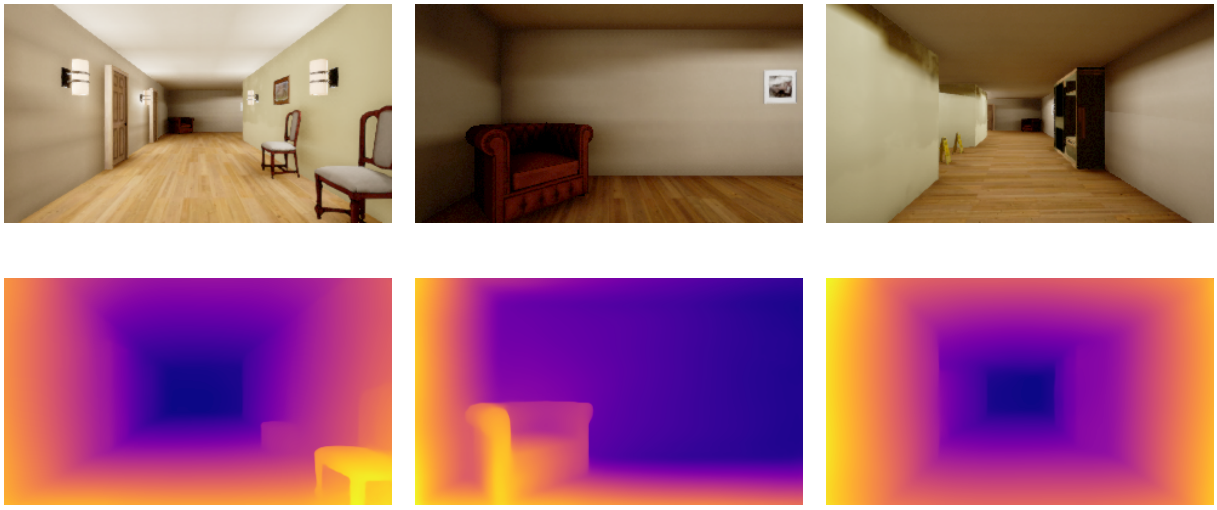


Figure 6.12: Examples of obtained results with the MiDaS pre-trained model. The first row contains the original images. The second row contains the inferred relative depth maps.

Although the model was not trained on images of the simulated environments, the obtained results seem very convincing. The model manages to correctly estimate a relative depth map based on a monocular image and highlights the different elements of the scene (for example, the chairs and the armchair in the first two images).

The mean processing time per image is 0.6 s, which is higher than the processing time of classification models.

Distance estimation

The MiDaS model provides a *relative* depth map, which means that the predicted values for each pixel in the image do not correspond to a real distance measurement from the camera.

As mentioned by the authors [73], it is possible to obtain an absolute measurement of the depth, which can give information about the distance to the camera, via the linear relation

$$absolute = a \times relative + b \tag{6.1}$$

where *relative* is the model prediction and a and b two factors to be determined.

However, the determination of the factors a and b cannot be done without additional measurements: the absolute depth of at least 2 pixels of the image must be measured and, based on the relative information from MiDaS, a system of 2 equations with 2 unknowns (a and b) must be formed and solved. In practice, more points can be used in order to have a more accurate measurement and reduce the risk of working with a degenerate case.

Practically, this requires the collection of depth information from the environment beforehand. This is not very convenient if we want to have a fully autonomous drone. Moreover, the obtained factors do not necessarily generalize to other environments. In

this work, depth estimation will therefore be used essentially to try to detect key points in the environment, and other obstacles, without obtaining a precise distance to them but simply based on the relative depth of each (zone of) pixel(s).

Key points detection

When an image is captured by the drone, it is given to the depth estimation model. The resulting relative depth map is sliced into a grid of 3×3 cells. The average depths (average of the values of each pixel) of the different cells are calculated. Based on these 9 values, the key points are deduced as follows.

- If the average depths of the central cells are *much* higher than those of the extreme cells, there is a good chance that the drone is in a corridor. The image is then considered to be a "not key point". All the other possible cases are considered to be "key point". It is, however, interesting to distinguish these cases.
- If the average depths of the central cells are *slightly* higher than those of the extreme cells, there is a chance that the drone is in a crossroads. The image is then considered to be a "key point".
- If the average depths of all the cells are the same or close to the same, there is a good chance that the drone is facing a wall. The image is then considered to be a "key point".
- If the average depths of the upper cells are higher than the average depths of the lower cells (or the contrary), there is a good chance that the drone is facing a staircase. The image is then considered to be a "key point".

Evaluation

In order to evaluate the quality of the key point detection procedure, the images of the testing set of the Indoor Corridor environment were used. Each image was classified as either "key point" or "not key point" according to the procedure presented previously. Being, once again, a classification problem, the precision and recall metrics were used. The model, combined with the key point detection procedure, obtained a precision of 0.84 and a recall of 0.81. These results are satisfactory but nevertheless much worse than those obtained via the DenseNet-161 model directly classifying the images.

The MiDaS model has been kept for the creation of a second module, which we will call `Depth`, allowing the implementation of the ANALYZE method.

6.4.3 Markers

The methods presented above do not allow, or hardly allow, an accurate estimation of the distance of the drone to the different elements of the scene. In order to overcome this, a method working with markers has been studied.

Markers are used in the field of robotics to guide the robot in its task. Taking advantage of very efficient algorithms to detect and decode them, they can be used as landmarks for localization and navigation [74] or precision tasks such as autonomous landing [75].

In this work, two types of popular markers were tested: QR codes and ArUco markers.

QR code

A *Quick Response* (QR) code is a type of two-dimensional bar code that allows information to be encoded via black modules arranged on a square with a white background [76]. One of their properties is particularly interesting in the field of robotics: they can be decoded from any position in the vicinity.

Being very popular markers, many detection and decoding algorithms exist. In this work, two of them have been tested: an algorithm coming from the OpenCV[1] library and an algorithm coming from the PyZBar[2] library. The latter will be qualified, respectively, by `QROpenCV` and `QRZBar`. Each algorithm is able to detect a QR code in an image (if there is one) and return the x and y positions of its 4 corners as well as its content.

In order to evaluate their performance, a set of 50 images of QR codes, in as many different positions as possible, was collected. Some samples are shown in Figure 6.13.

Figure 6.13: Examples of images collected for the evaluation of QR code detection and decoding methods.

A detection is considered correct if the algorithm used detects the QR code and extracts its content correctly. The mean processing times per image were also evaluated. The obtained results, on the set of 50 images collected, are presented in Table 6.5.

Method	**Score** (over 50)	**Mean processing time** (s)
`QROpenCV`	48	0.048
`QRZBar`	49	0.023

Table 6.5: Results of the evaluation of QR code detection and decoding methods.

[1] https://opencv.org/

[2] https://github.com/NaturalHistoryMuseum/pyzbar/

The obtained results show that the two algorithms allow a very good detection and decoding of QR codes. We observe, however, that the algorithm `QRZBar` is twice as fast as the algorithm `QROpenCV`.

ArUco markers

An ArUco marker [77] is a synthetic square marker with a wide black border and a binary matrix inside. The latter determines its identifier (ID). These markers are smaller than QR codes, and therefore store less information. Several versions of the ArUco markers exist (differing by their *dictionaries*, *i.e.* the values that can be encoded in the marker). In this work, the original dictionary was used, allowing the encoding of 1024 different identifiers (ranging from 0 to 1023).

The algorithm for creating, detecting and decoding ArUco markers from the OpenCV library was used. The latter is an adaptation of the original algorithm proposed by Garrido-Jurado et al. [77]. Like QR codes, this algorithm returns the x and y positions of the 4 corners of the marker as well as its content.

In order to evaluate its performance, a set of 50 images was collected, with as many different marker positions as possible. Some samples are presented in Figure 6.14.

Figure 6.14: Examples of images collected for the evaluation of the ArUco marker detection and decoding method.

The evaluation procedure is identical to that carried out for the QR codes. The obtained results are as follows: 50 were correctly detected out of 50. The mean processing time per image is 0.003 s.

The detection of ArUco markers seems to be excellent and especially much faster than QR codes. For these reasons, the ArUco marker has been preferred to QR code and been kept for the creation of a third module, which we will call `Marker`, allowing the implementation of the ANALYSE method.

Distance estimation

Based on the (known) width of the markers used, the x and y positions of their four corners and the camera parameters, it is possible to know, relatively accurately, the distance of the drone to the marker.

To do this, the camera used must be calibrated. In this work, this calibration was carried out on the basis of a simple procedure inspired by [78]. The latter simply compute

the focal length f of the camera on the basis of an image (whose distance to marker is known) using triangle similarity.

Since the camera of the drone is always the same, it only needs to be calibrated once (whatever the environment), making this distance estimation procedure very interesting.

6.5 Autonomous navigation algorithms

So far, several modules have been created to implement the ALIGN and ANALYZE methods of the generic navigation algorithm (Algorithm 2).

As a reminder, the `VPClassic` and `VPDeepLearning` modules allow the alignment of the drone (ALIGN) based on the detection of the vanishing point of an image, using, respectively, classical Computer Vision methods and a Deep Learning model. The `Vision`, `Depth` and `Marker` modules allow the detection of key points of the environment based on image analysis (ANALYZE), using, respectively, image classification, depth estimation and marker detection and decoding.

These modules were used to create several navigation algorithms, based on the generic navigation algorithm and the architecture presented in Section 6.2.

6.5.1 Vision-based algorithm

This first algorithm, called `NavVision`, uses only the `Vision` module to detect key points in the environment.

More precisely, when an image is captured by the drone, it is classified as either "key point" or "not key point" by the module. This classification is used as is to detect key points in the environment.

Note. In the rest of this chapter, the `Vision` module will be trained directly on images of the environment on which it is used. Navigation using Transfer Learning will be evaluated in Chapter 7.

6.5.2 Depth-based algorithm

This second algorithm, called `NavDepth`, uses only the `Depth` module to detect key points in the environment.

The procedure used to detect key points based on relative depth map presented previously has been used as is to detect if an image captured by the drone is a key point or not.

6.5.3 Marker-based algorithm

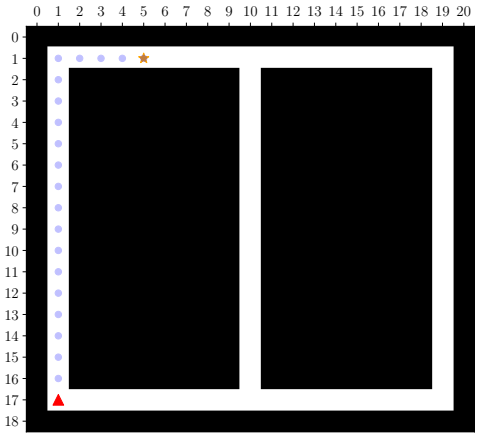
This third algorithm, called `NavMarker`, uses only the `Marker` module to detect key points in the environment.

More precisely, an ArUco marker is placed at each key point in the environment (the marker is identical for all key points and simply contains the information "1"). When the drone analyzes an image and locates a marker, it evaluates its distance to it. If the distance is less than a fixed threshold (in this work, a threshold of 1 m was chosen), the key point is considered to be detected and reached.

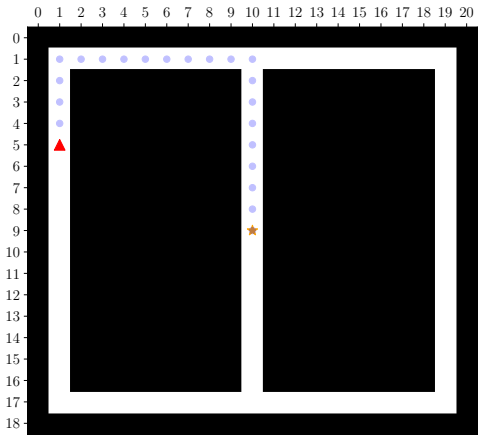
6.5.4 Evaluation

The different algorithms defined were evaluated on four trajectories, containing no obstacles that could hinder the drone in its navigation. Two trajectories were defined per simulated environment: a simple trajectory consisting of starting from a point, taking a turn and reaching the objective, and two difficult trajectories consisting of doing the same, but passing through several turns and/or crossroads.

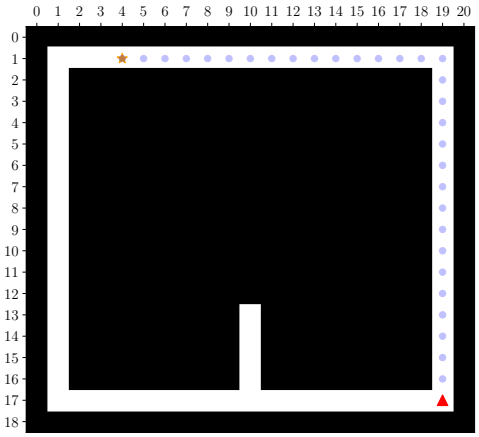
The different trajectories are shown in Figure 6.15.



