



DIOGO CARREIRA CRUZ PEREIRA CANECO

Bachelor's Degree in Electrical and Computer Engineering

AUTONOMOUS HIGH-PRECISION LANDING ON A UNMANNED SURFACE VEHICLE

THE MAIN GOAL OF THIS THESIS IS THE DEVELOPMENT OF AN
AUTONOMOUS HIGH-PRECISION LANDING SYSTEM OF AN UAV IN
AN AUTONOMOUS BOAT

MASTER IN ELECTRICAL AND COMPUTER ENGINEERING

NOVA University Lisbon
September, 2022

AUTONOMOUS HIGH-PRECISION LANDING ON A UNMANNED SURFACE VEHICLE

THE MAIN GOAL OF THIS THESIS IS THE DEVELOPMENT OF AN AUTONOMOUS
HIGH-PRECISION LANDING SYSTEM OF AN UAV IN AN AUTONOMOUS BOAT

DIOGO CARREIRA CRUZ PEREIRA CANECO

Bachelor's Degree in Electrical and Computer Engineering

Adviser: José Barata Oliveira

Full Professor, NOVA School of Science and Technology Caparica

Co-Adviser: Francisco Marques

Research Engineer, UNINOVA

Examination Committee

Chair: Doutor Nuno Manuel Ortega Amaro

Full Professor, FCT/UNL

Rapporteur: Doutor João Paulo Branquinho Pimentão

Full Professor, FCT/UNL

Co-Adviser: Mestre Francisco Antero Cardoso Marques

Research Engineer, UNINOVA

Autonomous High-Precision Landing on a Unmanned Surface Vehicle

Copyright © Diogo Carreira Cruz Pereira Caneco, NOVA School of Science and Technology, NOVA University Lisbon.

The NOVA School of Science and Technology and the NOVA University Lisbon have the right, perpetual and without geographical boundaries, to file and publish this dissertation through printed copies reproduced on paper or on digital form, or by any other means known or that may be invented, and to disseminate through scientific repositories and admit its copying and distribution for non-commercial, educational or research purposes, as long as credit is given to the author and editor.

ACKNOWLEDGEMENTS

I'd want to offer my heartfelt appreciation to my family, who have always provided me with unwavering support, both financially and emotionally, during this entire journey. My parents, in particular, shaped me into the person I am today, and without whom nothing I have done over my whole academic career would have been conceivable.

I would like to convey my appreciation to my dissertation advisor, Prof. José Barata, for giving me the wonderful opportunity to participate in the work of his exceptional research team. To the entire team at [Robotics and Industrial Complex Systems \(RICS\)](#) that I worked with and was responsible for overseeing my work throughout the course of the past year: Manuel Silva, Inês Oliveira, Nastaran Ghalati, and especially Francisco Marques, who gave me unwavering support and was always willing to lend a hand and discuss ideas whenever they could. In addition, I want to express my gratitude the Portuguese Armada for providing the laboratory space needed to conduct this dissertation work as well as the [NOVA School of Science and Technology \(FCT\)](#) Institution, where I have studied for the past five years.

The next group of acknowledgements are for all of my coworkers turned friends over the years, especially Afshin Lotfi, Gonçalo Valentim and André Teixeira, who I worked with for the majority of that time and to whom I wish the best of luck in his future endeavours.

Last but not least, I want to express my gratitude to my network of friends, who provided me with the encouragement and support I needed to redact this dissertation. And that throughout this ordeal, always found a way to make me laugh, and I will always be grateful for that.

ABSTRACT

In this dissertation, a collaborative method for **Multi Rotor Vertical Takeoff and Landing (MR-VTOL) Unmanned Aerial Vehicle (UAV)**s' autonomous landing is presented. The majority of common **UAV** autonomous landing systems adopt an approach in which the **UAV** scans the landing zone for a predetermined pattern, establishes relative positions, and uses those positions to execute the landing. These techniques have some shortcomings, such as extensive processing being carried out by the **UAV** itself and requires a lot of computational power. The fact that most of these techniques only work while the **UAV** is already flying at a low altitude, since the pattern's elements must be plainly visible to the **UAV**'s camera, creates an additional issue. An RGB camera that is positioned in the landing zone and pointed up at the sky is the foundation of the methodology described throughout this dissertation. Convolutional Neural Networks and Inverse Kinematics approaches can be used to isolate and analyse the distinctive motion patterns the **UAV** presents because the sky is a very static and homogeneous environment. Following real-time visual analysis, a terrestrial or maritime robotic system can transmit orders to the **UAV**.

