POLITECNICO DI TORINO

Master degree course in Data Science and Engineering

Master Degree Thesis

Visual Servoing for Autonomous Nano-drone Racing

Supervisors
Prof. Daniele Jahier Pagliari
Dr. Daniele Palossi
Dr. Alessio Burrello
Dr. Antonio Paolillo
Elia Cereda
Luca Crupi
Matteo Risso

Candidate
Lorenzo Scarciglia

October 2023

Abstract

Drones are nowadays used in many scenarios, from air surveillance and search-and-rescue missions to agriculture and cinematography. Drones' miniaturisation led to light nano-drones that fit the palm of a hand. Despite some downsides like the battery life of the order of minutes and limited computational power of the onboard microcontroller unit (MCU) belonging to the sub-100 mW power envelope, such systems can navigate in narrow spaces and being harmless around humans can be used in many scenarios. Over the last decade, autonomous drone racing (ADR) competitions have fostered research and provided an opportunity for scientists to create cutting-edge perception and control algorithms meant to operate directly onboard the drones. In such competitions, drones have to pass through a predefined set of gates as fast as possible while avoiding obstacles without human intervention. Recently, nano-drones faced ADR competitions.

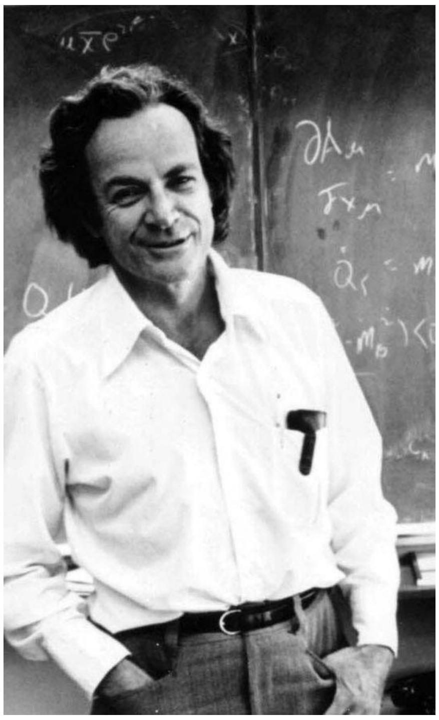
In this thesis, our focus centres on the Crazyflie 2.1, a nano-drone with only a 10 cm diameter and a weight of just 27g, within the context of a drone racing scenario featuring square gates. Such a drone is equipped with an ultra-low-power monochrome camera, sensors to estimate the drone's state, and the GAP8 MCU, which enables the execution of deep learning workloads directly onboard. To solve the gate-based navigation, i.e., the task of identifying and crossing the gate, we employ the image-based visual servoing (IBVS), a vision-based control aimed at aligning extracted image features with predefined objectives by issuing velocity commands to the drone.

We develop and compare two detection modules that extract the features required by IBVS, in the gate-based navigation case, the four corners of the gate. One consists of traditional computer vision (CV) algorithms, while the other relies on deep learning (DL) exploiting a convolutional neural network. Webots, a robotic simulator, is used to gather synthetic images to tune the CV module and to train the DL one. The synthetic training dataset is collected by randomly spawning the drone around the gate, taking care that each corner falls inside the image, and varying the background during the collection. Two testing datasets of increasing difficulty are collected in the same fashion without the background variation. Data augmentation is used to increase the number of training images and to mimic the real images. Overall, detection modules are tested on the two synthetic test sets and one of the actual camera images. The DL module error is 44% up to 77% lower than the CV one. To jointly test a detection module and the IBVS, we exploit Webots and define a flight task consisting of take-off, gate-based navigation, and landing. The same task is carried on in three worlds of increasing difficulty. The background increases in difficulty while the drone and relative position of the gate are the same in all the worlds. While the CV module succeeds only in the simplest of the worlds with a mean completion time of 84 seconds, the DL module outperforms it by completing it in 72 seconds. Moreover, it completes the task in all the worlds, showing the generalisation ability of the trained network.

Acknowledgements

A nine-month journey in which many people supported me, both technically and morally, has come to an end. Hence, this section is devoted to thank you all.
Thanks to the Politecnico, which was like a home for the past six years, and the IDSIA of Lugano, which welcomed me for my thesis period. Many words of thanks go to Daniele Jahier Pagliari, Daniele Palossi, Alessio Burrello, Antonio Paolillo, Elia Cereda, Luca Crupi, and Matteo Risso, my never-ending advisors. Many thanks for the opportunity, their suggestions, support, and patience with such a dummy head (almost like the detector in Sec. 4.1). I want to thank Simone, Gabri and the robotic family of IDSIA, with whom I shared good lunch o'clock times and sushi.
Thanks to Daniele, Matteo, Vitto and Bea for the table football matches, but essentially to Alessio, who taught me to stay humble. Thanks to Elia and Luca, who supported me and shared a lot of AFCs in the Lugano postal office. Thanks to Daniele, who had the patience and led me with his knowledge.
A huge thanks goes to my extended family, who always helped me and supported me in my decision to do my thesis abroad, even if abroad was not so Far Far Away. And to finish, a gargantuan thanks goes to all my friends, both the old and the new. There are too many, but thanks to the G∆NG, the Polenta, the Sopravvissuto D. & D. groups, and all the satellites of these groups. And, of course, thanks to Giulio, without him, I would have known less than half of you.

I think you, who are reading this thesis, have understood almost nothing, and that's fine but thanks to you too,
Lorenzo



“I was an ordinary person who studied hard. There are no miracle people. It happens they get interested in this thing and they learn all this stuff, but they’re just people.”

Richard Feynman

Acronyms

μDMA micro-DMA. 27

AI Artificial Intelligence. 23, 26, 37

ANN Artificial Neural Network. 40

AR Aspect Ratio. 13, 14, 45, 46, 47, 48, 55, 56, 66, 67, 68, 70

CL Cluster. 27

CNN Convolutional Neural Network. 19, 21, 34, 43, 52, 91, 92

CV Computer Vision. 9, 11, 13, 14, 15, 18, 21, 23, 34, 43, 44, 45, 46, 47, 49, 51, 57, 60, 63, 65, 67, 68, 69, 70, 71, 73, 74, 76, 78, 79, 80, 81, 83, 84, 91, 92, 97

DL Deep Learning. 9, 14, 15, 21, 34, 40, 43, 44, 52, 53, 55, 57, 59, 60, 61, 65, 67, 68, 69, 71, 73, 74, 76, 78, 79, 80, 81, 83, 84, 91, 92

DMA Direct Memory Access. 27

DOF Degrees of Freedom. 24, 33

DTW Dynamic Time Warping. 11, 15, 81, 82, 83, 84, 97

EKF Extended Kalman Filter. 19, 61

FC Fabric Controller. 26, 27

FOV Field of View. 54, 55

FPV First Person View. 17

GPU Graphic Processing Unit. 40

IBVS Image-based Visual Servoing. 9, 11, 15, 21, 29, 33, 49, 61, 62, 63, 73, 74, 75, 76, 77, 78, 79, 81, 84, 91, 92

IMU Inertial Measurement Unit. 18, 61

MAE Mean Absolute Error. 14, 15, 52, 67, 68, 69, 70, 72, 73, 77, 78

MCU Micro Controller Unit. 25, 26, 27

ML Machine Learning. 23, 37, 40, 44

NN Neural Network. 18, 19, 73

PID Proportional Integral Derivative. 18

PnP Perspective-n-Point. 19

PULP Parallel Ultra-Low Power. 11, 12, 21, 26, 40, 43, 44, 52, 53, 92

PWM Pulse-Width Modulation. 61

RISC Reduced Instruction Set Computer. 26

ROS Robot Operating System. 63

SIMD Single Instruction Multiple-Data. 27

SoC System-On-Chip. 12, 26

ToF Time of Flight. 26, 61

UAV Unmanned Aerial Vehicle. 24, 25

ULP Ultra-Low Power. 26

VIO visual-inertial odometry. 18, 19

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In drone racing competitions, professional players control their drones to fly through a predefined set of gates in the fastest way possible. The players control the drones in First Person View (FPV), which means that they receive real-time images from the drone's front camera. They see what the drones see. Years of training are required to achieve great results in terms of control and navigation skills. Drones are nowadays used in many fields, and being able to move rapidly in complex scenarios by matching human skills in terms of navigation and control skills can be very useful, for example, in search-and-rescue tasks.

Autonomous drone racing competition is a domain in which researchers can challenge each other to gauge their progress in perception, planning, and control algorithms, which are necessary for autonomous drones. The increasing interest in such a field is reported in [34], where a remarkable increase in drone racing papers in the last years is shown in Fig. 1.1

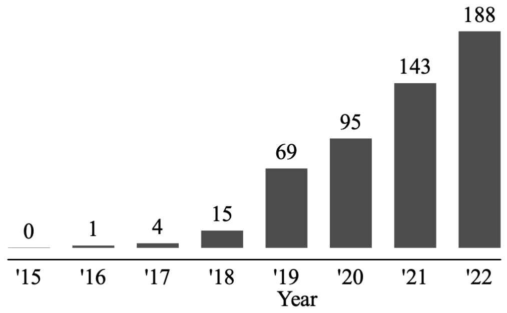


Figure 1.1: Number of drone racing papers by year [34].

In autonomous flying robotics, algorithms must be efficient, lightweight, and provide optimal decision and control behaviours in real time [34]. To encourage the research, many competitions like the IROS Autonomous Drone Racing Series [22] and projects like the European Research Council's Agile-Flight [25] have been launched in the last few years.

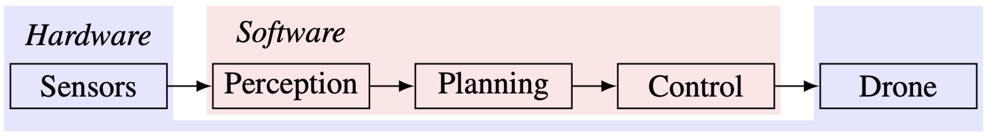


Figure 1.2: Image from [34], showing the architecture of an autonomous drone system.

Autonomous systems architectures feature a hardware part and a software one, as shown in Fig. 1.2. This project focuses on the development of two perception modules paired with visual servoing to control the drone. The control module is a cascaded PID controller. More in detail, we compare the perception modules where one relies on classic Computer Vision (CV) algorithms, and the other exploits a Neural Network (NN).

Perception modules estimate the vehicle state and perceive the environment using onboard sensors [34].
The most common solution for state estimation is the visual-inertial odometry (VIO), in which both the camera and the Inertial Measurement Unit (IMU) measurements are used to estimate the position, orientation, and velocity of the drone. Camera and IMU are used altogether to overcome the issues of one another. The former is affected by environmental conditions, poor illumination and motion blur are some examples, whilst the latter suffers from large drift in the measurements. This happens because IMU measurements are integrated over time, and the more time passes, the larger the accumulation of the drift. Since these two sensors complement each other in the state estimation task, they're commonly used jointly in flying vehicles [23].

Most of the VIO algorithms feature a front end and a back end. The former exploits camera images to estimate the motion of the sensors, whilst the latter fuses the output of the front end with the inertial measurements [34].
The front end is characterised by two main approaches, direct methods and feature-based methods. The latter is where the thesis fits. Direct methods use raw pixel intensities, extract image patches and track them to estimate

the camera trajectory [17], [13]. Instead, feature-based methods extract key points from camera images and track them to estimate the aforementioned trajectory [10], [14], [18].
The back end is devoted to fusing the output of the front end with the inertial measurements. The sensor fusion task relies on filtering methods and fixed-lag smoothing methods. The former is based on the Extended Kalman Filter (EKF) and its derivations. In these methods, the state of the system is broadcast with the inertial measurements and fused with the camera measurements in the update step [34]. Fixed-lag smoothing methods [14], [18] accumulate less linearisation error but are more computationally demanding since they aim to solve a non-linear optimisation problem that takes into account visual, inertial, and past states marginalisation residuals [34].

Learning-based approaches have been increasingly used extensively since they can deal with both high and low-dimensional input data. The replacement of some of the modules seen in Fig. 1.2 with a Neural Network is widely popular nowadays. The main problem of these approaches concerns the amount of data required to properly train the neural network. Data gathering can be performed in the real world with the labelling performed by humans or automated processes or using simulators. Using simulations to collect data has become popular in recent years [34].
The goal of a learned-perception module is to use camera images to detect landmarks and output useful representations like waypoints or the location of the gates [34]. It mainly replaces the front end in the VIO algorithm. In [21], given the input image, the network outputs the gate location along with its uncertainty. Such predictions are then fused with a EKF to estimate the position of the drone. [30] uses a Convolutional Neural Network (CNN) to detect gate corners, a Perspective-n-Point (PnP) algorithm to find the coordinate image of the corners, and again a EKF to estimate the drone's location.
To consume less onboard resources, lightweight CNNs like GateNet [26] and PencilNet [32] use optimised architectures for gate detection, still detecting gate centre locations, distance, and orientation relative to the drone.

Simulators are powerful tools that help researchers to test their algorithms and pipelines before deployment. Simulations are easier to set up, less expensive, and faster than tests in the real world. Moreover, the ease of data collection to train learning-based algorithms accelerated the progress of research in autonomous drone flight.
Many simulators have been developed over the years. Gazebo, the widely

popular simulation engine, was extended in 2016 to deal with multi-rotors, resulting in RotorS [15]. Such a simulator supports the physics to simulate the drone dynamics and has many easy-to-use plugins, but lacks of photorealistic details, which are necessary to simulate vision-based perception pipelines.
AirSim [19] is a photorealistic simulator built on Unreal Engine, which was introduced by Microsoft in 2018. With its impressive photorealism, it becomes feasible to replicate the entire perception and estimation pipeline. Moreover, thanks to its high-fidelity images, it also becomes feasible to transfer to real-world drone systems [34].
FlightGoggles [20] and Flightmare [28] are two other photorealistic simulators. The former uses Unity3D as the photorealistic engine and a dynamic simulation implemented in C++. Even the latter uses the Unity engine but has various physics engines that can be swapped to achieve the desired simulation fidelity. Both provide hardware-in-the-loop simulation functions.

Nowadays, drones miniaturisation has grown, making pocket-sized drones widely available. Being able to navigate in narrow spaces and inherently safe near humans, they can be deployed in many useful tasks like gas seeking or search-and-rescue in adversity environments. With a lower size come challenges in terms of onboard computation, power, and payload capability. Recently, nano-drones, which fit the palm of a hand, are being considered to compete in both autonomous and manual competitions. The first autonomous nano-drone competition was held in 2022 at the International Micro Air Vehicles (IMAV) conference at Delft. Sponsored by Bitcraze AB[1], their Crazyflie 2.1, equipped with the AI deck and the Flow deck, was the nano-drone at hand. Teams were asked to make the drone fly autonomously as far as possible through an indoor obstacle field, performing a vision-based obstacle avoidance to prevent collisions with obstacles such as pillars and panels. The field also featured square gates, which gave extra points if the drone flew through. Two rounds of 5 minutes each were performed, where the score was given by the amount of space covered during the round, meaning that going faster results in covering more space. Additional points were assigned to teams who achieved passing through gates and whose computations were performed onboard. For each team, only the best flight out of two was considered. To develop the vision-based obstacle avoidance, teams were provided with a robotic simulator, Webots [8], with a world mimicking the

[1]`https://www.bitcraze.io`

actual field and the model of Crazyflie 2.1.

Considering such a scenario, the thesis work focuses on developing a vision-based control pipeline to command the Crazyflie 2.1 nano-drone to cross a gate. Image-based Visual Servoing (IBVS) is the core of such a pipeline. Given a set of features and a set of desired positions in the image space, the control law minimises the difference between the two sets by supplying velocity commands to the drone at hand. At convergence, the set of features and the desired ones match, leading the drone in front of the gate. The four inner corners of the gate were chosen as the set of input features to the IBVS. To do that, two detection modules were developed and compared. One is based on classic CV algorithms (the CV detector), while the other one is learning-based, in particular, an adapted version of the PULP-Frontnet [27] CNN (the Deep Learning (DL) detector). The whole pipeline is then composed of:

- The detection module, which, given an input camera image, outputs the coordinates of the four corners, namely the set of features required by IBVS.
- The IBVS that, given the set of input features, provides the velocity commands to the drone to minimise the difference between the input features and the desired position of those features.

Tests were conducted on Webots, the open-source photorealistic robotics simulator used during the IMAV 2022 competition, because of the presence of both the Crazyflie 2.1 model and the indoor field.

The thesis is organised as follows: Chapter 2 presents the theory required to understand how the whole pipeline works. Chapter 3 explains how detection modules work, as well as the data collection and the description of the flight tests performed in the simulator. The comparison of detection modules on three test sets is presented in Chapter 4. Two are composed of synthetic images collected in Webots. The other was collected at the IDSIA Robotics Laboratory of Lugano, which features real images captured by an actual Crazyflie 2.1. The same chapter also presents the analysis of the flight tests performed in three different worlds of increasing difficulty in the amount of textures. In particular, the toughest is the one used for the IMAV 2022 competition. Finally, Chapter 5 concludes the work with a discussion on future works.

Chapter 2

Background and Related Works

The following sections are devoted to diving into the concepts and the theory behind this project work. Starting with a presentation of the considered drone system, the Crazyflie 2.1 by Bitcraze AB and two of its expansion decks: the AI deck and the Flow deck, then the focus is put into presenting the theory behind the visual servoing control, with an in-depth presentation of the image-based variety. The following section concerns the Computer Vision and the algorithms used to develop the traditional CV pipeline. After that, a brief presentation of some concepts concerning the Machine Learning field is given to introduce the algorithms used in both pipelines. A description of the adopted robotic simulator, Webots, ends the chapter.

2.1 The quadcopter

To better understand the following, this section starts with an introduction to some geometry concepts.
Figure 2.1 shows the fixed origin frame coordinate system (X, Y, Z) and the rigid body frame coordinate system (x, y, z). In this case, the rigid body is the quadcopter, namely a drone. In the 3D space, the position is defined as the point (X', Y', Z') in the space relative to the origin, and the orientation (or attitude) is the triplet of angles (φ, θ, ψ). Such angles are relative to the origin coordinate system and denote the rotation around, respectively, the axes x, y, and z of the drone. More in detail, φ is defined as the roll angle, θ as the pitch angle, and ψ as the yaw angle.

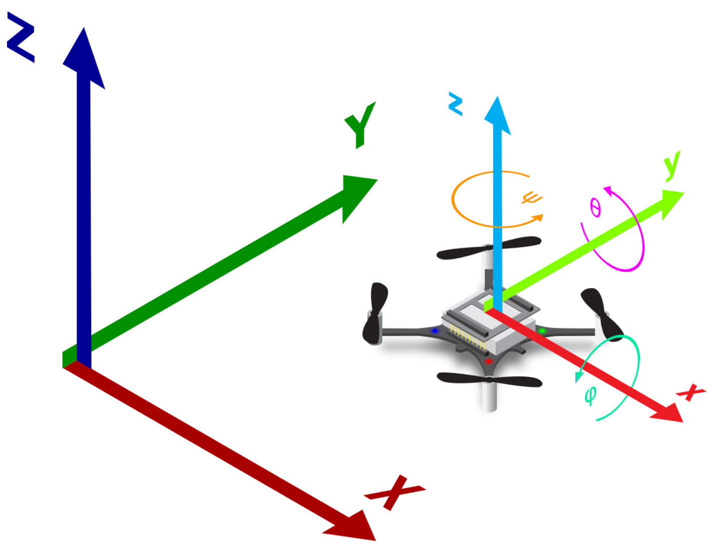


Figure 2.1: Being x the forward direction, y the left one, and z the upward direction, the rotations around the three axes are: φ, namely the roll; θ, the pitch; and ψ the yaw. Image from Bitcraze AB [36].

The quadcopter design is now very popular for indoor Unmanned Aerial Vehicle (UAV) applications. Quadcopters feature four propellers, namely the motors, used to control their position and attitude. The combination of the two, position (X, Y, Z) and orientation (φ, θ, ψ), defines the pose of the drone $(X, Y, Z, \varphi, \theta, \psi)$, namely its configuration. The six independent parameters of the pose denote the number of Degrees of Freedom (DOF) of the drone system. All the feasible combinations of parameters compose the configuration space $\mathcal{C}$. Actuators are the physical components in a robotic system that converts energy into physical motion. These components convert electrical, hydraulic, pneumatic, or other forms of energy into mechanical motion. In a robotic system, the number of actuators usually corresponds to the number of DOF that can be actively controlled. In a quadcopter, the number of DOF is six, but the number of actuators (the propellers) is four. Such a system is defined as underactuated and cannot directly access some points of the configuration space $\mathcal{C}$. More in detail, the propellers develop a thrust able to lift the drone and let it perform rotational motion. A movement in the forward or backward direction requires the drone to change its pitch angle. Instead, to move sideways, the roll angle has to be changed. Hence, movement in both forward and sideways directions requires variations of roll and pitch angles. In this sense, the system is underactuated. Having fewer actuators can be advantageous in terms of cost, complexity, and weight [33].

In the quadcopter system, to achieve balance, adjacent propellers must counter-rotate, and by varying the thrust produced by each propeller, the drone can move.

With small-sized drones, having fewer actuators is even more beneficial, where complexity, weight, and energy consumption must be optimal. The miniaturisation of Unmanned Aerial Vehicle (UAV)s, namely drones, has grown over the years, leading to pocket-sized drones being widely commercially available. With a lower size come challenges in terms of onboard computation, power, and payload capability. This project focuses on a nano-sized quadcopter named Crazyflie 2.1, developed and manufactured by Bitcraze AB[1].

2.1.1 The Crazyflie 2.1

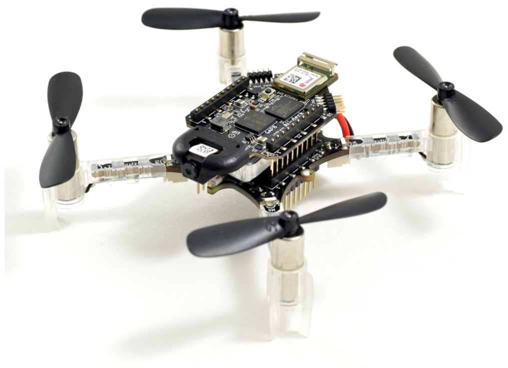


Figure 2.2: The Crazyflie 2.1 with the AI-deck on top. Image from Bitcraze AB [35].

Released in 2019, the Crazyflie 2.1 is a light (27g) nano-quadcopter of 10 cm diameter with up to 7 minutes of flight using stock batteries. Through the addition of expansion decks, its hardware can be enhanced in terms of sensing, positioning, and vision. Its basic hardware components are:

- The Micro Controller Unit STM32F405, which handles the low-level and high-level controls. High-level control updates the set point (the desired position) used by the low-level control, which updates the drone's state

[1]https://www.bitcraze.io

estimation and applies cascaded PID controllers to reach the set point defined by the high-level control.

- The nRF51822, another MCU designated for radio and power management.

Two widely used expansion decks are:

- The Flow deck, for visual odometry navigation. It allows the drone to detect the motion in any direction thanks to:
  - The VL53L1x Time of Flight (ToF), a laser-based sensor which measures the distance from the ground (the altitude) with high precision.
  - The PMW3901 optical flow sensor, which measures the displacement in the x, y direction as long as the altitude is at least 80mm.
- The AI deck, which comes with the Himax HM01B0, an Ultra-Low Power (ULP) 320x320 grayscale mono-camera, and the GAP8 System-On-Chip (SoC), which enables AI-based applications to run onboard thanks to the Parallel Ultra-Low Power (PULP) paradigm.