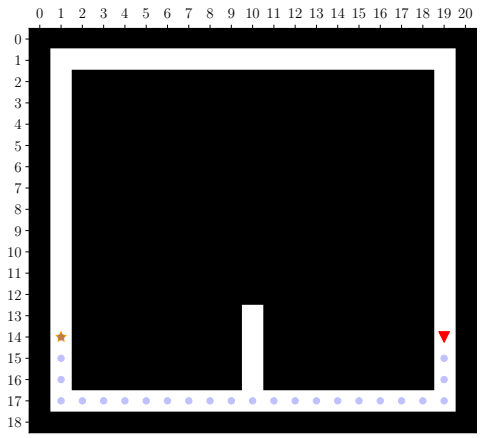
(a) Trajectory 1 (Indoor Corridor)



(b) Trajectory 2 (Indoor Corridor)



(c) Trajectory 3 (Indoor Staircase)



(d) Trajectory 4 (Indoor Staircase)

Figure 6.15: Trajectories used for the evaluation of the different navigation algorithms. The starting point of the drone, as well as its orientation, is represented by a red arrow. The objective to be reached is represented by a yellow star. The path to follow is represented by blue dots.

Each of the algorithms was evaluated with the two possible modules for vanishing point detection, `VPClassic` and `VPDeepLearning`, thus resulting in 6 different navigation algorithms.

Inspired by the work of Padhy et al. [22], a metric *Full-Flight Ratio* (FFR) was used. This is defined as the ratio of the number of times the drone has performed a valid flight over the total number of trials. A flight is considered valid when the drone goes from its starting point to its objective without taking a wrong path and without hitting any obstacles or walls.

Each trajectory was tested 30 times with each algorithm. At each new test, the starting position of the drone is very slightly modified (either a left or right horizontal displacement between 5 and 30 cm centimeters, drawn uniformly, or a rotation of between 5 and 30°, drawn uniformly, clockwise or counterclockwise). The noisy version of the simulated drone was used for all tests.

Two small examples of navigation are illustrated in videos available via the following links: `https://youtu.be/mODIlYtCzP8` and `https://youtu.be/AitgZ_cVE6k`. It should be noted that the second video was record at the beginning and then uses QR code instead of ArUco marker, but the principle is exactly the same. Moreover, the QR code was not used by the algorithm in the first video, neither for the training of the model, nor for the navigation (it was a mistake to let it on the wall during the record of the first video).

The obtained results are presented in Table 6.6.

Discussion

The obtained results allow several observations.

Firstly, there is no significant difference between the results of the algorithms using the `VPClassic` alignment module and those using the `VPDeepLearning` module. In all the tests carried out, very few errors were made due to a bad alignment of the drone. These two modules, in the context of the simulated environments, therefore appear to be effective for their task.

Secondly, except for the algorithms using markers, the difficult trajectories scored lower than the easy trajectories. The main difference was the presence of a crossroad. The latter seems to pose some difficulties for the different algorithms. This can be explained by the fact that a crossroad that allows the drone to continue straight ahead, like those of the two difficult trajectories, are not clearly and easily discernible from a simple straight line in a corridor. The various modules, especially those working with Deep Learning models, may therefore have difficulty in differentiating both (depending, perhaps, on the position of the drone at the time of image capture), leading to a possible confusion between a key point (crossroad) and a non-key point (straight line).

Thirdly, the algorithm `NavMarker` is the one providing the best results, closely followed by `NavVision`. The algorithm `NavDepth`, on the other hand, provides less good results. Despite its good results, the `NavMarker` algorithm is in reality difficult to use in practice for a navigation approach that is *fully* autonomous. Indeed, it requires that each key

Algorithm	**Trajectory**	**Full-Flight Ratio** (FFR)
`NavVision + VPClassic`	1	0.86
	2	0.73
	3	0.93
	4	0.76
`NavVision + VPDeepLearning`	1	0.93
	2	0.7
	3	0.96
	4	0.76
`NavDepth + VPClassic`	1	0.76
	2	0.46
	3	0.66
	4	0.56
`NavDepth + VPDeepLearning`	1	0.83
	2	0.53
	3	0.8
	4	0.6
`NavMarker + VPClassic`	1	0.96
	2	0.93
	3	0.96
	4	0.9
`NavMarker + VPDeepLearning`	1	1
	2	0.96
	3	0.93
	4	0.9

Table 6.6: Results of the evaluation of different navigation algorithms on four pre-defined trajectories.

point of an environment has been marked by a marker, which implies a prior preparation of the environment.

In view of the different results, the algorithm `NavVision` seems to be the most interesting for a fully autonomous navigation. Requiring only a training, it can then allow the drone to reach any objective from any starting point with only a few errors.

6.5.5 Robustness

The different algorithms created use only one module for the detection of key points in the environment. This can lead to a lack of robustness: for example, the detection of crossroads is not always good or a slight error (*e.g.* a bad prediction of a module) can be fatal (*e.g.* if a module predicts a turn in the middle of a corridor).

In an attempt to overcome this, algorithms using several modules have been created. More precisely, a main module is used to make predictions and a secondary module is

used to verify, under certain conditions, these predictions.

Providing the best results for fully autonomous navigation, the `NavVision` module was used as the primary module. Based on this, two robust algorithms were created.

- The first algorithm, called `NavVision + Depth`, uses depth estimation as a secondary module. When the `Vision` module predicts a key point, a relative depth map is inferred. If this seems more or less consistent with a key point, the key point is validated, otherwise the prediction is considered erroneous.
- The second algorithm, called `NavVision + Marker` uses markers *only* at difficult locations in an environment (*e.g.* crossroads). When the module predicts a key point with a probability below a certain threshold (usually when the model is not sure that it is a key point, which often happens at crossroads), the key point is only validated if a marker has been detected and decoded. In this work, the threshold was set at 80%.

An additional check has also been implemented. The latter checks, when the module predicts a key point, the position of the drone in the environment representation (see Section 5.3.3). If the drone is at a distance from a key point greater than a certain threshold (in this work, set to twice the accuracy of the environment representation) when a module predicts a key point, the prediction is considered erroneous and is not taken into account. This additional check will be qualified by `+ Env`.

For each robust algorithms, the module `VPDeepLearning` has been used to align the drone. These algorithms were evaluated in the same way as the non-robust algorithms on the two difficult trajectories (trajectories 2 and 4). The obtained results are shown in Table 6.7.

Algorithm	**Trajectory**	**Full-Flight Ratio** (FFR)
`NavVision + Depth`	2	0.9
	4	0.86
`NavVision + Depth + Env`	2	0.93
	4	0.83
`NavVision + Marker`	2	0.93
	4	0.96
`NavVision + Marker + Env`	2	0.96
	4	0.9

Table 6.7: Results of the evaluation of different robust navigation algorithms on two pre-defined trajectories.

Discussion

The obtained results allow several observations.

Firstly, the algorithms using the environment check (`+ Env`) do not give better results than the algorithms not using it. Thus, the modules do not (or very rarely) seem to make

mistakes when the drone is far from a key point (for example, a module will not tend to detect a key point by mistake when the drone is in the middle of a corridor).

Secondly, compared to the results of Table 6.6, the results obtained seem slightly better. These new navigation algorithms thus seem to allow a more robust navigation of the drone. We can also observe that, once again, the algorithms working with markers seem to provide better results.

We can therefore say that the idea of combining several modules to create robust algorithms is proving to be effective. In view of all the tests carried out (Tables 6.6 and 6.7), the algorithm `NavVision + Depth` seems to be the one providing the best results for a *fully autonomous* navigation. The use of markers, on the other hand, proves to be a very efficient solution but nevertheless not usable for a fully autonomous approach.

Note. The use of markers in autonomous navigation is therefore questionable; it all depends on the starting assumptions. If the objective of the drone is to be able to navigate in an environment that is totally unknown a priori and it is not possible to prepare this environment, the use of markers is impossible. On the contrary, if the drone's objective is to fly in an area known in advance (for example, to make deliveries in an area that is always the same), and this area can be prepared in advance, the use of markers is possible and will certainly be preferred.

6.6 Advanced navigation

The initial objective of this work was to make the drone navigate from a starting point to an objective autonomously in an indoor environment. However, an indoor environment is not limited to corridors and turns; several floors separated by staircases may be on the way. Moreover, in a context where the drone would be used for a long time, the problem of its low autonomy must be taken into account. To this end, and with the aim of pushing the results further, this section explores the integration and management of battery stations in an environment as well as the passage of staircases.

6.6.1 Battery station

In order to solve the problem of the drone's autonomy, charging stations could be placed at fixed locations in the environment. Depending on its battery level, the drone could plan paths passing by these stations. This aspect has already been discussed and implemented in the environment representation (see Section 5.3.4).

When the drone plans to pass by a charging station, this one could be seen as an intermediate objective. The drone would then use a navigation algorithm to, firstly, reach the charging station and, secondly, reach its objective. The main difficulty is therefore to ensure that the drone, once it arrives at the charging station, detects it and lands precisely on it.

To do this, the battery station could be localized with a marker, being at a fixed and known position in the environment. When the drone reaches the station, it could adjust

itself precisely with the help of the marker and land. After charging, the drone would take off again and navigate to its next objective.

In order to test this procedure, the module `Marker` was used. A (fictitious) battery station was imagined in the Indoor Corridor environment. Simulating a low battery level, the drone planned a path to this station and then to its objective. This path is shown in Figure 6.16.

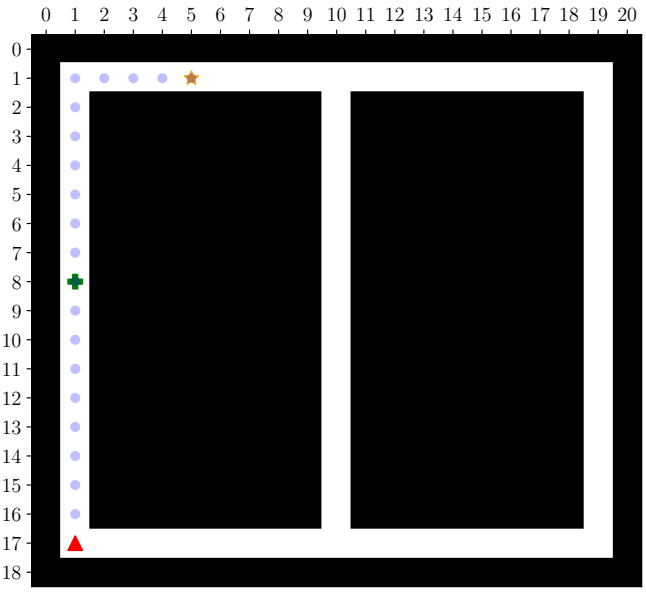


Figure 6.16: Example of a path planned by the drone and passing through a battery station (green cross) in the Indoor Corridor environment.

The drone uses the `NavVision + Depth` algorithm to move and reach its objectives. When the battery station is reached, the drone uses the `Marker` module to detect the marker and adjust to it. When the drone is aligned with the marker and at a fixed distance (depending on the position of the station), it lands. After a few seconds, the drone takes off again and flies to its final target.

This algorithm was evaluated 30 times on the trajectory of Figure 6.16, in the same way as all other navigation algorithms. The results obtained are as follows: the drone successfully completed 28 trials on 30 (FFR of 0.93). This result is very good and is in agreement with the previously obtained results. It shows that it is possible quite easily, based on the algorithms created, to include and manage charging stations in an environment.

6.6.2 Staircase

Staircases are particular key points in an environment. These can be detected via the appropriate modules (`Vision`, `Depth` or `Marker`). When a staircase is detected, it is possible to know its direction (up or down) based on the list of key points extracted from the environment (see Chapter 5).

When the drone has detected a staircase, the main difficulty is to be able to cross it. For this purpose, 3 methods have been created: a naive method, a method based on depth estimation and a method using markers.

Note. The staircases considered in this work are staircases going in one direction only. More complex staircases (with several directions or spiral) will not be addressed and are left for future work on the subject.

Naive method

This first method, named `StaircaseNaive`, consists of incremental displacements of the drone in a vertical and then horizontal direction, in a loop until the staircase is crossed.

More precisely, when the drone detects a staircase (and its direction, *e.g.* a staircase going up), the drone moves 30 cm up (or down if the staircase is going down) and 30 cm forward. The drone then captures a new image and analyzes it to deduce whether it is still a key point (in other words, whether it is still on a staircase). If it is, the drone repeats this procedure, otherwise the staircase is considered to be crossed.

Depth-based method

This second method, called `StaircaseDepth`, uses the `Depth` module.

More precisely, this method works in the same way as the naive method. However, unlike advancing by a fixed value of 30 cm, the drone captures an image of the staircase and estimates a relative depth map of it (via the `Depth` module). An example is shown in Figure 6.17.

(a) Original image

(b) Relative depth map inferred

Figure 6.17: Example of a relative depth map inferred of a staircase image via the module `Depth`.