The ultimate result is a model-free technique, or one that is not based on established patterns, that can help the **UAV** perform its landing manoeuvre. The method is trustworthy enough to be used independently or in conjunction with more established techniques to create a system that is more robust. The object detection neural network approach was able to detect the **UAV** in 91,57% of the assessed frames with a tracking error under 8%, according to experimental simulation findings derived from a dataset comprising three different films. Also created was a high-level position relative control system that makes use of the idea of an approach zone to the helipad. Every potential three-dimensional point within the zone corresponds to a **UAV** velocity command with a certain orientation and magnitude. The control system worked flawlessly to conduct the **UAV**'s landing within 6 cm of the target during testing in a simulated setting.

Keywords: **UAV** Autonomous Landing Systems, Robotic Cooperation, Convolutional Neural Networks, Object Detection, Inverse Kinematics, High-Level Control.

RESUMO

Nesta dissertação, é apresentado um método de colaboração para a aterragem autónoma de **Unmanned Aerial Vehicle (UAV) Multi Rotor Vertical Takeoff and Landing (MR-VTOL)**. A maioria dos sistemas de aterragem autónoma de **UAV** comuns adopta uma abordagem em que o **UAV** varre a zona de aterragem em busca de um padrão pré-determinado, estabelece posições relativas, e utiliza essas posições para executar a aterragem. Estas técnicas têm algumas deficiências, tais como o processamento extensivo a ser efectuado pelo próprio **UAV** e requer muita potência computacional. O facto de a maioria destas técnicas só funcionar enquanto o **UAV** já está a voar a baixa altitude, uma vez que os elementos do padrão devem ser claramente visíveis para a câmara do **UAV**, cria um problema adicional. Uma câmara RGB posicionada na zona de aterragem e apontada para o céu é a base da metodologia descrita ao longo desta dissertação. As Redes Neurais Convolucionais e as abordagens da Cinemática Inversa podem ser utilizadas para isolar e analisar os padrões de movimento distintos que o **UAV** apresenta, porque o céu é um ambiente muito estático e homogéneo. Após análise visual em tempo real, um sistema robótico terrestre ou marítimo pode transmitir ordens para o **UAV**.

O resultado final é uma técnica sem modelo, ou que não se baseia em padrões estabelecidos, que pode ajudar o **UAV** a realizar a sua manobra de aterragem. O método é suficientemente fiável para ser utilizado independentemente ou em conjunto com técnicas mais estabelecidas para criar um sistema que seja mais robusto. A abordagem da rede neural de detecção de objectos foi capaz de detectar o **UAV** em 91,57% dos fotogramas avaliados com um erro de rastreio inferior a 8%, de acordo com resultados de simulação experimental derivados de um conjunto de dados composto por três filmes diferentes. Também foi criado um sistema de controlo relativo de posição de alto nível que faz uso da ideia de uma zona de aproximação ao heliporto. Cada ponto tridimensional potencial dentro da zona corresponde a um comando de velocidade do **UAV** com uma certa orientação e magnitude. O sistema de controlo funcionou sem falhas para conduzir a aterragem do **UAV** dentro de 6 cm do alvo durante os testes num cenário simulado.

Traduzido com a versão gratuita do tradutor - www.DeepL.com/Translator

Palavras-chave: Sistemas Autónomos de Aterragem de **UAV**, Cooperação Robótica, Redes Neurais Convolucionais, Detecção de Objectos, Cinemática Inversa, Controlo de Alto Nível.

CONTENTS

List of Figures	ix
List of Tables	xii
Acronyms	xiv
1 Introduction	1
1.1 Dissertation Outline	2
1.2 The problem	2
1.3 Proposed Solution	4
2 State of the Art	6
2.1 Model and Control of a multi-rotor UAV	6
2.1.1 UAV Physics and Model Theory	6
2.1.1.1 General Remarks on multi-rotor UAVs	6
2.1.1.2 Quadrotor UAV Mathematical Model	8
2.1.1.2.1 Coordinate System and General Assumptions	9
2.1.1.2.2 Newton-Euler Formalism	10
2.1.1.2.3 Dynamic Model and Notes on Linearization	15
2.1.2 Control Theory	17
2.1.2.1 Low-level Control	18
2.1.2.2 High-level Control	18
2.1.3 Parrot Bebop 2 Overview	19
2.2 High-Precision multi-rotor UAV Landing Systems	20
2.2.1 Landing Routine of multi-rotor UAVs	20
2.2.1.1 High-Level Routine Overview	21
2.2.1.2 The Problem with UAV Landing	22
2.2.1.3 The Challenge of UAV Landing on Mobile Platforms	23
2.2.2 Detection Methods for Autonomous Landing	23
2.2.2.1 Related's Work Methodology	24
2.3 Gazebo Simulation Environment for multi-rotors UAV	26
2.3.1 Gazebo Overview	26
2.3.1.1 Description	26