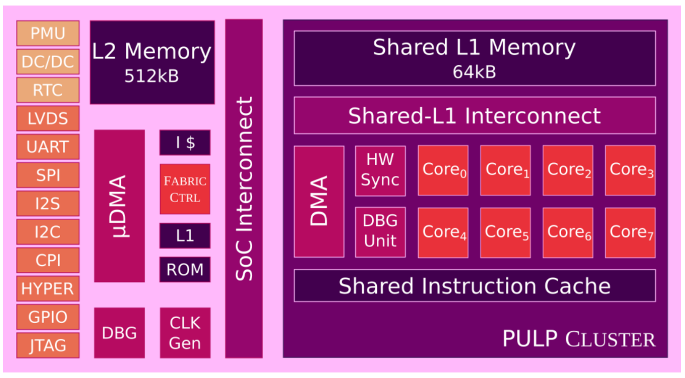


Figure 2.3: GAP8 SoC [27]

Being the "brain" of Crazyflie's AI deck, the GAP8 is an IoT application processor that enables the onboard execution of miniaturised neural networks thanks to the PULP paradigm.
Produced by GreenWaves Technologies[2], this ultra-low power processor features a total of nine identical RISC-V cores. One is called Fabric Controller

[2]`https://greenwaves-technologies.com`

(FC) and is devoted to being the main core in the Micro Controller Unit (MCU), controlling all the GAP8 operations. It enables and dispatches the workload to the Cluster (CL). The remaining eight cores compose the CL, the parallel general-purpose accelerator, which can be programmed to compute efficiently highly parallel workloads. The CL receives from the FC high computational workload in order to speed up the execution by exploiting the parallelism.
Fig. 2.3 depicts the architecture of GAP8. The two main regions are:

- The left one presents a shared L2 memory (512kB), which can be accessed from both FC and CL, the FC paired with the micro-DMA (μDMA) manage the external communications. Each interface is implemented as a separate and independent μDMA channel. The μDMA is a programmable subsystem connected directly to the L2 memory that can move data from memory to I/O and vice versa without involving the core.
- The other region contains the CL. It has an ultra-fast, tightly coupled data memory (64kB) organised in banks that directly communicates with an optimised DMA to move data between the two regions without limiting the core's execution. Moreover, the eight cores share a single instruction cache, optimised for the Single Instruction Multiple-Data (SIMD) paradigm.

2.2 Visual Servoing

To let the drone move towards our goal, the proposed vision-based pipeline relies on visual servoing, a well-known technique used to control robots by using the information coming from camera sensors. In this kind of control, visual features of the goal object, such as the inner corners of the gate, are extracted from the camera image and used to guide the drone towards the target position. In doing this, such a control system involves continuous measurement of the target, leading to robustness to possible errors [33].

Two possible configurations are:

- *Eye-to-hand*, where the camera is fixed in the world and looks at the target. In this case, the observed object is the one who moves.
- *Eye-in-hand*, where the camera is mounted on the robotic system, which can move. Hence, the camera is moving while the observed object is fixed. This is the configuration considered in this project.

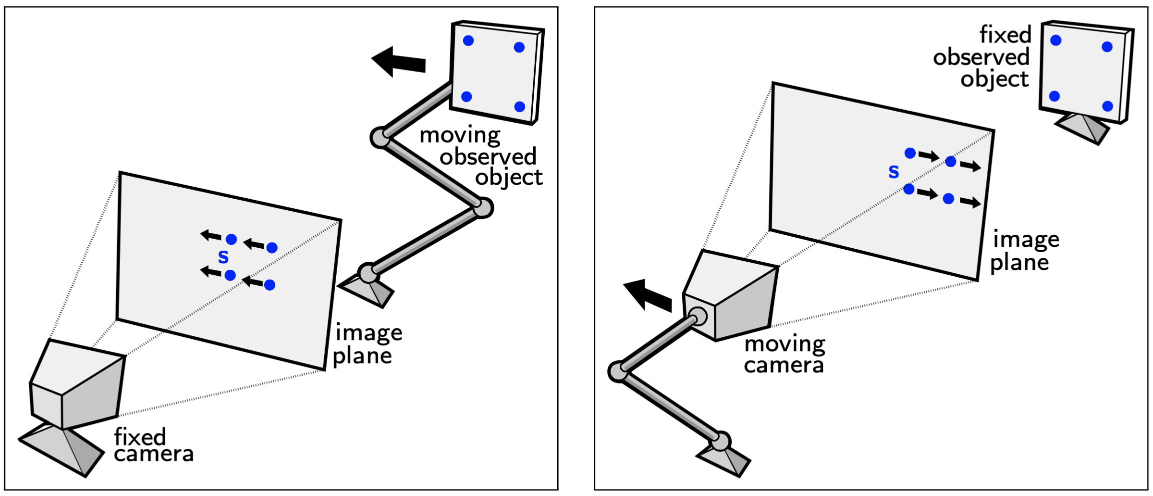


Figure 2.4: Eye-to-hand (left), and eye-in-hand (right) configurations. Image from [31].

The control aims to minimize the following error:

$$\boldsymbol{e}(t) = \boldsymbol{p}(t) - \boldsymbol{p}^*. \tag{2.1}$$

Such an error is the difference between the vector of the extracted features $\boldsymbol{p}$ and the vector containing the desired values of the features $\boldsymbol{p}^*$. The considered setting focuses on a non-moving target, e.g. the gate, and the goal poses to be fixed. This means that $\boldsymbol{p}^*$ is constant, and the changes in the feature vector $\boldsymbol{p}$ are due to camera motion. Depending on how $\boldsymbol{p}$ and $\boldsymbol{p}^*$ are characterised, two main approaches to visual servoing are possible: the position-based and the image-based. The former requires estimating the pose of the target through image features, which is computationally expensive and requires an accurate calibration of the camera, whilst the latter doesn't require direct pose estimation but uses directly the image-plane information. To develop a pipeline relying only on vision, the image-based visual servoing fits the problem at hand.

The visual servoing control law is obtained in the following way. Consider a point in real space and a moving camera with spatial velocity

$$\boldsymbol{\nu} = (\boldsymbol{v}, \boldsymbol{\omega}) = (v_x, v_y, v_z, \omega_x, \omega_y, \omega_z) \in \mathbb{R}^6$$

The point in the real space will move in the image space of the camera frame because of the camera motion, as shown in Fig. 2.4. The relationship between the time variation of the feature vector, namely the 2D vector describing the point in the image space, $\boldsymbol{p}$ and the velocity of the camera is

$$\dot{\boldsymbol{p}} = \boldsymbol{L_p}\boldsymbol{\nu} \tag{2.2}$$

where $\boldsymbol{L_p}$ is the *interaction matrix* [9].
The time variation of the visual servoing error is obtained by using Eq. 2.1 and Eq. 2.2. Remembering that $\boldsymbol{p}^*$ is constant, which leads to $\dot{\boldsymbol{p}}^* = 0$, the equation is:

$$\dot{\boldsymbol{e}} = \boldsymbol{L_e}\boldsymbol{\nu} \tag{2.3}$$

with $\boldsymbol{L_e} = \boldsymbol{L_p}$.
As discussed in [9], to ensure an exponential decoupled decrease of the error and have stable error dynamics, the derivative of the error is $\dot{\boldsymbol{e}} = -\lambda\boldsymbol{e}$. Using Eq. 2.3:

$$\boldsymbol{\nu} = -\lambda{\boldsymbol{L_e}}^+\boldsymbol{e} = -\lambda{\boldsymbol{L_e}}^+(\boldsymbol{p} - \boldsymbol{p}^*) \tag{2.4}$$

Where $\lambda > 0$ is the control gain, ${\boldsymbol{L_e}}^+ \in \mathbb{R}^{6\times n}$ is the Moore-Penrose pseudo-inverse of $\boldsymbol{L_e}$, and n is the number of extracted features. In this case, for a 3D point feature, only two coordinates are in the image plane.
Since it's difficult to know perfectly $\boldsymbol{L_e}$ or ${\boldsymbol{L_e}}^+$, an approximation or an estimation can be done [9].

2.2.1 The Image-based Visual Servoing (IBVS)

This type of visual servoing does not require estimating the relative pose of the target as the counterpart position-based. The relative pose of the target is implicit in the image features. Here, the control problem is expressed in terms of image coordinates. It aims to move the extracted features $\boldsymbol{p}$ towards the desired features $\boldsymbol{p}^*$, this movement in the 2D image plane is reflected by a movement in the 3D space of the camera, hence of the drone.

From 3D to 2D: Camera projection

An object in the 3D space is described by three coordinates, namely X, Y, and Z. In a picture of the same object, the depth dimension is lost, leaving only two coordinates to describe its position. Such a 2D space is called image space, and the coordinates used are u and v as in Fig. 2.5 In the following, considering Fig. 2.7 as a reference, the relation between these two spaces, namely the projection of a 3D point into the 2D image space, is presented considering a frontal pinhole camera model. The pinhole camera is a camera

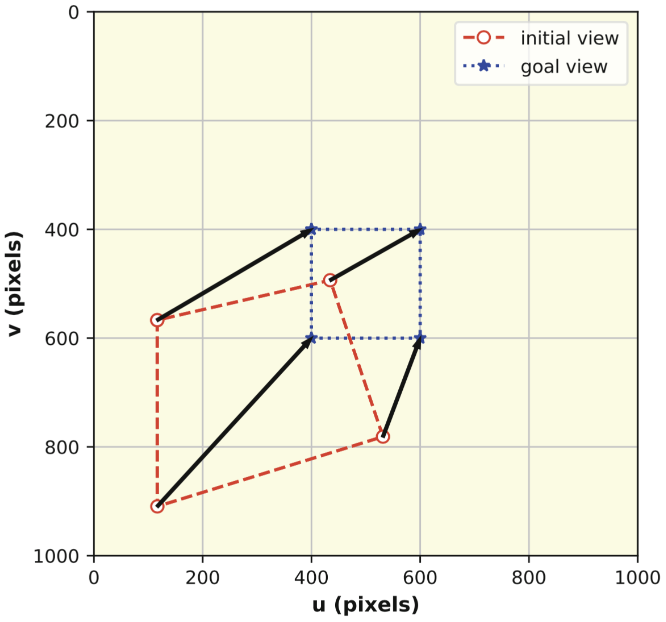


Figure 2.5: Evolution of the visual features in the image space, starting from the initial view (red circles) to the goal view (blue stars). Image from [33]

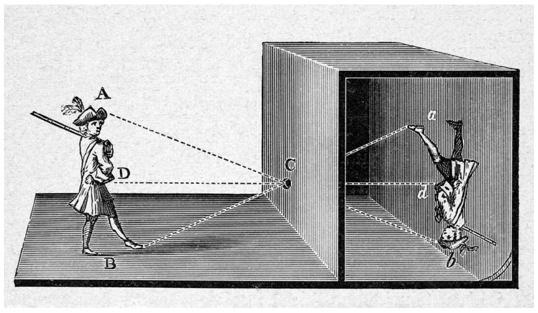


Figure 2.6: The camera obscura. An example of the pinhole camera.

featuring no lenses but only a tiny hole. The light passes through the hole, projecting an inverted image on the opposite side of the hole. In such a camera, there is no focus. Every object is in focus. The camera obscura shown in Fig. 2.6[3] is an example of a pinhole camera.

Considering a 3D point $\mathbf{P} = (X, Y, Z)$ whose position is relative to the

[3]Image from ResearchGate.

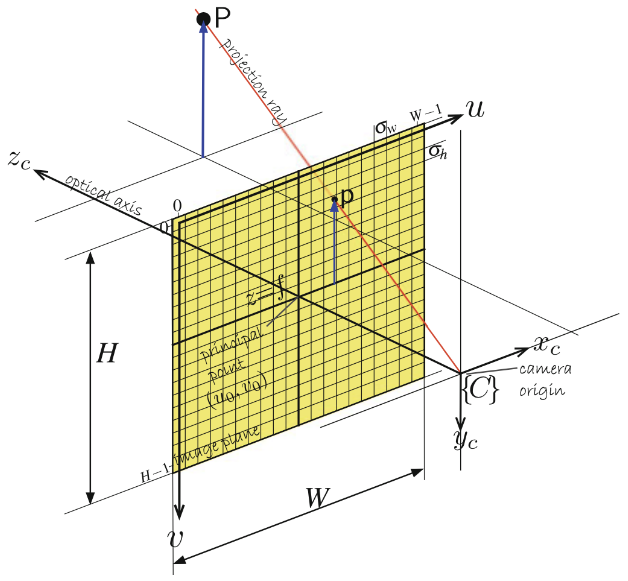


Figure 2.7: Camera projection schema from [33]. $\{C\}$ is the coordinate system of the camera.

camera, the perspective projection is:

$$x = f\frac{X}{Z},\ y = f\frac{Y}{Z} \tag{2.5}$$

where x, y are the image-plane coordinates and f is the focal length of the camera. Considering the *normalized image-plane coordinates* where $f = 1$, Eq. 2.5 can be rewritten as

$$x = X/Z,\ y = Y/Z$$

Taking into account a moving camera with body velocity $\boldsymbol{\nu} = (\boldsymbol{v}, \boldsymbol{\omega}) = (v_x, v_y, v_z, \omega_x, \omega_y, \omega_z) \in \mathbb{R}^6$, the velocity of $\mathbf{P}$ relative to the camera frame is

$$\dot{\boldsymbol{P}} = -\boldsymbol{\omega} \times \mathbf{P} - \boldsymbol{v} \Rightarrow \begin{cases} \dot{X} = -\omega_y Z + \omega_z Y - v_x \\ \dot{Y} = +\omega_x Z - \omega_z X - v_y \\ \dot{Z} = +\omega_y X - \omega_x Y - v_z \end{cases} \tag{2.6}$$

The relation between the feature velocity $\dot{\boldsymbol{p}} = [\dot{x}, \dot{y}]^T$, and the camera spatial velocity $\boldsymbol{\nu}$ is obtained by computing the temporal derivative of Eq.

2.5 and substitute it in Eq. 2.6:

$$\begin{bmatrix}\dot{x}\\ \dot{y}\end{bmatrix} = \underbrace{\begin{bmatrix}-1/Z & 0 & x/Z & xy & -(1+x^2) & y\\ 0 & -1/Z & y/Z & 1+y^2 & -xy & -x\end{bmatrix}}_{\boldsymbol{L_p}} \begin{bmatrix}v_x\\ v_y\\ v_z\\ \omega_x\\ \omega_y\\ \omega_z\end{bmatrix} \tag{2.7}$$

or, more concisely:

$$\dot{\boldsymbol{p}} = \boldsymbol{L_p}\boldsymbol{\nu} \tag{2.8}$$

Where $\boldsymbol{L_p} \in \mathbb{R}^{2\times 6}$ is the interaction matrix related to $\dot{\boldsymbol{p}}$.
The feature vector velocity $\dot{\boldsymbol{p}}$ can be expressed by exploiting the relationship between the image space coordinates (u, v) and the normalised image plane coordinates (x, y):

$$\begin{cases} u = f\frac{x}{\sigma_w} + u_0 \\ v = f\frac{y}{\sigma_h} + v_0 \end{cases}$$

Where:

- f is the focal length of the camera,
- $\sigma_i,\ i \in \{w, h\}$ is the pixel length in terms of width (w) and height (h),
- u_0, v_0 is the central point of the camera.

With $\bar{u} = u - u_0$ and $\bar{v} = v - v_0$ being the pixel coordinates relative to the central point of the camera, Eq. 2.2.1 can be rearranged into:

$$x = \frac{\sigma_w}{f}\bar{u},\ y = \frac{\sigma_h}{f}\bar{v} \tag{2.9}$$

with relative temporal derivative:

$$\dot{x} = \frac{\sigma_w}{f}\dot{\bar{u}},\ \dot{y} = \frac{\sigma_h}{f}\dot{\bar{v}} \tag{2.10}$$

Since the central point of the camera is fixed, it follows that $\dot{\bar{u}} = \dot{u}$ and $\dot{\bar{v}} = \dot{v}$. Starting from Eq. 2.7, the point velocity in terms of pixel coordinates is obtained by exploiting Eq. 2.9 and Eq. 2.10 finally having the relationship between the velocity of a point feature $\dot{\boldsymbol{p}}$ in terms of u, v, and the camera velocity:

$$\begin{bmatrix} \dot{u} \\ \dot{v} \end{bmatrix} = \begin{bmatrix} -\frac{f}{\sigma_w Z} & 0 & \frac{\bar{u}}{Z} & \frac{\sigma_h \bar{u}\bar{v}}{f} & -\frac{f^2+(\sigma_w \bar{u})^2}{\sigma_w f} & \frac{\sigma_h \bar{v}}{\sigma_w} \\ 0 & -\frac{f}{\sigma_h Z} & \frac{\bar{v}}{Z} & \frac{f^2+(\sigma_h \bar{u})^2}{\sigma_h f} & -\frac{\sigma_w \bar{u}\bar{v}}{f} & -\frac{\sigma_w \bar{v}}{\sigma_h} \end{bmatrix} \begin{bmatrix} v_x \\ v_y \\ v_z \\ \omega_x \\ \omega_y \\ \omega_z \end{bmatrix} \tag{2.11}$$

The interaction matrix $\boldsymbol{L_p}(u, v, Z)$ related to $\boldsymbol{p}$ depends only on u, v of the 2D image space and Z, the point's depth relative to the camera frame, in the 3D space. Such a depth varies at each iteration, requiring ongoing estimation. A simpler approach is to fix it to a chosen depth value. Without loss of convergence, approximating such interaction matrix by fixing Z to be the Z at convergence, where extracted features and desired ones match, namely when $\boldsymbol{e} = 0$ [33].

Using the interaction matrix of Eq. 2.11 in Eq. 2.4, the velocity commands needed to move the camera towards the goal position are finally obtained.

Until now, only one feature point in the 2D image space was considered, but one point is not enough to determine the pose of the camera at convergence. For example, to control the 6 DOF of a camera in the 3D space at least three points are required. In our case study, dealing with four points of the corners allows the pose estimation of the camera by stacking the interaction matrices of each point as shown in Eq. 2.12:

$$\begin{bmatrix} \dot{\boldsymbol{p}}_\mathbf{0} \\ \dot{\boldsymbol{p}}_\mathbf{1} \\ \dot{\boldsymbol{p}}_\mathbf{2} \\ \dot{\boldsymbol{p}}_\mathbf{3} \end{bmatrix} = \begin{bmatrix} \boldsymbol{L}_{\boldsymbol{p}_\mathbf{0}}(u_0, v_0, Z_0) \\ \boldsymbol{L}_{\boldsymbol{p}_\mathbf{1}}(u_1, v_1, Z_1) \\ \boldsymbol{L}_{\boldsymbol{p}_\mathbf{2}}(u_2, v_2, Z_2) \\ \boldsymbol{L}_{\boldsymbol{p}_\mathbf{3}}(u_3, v_3, Z_3) \end{bmatrix} \boldsymbol{\nu} \tag{2.12}$$

Finally, as shown above, the desired velocity of the camera to match all four points is obtained by exploiting Eq. 2.12 and Eq. 2.4:

$$\boldsymbol{\nu} = -\lambda \begin{bmatrix} \boldsymbol{L}_{\boldsymbol{p}_\mathbf{0}}(u_0, v_0, Z_0) \\ \boldsymbol{L}_{\boldsymbol{p}_\mathbf{1}}(u_1, v_1, Z_1) \\ \boldsymbol{L}_{\boldsymbol{p}_\mathbf{2}}(u_2, v_2, Z_2) \\ \boldsymbol{L}_{\boldsymbol{p}_\mathbf{3}}(u_3, v_3, Z_3) \end{bmatrix}^+ (\boldsymbol{p} - \boldsymbol{p}^*) \tag{2.13}$$

This section concludes with the IBVS control considered in the quadcopter case. Such a robotic platform is underactuated and has a coupling between velocity on the x axis and the pitch and on the y axis and the roll. This means that the drone cannot go forward without pitching forward, and the same happens with the horizontal movement and the roll. Due to such a system, the rolling rate ω_x and the pitching rate ω_y are not taken into consideration, meaning that only v_x, v_y, v_z and ω_z velocity commands are used.

2.3 Computer Vision

Vision is one of the fundamental senses that let humans perceive the world through light reflection. Starting in the late 60s with the aim to mimic the human visual system, the field of Computer Vision grew exponentially. Lots of algorithms were developed, from classic approaches like edge detection to Deep Learning approaches that nowadays outperform many classic CV algorithms for tasks like image classification or image segmentation. Classic CV algorithms are less computationally expensive with respect to the training of DL approaches. Their focus is on extracting specific features, and to solve a specific problem, it's often necessary to combine different algorithms to build a pipeline, resulting in plenty of parameters to be tuned. Instead, DL approaches for computer vision rely on task-specific datasets to train Convolutional Neural Network to solve the task at hand. These approaches are more computationally intensive, concerning the training part, but more flexible since the same architecture can be trained to solve different tasks by supplying custom datasets.

This section is devoted to presenting the classic CV algorithms employed in the project, while Sec. 2.4.3 will focus on the DL architecture adopted to solve the corner detection task.

2.3.1 Edge detection: the Canny edge detector

Presented in the mid-80s, the Canny edge detector [5] is a multi-stage algorithm that extracts reliable edges from grayscale images, whose output is a binary image.
The steps of such an algorithm are:

1. Edge detectors are sensitive to noise, hence a Gaussian filter is applied to smooth the input image.

2. The Sobel operator [2] is exploited to compute the approximated gradient in both directions x and y:.

   $$\mathbf{G}_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} * \mathbf{I}, \quad \mathbf{G}_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * \mathbf{I}$$

   Where $\mathbf{I}$ is the image and $*$ is the convolution operation.
   The resulting gradient magnitude is given by $\mathbf{G} = \sqrt{{\mathbf{G}_x}^2 + {\mathbf{G}_y}^2}$, and the relative orientation of the gradient is $\theta = \tan^{-1}(\mathbf{G}_y/\mathbf{G}_x)$.
   The angles are rounded to be in one of the following cases:

   – horizontal: $\theta = 0°$
   – diagonal: $\theta \in [45°, 135°]$
   – vertical: $\theta = 90°$

3. Non-maximum suppression is performed in each pixel along the directions found during the previous step. The purpose is to keep only the pixels with maximum gradient value along the gradient direction. This is useful to keep only the candidate edges and remove the pixels that are not considered to be part of the edge. This is an edge-thinning technique.

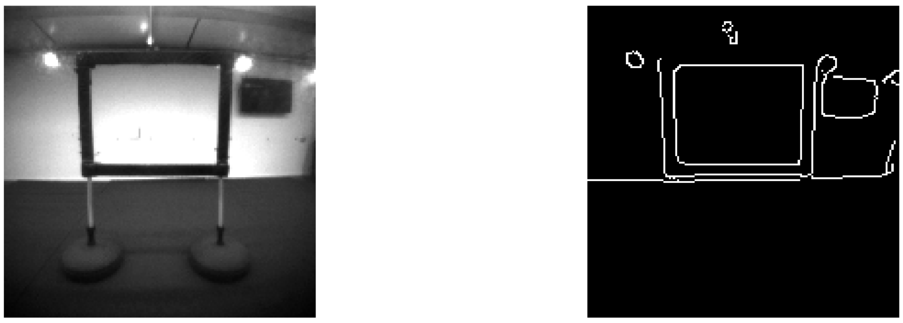
4. A double threshold $[t_1, t_2]$, $t_1 < t_2$ is used to classify pixels by their gradient intensity. If its intensity is greater than t_2, then the pixel is kept and labelled as "strong". Instead, if its intensity falls inside the interval $[t_1, t_2]$, it is labelled as "weak", and a further study is conducted. In the final case in which the intensity is below the lowest threshold t_1, the pixel is no longer considered to be part of an edge.