Based on the relative depths, the drone tries to estimate its position in relation to the staircase. More precisely, the image is cut horizontally into 3 areas of equal size. Based on the mean depths of each area, the drone tries to determine whether it is close to a step or not (in the same idea as the depth-based navigation algorithm) and moves accordingly.

Marker-based method

This third method, called `StaircaseMarker`, uses the `Marker` module.

This method, unlike the others, involves placing markers at different steps on the staircase. The number of markers to be placed depends on the size of the staircase: it is important to ensure that, no matter where the drone is on the staircase, it can always see one (or more) markers.

An example of staircase with markers on it is shown in Figure 6.18.

Figure 6.18: Example of a staircase with markers placed at different steps to help the drone crossing it.

This method works on the same principles as the previous ones. However, instead of advancing a fixed distance or inferring a depth map, it captures an image, analyzes it, detects the marker and calculates the distance to it (if several markers are present, only the closest is considered). Depending on the position of the marker relative to the drone and the distance from the drone to the marker, the drone adjusts its displacements. Specifically, the drone faces the marker and then moves up until it is no longer detected. This is repeated with the next marker. If no more markers are detected, the staircase is considered to be crossed.

Evaluation

In order to evaluate the 3 methods presented, a simple path was defined in the Indoor Staircase environment. The latter, presented in Figure 6.19, does not present any difficulty other than the staircase passage. When the drone has passed the staircase, its navigation is considered complete.

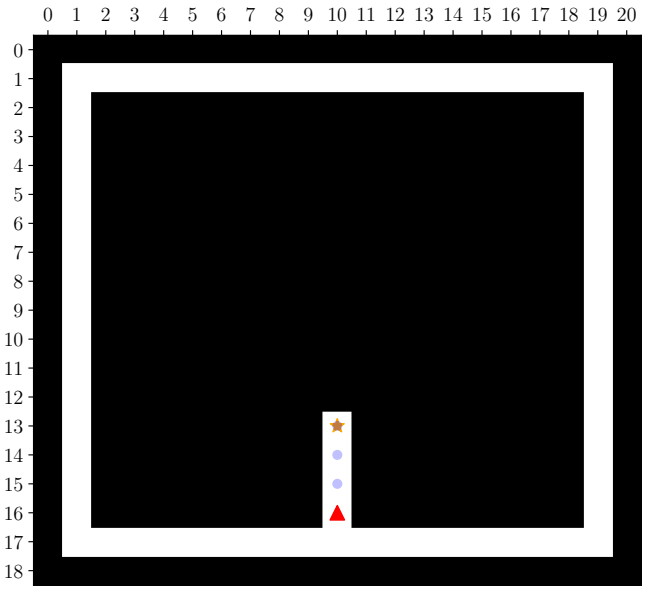


Figure 6.19: Example of a path planned by the drone and passing through a staircase in the Indoor Staircase environment.

The `NavVision + Depth` navigation algorithm was used. The trajectory was tested 30 times for each method, in the same way as all the other tests performed. The obtained results are presented in Table 6.8.

Algorithm	**Full-Flight Ratio** (FFR)
`NavVision + Depth + StaircaseNaive`	0.7
`NavVision + Depth + StaircaseDepth`	0.46
`NavVision + Depth + StaircaseMarker`	0.86

Table 6.8: Results of the evaluation of different staircase crossing methods on a pre-defined trajectory.

Discussion

The obtained results allow several observations.

Firstly, we observe that the `StaircaseDepth` method gives bad results. This can be explained by the fact that, as for the depth-based navigation algorithm, the estimation of the staircase depth is too approximate.

Secondly, it can be seen that the `StaircaseNaive` method, although naive, gives correct results. This can be explained by the fact that the drone systematically makes the same movements to pass the staircases. If its movements are not too noisy, there is a good chance that the passage will be successful. However, it is important to bear in mind that this method cannot be generalised to all types of staircases. Indeed, the values of vertical and horizontal displacements have to be adjusted according to the slope of the staircase.

Thirdly, the `StaircaseMarker` method provides good results, better than the two other methods. This can be explained by the fact that with the help of constantly visible markers, the drone can have, at any time, a precise idea of its distance to the staircases. It can therefore adjust its movements and manage staircases more easily, regardless of their slope.

In view of the obtained results, it can be seen that the depth-based method is not usable in practice, but that the naive and marker-based methods are. The naive method allows the passage of staircases without any prior preparation of the environment, but is not really accurate and not generalizable. The marker-based method provides good results but involves placing markers in advance.

6.7 Summary

In this chapter, first, a generic autonomous navigation algorithm has been designed. The latter, in order to be functional, implies the implementation of a method to align the drone (ALIGN) and a method to analyze an image and detect a key point (ANALYZE).

To align the drone, a small generic algorithm has been created. Based on the latter, several modules working with the vanishing point of an image have been implemented: `VPClassic` working with line detection, `VPEdgelets` working with edgelets detection and RANSAC and `VPDeepLearning` performing classification to predict the approximate position of the vanishing point. The modules `VPClassic` and `VPDeepLearning` provided very good results, while the module `VPEdgelets` was a little less good.

To analyze images detect key points, several modules have been realized: `NavVision` and `NavDepth`, both working with Deep Learning models, to perform, respectively, image classification and depth estimation as well as `NavMarker` allowing the detection and decoding of QR codes and ArUco markers.

These different modules have been combined to create navigation algorithms, based on the generic algorithm designed. In general, the algorithms performing image classification and ArUco marker detection provide good results. Robust version of these algorithms, obtained by combining several modules, have been tested and seem to improve the results.

Finally, staircase passage and battery station management have been tested. Three staircase passing methods were performed: a naive method, a method based on depth estimation and a method using ArUco markers. The naive and marker methods provided good results. The battery stations were integrated into the environment via ArUco markers allowing their detection. The drone successfully detected and landed on them.

Chapter 7

Real World Testing

Several navigation methods and algorithms have been designed in Chapter 6. These algorithms have been evaluated on simulated environments and have provided, for the most part, good results. The last major step of this work is to test these functional algorithms on the real world, with a real drone.

This chapter presents and discusses the results of the different methods and algorithms adapted and tested in the real world. The different obtained results will help to discuss the feasibility of autonomous drone navigation in an indoor environment.

7.1 Adaptation to real world

The real drone used is a Tello EDU, presented in Chapter 4. The controller of this drone, based on the high-level control interface presented in the same chapter, will be used.

The environment considered is the corridors the Montefiore Institute of the University of Liege. The representation of this environment presented in Section 5.4.3 will be used.

The various algorithms designed on the basis of the generic algorithm 2 will also be used without code modification. Their architecture, composed of several modules (see Section 6.2), is still valid for the real drone. The only real modifications concern the methods ALIGN and ANALYZE: the various modules created will be evaluated on real data in order to judge which modules could be used in practice or not.

7.2 Alignment

In order to align the drone on a straight trajectory when navigating in a corridor, two modules, working with the vanishing point, have been realized and tested on simulated environments: the `VPClassic` module detecting the vanishing point based on line detection and the `VPDeepLearning` module working with a Deep Learning model. Both modules were evaluated on real data.

7.2.1 Line detection

Two methods working with line detection had been evaluated previously: `VPClassic` and `VPEdgelets`. For its good results and its speed, the `VPClassic` method was retained.

The latter, with some adjustments of the different parameters of the methods (Canny algorithm and Hough transform) was evaluated on the real world. For this, a series of 60 images, varied in terms of the position of the vanishing point, was collected (by manually controlling the drone). These images were then annotated with their vanishing point in the same way as the images collected in the simulated environments. Some examples of the collected images are shown in Figure 7.1.

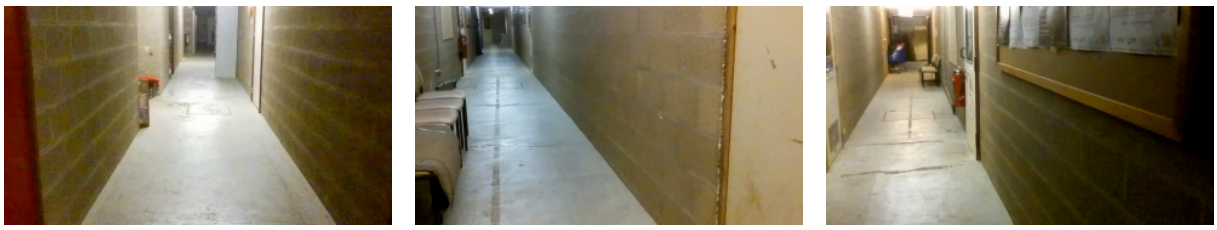


Figure 7.1: Examples of images (real world) collected for the evaluation of the `VPClassic` vanishing point detection method.

The `VPClassic` method was evaluated on these images in the same way as the images of the simulated environments (see Section 6.3.1). The obtained results are as follows: 36 vanishing points were correctly detected over the 60 images. The mean processing time per image is 0.05 s.

These results show that, despite its speed, this method is no longer efficient on real-world images. Only a little more than half of the vanishing points were detected correctly, which is too few to guarantee a correct navigation of the drone. The main problem seems to be the detection of the lines in the image. An example of a poorly detected vanishing point is shown in Figure 7.2.

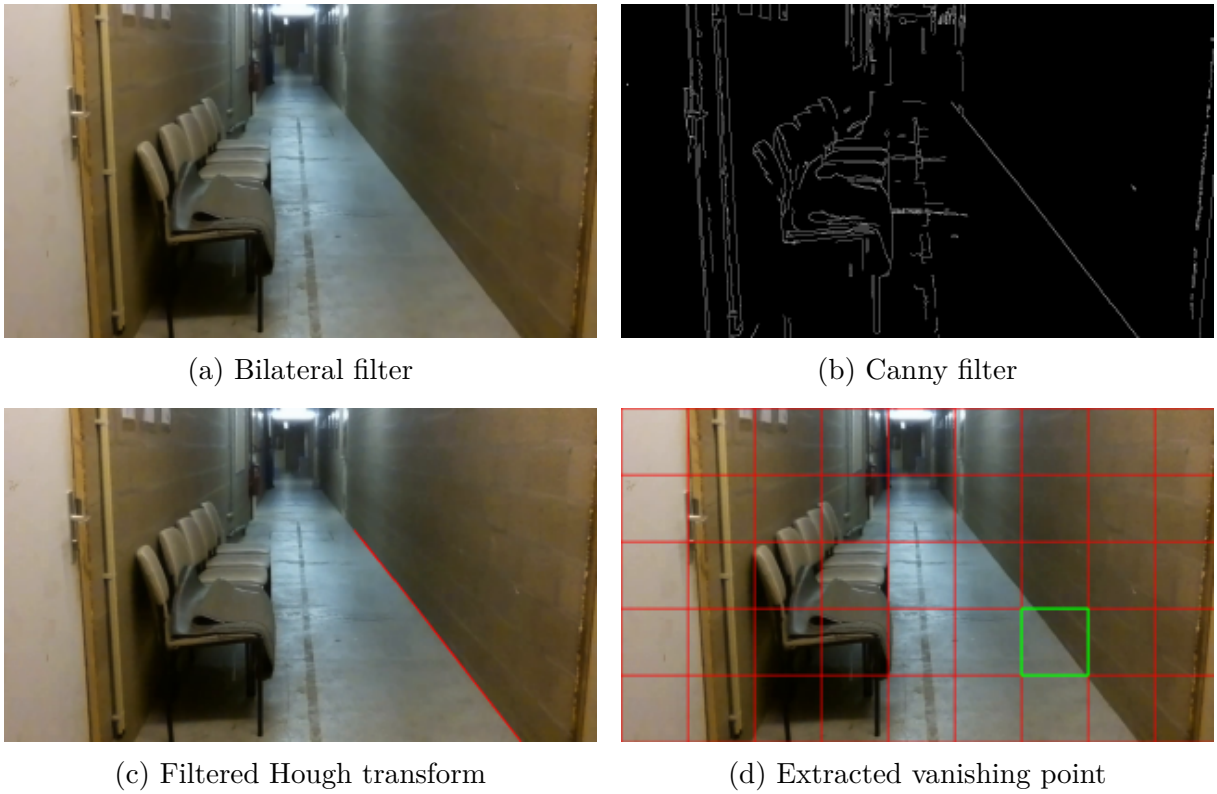
In this example, it can be seen that the (static) obstacles on the left of the image obstruct the correct detection of the lines. In addition, the lines separating the ceiling from the walls are not visible. Therefore, only one line was correctly detected, which is not enough to determine the vanishing point.

Because of these degrading results and in order not to risk damaging the drone in case of bad detection, the `VPClassic` method was not retained for testing.

Discussion

It is interesting to try to understand why the results deteriorate so much when working on real world images.

The main difficulty seems to come from line detection. As mentioned in Chapter 6, line detection is a task that is far from trivial and difficult to generalize. Indeed, an algorithm that works very well under certain conditions may no longer work under other conditions.



(a) Bilateral filter

(b) Canny filter

(c) Filtered Hough transform

(d) Extracted vanishing point

Figure 7.2: Example of results obtained via the `VPClassic` vanishing point detection method (image of real world).

The simulator images were of good quality, with good lighting conditions and clear distinction between floor, walls and ceiling. The real-world images, on the other hand, were sometimes blurred due to the movement of the drone, with sub-optimal lighting conditions and sometimes cluttered with static elements (*e.g.* chairs). These conditions certainly degrade line detection.

Note. It would certainly be possible to find improvements that would allow a better score on these images. However, it will remain difficult to design a very robust algorithm. For these reasons, and in view of the `VPDeepLearning` method presented in the next section, the `VPClassic` method has not been further developed in this work.

7.2.2 Deep Learning

A method working with a Deep Learning model has been evaluated on the simulated environments: `VPDeepLearning`. This method was also evaluated on real-world images.

For this, first, a data set of 650 images (of size 320×180) was collected (by manually controlling the drone in the environment, see Section 6.4.1). Each of the images was annotated with its cell containing the vanishing point, in the same way as the images of the simulated environments. The data set was then separated into a training set (70% of the images) and a testing set (30% of the images).

The same model was kept: a modified version of the DenseNet-161 pre-trained model.

The latter was trained in the same way and under the same conditions as for the images of the simulated environments.

The model was also evaluated using the same metrics. It obtained, on the testing set, a precision of 0.92 and a recall of 0.92. The mean processing time per image is 0.24 s. Although these results are slightly lower than those obtained on the simulated images, they are still very good. The problematic example via the line detection method was tested. The obtained results are shown in Figure 7.3.

Figure 7.3: Example of a vanishing point detected via the `VPDeepLearning` method on a real world image.

This example shows that the vanishing point has been correctly detected. In addition, the 60 images used for the evaluation of the `VPClassic` method were also tested (these images are not from the training or the testing set used for the Deep Learning model). The model obtained a score of 58 vanishing point correctly detected over 60.

These results show that the `VPDeepLearning` method performs better than the `VPClassic` method on real world images. The latter was therefore retained for the alignment of the drone in the real world.

Generalization

The Deep Learning model involves the collection of images of the considered environment. This is not always possible, especially if the environment is not known in advance.

It is therefore interesting to study the extent to which a model trained on a simulator can be generalized to the real world. For this purpose, the DenseNet-161 model trained *only* on the simulator images was evaluated on real world images. The model obtained, on the testing set, a precision of 0.21 and a recall of 0.13. These extremely poor results show that the model does not generalize at all to the real world. This can probably be explained by the fact that the simulator images are very different from the real world images and that the classification problem is complex (it is composed of 25 different classes).

Inspired by the work of Anwar et al. [43], the DenseNet-161 model trained on the simulator images was taken up and trained, again, on the real-world images. However, only the last layers of the network (the extra layers added to the basic DenseNet-161

model, see Appendix B) were re-trained; the others remained fixed. With these conditions, the obtained results are as follows: a precision of 0.68 and a recall of 0.59. These results, although still poor, are clearly more encouraging. They show that a model pre-trained on a simulator and then trained on real images can provide usable results, but is still far from the performances of a model fully trained on real images.

Better results, and therefore better generalization, could perhaps be obtained via pre-training on more and varied images of simulated environments. The simulated environments could also try to be even closer to the real world. In the context of this work, having only two simulated environments, this aspect has been left for future work on the subject.

7.3 Key points detection

Several methods for key point detection based on drone images were created and evaluated on the simulated environments: `Vision` working with a Deep Learning model to perform image classification, `Depth` working with depth estimation and `Marker` working with marker detection and decoding.

These different methods were evaluated on real-world images.

7.3.1 Image classification

Image classification in the simulated environments was performed via the DenseNet-161 model. Each image was classified as either "key point" or "not key point". This idea was retained and evaluated in the real world.

To do this, firstly, a data set of 1570 images, including 660 images of "key point" and 910 images of "not key point", was collected (by manually controlling the drone in the environment, see Section 6.4.1). The images were processed and annotated in the same way as the images of simulated environments. A sample is shown in Figure 7.4.

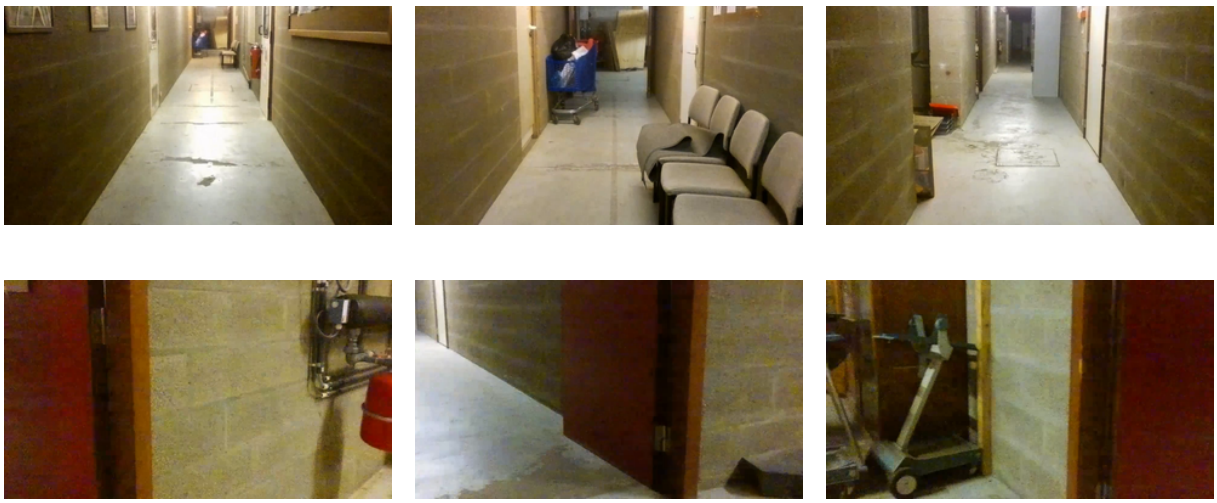


Figure 7.4: Example of images (real world) constituting the data sets for training the Deep Learning model. The first line contains images annotated as "not key point". The second line contains images annotated as "key point".

The data set was then separated into a training set (70% of the images) and a testing set (30% of the images).

The DensetNet-161 model was trained and evaluated under the exact same conditions as the images of the simulated environments. The obtained results, on the testing set, are the following: a precision of 1 and a recall of 0.99. The mean processing time per image is 0.25 s. These excellent results show that the model is perfectly capable of adapting to more complex images from the real world. The `Vision` module will therefore be retained to carry out the various tests.

Generalization

Again, it is interesting to test how well a model trained only on a simulated environment can generalize to the real world.

To do this, two versions of the DenseNet-161 model trained on the images of the simulated Indoor Staircase environment (since the real-world images contain staircases, only the Indoor Staircase environment was considered) were evaluated on the real-world images: one version trained on the images and another trained on the edges of the images. The results are presented in Table 7.1.

Training	**Evaluation**	**Precision**	**Recall**
Indoor Staircase	Real-world images	0.98	0.51
Indoor Staircase (on edges)	Real-world images	0.98	0.78

Table 7.1: Evaluation of the generalizability across (real and simulated) environments of the DenseNet-161 model for the image classification problem.

These results, which are much more encouraging than those of the vanishing point, show that the model seems to generalize well to the real world. This can be explained by

the fact that the classification problem only contains two different classes and that these classes are easily distinguishable (the model can easily makes a clear difference between a corridor and a wall).

Similar to the tests performed on simulated environments, training on image edges gives better results for the generalization of the model. These results are very interesting because they show that it could be possible to teach a drone to fly in an environment that is totally unknown to it on the basis of training carried out only on a simulator. This model, trained on the simulator's image edges, will be kept for flight tests and will be called `VisionGen`.

7.3.2 Depth estimation

Depth estimation via a pre-trained model was the second method tested on the simulated environments for key point detection (`Depth` module).

The chosen pre-trained model, MiDaS, was tested on real-world images under the same conditions as the images in the simulated environments. Some examples of the obtained results are shown in Figure 7.5.

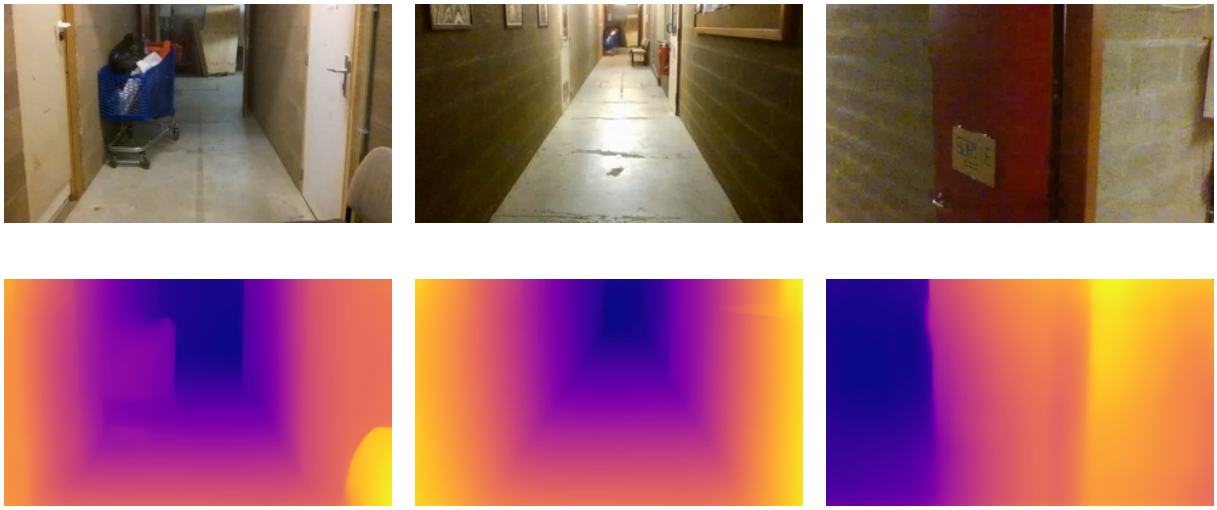


Figure 7.5: Examples of obtained results with the MiDaS pre-trained model. The first row contains the original images (real world). The second row contains the corresponding inferred relative depth maps.

The obtained results are, as for the images of the simulated environments, quite convincing. Moreover, the key point detection procedure has been evaluated on the testing set (real images) used for the DenseNet-161 model, as it was in the Chapter 6: a precision of 0.78 and a recall of 0.74 have been obtained.

The `Depth` module was therefore retained for the flight tests.

7.3.3 Markers

The use of markers to be detected and decoded was the third method used for key point detection (`Marker` module).

For their efficiency and speed of detection and decoding, only the ArUco markers were retained. The latter were evaluated on real-world images in the same way as the images of the simulated environments. For this purpose, a series of 100 images containing an ArUco marker, with as many different angles of view as possible, was collected. Some examples of the images are shown in Figure 7.6.

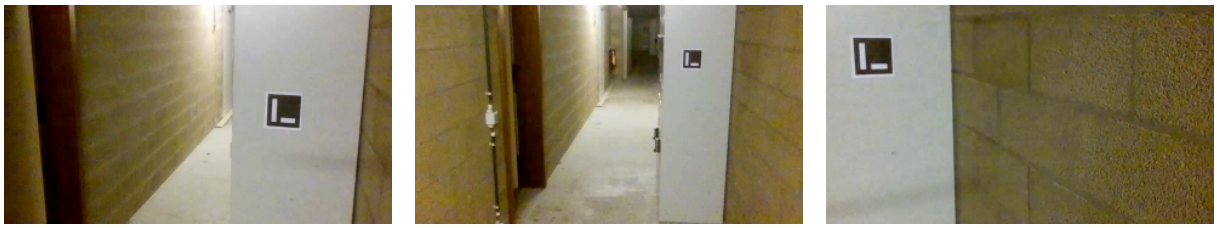


Figure 7.6: Examples of (real-world) images collected for the evaluation of the ArUco marker detection and decoding method.

The obtained results are as follows: 100 markers were detected and decoded correctly over the 100 images. The mean processing time per image is 0.006 s.

Not surprisingly, the obtained results are, as for the simulated environments, excellent. For this reason, the `Marker` module was retained for the flight tests.