   The thresholds $[t_1, t_2]$ are empirically defined and depend on the image under study.

5. In the final step, "weak" pixels are studied to choose whether to keep or discard them. Only those which are connected to "strong" pixels are kept. Otherwise, they are discarded.

2.3.2 Contour detection: the Suzuki Abe algorithm

A contour is defined as a curve joining all continuous points along a boundary having the same intensity.



(a) Canny input image. (b) Canny output image.

Figure 2.8: Canny example. Given the input image (a), the output of the Canny edge detector is shown in (b).

The contour detection algorithm proposed by Suzuki and Abe [4] works on binary images and is able to extract the hierarchy of the contours in the image.

Working with binary images, it considers 1-connected components, composed of pixels value equal to 1, and 0-connected components, the same as 1-components but with 0-value pixels, to extract the boundaries. The algorithm works in a TV-raster way, from left to right and from top to bottom. It starts scanning horizontally from the top left corner of the image. Considering 0-value pixels as the background, when the scanning finds a 1-value pixel, the computation of the contour starts following a clockwise and counter-clockwise manner. To keep the hierarchy of the contours, it sets integer indexes to the boundary values, and whenever it reaches the starting point of the boundary, it proceeds with the scanning. The process stops when the scanning reaches the bottom right pixel of the image.

2.3.3 Corner detection: the Harris detector

A corner is a point of interest where two edges of different directions meet each other. Since an edge can be defined as a sudden change in the image brightness, a corner features a variation in intensity.

The Harris corner detector [6] was introduced by Harris and Stephens in the late 80s. It is a gradient-based corner detection that works on grayscale images.
Considering an image I, an image patch W and its shifted version by $(\Delta x, \Delta y)$ is used to compute the sum of squared differences, which, using the Taylor

expansion, is the following:

$$f(\Delta x, \Delta y) = \sum_{(x_k,y_k)\ \in\ W} (I(x_k, y_k) - I(x_k + \Delta x, y_k + \Delta y))^2 \underbrace{\approx}_{\text{Taylor}} \begin{bmatrix} \Delta x & \Delta y \end{bmatrix} M \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix}$$

Where

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

is a 2×2 matrix, namely the second-moment matrix. This matrix is used to compute the Harris corner response score $R = \det(M) - k\,\mathrm{tr}(M)^2$, where k is an empirically determined constant. Both the trace and the determinant of M depend on the eigenvalues α, β of M. This means that based on the eigenvalues, R assumes different values. Based on such a value, a point is classified as a corner or not. More specifically:

- $|R|$ small represents a flat region.
- $R >> 0$ denotes a corner. Both the eigenvalues are large.
- $R < 0$, the pixel belongs to an edge. One eigenvalue is greater than the other.

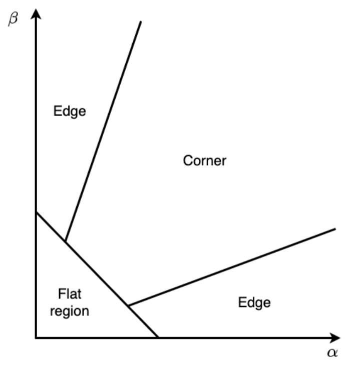


Figure 2.9: Harris eigenvalue response

To get an optimal value, a threshold of R is set to select the corners.

2.4 Machine Learning

Machine Learning (ML) is a sub-field of Artificial Intelligence (AI) that covers a large pool of algorithms able to solve task-specific problems by learning

from given data. In this sense, they mimic human behaviour.
A more formal and operational definition is given by T. Mitchell:

> A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P if its performance at tasks in T, as measured by P, improves with experience E.

According to the adopted learning approach, algorithms can be classified into different classes like supervised learning, unsupervised learning, self-supervised learning, reinforcement learning, etc. Throughout the project, the detection modules take advantage of algorithms belonging to supervised and unsupervised learning. A brief explanation is given in the following.

2.4.1 Supervised Learning

This approach relies on labelled datasets, which means that both the inputs and expected outputs are known. The aim of the learning algorithm is to build a mathematical model that maps the input to the right output through an optimisation process, the training.
The labelled dataset is divided into training and testing datasets. While the former is used to train the algorithm, the latter is used to test the trained algorithm on unseen data, hence giving a measure of how well the algorithm generalises. The training is performed through the optimisation of an objective function in an iterative way.

Models depend on some specific parameters, called *hyperparameters*, that change the resulting behaviour. Tuning such parameters is necessary to achieve the desired outputs. When a model learns "by heart" the training data, it "overfits" because it lacks generalisation capability. In this case, better tuning of the hyperparameters or regularisation techniques can mitigate this behaviour.

Most common algorithms belonging to the supervised learning category solve the task of classification or regression. In the former case, the model learns to map the input to a set of categorical values, while in the latter, the output of the model will be a value belonging to the set of real numbers.

Some algorithms, belonging to this category are SVM, Random Forests, Neural Networks, k-Nearest-Neighbor, and many others.

2.4.2 Unsupervised Learning

In contrast to the supervised case, in the unsupervised one, there are no labels in the data, and the aim is to find patterns in the data. Some applications are grouping/clustering and anomaly detection. The computer vision pipeline, described in Sec. 3.1, relies on a clustering algorithm named K-Means, presented in the following section.

K-Means

Belonging to the unsupervised learning approach, K-Means is a clustering algorithm that aims to partition the unlabelled data into K clusters by minimising the within-cluster sum-of-squares. Every cluster has a centroid, and each observation of the dataset is assigned to the cluster with the closest centroid.

This is an iterative algorithm originally proposed by Lloyd for signal processing [1]. Starting with a random initialisation of the k clusters, it iteratively assigns the data to the closest centroid and then computes the mean of each cluster, namely the new centroids. The process ends when the differences between the new centroids and previous ones are below a given threshold.

More in details, given a dataset $\mathbf{X} = (\mathbf{x}_1, \ldots, \mathbf{x}_n)$ the steps are:

1. Initialize the K centroids.

2. Assign the samples to the nearest centroid following

$$\arg\min_S \sum_{i=1}^{K} \sum_{\mathbf{x}\in S_i} ||\mathbf{x} - \boldsymbol{\mu}_i||^2 \tag{2.14}$$

   With S the set of K clusters,

$$\boldsymbol{\mu}_i = \frac{1}{|S_i|} \sum_{\mathbf{x}\in S_i} \mathbf{x}_i \tag{2.15}$$

   being the centroid of the i-th cluster, and $|S_i|$ its size.

3. Compute the new centroids using Eq. 2.15.

4. Iterate over steps 2 and 3 and stop whenever the differences between the new centroids and the previous ones are below a given threshold or the maximum number of iterations is reached.

When the algorithm stops, all the input data will be associated with one of the K clusters.

2.4.3 Deep Learning

Deep Learning concerns a subset of ML algorithms: the Artificial Neural Networks ANN with three or more layers. Such architectures mimic the biological neural networks of animals by stacking layers of artificial neurons connected to one another. By providing task-dependent data, such algorithms can be trained to solve the corresponding task without programming it explicitly. Deep learning refers to ANN with many layers of neurons where features are extracted from the input data throughout the architecture.
Graphic Processing Units are well suited to perform matrix and vector computations, which are fundamental in DL, and thanks to the evolution of GPUs, DL has been gaining interest in recent years. This led to a lot of research in this field. Thus, many architectures have been developed to solve different tasks. The next section presents PULP-Frontnet, the chosen architecture to solve our feature extraction task.

PULP-Frontnet

The neural network used in the DL pipeline, described in Sec. 3.2, is an adapted version of PULP-Frontnet, introduced by Palossi et al. [27]

Running fully onboard on the Crazyflie device, this network was developed to assess the relative pose estimation of the drone with respect to a moving human subject using 160 × 96 grayscale images.

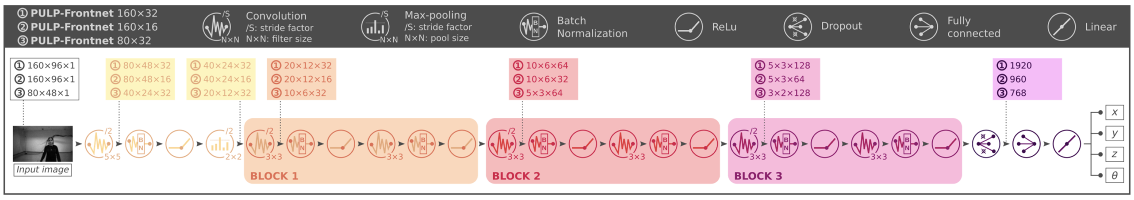


Figure 2.10: PULP-Frontnet architecture [27]

As reported in Fig. 2.10, PULP-Frontnet is a convolutional neural network. It features convolution, batch normalisation, ReLU, and max-pooling layers, reducing the input image by 4× thanks to the stride of 2. The three inner blocks are composed of convolution, batch normalisation and ReLU layers. The architecture ends with a dropout layer followed by a fully connected layer. To solve the pose estimation task, the network outputs the 3D coordinates x, y, z and the rotational angle θ.

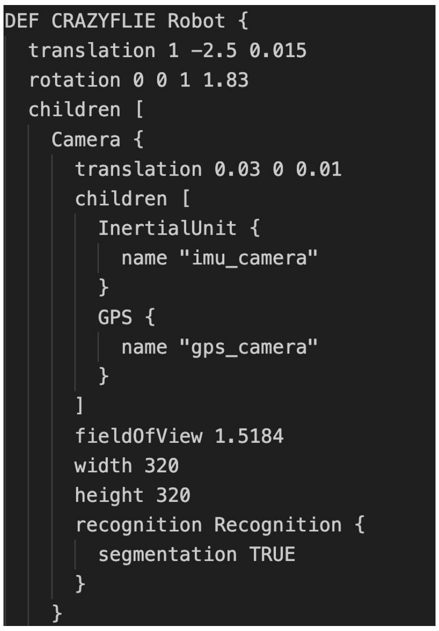
2.5 Webots

Webots [37] is an open-source, deterministic, and updated professional mobile robot simulator developed by Cyberbotics Ltd.
Webots is used to simulate many kinds of robots in 3D virtual worlds with physics properties relying on the ODE (Open Dynamics Engine) library. It allows the user to prototype robotics simulations in interactive environments with low effort. It features a large pool of robots, sensors, actuators, objects and materials. On the robot side, there are flying drones, modular robots, aerospace vehicles, and many others. The behaviour of a robot during a simulation is described via programming a Webots controller. Other than that, such a controller can also be used to read and collect data from the sensors of the robots. This can be done in plenty of programming languages: C, C++, Python, Java, and MATLAB. Moreover, it can interact with ROS using specific topics.

To start a simulation, the following is required:

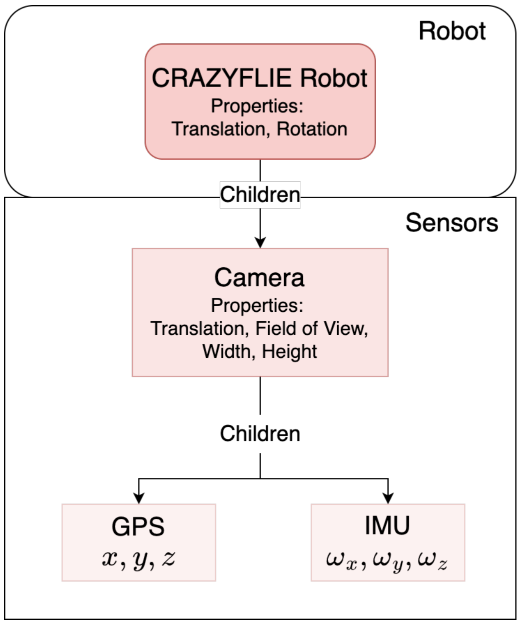
- A Webots world file, which contains the description of every object, like textures, position, orientation, appearance, and physical properties. Worlds are organised hierarchically. Hence, objects can contain other objects. For example, a mobile robot containing a set of sensors like a GPS, an IMU, and a camera. Such sensors are the *children* of the robot. Figure 2.11 shows an extract of the world file, which describes the robot, its properties, and its children.

- A robot can have at most one controller program. In this file, the user programs the behaviour of the robot during the simulation.
  A robot can be set as a *supervisor* and execute operations that normally can not be done by a real robot. For example, set its pose or the position of objects.



```
DEF CRAZYFLIE Robot {
  translation 1 -2.5 0.015
  rotation 0 0 1 1.83
  children [
    Camera {
      translation 0.03 0 0.01
      children [
        InertialUnit {
          name "imu_camera"
        }
        GPS {
          name "gps_camera"
        }
      ]
      fieldOfView 1.5184
      width 320
      height 320
      recognition Recognition {
        segmentation TRUE
      }
    }
```

(a) Extract of the world file concerning the CRAZYFLIE robot.



(b) Hierarchy visualization of the world description reported on the left.

Figure 2.11: In (b), the hierarchy visualisation of the code in (a) is reported. Rounded corners refer to robots, while square corners denote sensors. The filling colour goes from darker to lighter as the depth of the hierarchy increases. Both GPS and IMU are attached to the camera sensor of the drone. The GPS sensor outputs the x, y, and z displacement with respect to the origin of the world, while the IMU outputs the angular velocity with respect to the world frame.

Chapter 3

Methods

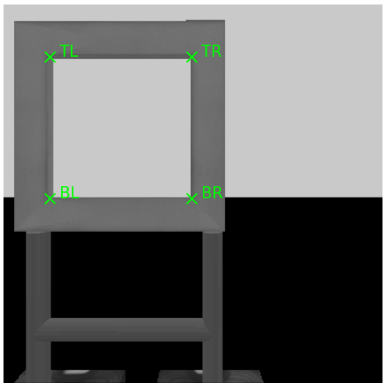


Figure 3.1: Desired output of the pipelines. Ground truths are shown in lime colour.

This chapter is devoted to presenting the core of the thesis work. In Sec. 3.1 and Sec. 3.2, a thorough presentation about the corner detection pipelines is given. These are necessary to extract and label the four inner corners of the gate, as shown in Fig. 3.1. The former pipeline relies on classic CV algorithms, while the latter exploits a CNN to perform the corner detection. The CV-based pipeline detects the corners by extracting edges from the input image and searching for contours (continuous and closed regions) on those edges. Then, corner detection on the extracted contours is performed. Instead, the DL-based pipeline relies on an adapted version of the PULP-Frontent CNN. Such pipelines are the core of the feature extraction module shown in Fig. 3.2. Both pipelines take their ground on the concepts presented in Chapter 2.

Both pipelines take as input grayscale images. The CV-based one takes as input 320x320 images, native from the camera. Instead, the DL-based one takes as input a binned version, 160x160, of the input images. In the input image, a square of 2×2 pixels is read out as just one pixel. In the first case, the images are processed without taking care of a possible future deployment, whilst in the other case, starting from the PULP-Frontnet architecture that receives 160x96 grayscale images, the adapted version is extended to work with 160x160.

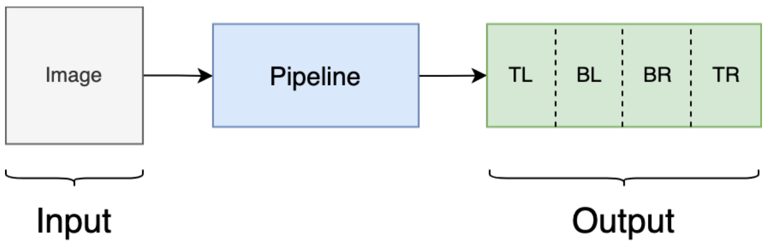


Figure 3.2: Structure of the feature extraction module. The input is fed to the pipeline, which extracts the four corners of the gate. The order is the top-left (TL), the bottom-left (BL), the bottom-right (BR), and the top-right (TR).

Section 3.3 dives deep into the details concerning the simulations where the corner detection pipeline feeds the extracted features to the visual servoing control. Finally, in Sec. 3.4, the setup adopted to collect some real data is presented.

3.1 CV-based pipeline

The following pipeline is labelled as CV-based since it relies on traditional CV algorithms with a pinch of Machine Learning. The main steps of the pipeline are shown in Fig. 3.3. The goal of the pipeline is to extract the image coordinates of the four inner corners. Using only corner detection algorithms like Harris [6] or FAST [11] is not sufficient since all the corners in the image are taken into consideration. This leads to the necessity of identifying the gate. To do that, the first step is to extract all the edges recognised by the Canny detector. The following consists of applying the contour detection algorithm of Suzuki and Abe to extract all the feasible contours. Out of all the extracted contours, only those which have a shape referable to a square are kept. The adopted discriminant is to consider the aspect ratio and keep only the contours with an aspect ratio near one. From

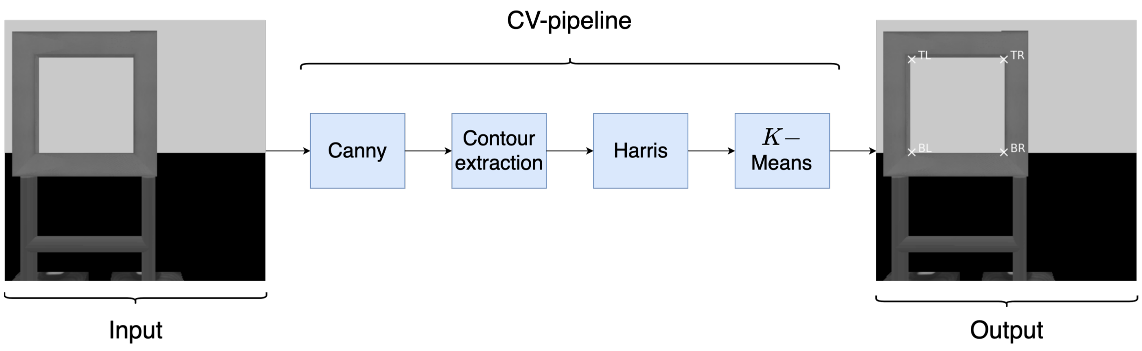


Figure 3.3: CV-based pipeline. Our goal is to extract the four corners of the gate, hence for the $K-$Means, $K = 4$.

those contours, the Harris detector extracts clusters of points whose centroids are computed by the $K-$Means and will be considered as the corners of the gate. A pseudocode of such a pipeline is reported in Alg. 1 where most of the parameters are not shown to lighten the reading.

Algorithm 1 CV-based pipeline pseudocode.

```
Require: Grayscale image $I \in \mathbb{N}^{320\times 320}$, AR thresholds, selection method
  $I \leftarrow$ GaussianBlur($I$)
  $I \leftarrow$ Dilation(Canny($I$))
  if Contour($I$) $\in$ AR thresholds then
      ContoursList$\leftarrow$ Contour($I$)
  end if
  for $c \in$ ContoursList do
      $H \leftarrow$ Harris($c$)
      $D \leftarrow$ createDataset($H$)
      CornerMatrix $\leftarrow$ ClassifyCorners($K-$Means($D$))
      BestCorners $\leftarrow$ CornerMatrix
  end for
  BestCornerMatrix $\leftarrow$ selection method(BestCorners)
  return BestCornerMatrix
```

After a presentation of the pseudocode, a detailed explanation of the working pipeline is provided. It is worth noting that the pipeline has many parameters that can be tuned, and it is very susceptible to the image. First of all, the input image is smoothed by the Gaussian blur filter, which can vary in kernel size and standard deviation. Then, the Canny detector requires two thresholds that depend on the image at hand. After applying the contour detection algorithm, an AR interval near one is chosen to discard contours that do not fit the squared form. Concerning the Harris detector, the size of the window W to take into account in the second-moment matrix computation is chosen along with the size of the Sobel operator (introduced in Sec. 2.3.1 while presenting Canny detector), and the constant k. Finally, the K-Means clustering algorithm has some parameters that have to be tuned, starting from the number of clusters K, the threshold used to check the differences between centroids of two subsequent iterations, as stopping criteria, and the number of maximum iterations. Canny is the first CV algorithm of the pipeline, and being sensitive to images, whenever conditions change, the pipeline has to be tuned accordingly to perform well. This pipeline was tested on an empty Webots' world, presenting a black floor, a plain light blue background, and the gate. Fig. 3.4 shows a picture of the empty world in Webots.

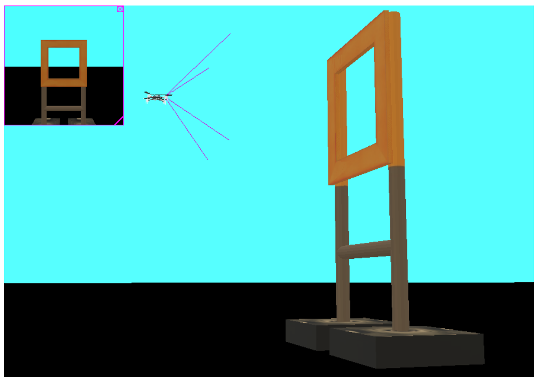


Figure 3.4: Empty world used to develop and test the CV-based pipeline. Purple lines are the camera frustum. They represent what the drone sees.

Gaussian blur filter is the first step. Given the kernel dimension, its standard deviation is computed as, according to OpenCV[1] implementation,

$$\sigma = 0.3 * ((ksize - 1) * 0.5 - 1) + 0.8$$

where $ksize$ is the kernel size. The resulting kernel is a matrix whose entries values are defined by Eq. 3.1:

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{\frac{-(x^2+y^2)}{2\sigma^2}} \tag{3.1}$$

[1]https://docs.opencv.org/4.x/d4/d86/group__imgproc__filter.html.

With (x, y) being, respectively, the horizontal and vertical displacements with respect to the central entry of the kernel matrix. Such a kernel matrix is then convolved with the input image, aiming to remove the noise from the image. This first step has to be tuned carefully. The main parameters to tune are the size of the kernel and the standard deviation. If the kernel dimension is too high or the standard deviation is not tuned correctly, the result is an image too smoothed. Then, when we apply the Canny detector, there will be no extracted edges. An example of how the blurring affects the Canny's output is reported in Figure 3.5.

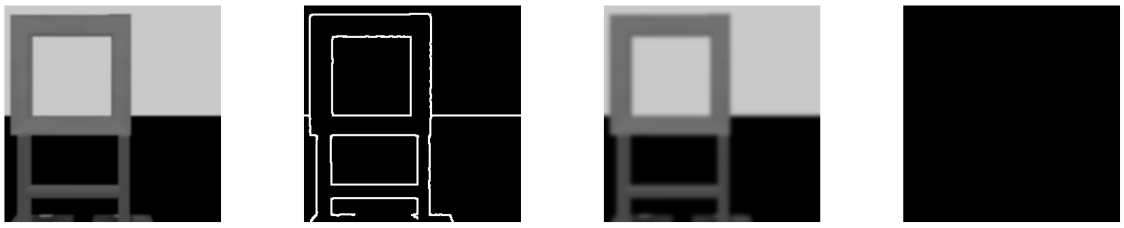

(a) (b) (c) (d)

Figure 3.5: Input and output of the Canny edge detector. Figure (a) and Figure (c) show two blurred images of Figure 3.1 obtained with, respectively, 7×7 and 19×19 kernel sizes. Figure (b) and Figure (d) show detected edges. While in the former case, there are edges, in the latter, none are recognised. Canny's thresholds are set to 100 and 200.

For the Canny detector, recalling from Sec. 2.3.1, the low and the high threshold are parameters to be tuned. Edges below the low one will be discarded, so it cannot be set too high. On the Canny's output, a dilation operation to enlarge the extracted edges since they would be too thin.