7.4 Autonomous navigation algorithms

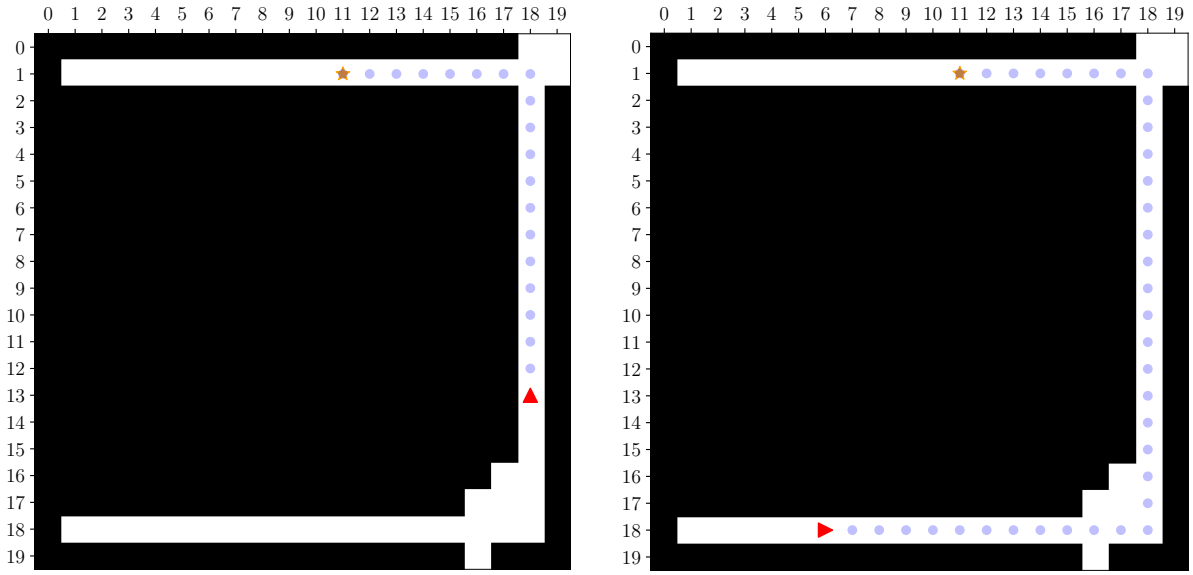
Several autonomous navigation algorithms have been created and evaluated on the simulated environments: `NavVision` using the `Vision` module, `NavDepth` using the `Depth` module and `NavMarker` using the `Marker` module.

These different algorithms have been tested in the real world. The different modules have been adapted to the real world, as explained above. A new navigation algorithm was also created based on the module `VisionGen`: `NavVisionGen`. The latter is exactly the same as the `NavVision` algorithm but works with the model trained on the simulator (`VisionGen`) in order to experiment with Transfer Learning.

The different robust algorithms, presented in Section 6.5.5, have also been evaluated. As mentioned in this same section, the verification based on the environment (`+ Env`) has not been kept.

Finally, each algorithm uses the `VPDeepLearning` module for the alignment of the drone.

Two trajectories were defined, in the Montefiore Institute, to evaluate the different algorithms. These are shown in Figure 7.7.



(a) Trajectory 1

(b) Trajectory 2

Figure 7.7: Trajectories used for the evaluation of the different navigation algorithms (real world). The starting point of the drone, as well as its orientation, is represented by a red arrow. The objective to be reached is represented by a yellow star. The path to follow is represented by blue dots.

The first is a simple trajectory containing only one turn. The second is a more difficult one containing two turns. The first turn of this difficult path is actually almost a crossroads as it is an open space with staircases and several recesses in the walls.

Each algorithm was evaluated in the same way as for the simulated environments. However, due to technical constraints with the battery autonomy of the drone, each trajectory was only tested 10 times. The obtained results are presented in Table 7.2.

A flight demonstration on the trajectory 1 using the `NavVision` algorithm is available via the following link: https://youtu.be/-GHyQnuI1Gk.

7.4.1 Discussion

The obtained results allow several observations.

Firstly, in general, the results show that the drone is able to navigate from a starting point to an objective autonomously in a real indoor environment, based solely on the monocular images of its RGB front camera.

Secondly, we observe that, once again, the algorithms working with markers provide the best results. This is not a surprise since markers are detected very efficiently and allow a (more or less accurate) estimation of the distance of the drone to the key point. However, as already mentioned, markers can only be used if the environment is accessible for pre-processing.

Thirdly, the algorithm `NavVision` provides good results. In the vast majority of trials,

Algorithm	**Trajectory**	**Full-Flight Ratio** (FFR)
`NavVision`	1	0.8
	2	0.7
`NavVisionGen`	1	0.7
	2	0.6
`NavDepth`	1	0.4
	2	0.2
`NavMarker`	1	0.9
	2	0.9
`NavVision + Depth`	1	0.9
	2	0.7
`NavVision + Marker`	1	1
	2	0.9

Table 7.2: Results of the evaluation of different navigation algorithms on two pre-defined trajectories (real world).

the drone succeeded in reaching its objective. In addition, the algorithm `NavVisionGen` also seems to provide correct results. Although a little less good, they nevertheless show that a model trained on simulator can be used in the real world to navigate a drone.

Fourthly, we observe that the `NavDepth` algorithm does not give good results. Having a too rough estimation of the environment (especially since it is a pre-trained model that has not been fine tuned on the considered environment), the drone tends to make too many errors in its detections.

Finally, we observe that robust algorithms provide slightly better results than non-robust algorithms.

In view of the obtained results, it is possible to fly a drone autonomously in an indoor environment. Markers are, for the moment, the most efficient solution. However, they cannot be used when the environment cannot be processed in advance. In this case, the algorithm `NavVision` can be used. The latter has proved, in all the tests carried out, to be really efficient and quite simple for the detection of key points in the environment. The good results obtained with the `NavVisionGen` algorithm are very promising and show that Transfer Learning can be a feasible solution for creating drones capable of navigating in environments that are totally unknown to them.

7.5 Advanced navigation

In order to further develop the results, the management of battery stations in the environment and the passage of staircases were tested in simulated environments. These were also tested in the real world.

7.5.1 Battery station

In order to test the drone's ability to plan a path to a battery station and land there, a fictitious battery station was imagined at the Montefiore Institute. This can be located by an ArUco marker placed directly above it on the wall.

Based on its representation of the environment and the position of this known charging station, a simple path was planned (Figure 7.8).

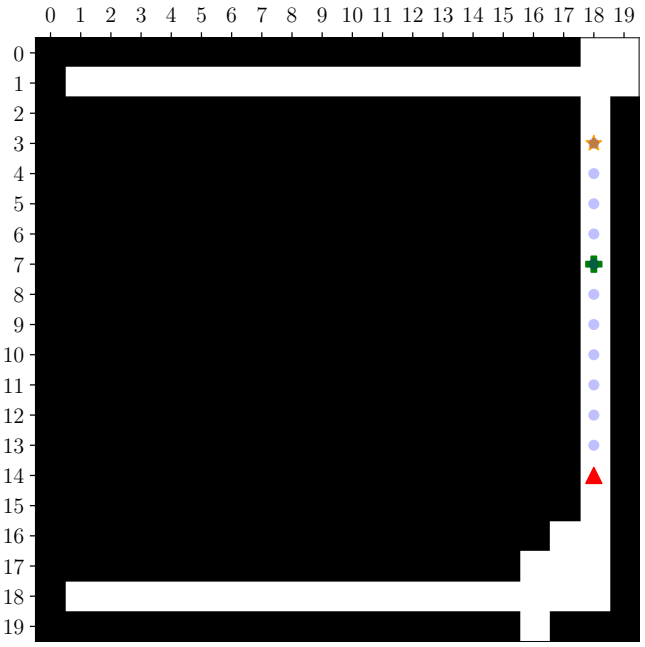


Figure 7.8: Example of a path planned by the drone and passing through a battery station (green cross) in the real-world environment.

Similar to the tests performed in the simulated environment, the drone uses the `NavVision + Depth` algorithm to move and reach its objectives. When the drone reaches the battery station, it performs a 360° rotation to try to locate the ArUco marker. When the ArUco marker is located, the drone uses the `Marker` module to align itself and land on the battery station. It stops for a few seconds and then heads back to its final destination.

A small demonstration is available in a video via the following link: `https://youtu.be/M0i27bwo5Z8`.

This algorithm was evaluated 10 times in the same way as the one used for the simulated environment. The results obtained are as follows: the drone successfully completed 9 of the 10 trials. This very good result shows that it is possible quite easily, even in a real environment, to include charging stations in the environment, which is a real advantage for drones with low autonomy.

7.5.2 Staircase

Different methods for staircase crossing were created and tested on the simulated environments: a naive method, a method based on depth estimation and a method using markers. As the depth estimation method did not provide satisfactory results in the simulated environment, it was not tested in the real one.

To test the different methods, a simple path through a staircase in Montefiore was defined. This path, shown in Figure 7.9, presents no other difficulties apart from the staircases.

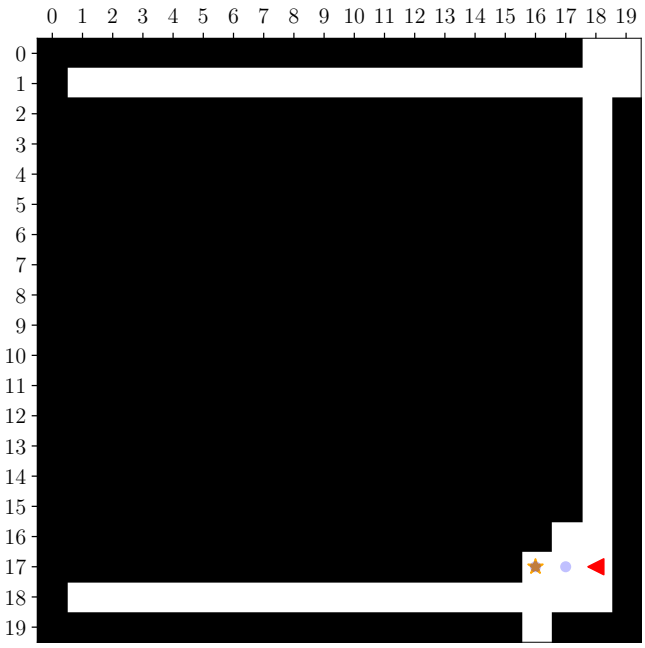


Figure 7.9: Example of a path planned by the drone and passing through a staircase in the real-world environment.

It should be noted that this staircase, in the real world, turns in several places before reaching the top floor. Since the algorithms are not designed for this type of staircase, the drone is stopped when it reaches the first landing and the passage is considered successful.

Identical to the simulated environment tests, the `NavVision + Depth` algorithm was used. The trajectory was tested 10 times. The obtained results are shown in Table 7.3.

Algorithm	**Full-Flight Ratio** (FFR)
`NavVision + Depth + StaircaseNaive`	0.8
`NavVision + Depth + StaircaseMarker`	0.9

Table 7.3: Results of the evaluation of different staircase crossing methods on a pre-defined trajectory (real-world).

A small demonstration, using `StaircaseMarker`, is available via the following link: https://youtu.be/BaQX4N4wwAM.

It can be seen that, as with the simulated environment tests, both methods appear to be effective for staircase passage. However, as explained above, the naive method cannot be generalized to all types of staircases and may not work in all types of environment. In addition, it is not robust. On the contrary, the method using markers can be adapted to several types of staircases (differing by their slope) and is more robust since it allows the drone to know its distance to the staircases.

These fairly simple tests obviously do not allow for a complete passage of difficult staircases but are nevertheless very encouraging and pave the way for future work on complex staircases.

7.6 Summary

In this chapter, the different methods tested on simulator have been adapted and evaluated in the real world.

First, the modules `VPClassic` and `VPDeepLearning` allowing the alignment of the drone via the calculation of the vanishing point of an image were tested. The module `VPClassic` gives much less good results than on simulator. The module `VPDeepLearning` still gives, on the other hand, good results.

The modules `NavVision`, `NavDepth` and `NavMarker` were also tested. The obtained results on the real world still seem good. The generalization of the model used for the module `NavVision` has also been tested. For that, the model was trained on simulator and directly used on the real world. The obtained results show that this solution, despite slightly lower scores, is promising.

Based on the modules providing good results, different navigation algorithms have been tested. The algorithm working with depth estimation does not provide good results. The other algorithms, working with image classification and marker detection, provide good results. The robust algorithms have also been tested and, once again, improve the obtained results.

Finally, the staircase passage and battery stations have been tested. The naive method and the method using markers allowed the drone to successfully pass staircases. The battery stations, on the other hand, were successfully detected using ArUco markers.

Chapter 8

Future of Autonomous Drones

This work focused on the realization of autonomous navigation algorithms for miniaturized drones in indoor environments free of dynamic obstacles. However, this is only a small part of the very rich and growing field of autonomous drones.

In order to further develop such systems, it is interesting to ask about their future: are current technologies suitable for developing such systems in all kinds of environments and situations? Are they viable in our society? Can they replace humans in their work in the near future? Do current laws allow such integration?

As a prelude to the conclusion, this chapter looks at the future of autonomous drones, from a technical and legal point of view.

8.1 Technical point of view

The information in this section comes, mostly, from [79].

Autonomous drones are a real technological challenge. Despite their small size, they are generally equipped with multiple sensors allowing them to perceive their environment. Technological advances aim to improve and optimize these flying machines and their components as much as possible.

8.1.1 Data processing

An autonomous drone usually collects a very large amount of data from its multiple sensors. This data requires processing, sometimes time-consuming, which can slow down the drone and prevent smooth real-time navigation. Although processors and graphics cards are evolving and becoming smaller, it seems difficult to envisage complete processing of all the data directly on board the drone, especially with complex sensors or huge Deep Learning models.

One solution is to use cloud technologies. Powerful Artificial Intelligence models, constantly fed with data, could run and perform heavy calculations remotely. The drone would transmit its data to the servers and receive analysis and results. This would drastically reduce their computing load.

However, for smooth and fast navigation, the transmission and reception of data must be very fast. Currently, even the fastest 4G network is limited in performance for such transmission. Recently, autonomous flight trials using 5G have shown very promising results [80]: the very high data rate of 5G has allowed for better real-time video quality and faster data processing.

In a future where big data will be the main driver of autonomous systems, and where processing operations will be done in the cloud, a development of fast communication means is essential.

8.1.2 Communication

A drone can act alone but also in collaboration with other drones. We then talk about a "drone swarm". To be effective, a swarm of drones must be able to manage itself for a large part of the operations: each drone must know the position of its neighbors, all the drones must be able to coordinate, etc. To this end, good communication between each drone is essential.

Communication does not stop with swarming drones. One could imagine communication between autonomous systems (*e.g.* between autonomous cars and autonomous drones) for fast and efficient transmission of information (*e.g.* warning of imminent danger).

Good communication is really one of the keys to successful autonomous systems. Whether they are a swarm drones or other machine, means of communication, directly between devices or passing through relays, exploiting notably 5G, are studied, for example in [81] or [82].

8.1.3 Battery

The battery is one of the main limitations of drones. Ranging from a few minutes on average to several hours for the most powerful drones, it does not allow for long term use.

This problem, which also affects many technological devices, is constantly improving: the batteries produced are increasingly powerful and the technologies increasingly efficient, thus reducing battery usage.

However, due to the very small size of some drone models, there will always be a physical limitation. One solution would be to set up drone charging stations (much like charging stations for electric cars). The drone could locate these stations and land there to charge. These stations could be placed in places such as building roofs, for example. Such technologies have already been studied, for example in [83] or [84].

8.1.4 Limits of artificial intelligence

Modern artificial intelligence methods, such as the ones explored in this work, are able to perform complex tasks. However, it is interesting to reflect on their limits: are these methods reliable? To what extent can we trust them?

These systems work on the basis of training on large amounts of data. If these data are biased, for example because of wrong annotations or because they are too specific to a particular situation, the models may not work correctly in all situations. Moreover, the collection of this data may raise confidentiality issues: would an autonomous drone be allowed to keep the images captured in an urban environment, potentially containing the identities of several people?

Finally, although powerful, these systems have a cost: a powerful hardware is necessary to realize the trainings and large storage bays are necessary to keep the large quantities of data used.

It is obvious that artificial intelligence systems will be at the heart of the development of new technologies, such as autonomous drones, but it is important to keep in mind the various existing limitations and to say that these models are far from perfect.

8.2 Legal point of view

The rules concerning drones are managed by competent authorities depending on the geographical area. In Europe, it is the European Union Aviation Safety Agency (EASA) [85] that has set the main rules.

Each country also provides regulations concerning drones. For example, the Belgian State has a Royal Decree (10 April 2016) on the use of remotely piloted aircraft on its territory [86]. In general, drones do not yet have unified international regulations. As they are developing very rapidly, the laws sometimes struggle to keep up with the technological advances in the field.

Indeed, drones qualified as autonomous by the EASA ("*An autonomous drone is able to conduct a safe flight without the intervention of a pilot. It does so with the help of artificial intelligence, enabling it to cope with all kinds of unforeseen and unpredictable emergency situations.*" [87]) fall in a category called "specific". The latter requires the pilot to make a declaration, or even obtain authorization from the agency, for the flight of the drone. In addition, a wide range of constraints are imposed: there must always be a pilot associated with the drone and this pilot must be present during the flight, the drones can only fly in areas designated for this purpose, and these areas generally exclude public urban areas.

In general, the limitations are still too strong to expect to see autonomous drones in tomorrow's society. Moreover, there is no international consensus on these limitations. Although technologies are evolving rapidly, laws need to adapt to see such systems develop.

8.3 Discussion

At present, the technologies are still under development and the laws are not sufficiently adapted to expect to see fully autonomous drones in our society in the very near future.

Nevertheless, it is clear that drones will be an integral part of our society within a few years. Their multiple applications and benefits are no longer up for debate. A large number of companies focusing on drones have emerged in recent years, gradually pushing these machines to become more popular and democratic, while becoming more sophisticated.

Among others, we can notably mention the company Humanitas [88] working with drones for rescue missions. Founded about ten years ago by a doctor who realized the critical lack of technology during catastrophic situations (*e.g.* the earthquake in Haiti in 2010), the company now provides solutions working with one or more autonomous drones to map an environment, locate people, extend a communication network, and many other applications.

Chapter 9

Conclusion and Future Work

9.1 Summary

The aim of this work was to study, create and evaluate autonomous navigation algorithms allowing a small drone with few sensors (a Tello EDU, mainly with a single RGB frontal camera) to navigate without the supervision of a human pilot in an indoor environment (the corridors of the Montefiore Institute) free of dynamic obstacles. The assumption that the drone has access to a simple representation of its environment was made.

Firstly, after a preliminary review of the state of the art about the subject, a simulated environment was set up to perform initial tests in a safe way and without real world constraints. The simulator chosen was Unreal Engine 4 with the AirSim plugin and two simulated indoor environments have been created.

Secondly, a high-level control interface was designed. The latter is an additional layer to existing drone's manufacturer control interface allowing navigation algorithms to be usable with any drone model.

Thirdly, a representation of the environment, exploitable by the drone, was created. Using a binary occupancy grid, this representation allows the planning of paths and the extraction of information (called "key point") such as the position of turns, crossroads and staircases and their associated actions.

Fourthly, based on the implemented resources, several autonomous navigation algorithms have been designed and tested, first on simulator and then in the real world. Vanishing point detection methods, via line detection and neural network, are used to align the drone. Three main methods have been used to design algorithms: image classification (using Deep Learning), depth estimation (using Deep Learning) and markers (ArUco) detection. Initially designed to follow a simple path in a single floor, more advanced techniques to manage staircases but also battery stations have been implemented.

Finally, the future of such autonomous systems, from a technical and legal point of view, was discussed.

9.2 Results

The algorithms developed were tested in the simulator and then in the real world. The obtained results show that it is possible to navigate the drone autonomously: in the majority of cases, the drone successfully reached its objective without any human supervision.

The algorithms working with image classification using Deep Learning models provide good results and seem to be robust to variations in the environment (change of scenery, variation in luminosity, etc.) but have difficulty in handling complex areas (*e.g.* crossroads). The latter have been improved by combining the other methods developed (depth estimation and markers). The algorithms working only with depth estimation showed mitigated results: they mainly lacked precision and robustness. The utilization of markers are very reliable but can only be used for a complete navigation if the environment can be prepared in advance. They have therefore been used mainly to improve the robustness of the other algorithms, especially in difficult passages such as crossroads or staircases instead of a real stand-alone solution.

Finally, the more advanced tests concerning staircase passage and battery station management proved successful. Guided mainly by markers, the drone managed to pass simple staircases and to detect a battery station and land on it accurately.

In conclusion, autonomous drone navigation in an indoor environment free of dynamic obstacles is possible based on drone vision only with Deep Learning models, and sometimes guided by markers if possible. Although promising, it is important to keep in mind that the results were obtained in simple environments. In a real-world case, as drones will be flying in complex environments, these methods pave the way for future work on the subject which would mainly aim at improving their robustness and safety.

9.3 Future work

Autonomous navigation of drones is a too complex subject to cover all aspects in a single work. The autonomous systems implemented in this project still have a lot of possible improvements in several aspects.

9.3.1 Complex environment

The environment considered was relatively simple: it consisted mainly of straight lines and right-angle turns. In a more advanced use of drones, more complex environments should be considered: turns of any angle, open areas (*e.g.* a large entrance hall), restricted areas and obstacles to cross (*e.g.* a door to pass through), etc.

This would imply redesigning not only the representation of the environment but also the way the drone moves: the vanishing point would no longer be specifically adapted to all rooms, turns would no longer be limited to a simple 90° rotation, Deep Learning models would have to handle a larger collection of different key points, etc.

9.3.2 Dynamic obstacles

In this work, the environment considered was free of dynamic obstacles (*e.g.* a person walking through the corridors). However, these are unavoidable in the real world and must be considered.

Deep learning models of depth estimation have their limitations in detecting unfamiliar and unusual features of the training environment. Object detection models, such as YOLO [89], exist and could allow real-time detection of obstacles.

A simple algorithm could be to analyze, via YOLO, each image captured by the drone and land when a nearby obstacle is detected. As long as the obstacle is present, the drone waits. When the obstacle is no longer in the way, the drone could take off again and continue. A more complex algorithm could implement a way to dodge obstacles during flight.

9.3.3 On-board processing

The Tello EDU is a drone that receives its instructions from a remote computer via Wi-Fi. No processing is directly on board; everything is calculated on an external machine, thus taking advantage of high computing power.

To minimize the delay caused by wireless communication, the drone should process the data itself. This would mean working with limited computing power and would pose a whole series of optimization problems to solve.

As discussed in Chapter 8, cloud computing is a technology that is developing more and more and allows heavy calculations to be carried out on a remote machine. The efficiency, advantages and disadvantages of this technology could be compared to on-board processing in order to obtain an optimal solution according to the needs.

9.3.4 Robustness and safety

If autonomous drones are ever used extensively in society, they must be robust and safe.

A robust drone should be able to minimize errors in its movements and behave appropriately, even if an error is made. Currently, in this work, if a module provides an erroneous prediction, or if a marker is obscured by an obstacle, the drone is very likely to fail to act correctly. This rarely happens in the simple environments considered in this project but could happen more often in complex environments.

A safe drone should be able to act in such a way as to minimize, if not completely avoid, any danger to others. Like other autonomous vehicles, such as cars, the drone could try to predict potential dangers in advance, and thus avoid them, but also communicate with other drones and relay important information. The drone could also maintain a minimum safe distance and fly, at most, at a relatively high altitude.

Bibliography

[1] Wikipedia. "Unmanned aerial vehicle". 2021. URL: https://en.wikipedia.org/wiki/Unmanned_aerial_vehicle (visited on 04/10/2021) (pages 1, 4).

[2] Kashyap Vyas. "A Brief History of Drones: The Remote Controlled Unmanned Aerial Vehicles (UAVs)". 2020. URL: https://interestingengineering.com/a-brief-history-of-drones-the-remote-controlled-unmanned-aerial-vehicles-uavs (visited on 04/10/2021) (page 1).

[3] Wikipedia. "Aircraft". 2021. URL: https://en.wikipedia.org/wiki/Aircraft (visited on 04/16/2021) (page 4).

[4] AltiGator. "Drone, UAV, UAS, RPA or RPAS..." 2021. URL: https://altigator.com/en/drone-uav-uas-rpa-or-rpas/ (visited on 04/16/2021) (page 4).

[5] Le Robert. "Drone". 2021. URL: https://dictionnaire.lerobert.com/definition/drone (visited on 04/16/2021) (page 4).

[6] studioSPORT. "À la découverte des principaux composants d'un drone". 2021. URL: https://www.studiosport.fr/guides/drones/de-quoi-est-compose-un-drone.html (visited on 04/16/2021) (page 5).

[7] Ansys. "The Challenges to Developing Fully Autonomous Drone Technology". 2019. URL: https://www.ansys.com/blog/challenges-developing-fully-autonomous-drone-technology (visited on 04/16/2021) (page 5).

[8] SkyHopper. "The Current Development And Application Of Autonomous UAVS". 2021. URL: https://www.skyhopper.biz/autonomous-uavs/ (visited on 04/16/2021) (page 5).

[9] NVIDIA. "NVIDIA Jetson Solutions For Drones And UAVS". 2021. URL: https://www.nvidia.com/fr-fr/autonomous-machines/ (visited on 04/16/2021) (page 6).

[10] SenseFly. "Drones". 2021. URL: https://www.sensefly.com/drones/ (visited on 04/16/2021) (page 6).

[11] DJI. "DJI Manifold 2 Onboard Supercomputer Transforms Drones Into Autonomous Robots". 2021. URL: https://www.dji.com/lv/newsroom/news/dji-manifold-2-onboard-supercomputer-transforms-drones-into-autonomous-robots (visited on 04/16/2021) (page 6).

[12] Philipp Foehn, Dario Brescianini, Elia Kaufmann, Titus Cieslewski, Mathias Gehrig, Manasi Muglikar, and Davide Scaramuzza. "Alphapilot: Autonomous drone racing". In: *arXiv preprint arXiv:2005.12813* (2020) (page 6).

[13] William Power, Martin Pavlovski, Daniel Saranovic, Ivan Stojkovic, and Zoran Obradovic. "Autonomous Navigation for Drone Swarms in GPS-Denied Environments Using Structured Learning". In: *IFIP International Conference on Artificial Intelligence Applications and Innovations*. Springer. 2020, pp. 219–231 (page 6).