Then, the contour extraction algorithm presented in Sec. 2.3.2 is performed. In this case, only the contours with an Aspect Ratio (AR) falling into a given interval are taken into consideration for the next steps. The AR is the ratio between the width and height of the extracted contour. Even in this case, such an interval must be studied because if no constraints are considered, namely $AR = [0, +\infty)$, every contour will be taken into consideration in the next step. Knowing that the gate has a square form, an interval near to 1 is a good choice. Tolerating ARs, which are not exactly 1, skewed versions of the gate can be processed. This is required since the drone won't be facing the gate every time. Degenerate cases like the drone seeing the gate from the side are not taken into account. Moreover, with such an interval, the amount of contours to be processed in the following steps can be reduced.

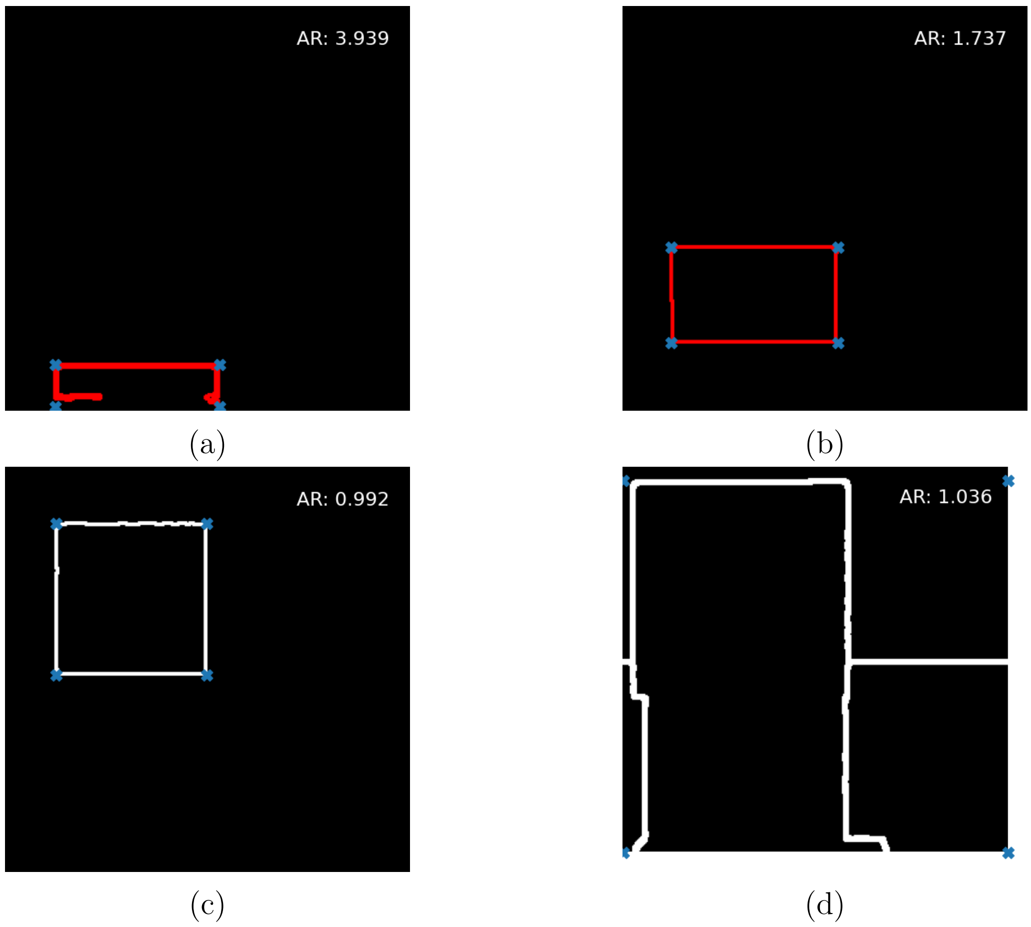


Figure 3.6: Output of the contour extraction given the Canny's output as input. The light blue crosses show the points used to compute the Aspect Ratio (AR), which are the extremity points of the contour. In this case, the given AR interval is [0.6,1.25]. Red contours are the discarded ones, whilst the white ones are kept for the next step.

The Harris detector is then applied to the selected contours. In this case, the parameters to tune are the block size (the window W, Sec. 2.3.3) over which the displacement is computed, the dimension of the Sobel operator (the one that computes the derivatives), the k in the score function, and the threshold used to label a pixel as a corner or not. The resulting output is a binary image with clusters of pixels.
The final step is to perform the $K-$Means to compute the centroids of $K = 4$ clusters. Such centroids will be the image coordinates of the gate's corners.

To compute them, a temporary dataset is built. Such a dataset consists of all pixels labelled as corners. Each row denotes a pixel, and its features are the u, v coordinates of the pixel. On this temporary dataset, $K-$Means is run, and then, the four centroids are extracted and classified into:

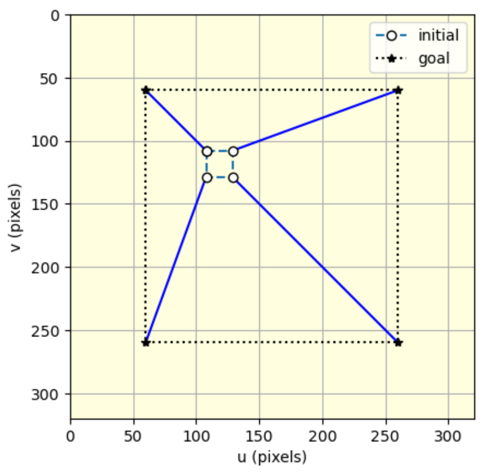
0. Top-Left (TL)

1. Bottom-Left (BL)

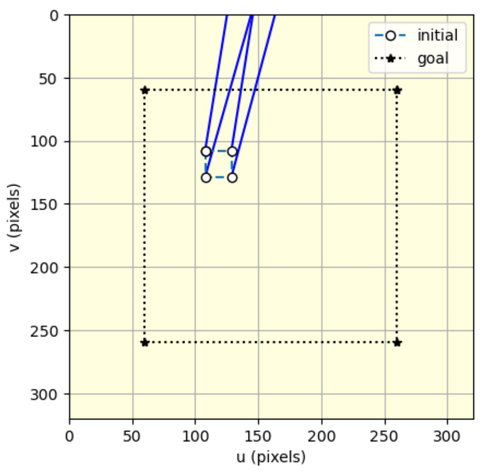
2. Bottom-Right (BR)

3. Top-Right (TR)

This classification is necessary since the correspondence between extracted features and the desired position is a must in IBVS. Fig. 3.8 shows the importance of the correspondence.



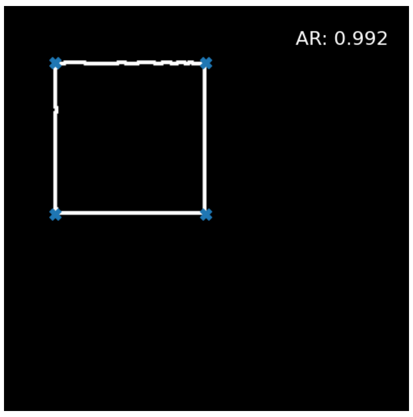
(a)



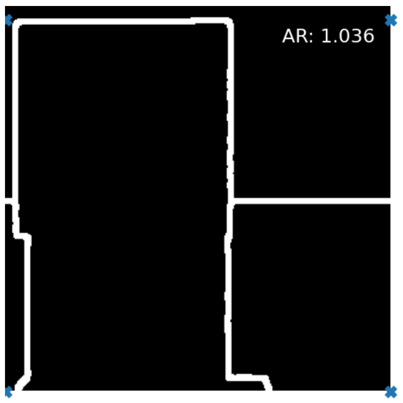
(b)

Figure 3.8: The order of the feature points is fundamental. In (a), feature points correspond point-wise with the desired ones. In (b), reversing order is considered, e.g., the order is TR, BR, BL, and TL. In this case, the velocity commands sent to the controller lead the feature points outside the image without convergence.

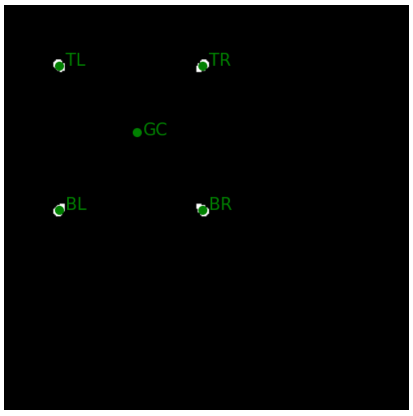
The last step is to select one set among all the sets of extracted corners. To do that, one out of two simple heuristics is considered. One is called *highest*, the other *central point*. The former selects the set of corners with the highest gate centre (GC), namely the mean of the corners' coordinates.



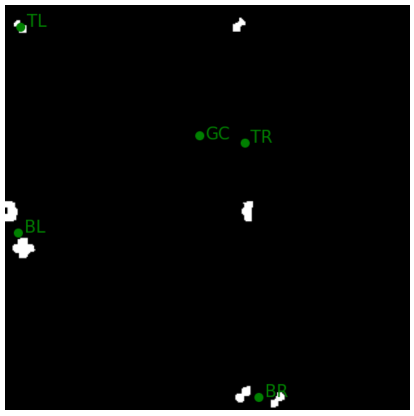
(a)



(b)



(c)

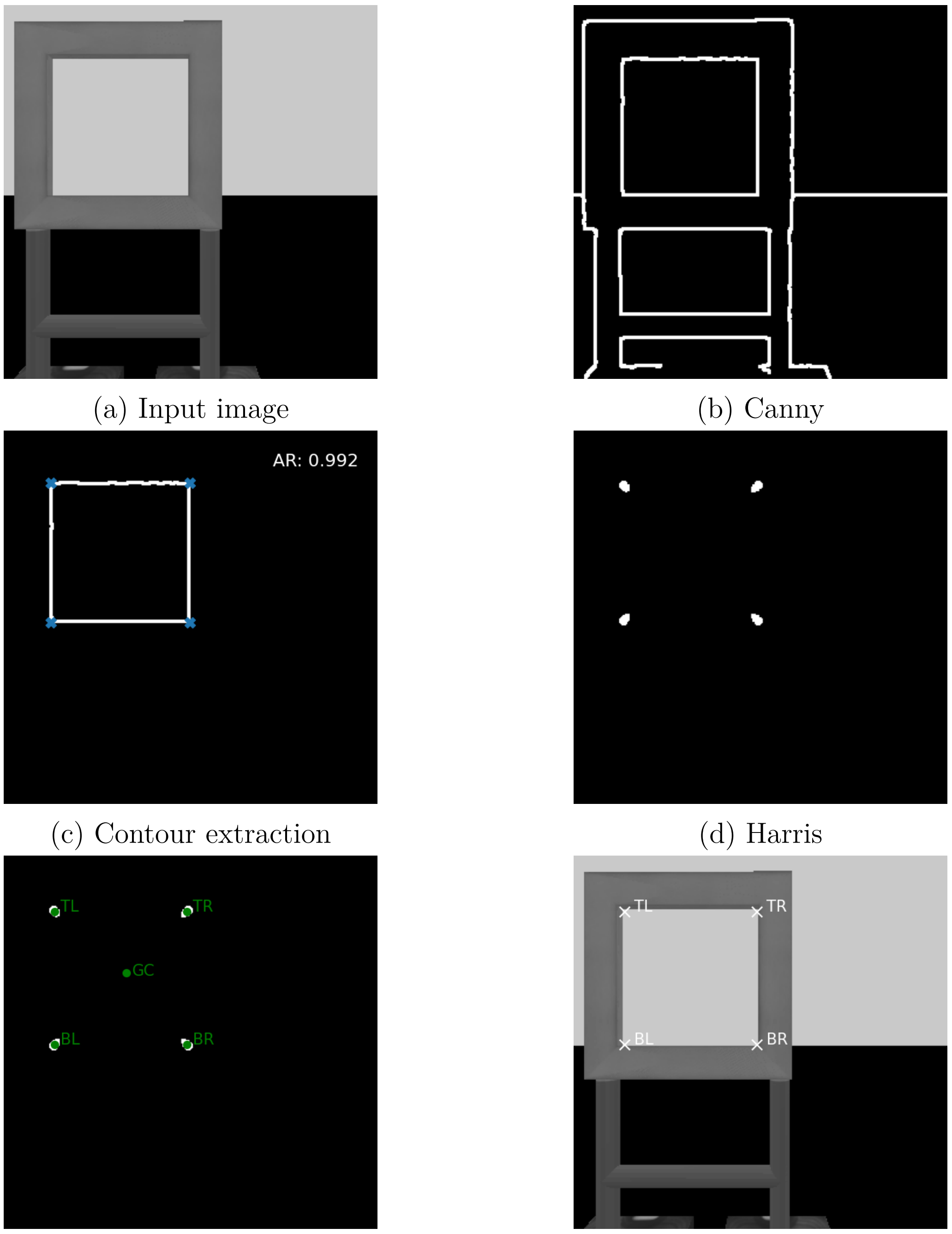


(d)

Figure 3.7: To the selected contours, we first apply the Harris corner detector (which gives the white clusters) and then compute the $K-$Means. The centroids are then labelled as TL, BL, BR, and TR. The GC label (which stands for 'Gate Centre') is the mean of the centroids and is used to select the set of corners according to a given heuristics. In this case, the set of corners with the highest gate centre is selected and shown in (c).

The idea is that in simple worlds featuring only the gates and no background, the contour to be extracted will always be the highest one. Instead, the latter selects the set with the nearest gate centre to the central point of the camera. In this case, the idea behind this is that, considering that the drone is flying at the gate level, the set of correct corners is the nearest to the central point of the camera. In both cases, only the v coordinate of the image space is taken into consideration since the gate can be located anywhere in the u direction.

To conclude the description of the CV-based pipeline, Fig. 3.9 provides a visualisation of the working pipeline.



(a) Input image (b) Canny

(c) Contour extraction (d) Harris

(e) $K-$Means (f) Output

Figure 3.9: A visualization of how the CV-based pipeline works inside.

3.2 DL-based pipeline

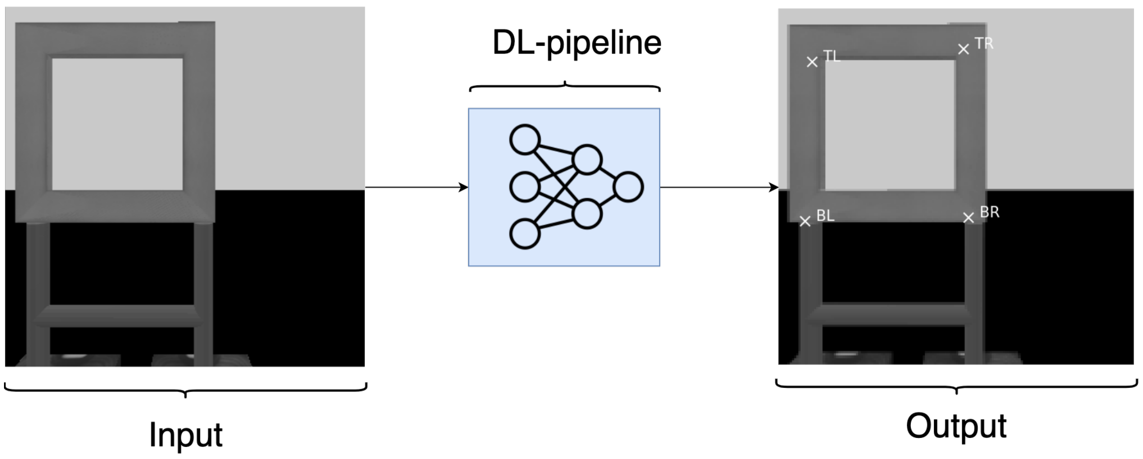


Figure 3.10: DL-based pipeline.

This pipeline exploits a Convolutional Neural Network (CNN) to solve the corner detection task. With the aim of being deployed onboard the Crazyflie 2.1, the used architecture is the well-tested PULP-Frontnet introduced in Sec. 2.4.3. The original network processes grayscale images of size 160×96, producing the relative position and angle displacement of the person in front of the drone. Considering the use case of this thesis work, the network is adapted to work with 160×160 grayscale images and output the coordinates of the four corners, resulting in eight image-space coordinates. The rearranged structure is reported in Tab. 3.1 with an order of trainable parameters of 300k. The training set was composed of 75k synthetic 160×160 images collected in Webots, which became 750k after augmentation. The training was performed on such a dataset using batches of 64 samples, the $L1-$loss and the Adam optimizer [12] with a learning rate of 0.001. $L1-$loss, or Mean Absolute Error (MAE), is defined as follows:

$$\mathrm{MAE} = \frac{\sum_{i=1}^{n} |y_i - \hat{y}_i|}{n} \tag{3.2}$$

where y_i are the predictions, and $\hat{y}_i$ are the desired values. An early stopping criteria of 10 epochs was adopted to check validation loss improvements. After ten epochs, if the validation loss doesn't improve, the training stops. On a training of 30 epochs, the 20th was the most performing one.

Both validation and test sets feature never-seen images during the training.

The following sections further describe the dataset collection and augmentation processes.

Input Size	**Layer**	**Stride**	**Filter Shape**	**Output Size**
160x160x1	Conv	2	5x5x32	80x80x32
80x80x32	BatchNorm + ReLU			80x80x32
80x80x32	MaxPooling	2		40x40x32
40x40x32	Conv	2	3x3x32	20x20x32
20x20x32	BatchNorm + ReLU			20x20x32
20x20x32	Conv	1	3x3x32	20x20x32
20x20x32	BatchNorm + ReLU			20x20x32
20x20x32	Conv	2	3x3x64	10x10x64
10x10x64	BatchNorm + ReLU			10x10x64
10x10x64	Conv	1	3x3x64	10x10x64
10x10x64	BatchNorm + ReLU			10x10x64
10x10x64	Conv	2	3x3x128	5x5x128
5x5x128	BatchNorm + ReLU			5x5x128
5x5x128	Conv	1	3x3x128	5x5x128
5x5x128	BatchNorm + ReLU			5x5x128
5x5x128	Dropout			5x5x128
5x5x128	Fully Connected		1x3200	8

Table 3.1: Network architecture used to solve the corner detection task. It is an adapted version of PULP-Frontnet [27].

3.2.1 Dataset collection

To properly train the network, a lot of time was devoted to collecting suitable data. To let the network know how to extract the four corners, the dataset must be composed of grayscale images showing the gate along with its corners' coordinates. Hence, given an image input, the output consists of eight coordinates, two for each corner. The following paragraphs are devoted to presenting the collection of the training dataset.

Corner	*u*	*v*
Top-Left (TL)	65	6
Bottom-Left (BL)	62	110
Bottom-Right (BR)	131	103
Top-Right (TR)	125	39

Table 3.2: Image space coordinates of the corners shown in Fig. 3.11 (b).

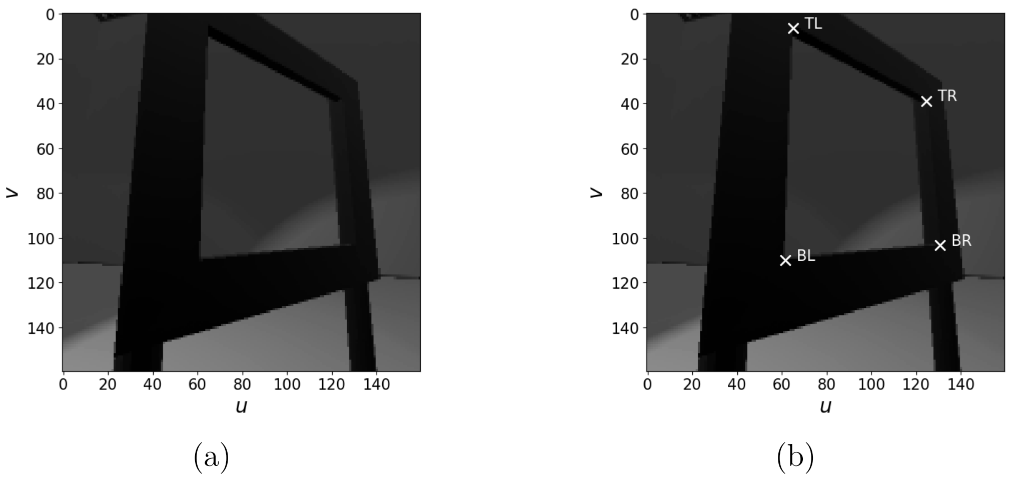
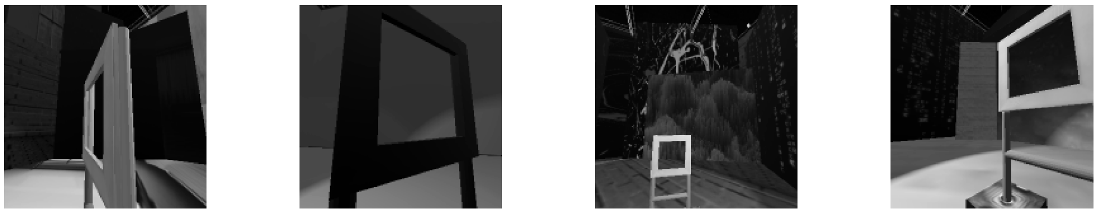


Figure 3.11: Example of dataset image in (a). (b) shows the position of the ground truths.

Images along with the corresponding corners' coordinates were collected in Webots. An example image along with the corners is shown in Fig. 3.11 and Table 3.2. The setup was the following: the gate is fixed and rotates in place. Its colour can change, and some lights are turned on and off and changed in intensity to change the luminosity of the scene. The scene is surrounded by panels in which background images are put to mimic different scenes. These images change in colour and scale. After a while, those walls are randomly spawned and changed in scale around the gate. The images are picked randomly from a pool of background images. Some images are realistic, others are not. Moreover, when the walls are spawned randomly, they can overlap each other. The aim is to create random backgrounds to make the network robust and just focus on the gate shape. In the same way, the floor panel is spawned randomly, and its image is randomly picked.

The drone is randomly spawned between the gate and the wall panels. To always have the four corners in the Field of View (FOV) of the camera, as the first thing, x and y coordinates are computed by uniformly sampling a radius and the angle in polar coordinates. Then, the height, the pitch, and the yaw are sampled to have the four corners in the FOV of the camera. Some images of the resulting dataset are shown in Fig. 3.12.



(a) (b) (c) (d)

Figure 3.12: Example of training dataset images.

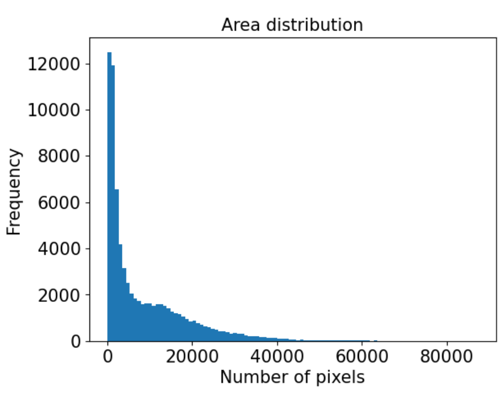
All of this is done via the Webots controller of the Crazyflie robot set with the *supervisor* property (see Sec. 2.5).