[14] Alireza Nemati, Mohammad Sarim, Mehdi Hashemi, Eric Schnipke, Steve Reidling, William Jeffers, Jesse Meiring, Padmapriya Sampathkumar, and Manish Kumar. "Autonomous navigation of uav through gps-denied indoor environment with obstacles". In: *AIAA SciTech: Infotech@ Aerospace, paper AIAA* 715 (2015) (page 7).

[15] Stefan Kohlbrecher, Oskar Von Stryk, Johannes Meyer, and Uwe Klingauf. "A flexible and scalable SLAM system with full 3D motion estimation". In: *2011 IEEE international symposium on safety, security, and rescue robotics*. IEEE. 2011, pp. 155–160 (page 7).

[16] Roland Brockers, Martin Hummenberger, Stephan Weiss, and Larry Matthies. "Towards autonomous navigation of miniature UAV". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*. 2014, pp. 631–637 (page 7).

[17] Simon J Julier and Jeffrey K Uhlmann. "New extension of the Kalman filter to nonlinear systems". In: *Signal processing, sensor fusion, and target recognition VI*. Vol. 3068. International Society for Optics and Photonics. 1997, pp. 182–193 (page 7).

[18] Wilbert G Aguilar, Vinicio S Salcedo, David S Sandoval, and Bryan Cobeña. "Developing of a video-based model for UAV autonomous navigation". In: *Latin American Workshop on Computational Neuroscience*. Springer. 2017, pp. 94–105 (page 7).

[19] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. "Deep learning". Vol. 1. 2. MIT press Cambridge, 2016 (page 7).

[20] Karim Amer, Mohamed Samy, Mahmoud Shaker, and Mohamed ElHelw. "Deep convolutional neural network based autonomous drone navigation". In: *Thirteenth International Conference on Machine Vision*. Vol. 11605. International Society for Optics and Photonics. 2021, p. 1160503 (page 8).

[21] Karen Simonyan and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition". In: *arXiv preprint arXiv:1409.1556* (2014) (pages 8, 42).

[22] Ram Prasad Padhy, Sachin Verma, Shahzad Ahmad, Suman Kumar Choudhury, and Pankaj Kumar Sa. "Deep neural network for autonomous uav navigation in indoor corridor environments". In: *Procedia computer science* 133 (2018), pp. 643–650 (pages 8, 10, 45, 57).

[23] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. "Densely connected convolutional networks". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 4700–4708 (pages 8, 94, 95).

[24] HY Lee, HW Ho, and Y Zhou. "Deep Learning-based Monocular Obstacle Avoidance for Unmanned Aerial Vehicle Navigation in Tree Plantations". In: *Journal of Intelligent And Robotic Systems* 101.1 (2021), pp. 1–18 (page 8).

[25] Dhiraj Gandhi, Lerrel Pinto, and Abhinav Gupta. "Learning to fly by crashing". In: *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2017, pp. 3948–3955 (page 8).

[26] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems* 25 (2012), pp. 1097–1105 (page 8).

[27] Alexandros Kouris and Christos-Savvas Bouganis. "Learning to fly by myself: A self-supervised cnn-based approach for autonomous navigation". In: *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2018, pp. 1–9 (pages 8, 10).

[28] Dashuai Wang, Wei Li, Xiaoguang Liu, Nan Li, and Chunlong Zhang. "UAV environmental perception and autonomous obstacle avoidance: A deep learning and depth camera combined solution". In: *Computers and Electronics in Agriculture* 175 (2020), p. 105523 (page 9).

[29] Intel RealSense. "Depth Camera D435". 2021. URL: https://www.intelrealsense.com/depth-camera-d435/ (visited on 04/17/2021) (page 9).

[30] Joseph Redmon and Ali Farhadi. "Yolov3: An incremental improvement". In: *arXiv preprint arXiv:1804.02767* (2018) (page 9).

[31] Po-Heng Chen and Chen-Yi Lee. "UAVNet: An efficient obstacel detection model for UAV with autonomous flight". In: *2018 International Conference on Intelligent Autonomous Systems (ICoIAS)*. IEEE. 2018, pp. 217–220 (page 9).

[32] Heng Lu, Xiao Fu, Chao Liu, Long-guo Li, Yu-xin He, and Nai-wen Li. "Cultivated land information extraction in UAV imagery based on deep convolutional neural network and transfer learning". In: *Journal of Mountain Science* 14.4 (2017), pp. 731–741 (page 9).

[33] Fatih Kucuksubasi and Arzu Sorguc. "Transfer learning-based crack detection by autonomous UAVs". In: *arXiv preprint arXiv:1807.11785* (2018) (page 9).

[34] Kun Shao, Zhentao Tang, Yuanheng Zhu, Nannan Li, and Dongbin Zhao. "A survey of deep reinforcement learning in video games". In: *arXiv preprint arXiv:1912.10944* (2019) (page 9).

[35] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. "Mastering the game of Go with deep neural networks and tree search". In: *nature* 529.7587 (2016), pp. 484–489 (page 9).

[36] Jens Kober, J Andrew Bagnell, and Jan Peters. "Reinforcement learning in robotics: A survey". In: *The International Journal of Robotics Research* 32.11 (2013), pp. 1238–1274 (page 9).

[37] Nursultan Imanberdiyev, Changhong Fu, Erdal Kayacan, and I-Ming Chen. "Autonomous navigation of UAV by using real-time model-based reinforcement learning". In: *2016 14th international conference on control, automation, robotics and vision (ICARCV)*. IEEE. 2016, pp. 1–6 (page 9).

[38] Huy X Pham, Hung M La, David Feil-Seifer, and Luan V Nguyen. "Autonomous uav navigation using reinforcement learning". In: *arXiv preprint arXiv:1801.05086* (2018) (page 9).

[39] Ory Walker, Fernando Vanegas, Felipe Gonzalez, and Sven Koenig. "A deep reinforcement learning framework for uav navigation in indoor environments". In: *2019 IEEE Aerospace Conference*. IEEE. 2019, pp. 1–14 (page 9).

[40] Chao Wang, Jian Wang, Yuan Shen, and Xudong Zhang. "Autonomous navigation of UAVs in large-scale complex environments: A deep reinforcement learning approach". In: *IEEE Transactions on Vehicular Technology* 68.3 (2019), pp. 2124–2136 (page 9).

[41] Lei He, Aouf Nabil, and Bifeng Song. "Explainable Deep Reinforcement Learning for UAV Autonomous Navigation". In: *arXiv preprint arXiv:2009.14551* (2020) (page 9).

[42] Anna Guerra, Francesco Guidi, Davide Dardari, and Petar M Djuric. "Reinforcement Learning for UAV Autonomous Navigation, Mapping and Target Detection". In: *2020 IEEE/ION Position, Location and Navigation Symposium (PLANS)*. 2020, pp. 1004–1013 (page 9).

[43] Aqeel Anwar and Arijit Raychowdhury. "Autonomous navigation via deep reinforcement learning for resource constraint edge nodes using transfer learning". In: *IEEE Access* 8 (2020), pp. 26549–26560 (pages 10, 13, 69, 98).

[44] Open Robotics. "Gazebo". 2021. URL: http://gazebosim.org (visited on 04/10/2021) (page 11).

[45] Open Robotics. "Open Robotics". 2021. URL: https://www.openrobotics.org (visited on 04/10/2021) (page 11).

[46] Open Robotics. "Robot Operating System (ROS)". 2021. URL: https://www.ros.org (visited on 04/10/2021) (page 11).

[47] Epic Games. "Unreal Engine". 2021. URL: https://www.unrealengine.com/en-US/ (visited on 04/10/2021) (page 12).

[48] Epic Games. "Epic Games". 2021. URL: https://www.epicgames.com/site/fr/home (visited on 04/10/2021) (page 12).

[49] Shital Shah, Debadeepta Dey, Chris Lovett, and Ashish Kapoor. "Airsim: High-fidelity visual and physical simulation for autonomous vehicles". In: *Field and service robotics*. Springer. 2018, pp. 621–635 (page 12).

[50] Wikipedia. "Parrot AR.Drone". 2021. URL: https://en.wikipedia.org/wiki/Parrot_AR.Drone (visited on 04/10/2021) (pages 16, 17).

[51] Free PNG. "Parrot AR Drone 2.0". 2021. URL: https://www.freepng.fr/png-i11vr7/ (visited on 04/10/2021) (page 16).

[52] Grosbill. "Parrot AR Drone 2.0 - Elite Edition". 2021. URL: https://www.grosbill.com/4-parrot_parrot_ar_drone_2_0_elite_edition_-619523-loisirs_et_jouets_connectes-robot_espion (visited on 04/10/2021) (page 17).

[53] Ryze Robotics. “Tello EDU”. 2021. URL: https://www.ryzerobotics.com/tello-edu (visited on 04/10/2021) (pages 17, 18).

[54] Robot Advance. “DJI Tello EDU”. 2021. URL: https://www.robot-advance.com/art-drone-dji-tello-edu-2610.htm (visited on 04/10/2021) (page 17).

[55] Microsoft. “Simple flight”. 2021. URL: https://microsoft.github.io/AirSim/simple_flight/ (visited on 04/10/2021) (page 19).

[56] Ryze Robotics. “Tello EDU User Manual”. 2021. URL: https://dl-cdn.ryzerobotics.com/downloads/Tello/20180404/Tello_User_Manual_V1.2_EN.pdf (visited on 04/10/2021) (page 22).

[57] Wilbert G Aguilar, Guillermo A Rodriguez, Leandro Alvarez, Sebastian Sandoval, Fernando Quisaguano, and Alex Limaico. “Visual SLAM with a RGB-D camera on a quadrotor UAV using on-board processing”. In: *International Work-Conference on Artificial Neural Networks*. Springer. 2017, pp. 596–606 (page 23).

[58] Mitch Bryson and Salah Sukkarieh. “Building a Robust Implementation of Bearing-only Inertial SLAM for a UAV”. In: *Journal of Field Robotics* 24.1-2 (2007), pp. 113–143 (page 23).

[59] Wikipedia. “Occupancy grid mapping”. 2021. URL: https://en.wikipedia.org/wiki/Occupancy_grid_mapping (visited on 04/11/2021) (page 24).

[60] Peter E Hart, Nils J Nilsson, and Bertram Raphael. “A formal basis for the heuristic determination of minimum cost paths”. In: *IEEE transactions on Systems Science and Cybernetics* 4.2 (1968), pp. 100–107 (page 26).

[61] Nicholas Swift. “Easy A star Pathfinding”. 2017. URL: https://medium.com/@nicholas.w.swift/easy-a-star-pathfinding-7e6689c7f7b2 (visited on 05/28/2021) (page 26).

[62] Wikipedia. “Bilateral filter”. 2021. URL: https://en.wikipedia.org/wiki/Bilateral_filter (visited on 04/18/2021) (page 38).

[63] John Canny. “A computational approach to edge detection”. In: *IEEE Transactions on pattern analysis and machine intelligence* 6 (1986), pp. 679–698 (page 38).

[64] Nahum Kiryati, Yuval Eldar, and Alfred M Bruckstein. “A probabilistic Hough transform”. In: *Pattern recognition* 24.4 (1991), pp. 303–316 (page 39).

[65] SZanlongo. “vanishing-point-detection”. 2019. URL: https://github.com/SZanlongo/vanishing-point-detection (visited on 04/18/2021) (page 39).

[66] Krishnendu Chaudhury, Stephen DiVerdi, and Sergey Ioffe. “Auto-rectification of user photos”. In: *2014 IEEE International Conference on Image Processing (ICIP)*. IEEE. 2014, pp. 3479–3483 (page 40).

[67] chsasank. “Automated Rectification of Image”. 2020. URL: https://github.com/chsasank/Image-Rectification (visited on 04/18/2021) (page 40).

[68] Chin-Kai Chang, Jiaping Zhao, and Laurent Itti. “DeepVP: Deep learning for vanishing point detection on 1 million street view images”. In: *2018 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2018, pp. 4496–4503 (page 41).

[69] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014) (page 42).

[70] Mohammad Hossin and MN Sulaiman. "A review on evaluation metrics for data classification evaluations". In: *International Journal of Data Mining And Knowledge Management Process* 5.2 (2015), p. 1 (page 44).

[71] Rene Ranftl, Katrin Lasinger, David Hafner, Konrad Schindler, and Vladlen Koltun. "Towards robust monocular depth estimation: Mixing datasets for zero-shot cross-dataset transfer". In: *arXiv preprint arXiv:1907.01341* (2019) (page 50).

[72] PyTorch. "MiDaS". 2021. URL: https://pytorch.org/hub/intelisl_midas_v2/ (visited on 04/20/2021) (page 50).

[73] intel-isl. "What does the predicted depth signify?" 2020. URL: https://github.com/intel-isl/MiDaS/issues/42 (visited on 04/26/2021) (page 51).

[74] Huijuan Zhang, Chengning Zhang, Wei Yang, and Chin-Yin Chen. "Localization and navigation using QR code for mobile robot in indoor environment". In: *2015 IEEE International Conference on Robotics and Biomimetics (ROBIO)*. IEEE. 2015, pp. 2501–2506 (page 52).

[75] Mengyin Fu, Kuan Zhang, Yang Yi, and Chao Shi. "Autonomous landing of a quadrotor on an UGV". In: *2016 IEEE International Conference on Mechatronics and Automation*. IEEE. 2016, pp. 988–993 (page 52).

[76] Wikipedia. "QR code". 2021. URL: https://en.wikipedia.org/wiki/QR_code (visited on 04/19/2021) (page 53).

[77] Sergio Garrido-Jurado, Rafael Munoz-Salinas, Francisco Jose Madrid-Cuevas, and Manuel Jesus Marin-Jimenez. "Automatic generation and detection of highly reliable fiducial markers under occlusion". In: *Pattern Recognition* 47.6 (2014), pp. 2280–2292 (page 54).

[78] Adrian Rosebrock. "Find distance from camera to object/marker using Python and OpenCV". 2015. URL: https://www.pyimagesearch.com/2015/01/19/find-distance-camera-objectmarker-using-python-opencv/ (visited on 04/27/2021) (page 54).

[79] Dario Floreano and Robert J Wood. "Science, technology and the future of small autonomous drones". In: *Nature* 521.7553 (2015), pp. 460–466 (page 79).

[80] Percepto. "2021: The Future of Autonomous Drone Flight is Already Here". 2021. URL: https://percepto.co/2021-future-of-autonomous-drone-flight/ (visited on 04/13/2021) (page 80).

[81] Mitch Campion, Prakash Ranganathan, and Saleh Faruque. "UAV swarm communication and control architectures: a review". In: *Journal of Unmanned Vehicle Systems* 7.2 (2018), pp. 93–106 (page 80).

[82] Yong Zeng and Rui Zhang. "Energy-efficient UAV communication with trajectory optimization". In: *IEEE Transactions on Wireless Communications* 16.6 (2017), pp. 3747–3760 (page 80).

[83] Edronic. “Edronic”. 2021. URL: http://www.edronic.com/ (visited on 04/13/2021) (page 80).

[84] Ali Rohan, Mohammed Rabah, Muhammad Talha, and Sung-Ho Kim. “Development of intelligent drone battery charging system based on wireless power transmission using hill climbing algorithm”. In: *Applied System Innovation* 1.4 (2018), p. 44 (page 80).

[85] European Union Aviation Safety Agency. “European Union Aviation Safety Agency”. 2021. URL: https://www.easa.europa.eu/ (visited on 04/13/2021) (page 81).

[86] Assurance Drone. “L’essentiel sur la législation drone en Belgique”. 2021. URL: https://www.assurance-drone.be/legislation-drone/ (visited on 04/13/2021) (page 81).

[87] European Union Aviation Safety Agency. “What is the difference between autonomous and automatic drone?” 2021. URL: https://www.easa.europa.eu/faq/116449 (visited on 04/13/2021) (page 81).

[88] Humanitas. “Humanitas - Technologie innovante pour les réponses d’urgence”. 2021. URL: https://www.fr.humanitas.io/ (visited on 04/24/2021) (page 82).

[89] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. “You only look once: Unified, real-time object detection”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 779–788 (page 85).

Appendix A

Technical Specifications

In order to be able to reproduce the various tests carried out in this project, this appendix lists the precise technical specifications of the various elements used.

The simulator used is Unreal Engine 4[1] (version 4.25.4) with the AirSim[2] plugin from Microsoft. As the latter is still actively under development, a version directly cloned from the official GitHub[3] was used. As it is updated very regularly, it is difficult to give the exact version used in this project. However, the AirSim files used can be found directly on the official GitHub[4] of this project. The Unreal Engine plugin corresponding to this version is embedded directly in the simulated environments created for this work.

All simulations and algorithm performances were tested on a computer running Windows 10, with an AMD Ryzen 7 3700X 8-Core (3.6 GHz) processor, and NVIDIA GeForce GTX 970 graphics card and 32 GB of RAM. The training of most of the Deep Learning models was performed on the Montefiore GPU cluster ("Alan GPU Cluster"[5]).

The real drone used is a (DJI) Tello EDU[6] (2020 model).

The programming language used for this project is Python 3[7] (version 3.8.8). A file `environment.yml` is available on the official GitHub of the project to install all necessary dependencies. The Deep Learning models have been built with the PyTorch[8] framework.

[1] https://www.unrealengine.com/
[2] https://microsoft.github.io/AirSim/
[3] https://github.com/microsoft/AirSim
[4] https://github.com/meurissemax/autonomous-drone
[5] https://github.com/montefiore-ai/alan-cluster
[6] https://www.ryzerobotics.com/tello-edu
[7] https://www.python.org
[8] https://pytorch.org

Appendix B

Neural Networks Architectures

The architectures of the different neural networks used in this work are shown in this appendix.

B.1 DenseNet models

Two versions of the DensetNet model [23] were used in this work: DenseNet-121 and DenseNet-161. Pre-trained versions of these models were imported (from PyTorch) and modified. The modifications made are presented in Tables B.1 and B.2.

All input images are RGB (3 channels) images of size 320×180. By default, when these networks are trained, the pre-trained layers are not frozen and are therefore also updated.

Layer	**Neurons**	**Kernel**	**Stride**	**Padding**
Conv1	3	1	1	0
Usual DenseNet-121 architecture [23]				
Conv2	128	5	1	0
Conv3	16	1	1	0
Flatten	-	-	-	-
Dense	2	-	-	-

Table B.1: Modified architecture of the DenseNet-121 neural network.

Layer	Neurons	Kernel	Stride	Padding
Conv1	3	1	1	0
Usual DenseNet-161 architecture [23]				
Conv2	1024	1	1	0
Conv3	128	5	1	0
Conv4	16	1	1	0
Flatten	-	-	-	-
Dense	2	-	-	-

Table B.2: Modified architecture of the DenseNet-161 neural network.

All the activation functions used are ReLU, except for the final layer which is a Softmax.

B.2 Personal model

A custom convolutional neural network has been designed: SmallConvNet. The architecture of the latter is shown in Table B.3.

Layer	Neurons	Kernel	Stride	Padding
Conv1	3	1	1	0
DoubleConv1	32	3	1	0
MaxPool	-	-	-	-
DoubleConv2	32	3	1	0
MaxPool	-	-	-	-
DoubleConv3	64	3	1	0
MaxPool	-	-	-	-
DoubleConv4	64	3	1	0
MaxPool	-	-	-	-
DoubleConv5	128	3	1	0
MaxPool	-	-	-	-
Dropout	-	-	-	-
Flatten	-	-	-	-
Dense	4096	-	-	-
Dense	2048	-	-	-
Dense	128	-	-	-
Dense	2	-	-	-

Table B.3: Architecture of the SmallConvNet network.

All max pool layers have a kernel size of 2. The drop out layer has a probability of 0.8. All the activation functions used are ReLU, except for the final layer which is a Softmax.

Appendix C

Reinforcement Learning Approach

Several methods, in the scientific literature, focus on the use of Reinforcement Learning to attempt to explore environments that are *unknown*. The objective is to determine the optimal set of actions to perform (the *optimal policy*) in order to reach the objective.

In this work, a simple Reinforcement Learning algorithm working with drone's position has been tested to explore the subject.

C.1 Formalization

The agent considered is the drone and the environment, in the sense of Reinforcement Learning, is the environment in which the drone navigates (*e.g.* Indoor Corridor).

The environment is considered as a two-dimensional grid. The position of the agent in the environment is given by its coordinates i and j in the grid. The latter represents the agent's state (noted x).

The agent interacts with its environment through a series of deterministic actions: move forward, move back, turn left and turn right. More formally, the action space U is given by

$$U = [(1, 0), (-1, 0), (0, 1), (0, -1)] \tag{C.1}$$

When the agent interacts with its environment, it receives some *rewards*: each displacement of the agent earns it a -1 reward, each obstacle taken earns it $-\infty$ and the objective reached earns it 100. The agent's main goal is to maximize his total reward, *i.e.* to reach the goal as quick as possible without hitting any obstacle.

C.2 Optimal policy

In order to calculate the optimal policy μ^* of the agent, a simple *Q-learning* algorithm has been used. The optimal policy is given by

$$\mu_N^*(x) = \arg\max_{u \in U} Q_N(x, u) \tag{C.2}$$

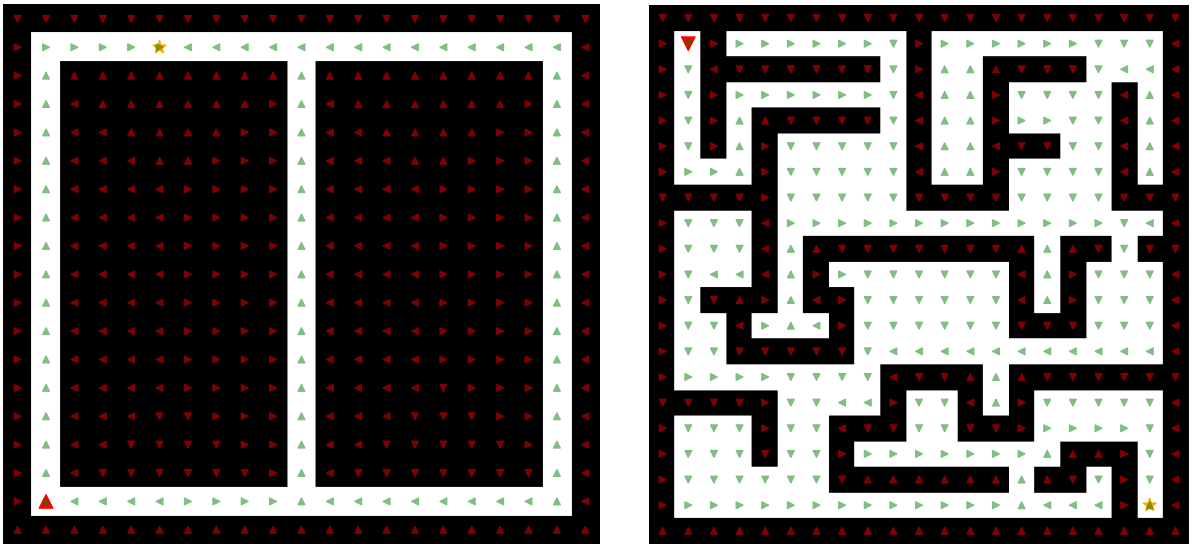
where x is the agent's state, u is an action and $Q_N(x, u)$ is the state-action value.

The value $Q_N(x, u)$ can be calculated via the recurrence equation

$$Q_N(x,u) = r(x,u) + \gamma \sum_{x' \in X} p(x' \mid x,u) \max_{u' \in U} Q_{N-1}(x',u'), \quad \forall N \geq 1 \tag{C.3}$$

where $r(x, u)$ is the reward, γ is a *discount factor* reducing the importance of future rewards compared to immediate rewards and $p(x' \mid x, u)$ the transition probability of the *Markov Decision Process* (MDP) corresponding to the domain. With a sufficient number of iterations N (fixed so that the difference between the rewards of two optimal policies calculated is lower than a fixed threshold set to 0.001), it is thus possible to calculate Q_N and to deduce the optimal policy μ_N^*.

This procedure was applied to the Indoor Corridor environment and to another complex environment. The obtained results are shown in Figure C.1.



(a) Indoor Corridor (b) Complex

Figure C.1: Optimal policies obtained via a simple Q-learning algorithm. The agent's position and orientation is represented by a red arrow. Its objective is represented by a yellow star. The action to be taken at each location in the environment is represented by a small arrow (green when the location is accessible and red when the location is inaccessible).

C.3 Discussion

The obtained results show that this procedure is effective in determining the optimal policy leading the drone from its starting point to the objective, in a totally unknown environment.

Although explored in several scientific papers (see Chapter 2), this procedure does not take into account the noisy displacements of the drone and the unanticipated dynamic obstacles in the environment. Moreover, the procedure experimented here considers that

the environment is deterministic (whereas in real life it is stochastic) and totally observable. Also, this procedure works with the drone's position, which is not really realistic since this position can not be easily obtained in practice.

These simplifying assumptions allow us to obtain results showing that Reinforcement Learning can be a solution for autonomous navigation. However, more complex techniques need to be explored (*e.g.* Deep Reinforcement Learning working with images) to hope to put this into practice in the real world, especially if we only have access to the drone's images (and not directly its position). Furthermore, since Reinforcement Learning works by trial and error, it is difficult to train a real drone in a real environment. It is then necessary to train a model on a simulator and apply it to the real world via Transfer Learning (see [43]).

The further exploration of Reinforcement Learning methods, especially using only drone's images, will not be addressed in this work and are left for future work on the subject.