The dataset needs to have images of the gate seen from different angles and distances, and since the drone will move towards the gate, a considerable number of images near the gate is required. An analysis of the resulting dataset is reported in the following:

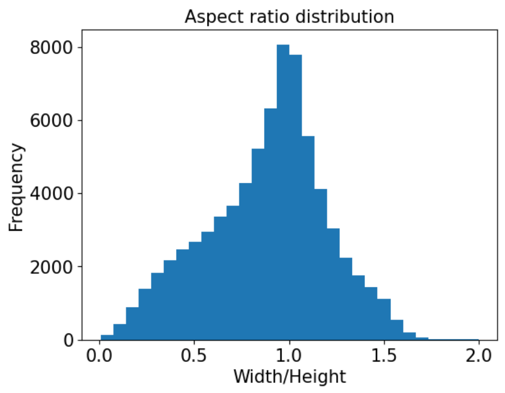
- Area and Aspect Ratio distributions, shown in Fig. 3.13. In this case, the area and the AR concern the polygon formed by the four corners of the gate. Considering the former distribution, most of the images results with area values close to zero. This happens because of the distance and relative orientation of the drone concerning the gate position. This is something that should be further studied and improved. The latter distribution, instead, shows that in most images, the polygon formed by the corners is regular. Moreover, there is a remarkable number of pictures showing skewed images of the gate. These skewed perspectives are crucial for gate detection when the camera is not positioned directly in front of it.

- The corners' distribution on the image in Fig. 3.14. These images show that each corner is well distributed in the image space. Some areas are not explored at all, as all corners are required to be within the image.

- The drone's pose distribution in the 3D space around the gate in Fig. 3.15. The plots show the space distribution of the drone's position. The $X - Y$ plane provides the space in which the drone is spawned. The



(a) Area distribution



(b) Aspect Ratio distribution

Figure 3.13: Distribution of the area and the AR.

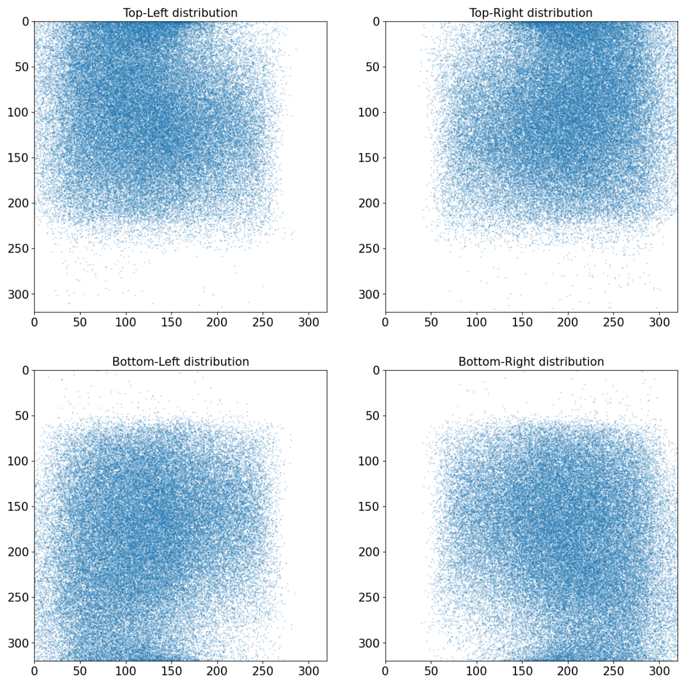


Figure 3.14: Corners' distribution

radius limits are [0.3,2.5] meters. An oversampling of the area near the gate was required to increase the amount of images close to the gate. In the $X - Z$ and $Y - Z$ planes, the hourglass form in the centre is remarkable due to the height, pitch, and yaw selection to have all the corners falling inside the image.

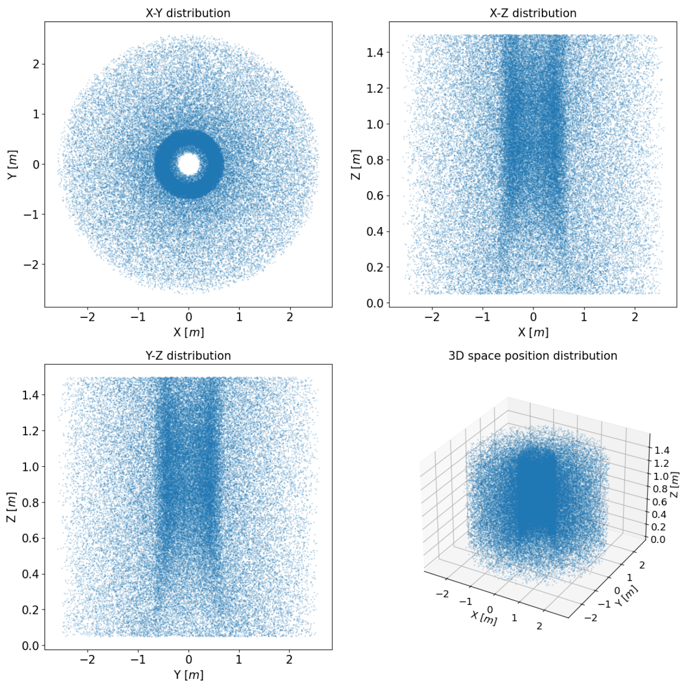


Figure 3.15: Drone position distribution

In total, three datasets of 320×320 grayscale images were collected. The train one is collected as described above. The test and the validation ones are collected in the empty world used to test the CV-based pipeline. The test one is collected without changing anything but the gate and drone poses. Such a test set is labelled as *Simple*. What differs from the test and the validation is the colour of the background, the floor, the gate, and the lightning. In any case, in the validation set, there are no images with the same configuration as the test set. Table 3.3 shows the number of images per dataset, and Figure 3.16 shows images from the training, the validation, and the testing set.

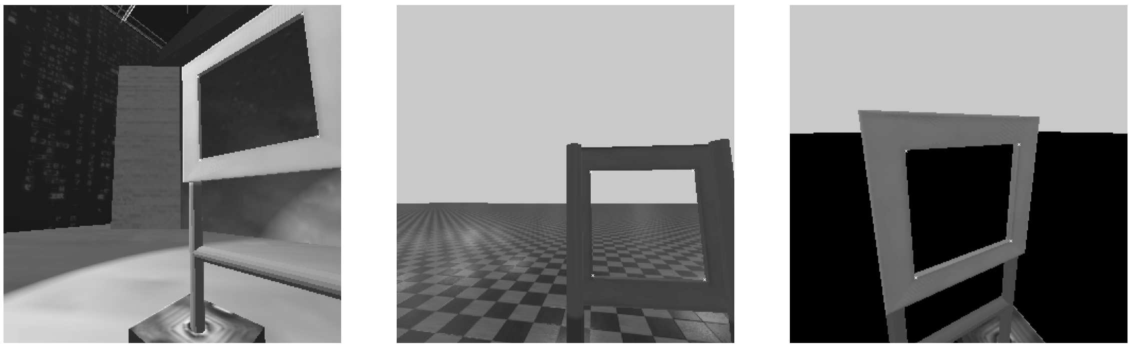

(a) Train (b) Validation (c) Test

Figure 3.16: Example images of train, validation, and test dataset. The (c) reports an image from the *Simple* test set.

Dataset	**Number of images**
Training	75k
Validation	10k
Test	7.5k

Table 3.3: Number of images per dataset.

To take a step towards testing the network in the real world, the *Laboratory* test set was collected. Featuring never-before-seen real images taken at the IDSIA Robotics Laboratory in Lugano and taking advantage of the four wall panels, the IDSIA Robotics Laboratory was recreated in Webots. Fig. 3.17 shows a representative example.

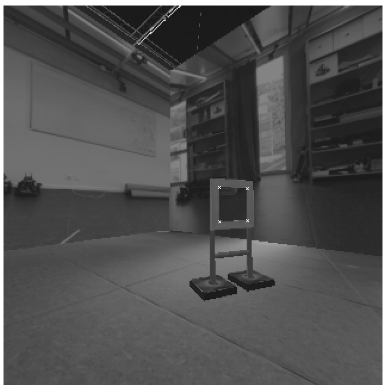


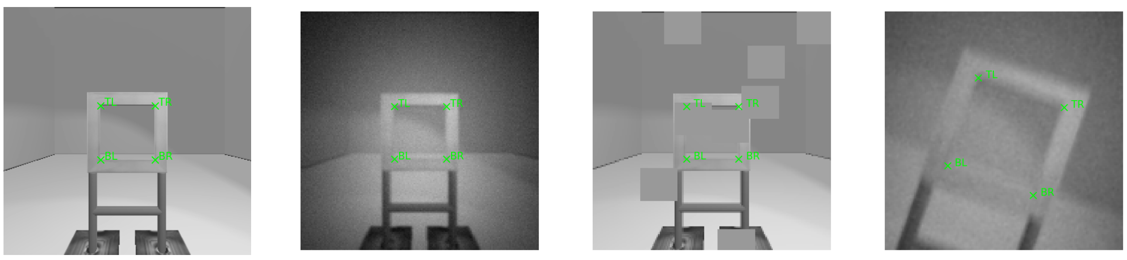
Figure 3.17: Example image from the *Laboratory* test set.

3.2.2 Data Augmentation

Data augmentation is a good technique to increase the amount of available training data. Starting from the simulator images, some photometric and geometric transformations are applied. With the goal of testing on in-field data, some photometric transformations are used to mimic the real behaviour of the Himax HM01B0, the available camera on the AI deck (see Sec. 2.1.1). This augmentation pipeline, named *Himax augmentation*, is achieved with the following transformations:

- Motion blur
- Gaussian blur
- Gaussian noise
- Vignetting
- Exposure change

Other adopted augmentations are the horizontal random flip, changes in scale, rotations, shift, shearing (geometry distortion), and perspective. Moreover, random erasure [16] (also known as cutout) was used to make the network more robust. This augmentation consists of covering parts of the image with random patches of plain colour. From the original training dataset composed of $75k$ images, each image was ten times, resulting in a dataset of 750k samples. Some examples of augmented images are shown in Fig. 3.18.



(a) (b) (c) (d)

Figure 3.18: Some augmentation examples. (a) shows the initial image, (b) the augmented image after Himax augmentation, (c) random erasure, (d) Himax, rotation, and scale-increasing augmentation.

In applying all those augmentations, it's fundamental to take into account the corners' coordinates. As long as the augmentations only concern photometric changings, corners' coordinates are not affected. Problems arise with geometric transformations like horizontal flip, scale, rotation and so on. All geometric augmentations, but horizontal random flip, are resolved by the *Albumentations* library [24]. By supplying corners' coordinates as key points along with their label, such a library applies the same transformations to the key points. It also supports the horizontal random flip transformation, but in this case, the transformation to the key points should not be applied because the aim is to have a detection invariant to the position of the gate. Whether the gate is seen from the front or the back, no distinction is made. Hence, after the random horizontal flip transformation, a relabelling of the corners is needed.

As said above, for each collected image in the simulator, ten augmented images are produced. Each augmentation is randomly performed. In this way, the resulting dataset has both images simulator-like and others that mimic the actual camera. Augmentations are performed in the following way:

1. Himax augmentation
2. Horizontal flip
3. Scale
4. Rotate
5. Shift
6. Shear
7. Perspective
8. Random erasure

During the augmentation process, some of the features may fall outside the image. In this case, augmented images are generated until ten proper images are produced, namely with all the features fitting the image.

3.3 Simulation

This section is devoted to describing the methods adopted in the simulation regarding the fusion of the corner detection module, both CV-based and DL-based, with the visual servoing one.

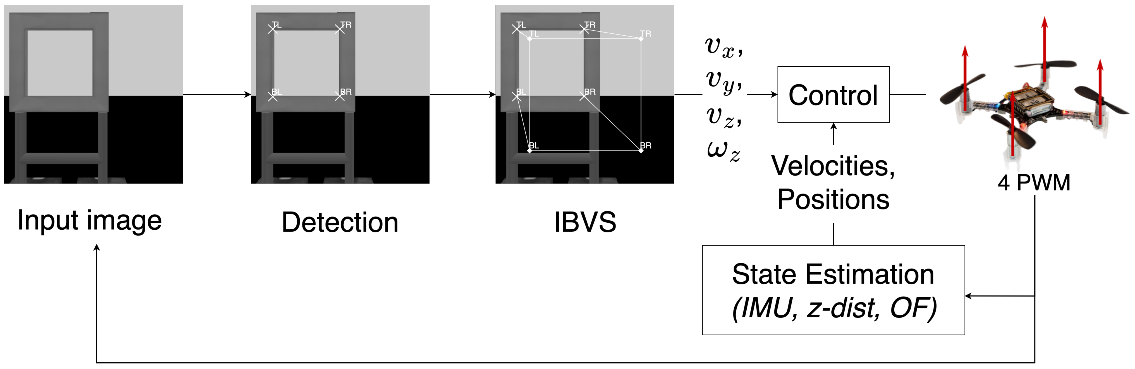
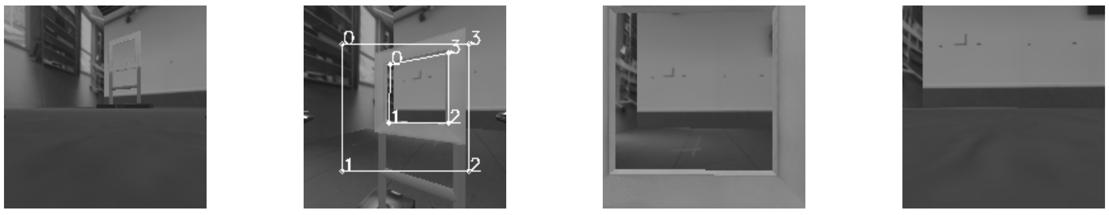


Figure 3.19: Full pipeline

Figure 3.19 reports the overall working pipeline. Given an image, the detection module predicts eight scalars, namely the image coordinates of the gate's four corners: *TL, BL, BR, TR*. Such features feed the visual servoing module that outputs four velocities to move the drone in the desired position. These velocities are then sent to the stock controller of the Crazyflie 2.1 that outputs four PWM to the motors to move the drone. Inputs to the control module are the velocities and positions computed by the state estimation module. In the real drone, it is computed by an EKF using the IMU and the Flow deck measurements. The Flow deck measures the optical flow and the altitude measurement thanks to the ToF. In the simulator, GPS and IMU measurements of the drone are retrieved to compute its velocity and pose.

To test the whole pipeline, composed of perception and control, a flight task to pass through the gate is defined as the following:



(a) (b) (c) (d)

Figure 3.20: Visualization of the flight task in the Laboratory world (see Sec. 4.2) using the DL-based detection module. Take-off (a), gate-based navigation (b), pass through the gate (c), landing (d).

1. Take-off
2. IBVS

3. Pass through the gate
4. Landing

An example is reported in Fig. 3.20.

More in detail, start the drone in a position such that it can see the gate and perform the take-off step (Fig. 3.20 (a)). It doesn't involve any perception pipeline yet. The height desired is provided as an altitude setpoint. Following such a setpoint, the drone reaches the desired height, then the IBVS task, namely the *gate-based navigation* (Fig. 3.20 (b)) is triggered. During this step, the drone will move towards the gate thanks to the features from the detector and supplied to the IBVS. By matching the features with the desired ones, the drone will move in the 3D space ending in front of the gate. Hence, a state machine sets a forward velocity command of 0.1 m/s for 10 s, resulting in a forward motion of one meter (Fig. 3.20 (c)). Landing is the last step and is achieved by supplying an altitude setpoint near zero and then switching off the motors (Fig. 3.20).

Both pipelines were tested on images without temporal consistency, i.e., provided images do not show consecutive moments. When used in the simulation with the visual servoing module, both corner detection modules showed a noisy behaviour. This led to the failure of the drone flight, but the noisy behaviour was not the only source of such a failure. More in detail, noisy predictions lead to different velocity commands to the controller, resulting in unstable movements. Another problem concerns the output velocity commands of the IBVS module. This is associated with the IBVS error $\boldsymbol{e}$ of the first iterations. To give a scalar value to such an error, defined in Eq. 2.1, the Frobenius norm [7] is applied to measure the magnitude or size of the error matrix. It is the square root of the sum of the squares of each element of the matrix. Namely, given the error matrix $\mathbf{e} \in \mathbb{R}^{m \times n}$, the Frobenius norm of such a matrix is the following:

$$||\mathbf{e}||_F = \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{n} |e_{i,j}|^2} \tag{3.3}$$

From now on, the referred IBVS error is actually its Frobenius norm.
As stated above, the first iterations of the gate-based navigation are the toughest due to large errors resulting in larger velocity commands in output. To deal with it, we implemented the *smooth start* to increase the error exponentially [33]. Hence, the image-plane error 2.1 is replaced by:

$$\boldsymbol{e}' = \boldsymbol{e} - \boldsymbol{e}_0 e^{-\mu t} \tag{3.4}$$

Where $\boldsymbol{e}_0$ is the error $\boldsymbol{e}$ at $t = 0$. This adjustment mitigated the problem. The drone dynamic was still very noisy. To further alleviate the problem, *exponential smoothing* was applied to both velocities commands and corner extraction pipelines. The implemented exponential smoothing is the simplest:

$$\begin{cases} s_0 = x_0 \\ s_t = \alpha x_t + (1-\alpha)x_{t-1}, \quad t > 0 \end{cases} \tag{3.5}$$

Where s_t is the smoothed output at timestep t and $\alpha \in [0,1]$ is the smoothing factor. By varying α, past states are weighted more or less. A manual tuning was performed for both detections and velocities, resulting in $\alpha = 0.9$ for both. Lower values of α led the drone to fail the gate-based navigation. These expedients allowed the drone to fly.

Unfortunately, the more the drone is near the gate, the more the noise increases (remarkably in the CV-based one), making it impossible to reach a very low IBVS error. An error threshold is empirically set to be 15.6% of the image size. In the 320×320 case, it corresponds to 50 pixels and halved in the 160×160. When the IBVS error reaches such a threshold, the drone is already in front of the gate, and the pocket-size dimension of the drone allows it to pass through. Moreover, to make sure the error is below such a threshold, the first time the error reaches the threshold, errors start to be collected for three seconds. Then, the median of the errors is computed and if the median value is below the threshold, the drone switches to the 'Pass through the gate' task. Otherwise, the error collection and median value computation don't stop until the median value of the last three seconds is below the threshold presented above.

3.4 In-Field dataset collection

The ultimate goal is to test both pipelines in the real world. Hence, in the IDSIA Robotics Laboratory of Lugano, a dataset of real images and poses using a Crazyflie 2.1 nano-drone was collected. The images were streamed via Wi-Fi to a workstation and collected along with the drone and the corners' poses with ROS. All the poses were collected with the OptiTrack motion capture system. For the event, a black gate was built. The drone was controlled by an operator who made it fly near the gate for three minutes straight. With a camera frequency of 20 Hz, a total of 4k 160×160 frames were collected. Almost half of the frames were discarded because of tracking errors in the corners. Moreover, most of the time, the gate was in the centre of the camera. Fig. 3.21 reports an image from such a dataset. This test set is labelled as *Real*.

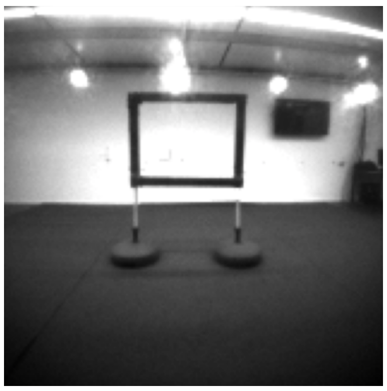


Figure 3.21: Example image from the *Real* test set.

Chapter 4

Results

This chapter reports and presents the results obtained.

Starting with the setup, the Python version used is the 3.8 with the following libraries and versions: albumentations 1.3.1, imgaug 0.4.0, machinevision-toolbox-python 0.9.6, numpy 1.24.4, opencv-python 4.8.0.74, pandas 1.2.4, scikit-learn 0.24.1, scipy 1.10.1, torch 1.4.0, torchsummary 1.5.1, torchvision 0.5.0. Concerning the simulator, the Webots version is the 2023a.

A presentation of the overall working pipeline behind the drone's control is conducted. Sec. 4.1 presents the performance of the detection modules on three test sets: the *Simple*, the *Laboratory* and the *Real* presented in Sec. 3.2.1 and Sec. 3.4.
Finally, in Section 4.2, the results of the simulations are presented, evaluating the entire pipeline that incorporates the joint utilization of both the detection and visual servoing modules.

4.1 Detection

Detection modules are tested on the test sets presented in Sec. 3.2.1 and Sec. 3.4, where two are composed of synthetic images and one by real ones. In addition to the CV and DL, a *Dummy* classifier is defined to always output the mean position of the corners of the testing dataset at hand.

At first, the CV-based module was tuned to work on the Simple dataset, resulting in the working setup of parameters reported in Tab. 4.1. Then, such a module was tuned to work with the *Laboratory* and the *Real* dataset. The label working setup denotes a set of parameters used to accomplish the flight task. Mainly, the kernel size of the Gaussian blur and the thresholds of the Canny detector were touched to retrieve proper edges. To work on

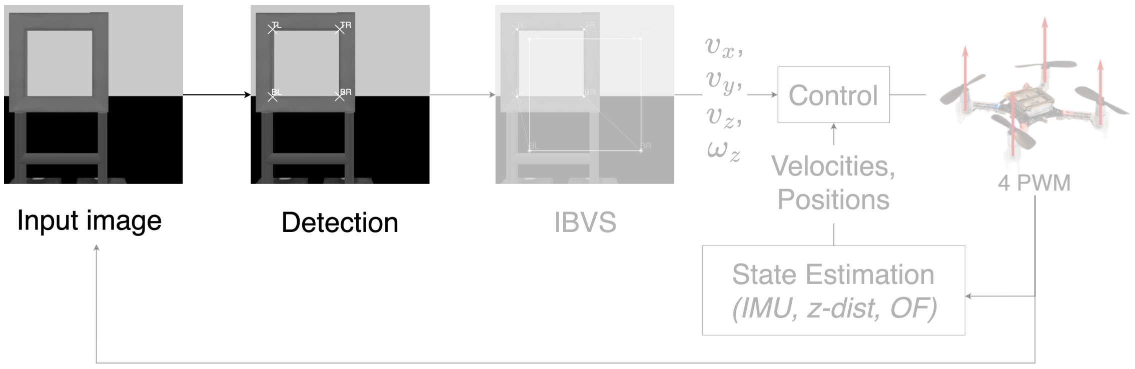

TL
TR
BL
BR
Input image
Detection
IBVS
v_x, v_y, v_z, ω_z
Control
Velocities, Positions
4 PWM
State Estimation (IMU, z-dist, OF)

Figure 4.1: Feature extraction modules.

the *Simple* dataset, the AR interval was set to [0.6,1.25] to recognise skewed figures. In the *Simple* world, the drone was spawned around the gate on a spherical shell to collect a dataset of gate images from different angles to perform a qualitative study to choose the low threshold. Different thresholds were used, starting from 0.8 and lowering it by 0.1 each time until reaching 0.1. It resulted that at 0.6, the module detects the gate from a relative angle of almost ±50°. Going further is not required, and in the same conditions, lowering the threshold to 0.1 increases the computation time up to 1.34 times the one having 0.6 as the lower threshold. Figure 4.2 shows an example image in which a low threshold of 0.7 fails while succeeding at 0.6.

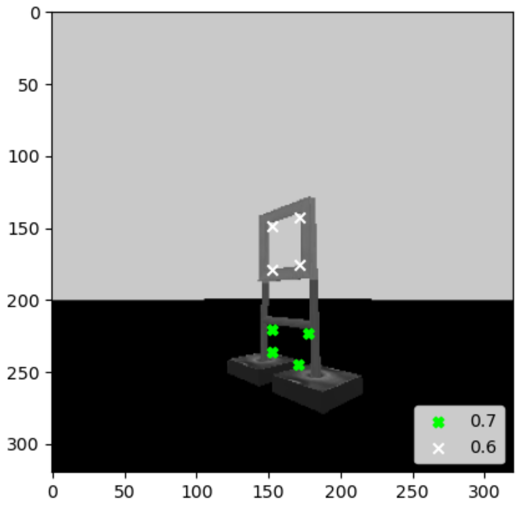


Figure 4.2: AR interval comparison in extracting corners with a low threshold of 0.7 in lime and a low threshold of 0.6 in white.

The other parameters of the CV-based detector were slightly changed to deal with the increment of the textures in the *Laboratory* and *Real* datasets.

Dataset	Blur	Canny	AR	Harris	Heuristic
Simple	7×7	$100, 200$	0.6, 1.25	$W = 10 \times 10$, Sobel k. $= 3 \times 3$, $k = 0.1$	Highest
Laboratory	3×3	$50, 150$	0.6, 1.25	$W = 10 \times 10$, Sobel k. $= 3 \times 3$, $k = 0.1$	Highest
Real	7×7	$100, 200$	0.8, 1.25	$W = 10 \times 10$, Sobel k. $= 3 \times 3$, $k = 0.1$	Central point

Table 4.1: CV-based module configurations for each dataset.

Table 4.1 reports the configuration used for the CV-based module in each dataset. The table reports the kernel shapes and the various thresholds adopted. Concerning the Harris column, such a table reports the shape of the window, the kernel of the Sobel operator, and the parameter k used to compute the Harris score.

Concerning the other module, the training procedure is presented in Sec. 3.2.

Full dataset			Samples with AR $\in$ [0.6,1.25]		
Detector	**Norm MAE**	**% Fail**	**Detector**	**Norm MAE**	**% Fail**
Dummy	0.159	0	Dummy	0.156	0
CV	0.095	16.52	CV	0.077	6.4
DL	0.053	0	DL	0.047	0

Table 4.2: *Simple* dataset. Left-most table, the whole dataset of 7.5k images. The right-most table shows the left-most table depurated from samples with AR outside of the range. Only 5030 samples were left.

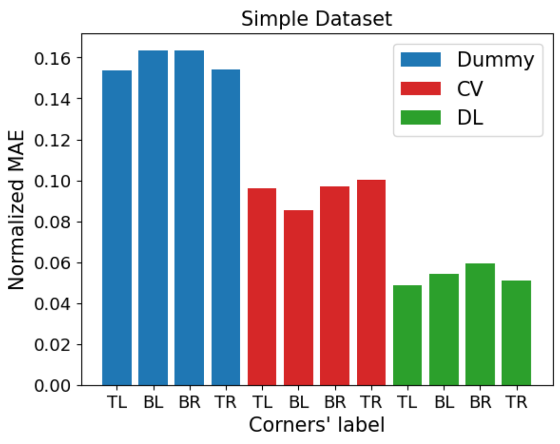
The results on the *Simple* dataset of the three detection modules are reported in Table 4.2. On the left, the whole dataset is considered, whilst on the right, it is reduced to the samples with an AR that matches the one adopted in the CV-based module. In the tables, the **% Fail** column reports the percentage of samples where no corners were detected. Since the CV-based detector and the DL-based one work on images of different shapes, the *normalized* MAE of the four corners is considered. The normalization concerns the MAE with respect to the image size, meaning that the error is divided by the image size, resulting in a pure number. The DL-based module is the best-performing one with the lowest MAE and zero failures because it always outputs some coordinates. This is because the training dataset contains only images with corners and none without. Hence, the network cannot learn to output null values. It will always output the eight coordinates, even if the gate is not seen. Instead, the CV module fails the detection in 16.52% of the cases. Such errors are due to the edge and contour extraction but also to the discarded contours that fall outside the considered AR interval. This leads us to analyse the aforementioned smaller part of the dataset, where the results are shown on the right portion of Table 4.2. Here, the percentage of failed detections of the CV-based module decreases substantially along with its MAE, which is almost 20% lower. A slight MAE reduction of both the Dummy and the DL-based detector can be noticed. Considering the normalized MAE, the CV module is 0.6 times lower than the Dummy counterpart and almost twice the DL detector. In this case, the Dummy is the worst, whilst the DL module is the best. Such a module achieves an error 0.3 times the Dummy one and 0.56 times the CV-based detector counterparts.

Considering the full dataset, Fig. 4.3 (a) shows the normalized MAE per each corner and reports in (b) the normalized MAE distribution with respect to the area and in (c) with respect to the AR of the three detectors in the *Simple* dataset. It can be noticed that the CV-based module achieves its best performance at great areas, namely near the gate, and in the given AR interval.

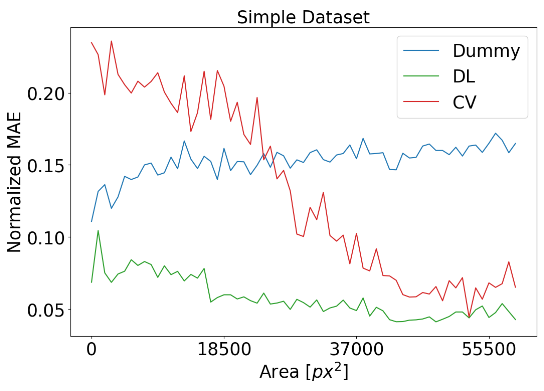
Full dataset			Samples with AR $\in$ [0.6,1.25]		
Detector	**Norm MAE**	**% Fail**	**Detector**	**Norm MAE**	**% Fail**
Dummy	0.159	0	Dummy	0.156	0
CV	0.237	2.57	CV	0.235	2.39
DL	0.055	0	DL	0.049	0

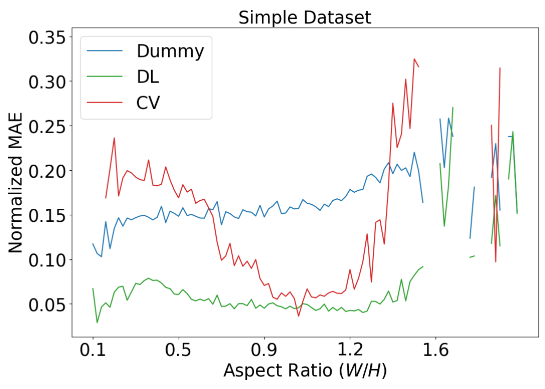
Table 4.3: *Laboratory* dataset.

Table 4.3 reports the results of the *Laboratory* test set. Now, the MAE of the CV module is pretty high in both cases. This shows the limit of the CV-based module. In the *Simple* dataset, the world featured only the gate and a plain background, whilst, in *Laboratory* one, the number of features increased, hindering the detection. In a textured environment, the abundance



(a)

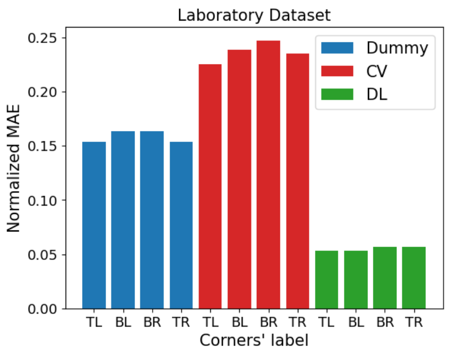




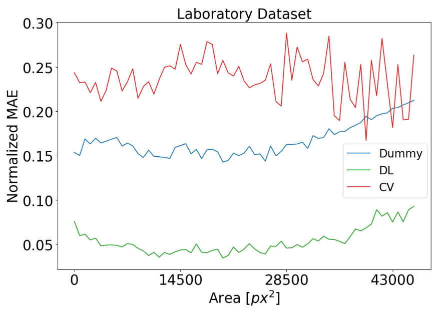
(b) (c)

Figure 4.3: *Simple* dataset. (a) shows the normalized MAE performance concerning each corner. Figures (b) and (c) show the same metric with respect to area and aspect ratio.

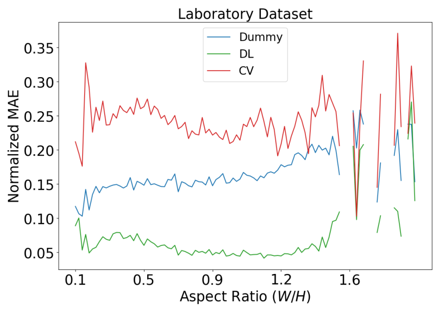
of viable contours results in a low failure rate, even across the entire dataset. The MAE of the Dummy predictor in the *Simple* and the *Laboratory* dataset are exactly alike. This is because the dataset collection is deterministic and was performed in the same fashion. Hence, the corner distribution is equivalent because of the spawning pattern of the drone and gate orientation. Referring to the errors, the CV module with 1.5 and 4.3 times the Dummy, and the DL errors, results being the worst one. Once again, the DL module is the best, with an error of 0.35 times the Dummy and 0.23 with the CV detector.



(a)



(b)



(c)

Figure 4.4: *Laboratory* dataset. (a) shows the normalized MAE performance concerning each corner. Figures (b) and (c) show the same metric with respect to area and aspect ratio.

Full dataset			Samples with AR ∈ [0.8,1.5]		
Detector	**Norm MAE**	**% Fail**	**Detector**	**Norm MAE**	**% Fail**
Dummy	0.073	0	Dummy	0.068	0
CV	0.185	2.14	CV	0.182	1.41
DL	0.058	0	DL	0.056	0

Table 4.4: *Real* dataset. The full dataset consists of 1867 samples, while the reduced version has 1559 samples.

Results on the *Real* dataset are shown in Table 4.4. Since the gate is rectangular, a change in the AR interval of the CV module to [0.8,1.5], as stated in Table 4.1, was conducted. Even in this case, the reduced dataset shows only a slight improvement for the CV module. The Dummy detector performs pretty well compared to the previous tests. The reason behind this is the distribution of the corners in the dataset. This is due to the fact that

most of the time, the drone was facing the centre of the gate. Figure 4.5 shows the gate centre distribution.
Even in this case, the winning detector is the DL-based one, with an error of 20% lower than the dummy, and of almost 70% lower than the CV one. Considering the CV module, its performance, with an error of 0.185, is slightly better than the one of the *Laboratory* dataset.

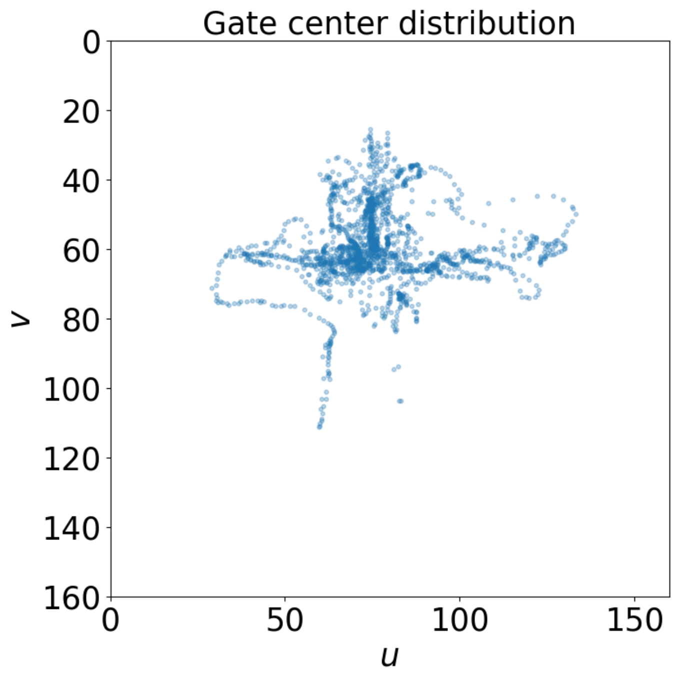
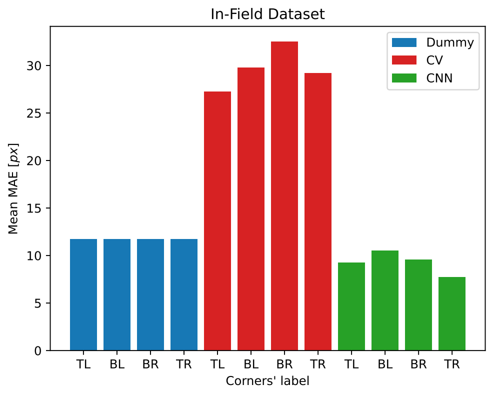
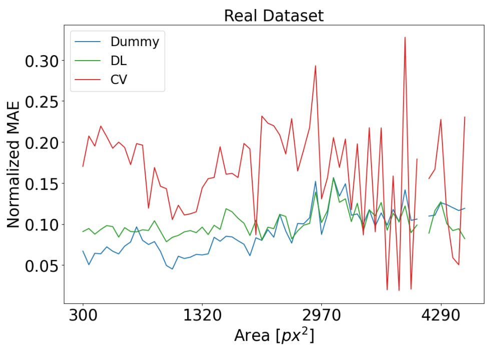
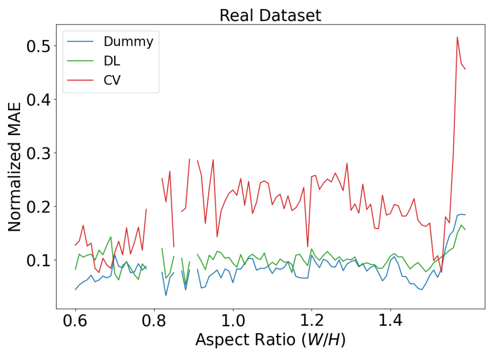


Figure 4.5: *Real* dataset. Gate center distribution with a mean value $(\mu_u, \mu_v) = (76, 59)$ and standard deviation $(\sigma_u, \sigma_v) = (17,12)$.



(a)





(b) (c)

Figure 4.6: *Real* dataset. (a) shows the normalized MAE performance concerning each corner. Figures (b) and (c) show the same metric with respect to area and aspect ratio.

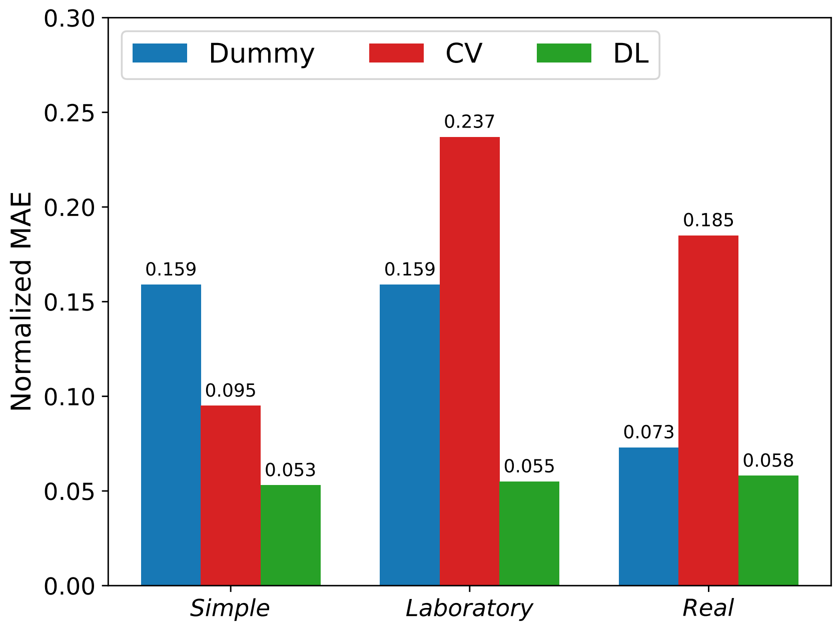


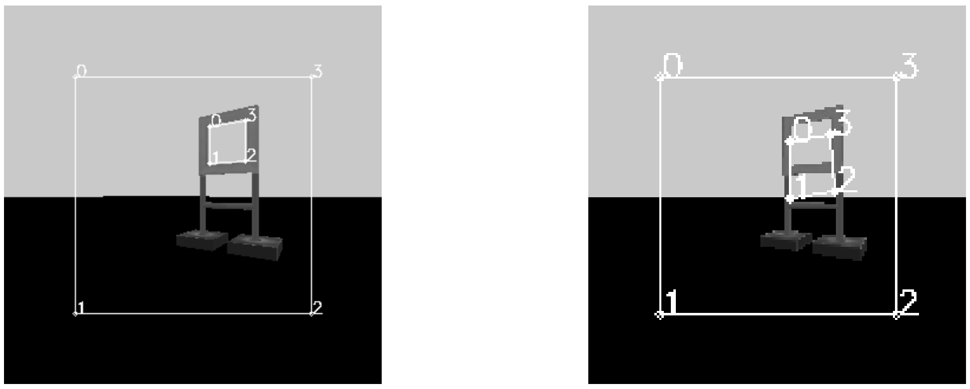
Figure 4.7: Normalized mean MAE in each dataset for every detector.

To conclude and visualize the performance of each detector in the datasets, Fig. 4.7 reports the results of Tab. 4.2, Tab. 4.3, and Tab. 4.4. In each case, the DL module outperforms the others except for the *Real* case because of the dataset collection. The CV module suffers environments rich in textures, leading to poor performance in the *Laboratory* and the *Real* test sets where the MAE is respectively 4.3 and 1.97 times DL counterpart. Thanks to the data augmentation, the NN manages to achieve a lower error on the *Real* dataset, while it has never seen real images during the training phase. The network reaches an error of 0.071 without data augmentation, while the error lowers to 0.058 with the network trained with augmentations, reducing the error by almost 20%.

4.2 Simulation

For what concerns the simulation part, a test of the whole pipeline reported in Fig. 3.19 on the flight task presented in Sec. 3.3 is conducted. The drone starts with the take-off, then detection and IBVS take over in the gate-based navigation, and once the IBVS error is below the threshold, the drone crosses the gate and finally lands.

Both predictions of the detection modules are filtered by exploiting the exponential smoothing introduced in Sec. 3.3 with a $\alpha = 0.9$. Such a high value weighs current prediction very low with respect to previous predictions, leading to a smoothed prediction.



(a) 320×320 image, CV-based detector (b) 160×160 image, DL-based detector

Figure 4.8: The corners of the square are the IBVS goal positions

In the IBVS, the goal positions in the image space are the corners of a centred 200 pixels wide square in the case of the 320×320 image size and 100 pixels in the 160×160 case. This means that at convergence, the drone will be at the same height as the gate centre. Figure 4.8 shows the goal positions (the corners of the big square) along with the output of the detectors (the corners of the small polygon).
The Z parameter of Eq. 2.7 is set to be 0.34 m and was obtained by exploiting the camera projection of the position of the corners, namely the ground truths (GTs), at convergence of IBVS where GTs match the goal positions and $\boldsymbol{e} = 0$. Following [33], the λ of Eq. 2.4 was set to 0.08. In Table 4.5, all the parameters used in Eq. 2.11 for the 320×320 case are reported. In the 160×160 case, the $\sigma_{w,h}$ is doubled.

f [mm]	$\sigma_{h,w}$ [μm]	Z [m]
0.6	3.6	0.34

Table 4.5: IBVS parameters of Eq. 2.11 in the 320×320 image case.

Concerning the control part of Fig. 3.19, the IBVS output velocities were filtered to avoid turbulent flight behaviour. The same filtering adopted for the predictions was applied here, the exponential smoothing, using $\alpha = 0.9$.

To test the pipeline, three different worlds of increasing difficulty were used.

The first two are those in which the *Simple* and the *Laboratory* test sets were collected. While the first features only the gate with a light blue background and black floor, the other presents the gate surrounded by four walls showing real images of the IDSIA Robotics Laboratory. These worlds are referred to by the names of the correspondent testing sets. The third world, labelled as *Competition*, imitates the arena used in the IMAV 2022 nano-quadcopter competition. In this arena, there are 3D objects and not only images or a plain background, as in the other two cases. This is the most challenging condition. Figure 4.9 shows the Crazyflie in a white circle and the gate in the worlds.

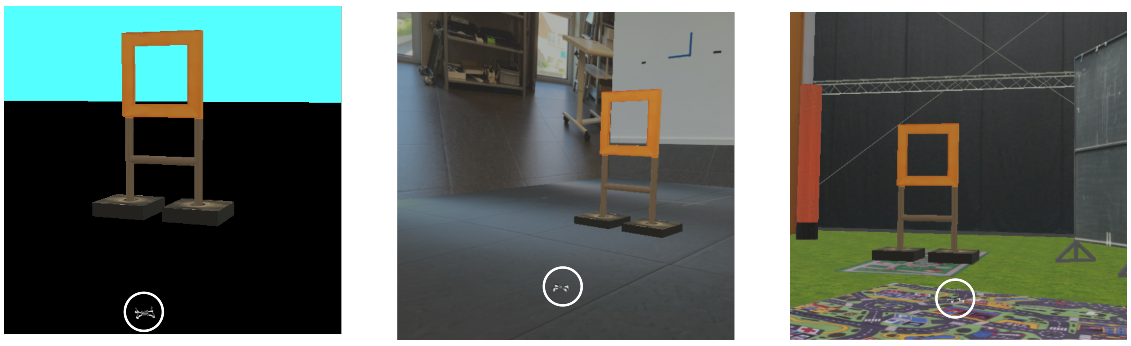

(a) Simple (b) Laboratory (c) Competition

Figure 4.9: Simulation worlds

Ten runs, with the drone's position relative to the gate kept constant, are conducted in every world. The drone is spawned 2 m away from the gate with an orientation angle θ of 30° relative to the gate. The bottom corners' height is 0.8 m, while the top ones are at 1.2 m, and the gate centre is 1 m high. In the flight task, all the output velocities are tested. Forward and sideways velocities (v_x, v_y) are tested by shifting the drone with respect to the gate centre, thanks to the angle displacement, the yaw rate (ω_z) is tested, and finally, by setting the take-off height to 0.5 m, the drone has to reach the 1 m height of the gate centre supplying height velocity $v_z > 0$.
If the drone completes the flight task, the run is labelled as successful or failed otherwise. During the simulation, the position of the ground truths is taken into account, and if GTs fall out of the image, the gate-based navigation is taken over by the landing task, labelling the run as failed.

Out of ten runs, the ratio between successes and failures with the average completion time of the flight task is reported in Table 4.6.
The ideal case is using GTs as input to the IBVS. The trajectory is the same

World	**Detector**	$\mu_{CT} \pm \sigma_{CT}$ $[s]$	**S/F**
Simple	GT	74.8 ± 0.0	1
	CV	84.2 ± 12.7	1
	DL	72.7 ± 0.8	1
Laboratory	GT	74.8 ± 0.0	1
	CV	17.8 ± 0.6	0
	DL	74.2 ± 2.5	0.9
Competition	GT	74.8 ± 0.0	1
	CV	15.6 ± 1.4	0
	DL	82.8 ± 6.5	1

Table 4.6: For each world and each detector, the table reports the average completion time μ_{CT} with its standard deviation σ_{CT}, and the ratio between the number of successful and failed runs (column **S/F**).

in all the worlds because of the relative positioning of the drone and the gate. This is reflected by the same completion time in all the worlds. In this case, only the IBVS part is exploited. The resulting trajectory will be considered as the optimal one, and in Sec. 4.2.2, a comparison with the resulting ones of CV and DL modules will be discussed.
The CV detector succeeds only in the Simple world. With a $\sigma_{CT,CV} = 0.15\ \mu_{CT,CV}$, it shows that the detection can be imprecise sometimes but still completes the task. In the other two worlds, the detection fails, leading the drone to a position where it cannot see the gate, ending with the landing taking over.
Concerning the DL detector, it succeeds every run in each world, except the Laboratory one. In this case, it fails one run out of ten, in particular when the drone gets too near the gate without being able to correctly detect the corners. This is due to the low quantity of training images in such a condition. Being too near, the corners fall outside the image space, and the landing supplants the gate-based navigation. In the Simple world, $\sigma_{CT,DL} = 0.8$ shows the robustness in such an environment. In the Competition world, with an increasing amount of textures and 3D objects, the detection is challenged. The detection becomes more challenging with a higher time to completion reaching a $\mu_{CT,DL} = 82.8s$ and $\sigma_{CT,DL} = 6.5s$. Overall, given the same starting conditions, the DL module is able to complete the task, at least once, in every world. Moreover, such a module achieves a 100% success ratio in the Competition world.

The following sections are devoted to presenting the results of the errors computed during the gate-based navigation and resulting trajectories.

4.2.1 Errors

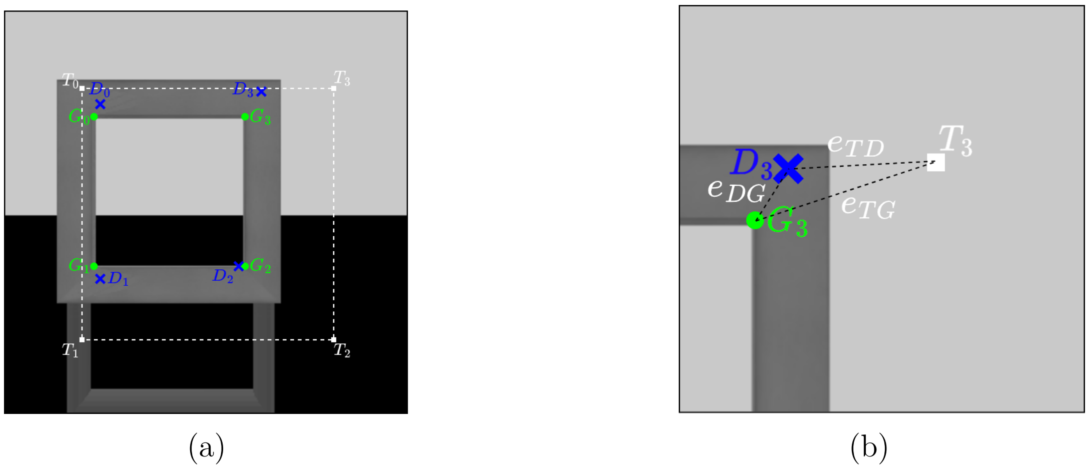


Figure 4.10: Example frame showing GTs in green, detection outputs in blue, and the features' goal position in white.

To introduce the errors, Figure 4.10 (a) shows an example of a frame taken during the gate-based navigation. Three kinds of labels are reported: white Ts, green Gs, and blue Ds. The former are the target positions of the IBVS, while the second and third are, respectively, the GTs and the detected features.

During each simulation, detected features and GTs, along with the IBVS error, are collected to compute the following errors:

- e_{DG}, namely the normalized MAE between detected features and GTs.
- e_{TD}, this is the normalized IBVS error of Eq. 2.1.
- e_{TG}, this is the normalized IBVS error using the GTs. It tells what the error would be if the detection was perfect.

Fig. 4.10 provides a visualization of such errors. To be compared, all the errors are normalized with respect to the image size.

Table 4.7 reports the values of the aforementioned errors during the last iteration of the gate-based navigation, considering only successful runs.

World	Detector	e_{DG}	e_{TD}	e_{TG}
Simple	CV	0.044 ± 0.021	0.130 ± 0.014	0.153 ± 0.029
	DL	0.031 ± 0.006	0.122 ± 0.014	0.169 ± 0.006
Laboratory	DL	0.034 ± 0.006	0.151 ± 0.017	0.214 ± 0.009
Competition	DL	0.067 ± 0.036	0.130 ± 0.015	0.185 ± 0.021

Table 4.7: Mean and standard deviation of the errors during the last iteration of the gate-based navigation. Only successful runs were considered.

In the Simple world, CV and DL modules reach comparable e_{DG}, with the former showing a high variability. Since the last IBVS iteration is taken into account, the high variability of the e_{DG} denotes how far the CV detection is from being robust, even close to the gate with an empty background. In this setting, the DL detector shows less variability. IBVS errors, e_{TD}, of both detectors are well below the requested median error threshold of 0.156 to cross the gate. While the e_{TG} of the CV-based module is still below the threshold, this is not the case for the DL one. This is due to the fact that the network tends to output bigger corners, leading to a lower e_{TD}.
Successful runs on the Laboratory and the Competition worlds were achieved only by the DL detector. On the latter, the MAE is almost double of the former, evidencing the detection hindering due to the higher world's complexity. This can be noticed by the variability rising. Concerning e_{TD} and e_{TG}, Laboratory values are higher than the competition ones. e_{TG} shows, in particular, that the drone crosses the gate before being actually in the goal position because of the network's output discussed earlier.

An example of a simulation for each world and each detector is provided in the remaining part of this section.

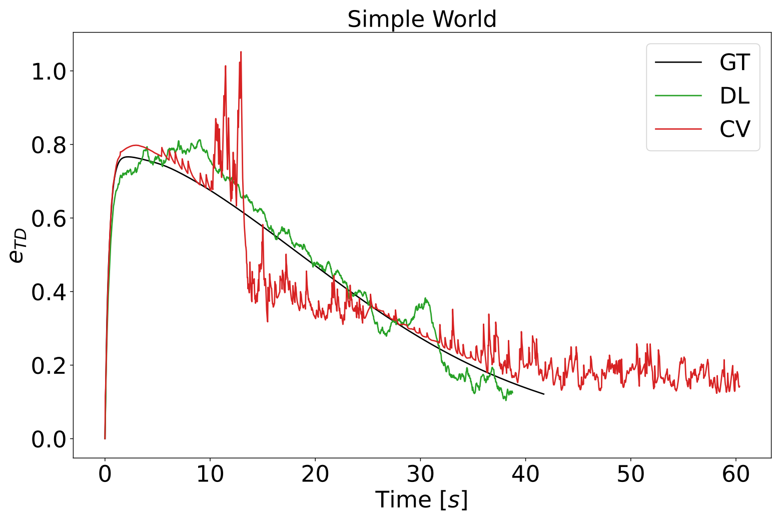


Figure 4.11: Simple world. The normalized e_{TD} for each detector.

Figure 4.11 reports the three e_{TD} errors concerning the use of GTs in black, the CV module in red, and the DL in green. The time window refers only to the gate-based navigation, meaning that it shows the time required to get closer to the gate. As shown in Tab. 4.6, DL accomplishes the flight task with the lowest time, followed by the GT and the CV module. This fits the analysis concerning e_{TG} discussed earlier. Both DL and CV IBVS errors show noisy trends. This is the direct consequence of the prediction problem.

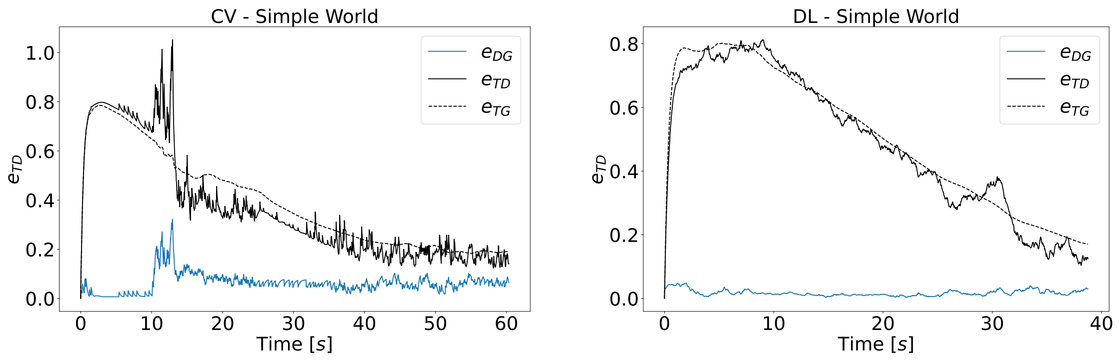


Figure 4.12: Errors distribution for each detector. CV module in (a), DL module in (b).

More in details, Fig. 4.12 show e_{DG}, e_{TD}, and e_{TG} for both detection modules. From Fig. 4.10 (b), the relation between the three errors is, at the high level, $e_{TD} = e_{DG} + e_{TG}$ explaining why the peaks in e_{DG} are reflected in e_{TD}.

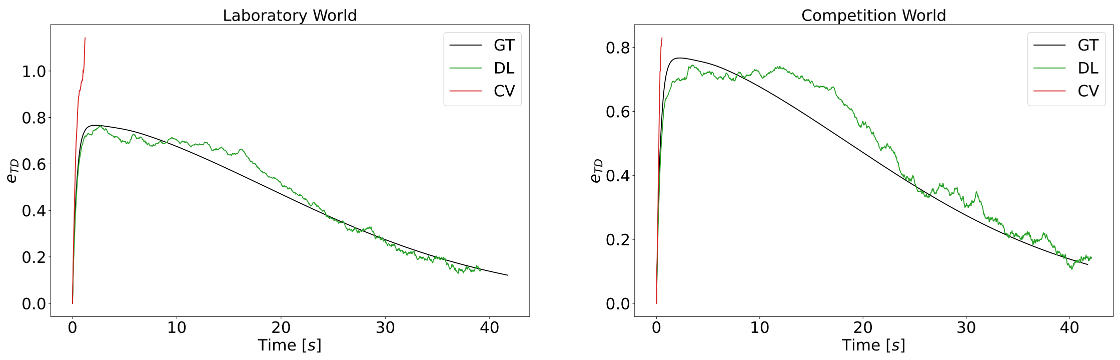


Figure 4.13: Normalized e_{TD} in the Laboratory world in (a) and in the Competition world in (b).

Likewise, Fig. 4.13 shows the e_{TD} trends and the errors of both detectors for the Laboratory and the Competition cases. In both runs, the CV module fails to detect the gate, leading to high e_{TD} errors and followed by a premature landing. Instead, the DL module completes the task in both cases, with a faster convergence in the Laboratory world than the Competition one.

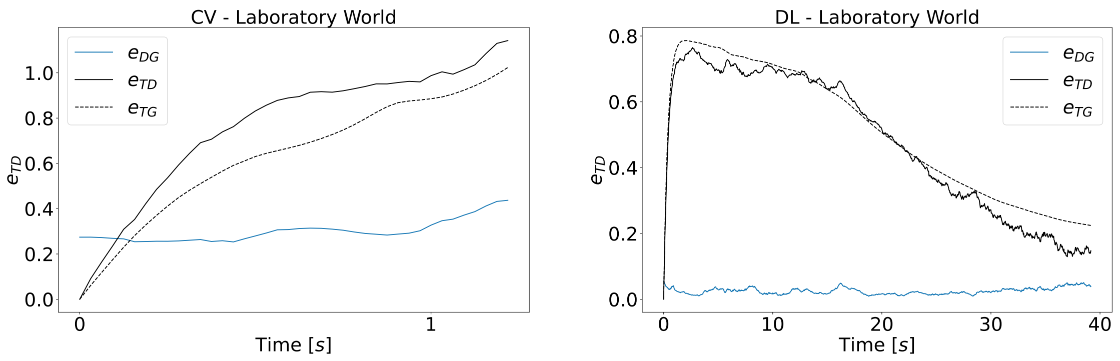


Figure 4.14: Trend of the errors in the Laboratory world.

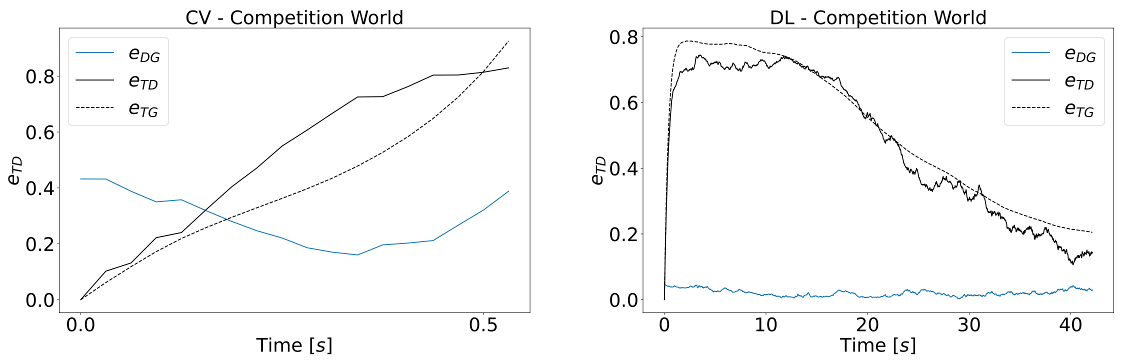


Figure 4.15: Trend of the errors in the Competition world.

Fig. 4.14 and Fig. 4.15 report the Laboratory and the Competition analogous cases of Fig. 4.12.

4.2.2 Trajectories

Within this section, a comparison of the resulting trajectories using CV and DL modules with respect to the optimal one is conducted. The optimal trajectory is defined as the resulting one using GTs as input to the IBVS module. Note that in this context, the term "optimal" serves as a label rather than an absolute definition, as there may exist alternative paths that satisfy the criteria for gate-based navigation.

Following [29], the Dynamic Time Warping (DTW) [3] score is chosen to assess the similarity between two trajectories. A trajectory is a sequence of coordinates in the 3D space, namely a time series. DTW matches the point near in the space of two time series of different lengths. The matching is performed according to the Euclidean distance.
Given two time series $X \in \mathbb{R}^{m,k}$ and $Y \in \mathbb{R}^{n,k}$, where k is the time series' number of features, and m and n denote their length. A cost matrix $C \in \mathbb{R}^{m,n}$ is computed as $C_{u,v} = f(X_u, Y_v)$, with $1 \leq u \leq m$, $1 \leq v \leq n$, and f as the Euclidean distance

$$f(X_u, Y_v) = \sqrt{\sum_{i=1}^{k} (X_{u,i} - Y_{v,i})^2} \tag{4.1}$$

In this case, each time series is a temporal sequence of made of 3D coordinates expressed in meters. Hence, the unit of measure of the Euclidean distance is meters.

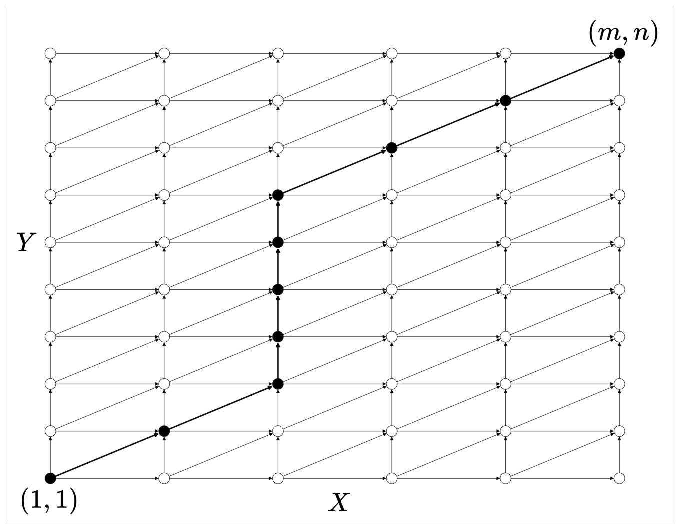


Figure 4.16: Graphical representation of DTW between two time series X and Y.

A warping path is then defined as the sequence $p = (p_1, \ldots, p_S)$ satisfying the following constraints:

- $\forall s \in \{1, \ldots, S\}$, $p_s = (u_s, v_s) \in \{1, \ldots, m\} \times \{1, \ldots, n\}$;
- $p_1 = (1,1)$ (starting point) and $p_S = (m, n)$ (finish point);
- $\forall s \in \{1, \ldots, S - 1\}$, $p_{s+1} - p_s \in \{(0,1), (1,0), (1,1)\}$

The cost associated with a warping path p is defined as the sum of the costs of the states visited by the path as

$$C_p(X, Y) = \sum_{s=1}^{S} C_{u_s,v_s} \tag{4.2}$$

Finally, the Dynamic Time Warping score is defined as the minimum cost among all the warping paths $p \in \mathcal{P}$:

$$\text{DTW}(X, Y) = \min_{p \in \mathcal{P}} C_p(X, Y) \tag{4.3}$$

A graphical representation of the warping path associated with the minimum cost and the relative DTW is presented in Figure 4.16.
The DTW score of two identical trajectories achieves the lowest value since

World	Detector	DTW score $[m]$
Simple	CV	4.26 ± 0.85
	DL	3.6 ± 0.13
Laboratory	CV	52.77 ± 1.68
	DL	15.59 ± 0.67
Competition	CV	61.31 ± 7.82
	DL	14.90 ± 0.78

Table 4.8: The table reports the mean and standard deviation of the DTW score of the ten runs performed by each detector in every world.

all the Euclidean distances are zeroed. The resulting graphical representation of the matches is a diagonal (Fig. 4.17 (a)).

For each world and detector, Table 4.8 reports the mean and standard deviation of the DTW score of the ten runs. The lower the score, the more similar the trajectory is to the optimal one. In the Simple world, both detectors accomplish the gate-based navigation task with low DTW scores, meaning trajectories are not so different. This is not true for the Laboratory and the Competition cases where the DTW scores of the CV module are the highest due to the lack of completion. Due to the challenging environments, even the DL scores are higher.

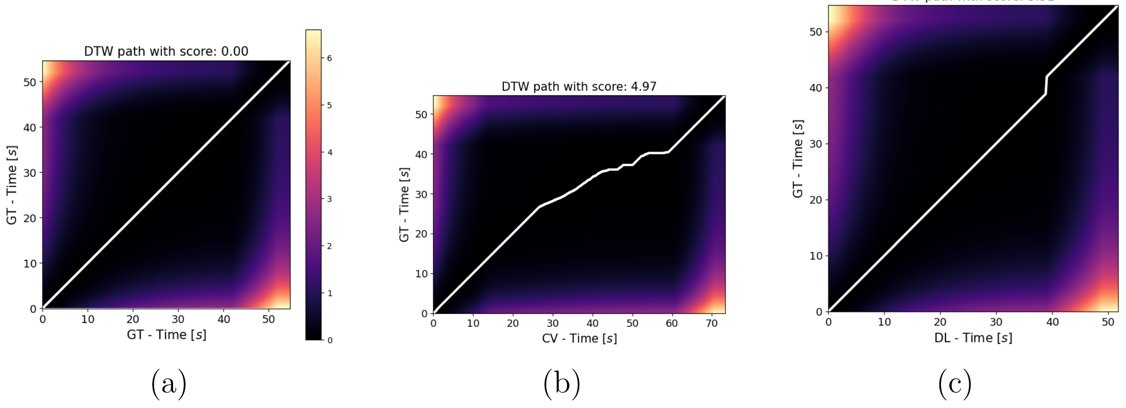
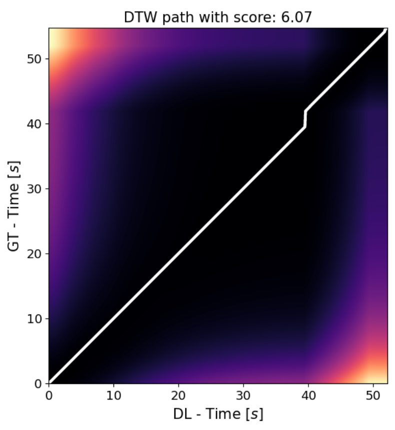
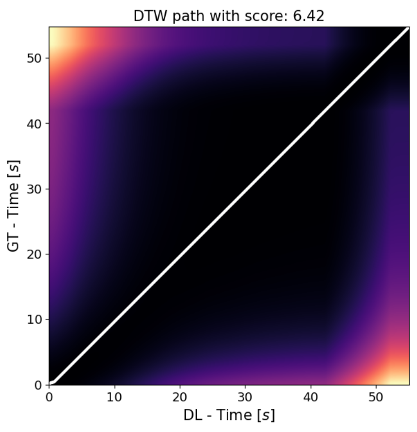


Figure 4.17: DTW scores and visualization of the path. From a run in the Simple world.

Figure 4.17 shows the DTW scores, along with the visualisation of the matching of a run in the Simple world. In (a), the DTW score of the same trajectory is shown to denote the perfect matching and the zero score.



(a)

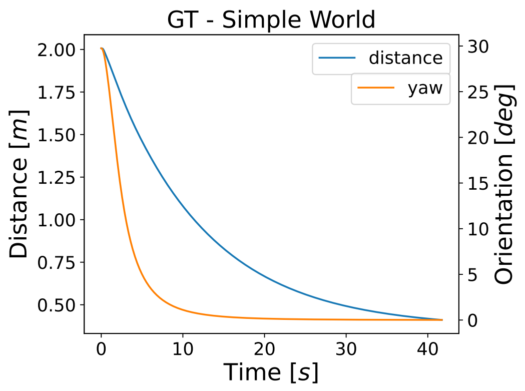


(b)

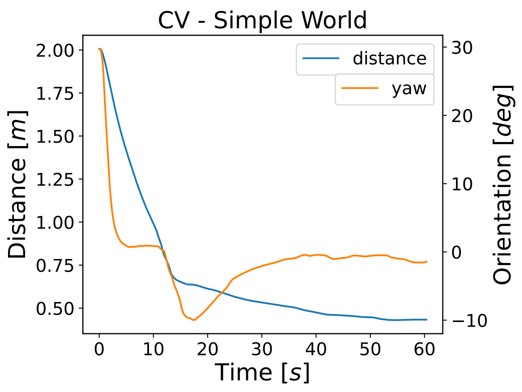
Figure 4.18: DTW scores and visualization of the path. From a run using the DL detector in the Laboratory world (a) and in the Competition world (b).

Figure 4.18 shows only the results of the DL module because both runs of the CV failed. To be exhaustive, these are reported in Fig. A.1 of the appendix.

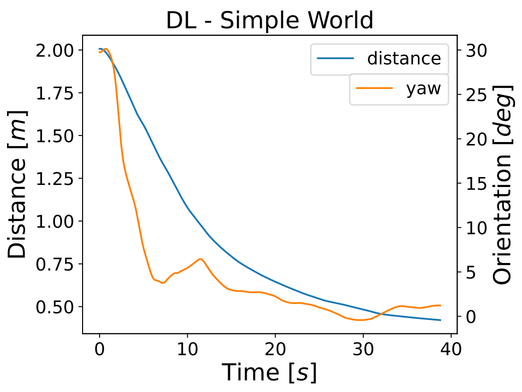
In the remainder of this section, a run trajectory is shown for each detector, along with the optimal one obtained using GTs in the IBVS. DL-based navigation completes the task in each world, while the CV one succeeds only in the simplest.
Fig. 4.19, Fig. 4.21, and Fig. 4.23 show the distance and the orientation changing of the drone through time concerning the gate pose.
Instead, Fig. 4.20, Fig. 4.22, and Fig. 4.24 show the trajectory in both 3D and 2D space.
Worth mentioning is that the GT-based navigation trajectory is identical in every world since it does not rely on images.



(a)

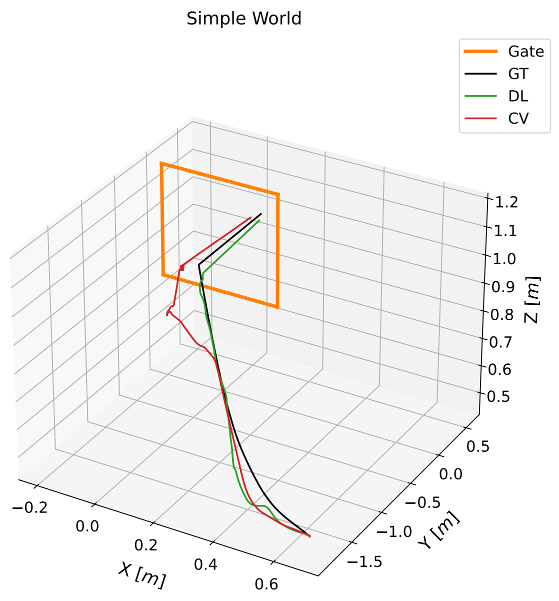


(b)

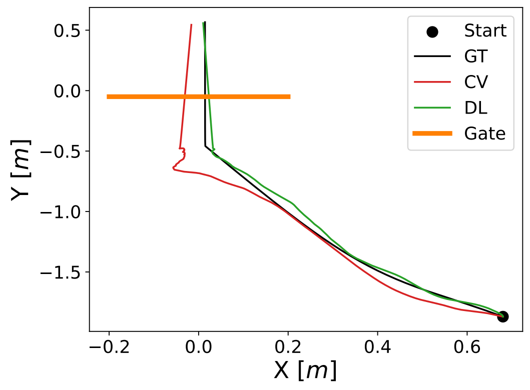


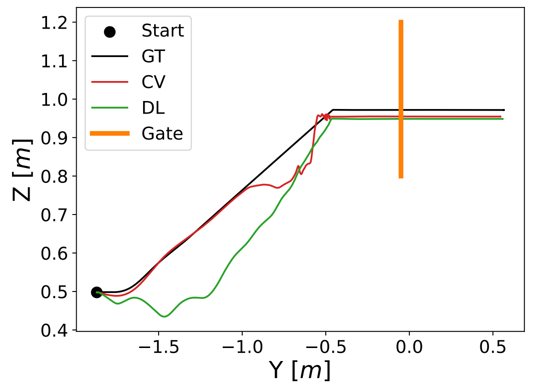
(c)

Figure 4.19: Simple world. Distance and orientation of the drone with respect to the gate.



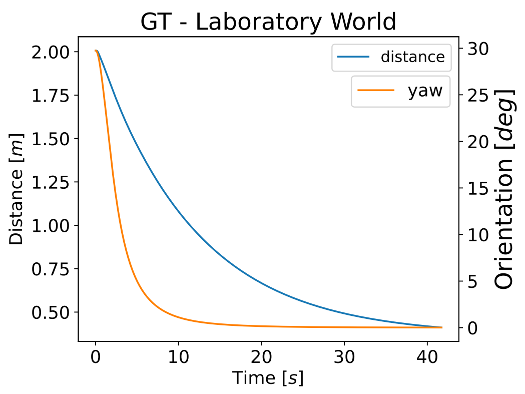
(a)



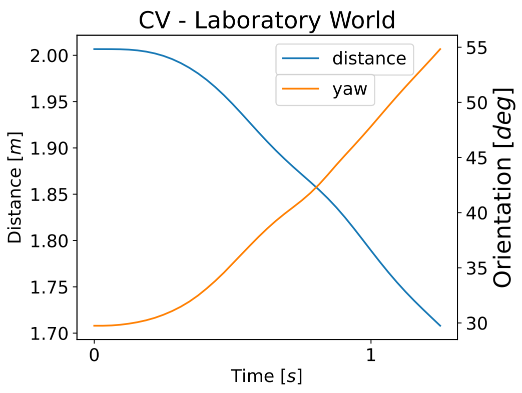


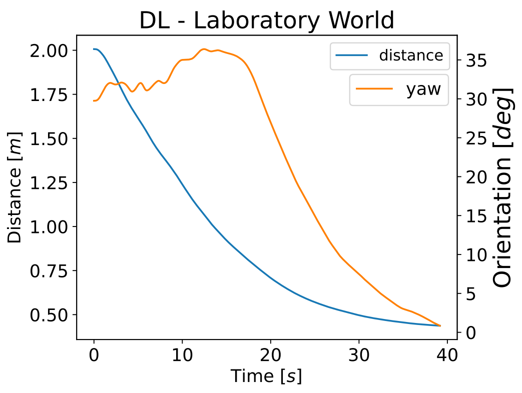
(b) (c)

Figure 4.20: Simple world. Trajectory is shown in 2D and 3D. It focuses on the gate-based navigation and crossing of the gate.



(a)

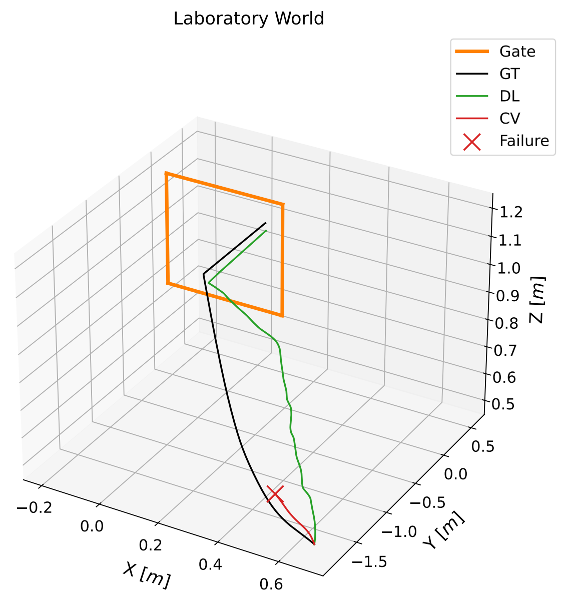




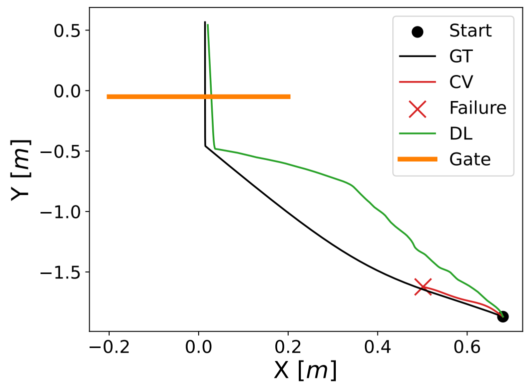
(b)

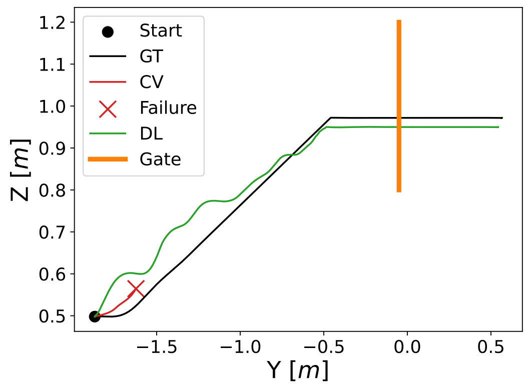
(c)

Figure 4.21: Laboratory world. Distance and orientation of the drone with respect to the gate.



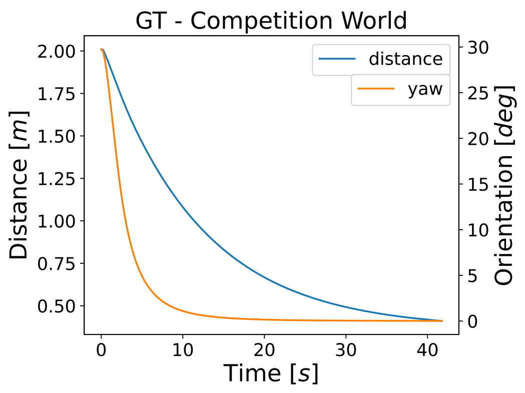
(a)



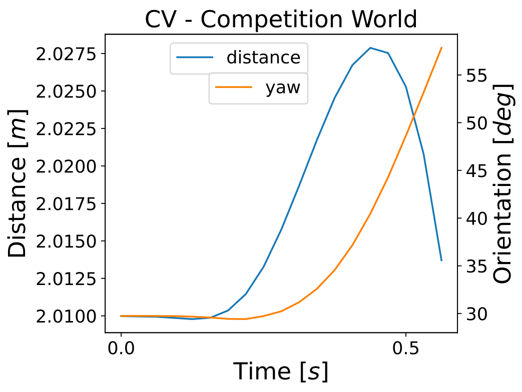


(b) (c)

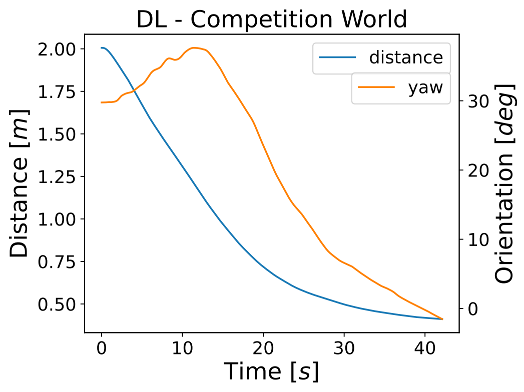
Figure 4.22: Laboratory world. Trajectory is shown in 2D and 3D. It focuses on the gate-based navigation and crossing of the gate.



(a)

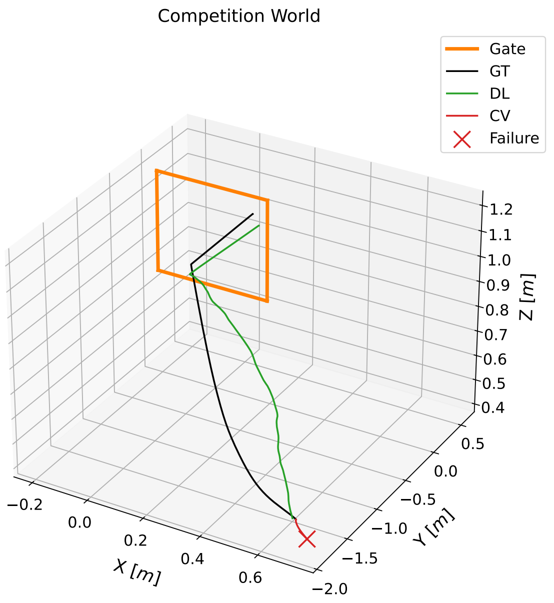


(b)

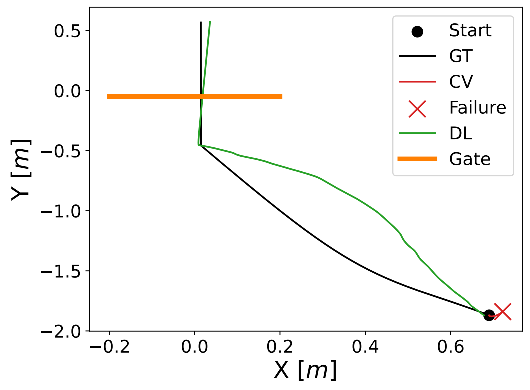


(c)

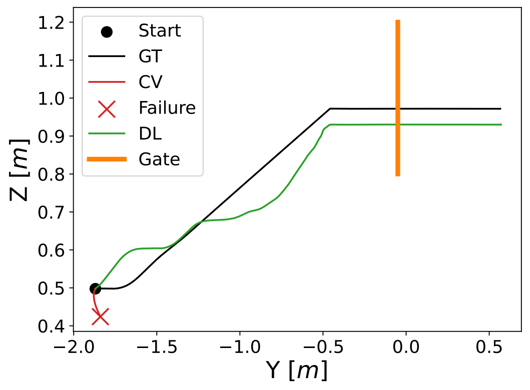
Figure 4.23: Competition world. Distance and orientation of the drone with respect to the gate.



(a)



(b)



(c)

Figure 4.24: Competition world. Trajectory is shown in 2D and 3D. It focuses on the gate-based navigation and crossing of the gate.

Chapter 5

Conclusions and future work

This thesis project puts the focus on developing a vision-based control pipeline to make the Crazyflie 2.1 nano-quadcopter cross a gate successfully. The gate-based navigation exploits the IBVS to identify and approach the gate. Such a control requires a detection module to extract the corners of the gate. Hence, two modules based on different approaches were proposed. One relies on classic CV algorithms, while the other is learning-based, taking advantage of a CNN. Both detection modules were tested on both synthetic and real images, with the DL-module always outperforming the CV one. Synthetic images were collected in Webots, an open-source robotic simulator in which the training set used to train the CNN was collected. To deal with real images, data augmentation mimicking the actual Himax camera images was employed, reducing the error by almost 20% compared with a model trained without augmentations. The CV-based detector performed well in a simple environment with a plain background and nothing else except for the gate. Whenever the amount of textures increases, e.g., no more plain background, the detection hinders, leading to poor performance. Moreover, such a detector relies on many classic CV algorithms like the Canny edge detector, the Suzuki-Abe contour detection, and the Harris corner detector, and heuristics to choose the corresponding gate shape and the set of corners. In this sense, such a pipeline has many parameters to be tuned according to the environment in which it is deployed. Moreover, it is more computationally intensive than the DL counterpart.

The whole control pipeline, featuring both detection and IBVS modules, was then tested in three Webots worlds of increasing difficulty, having the

IMAV 2022 competition world as the most challenging. Having the IBVS module fixed, ten runs for each detection module were conducted in every world, always in the same starting position. As expected, and following the performance obtained during the detection testing, the DL detector outperformed the CV one by completing all the runs in each world except for one case in which the drone was too near the gate, and could not be able to precisely outputs the four corners because few training data of such a position were present. Instead, the CV detector achieved to complete all the runs only in the simplest of the worlds, the one featuring nothing else than the gate and a plain background. During the detection test, both detectors were tested on images without temporal consistency. Instead, in this case, images are subsequent, and both detectors showed noisy behaviour during the detection. Exponential smoothing weighing more past states than the current reduced such a behaviour. In addition, the same smoothing was performed to the output velocities of the IBVS module because, without it, the drone couldn't accomplish the task.

Future works will involve the deployment of the CNN-based IBVS control pipeline on the actual Crazyflie 2.1 to work fully onboard. To this purpose, the adopted CNN is an adjusted version of the PULP-Frontnet, which was already proven to be working onboard. More thorough data collection could be assessed to pursue better predictions and avoid wrong predictions whenever the drone is too close. Concerning the noisy behaviour, time consistency can be considered during the training, and other than that, different loss functions can be studied, like the $L2-$loss rather than the simple $L1-$loss.

Bibliography

[1] Stuart Lloyd. “Least square quantization in PCM”. In: *Bell Telephone Laboratories Paper* (1957).

[2] I. Sobel and G. Feldman. “A 3x3 Isotropic Gradient Operator for Image Processing”. Stanford Artificial Intelligence Project (SAIL). 1968.

[3] Hiroaki Sakoe and Seibi Chiba. “Dynamic programming algorithm optimization for spoken word recognition”. In: *IEEE transactions on acoustics, speech, and signal processing* 26.1 (1978), pp. 43–49.

[4] Satoshi Suzuki and Keiichi Abe. “Topological structural analysis of digitized binary images by border following”. In: *Computer Vision, Graphics, and Image Processing* 30.1 (1985), pp. 32–46. ISSN: 0734-189X. DOI: `https://doi.org/10.1016/0734-189X(85)90016-7`.

[5] John Canny. “A Computational Approach to Edge Detection”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-8.6 (1986), pp. 679–698. DOI: `10.1109/TPAMI.1986.4767851`.

[6] Chris Harris, Mike Stephens, et al. “A combined corner and edge detector”. In: *Alvey vision conference*. Vol. 15. 50. Citeseer. 1988, pp. 10–5244.

[7] Charles F Van Loan and G Golub. “Matrix computations (Johns Hopkins studies in mathematical sciences)”. In: *Matrix Computations* 5 (1996).

[8] O. Michel. “Webots: Professional Mobile Robot Simulation”. In: *Journal of Advanced Robotics Systems* 1.1 (2004), pp. 39–42. URL: `http://www.ars-journal.com/International-Journal-of-%20Advanced-Robotic-Systems/Volume-1/39-42.pdf`.

[9] François Chaumette and Seth Hutchinson. “Visual servo control. I. Basic approaches”. In: *IEEE Robotics & Automation Magazine* 13.4 (2006), pp. 82–90.

[10] Anastasios I Mourikis and Stergios I Roumeliotis. “A multi-state constraint Kalman filter for vision-aided inertial navigation”. In: *Proceedings 2007 IEEE international conference on robotics and automation.* IEEE. 2007, pp. 3565–3572.

[11] Edward Rosten, Reid Porter, and Tom Drummond. “Faster and better: A machine learning approach to corner detection”. In: *IEEE transactions on pattern analysis and machine intelligence* 32.1 (2008), pp. 105–119.

[12] Diederik P Kingma and Jimmy Ba. “Adam: A method for stochastic optimization”. In: *arXiv preprint arXiv:1412.6980* (2014).

[13] Michael Bloesch et al. “Robust visual inertial odometry using a direct EKF-based approach”. In: *2015 IEEE/RSJ international conference on intelligent robots and systems (IROS).* IEEE. 2015, pp. 298–304.

[14] Stefan Leutenegger et al. “Keyframe-based visual–inertial odometry using nonlinear optimization”. In: *The International Journal of Robotics Research* 34.3 (2015), pp. 314–334.

[15] Fadri Furrer et al. “Rotors—a modular gazebo mav simulator framework”. In: *Robot Operating System (ROS) The Complete Reference (Volume 1)* (2016), pp. 595–625.

[16] T. DeVries and Graham W T. “Improved regularization of convolutional neural networks with cutout”. In: *arXiv preprint arXiv:1708.04552* (2017).

[17] Jakob Engel, Vladlen Koltun, and Daniel Cremers. “Direct sparse odometry”. In: *IEEE transactions on pattern analysis and machine intelligence* 40.3 (2017), pp. 611–625.

[18] Tong Qin, Peiliang Li, and Shaojie Shen. “Vins-mono: A robust and versatile monocular visual-inertial state estimator”. In: *IEEE Transactions on Robotics* 34.4 (2018), pp. 1004–1020.

[19] Shital Shah et al. “Airsim: High-fidelity visual and physical simulation for autonomous vehicles”. In: *Field and Service Robotics: Results of the 11th International Conference.* Springer. 2018, pp. 621–635.

[20] Winter Guerra et al. “Flightgoggles: Photorealistic sensor simulation for perception-driven robotics using photogrammetry and virtual reality”. In: *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS).* IEEE. 2019, pp. 6941–6948.

[21] Elia Kaufmann et al. "Beauty and the beast: Optimal methods meet learning for drone racing". In: *2019 International Conference on Robotics and Automation (ICRA)*. IEEE. 2019, pp. 690–696.

[22] Hyungpil Moon et al. "Challenges and implemented technologies used in autonomous drone racing". In: *Intelligent Service Robotics* 12 (2019), pp. 137–148.

[23] Davide Scaramuzza and Zichao Zhang. "Visual-inertial odometry of aerial robots". In: *arXiv preprint arXiv:1906.03289* (2019).

[24] Alexander Buslaev et al. "Albumentations: Fast and Flexible Image Augmentations". In: *Information* 11.2 (2020). ISSN: 2078-2489. DOI: `10.3390/info11020125`. URL: `https://www.mdpi.com/2078-2489/11/2/125`.

[25] European Research Council. *Low-latency Perception and Action for Agile Vision-based Flight*. `https://cordis.europa.eu/project/id/864042`. [ONLINE]. 2020.

[26] Tongwen Huang et al. "GateNet: gating-enhanced deep network for click-through rate prediction". In: *arXiv preprint arXiv:2007.03519* (2020).

[27] Daniele Palossi et al. "Fully Onboard AI-powered Human-Drone Pose Estimation on Ultra-low Power Autonomous Flying Nano-UAVs". In: *IEEE Internet of Things Journal* (2021). ISSN: 2327-4662. DOI: `10.1109/JIOT.2021.3091643`.

[28] Yunlong Song et al. "Flightmare: A flexible quadrotor simulator". In: *Conference on Robot Learning*. PMLR. 2021, pp. 1147–1157.

[29] Yaguang Tao et al. "A comparative analysis of trajectory similarity measures". In: *GIScience & Remote Sensing* 58.5 (2021), pp. 643–669.

[30] Philipp Foehn et al. "Alphapilot: Autonomous drone racing". In: *Autonomous Robots* 46.1 (2022), pp. 307–320.

[31] Antonio Paolillo. *Visual Servoing*. University Lecture. 2022.

[32] Huy Xuan Pham et al. "Pencilnet: Zero-shot sim-to-real transfer learning for robust gate perception in autonomous drone racing". In: *IEEE Robotics and Automation Letters* 7.4 (2022), pp. 11847–11854.

[33] Peter Corke. *Robotics, Vision and Control: Fundamental Algorithms in Python*. Vol. 146. Springer Nature, 2023.

[34] Drew Hanover et al. "Autonomous drone racing: A survey". In: *arXiv e-prints, pp. arXiv–2301* (2023).

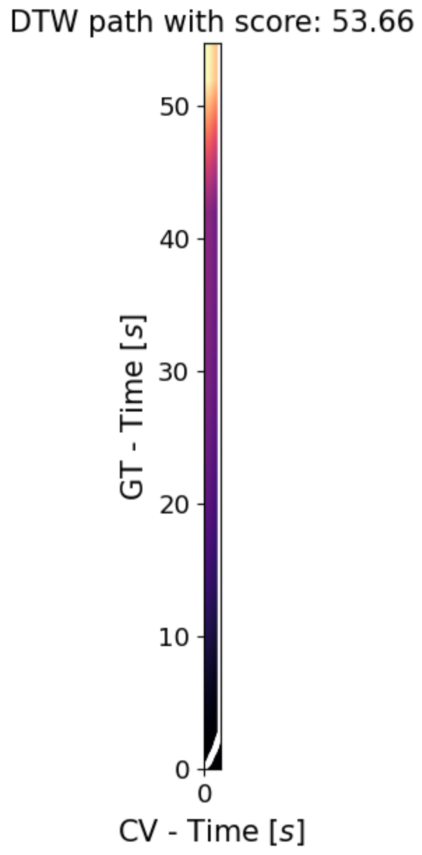
[35] *Latest Update on the AI deck*. `https://www.bitcraze.io/2020/05/latest-update-on-the-ai-deck/`. Accessed: 2023-09-25.

[36] *The Coordinate System of the Crazyflie 2.X*. `https://www.bitcraze.io/documentation/system/platform/cf2-coordinate-system/`. Accessed: 2023-09-25.

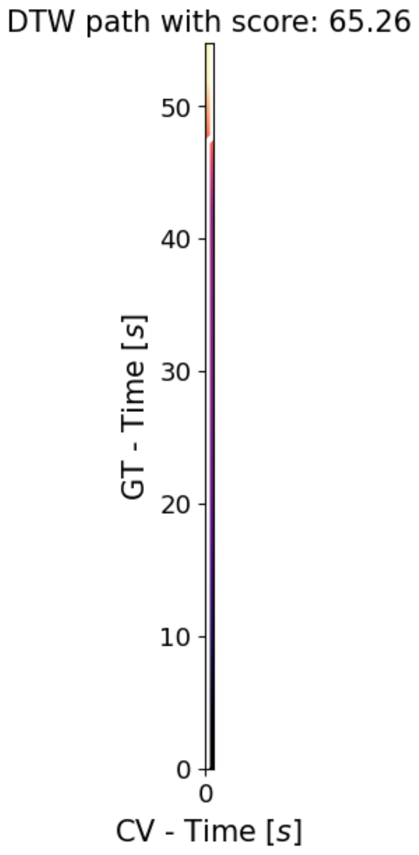
[37] Webots. *http://www.cyberbotics.com*. Ed. by Cyberbotics Ltd. Open-source Mobile Robot Simulation Software. URL: `http://www.cyberbotics.com`.

Appendix A

Results



(a)



(b)

Figure A.1: DTW scores and visualization of the path. From a run using the CV detector in the Laboratory world (a) and in the Competition world (b).