2.3.1.2	Architecture and ROS Integration	26
2.3.1.3	Context of simulation in multi-rotors UAV experiments	29
2.3.2	Software-in-the-loop (SIL) Simulation	29
2.3.2.1	Fundamental Concepts and Major Features	29
2.3.2.2	RotorS - SIL Simulation ROS packages for multi-rotor UAVs	29
2.3.3	BebopS - Parrot Bebop 2 + RotorS	31
2.3.4	Parrot's Sphinx Simulation	32
2.3.5	Comparison between Simulators	33
3	Proposed Model	36
3.1	General Overview	36
3.2	UAV Detection	39
3.2.1	Detection	40
3.2.2	Image + Detection Data Overlap	41
3.3	Middleware Communication Interface	42
3.3.1	Landing Target Computation	43
3.3.2	Image + Target Data Overlap	47
3.4	UAV Position Controller	48
3.4.1	Precision Landing Manager	48
3.4.2	Position Controller	49
4	Implementation	52
4.1	Experimental Setup	52
4.2	Darknet-53 + YOLOv3 CNN Framework	53
4.2.1	Architecture	53
4.2.1.1	Backbone Architecture	54
4.2.1.2	Multi-Scale Feature Pyramid Network	54
4.2.1.3	YOLOv3	56
4.2.2	Framework Files	57
4.3	Training Procedures	57
4.3.1	Data Gathering	58
4.3.2	Image Annotation	62
4.3.3	Data Augmentation	62
4.3.4	Parameters	68
4.3.5	Training	69
5	Experimental Results	72
5.1	Experimental Setup	72
5.1.1	ROS Integration	73
5.2	Preliminary Results	74
5.2.1	UAV Detection	74
5.2.1.1	YOLOv3 vs Other Frameworks Benchmark	74

5.2.1.2	Training Evaluation	76
5.2.1.2.1	Training Results	77
5.2.1.2.2	Training Results Analysis	81
5.2.1.3	Inference Evaluation	81
5.2.1.3.1	Test set Structuring	82
5.2.1.3.2	Inference Results	83
5.2.1.3.3	Inference Results Analysis	84
5.2.2	Position Controller	85
5.2.2.0.1	Simulation Environment	86
5.2.2.0.2	Simulation Results	88
5.2.2.0.3	Simulation Results Analysis	88
5.3	Real World Experimental Results	90
6	Conclusions and Future Work	94
6.1	Conclusions	94
6.2	Future Work	96
	Bibliography	98

LIST OF FIGURES

1.1	Proposed solution based on vision-based GCS method for landing MR-VTOL UAVs autonomously (adapted from [1]). The green cone represented depicts the camera Field of View (FOV) of the proposed robotic landing platform stationed on the autonomous boat.	4
2.1	Concept of UAV system architecture.	7
2.2	Simple diagram of a quadrotor UAV architecture (adapted from [3]).	8
2.3	Structural model of a quadrotor UAV.	8
2.4	General basic linear landing control system (adapted from [24]).	22
2.5	Vision-based most common method for landing MR-VTOL UAVs autonomously (adapted from [1]). The red cone represented depicts the camera mounted on the quadrotor Field of View (FOV)	24
2.6	Illustration of the main processes of Gazebo (adapted from [35]).	27
2.7	General environment structure in integration with ROS (simplified from [34]).	27
2.8	Schematic of a simple Drone + Camera communication over ROS.	28
2.9	Simple Architecture of RotorS Simulator (adapted from [38]).	30
2.10	Package Structure of RotorS Simulator. (adapted from [38]).	31
2.11	BebopS Control Scheme (obtained from [20]).	32
2.12	Architecture of Sphinx's Gazebo Simulator.	33
3.1	Proposed technique physical system, based on vision-based GCS method for landing MR-VTOL UAVs autonomously (adapted from [1]).	37
3.2	The proposed model structure is organised into three primary processes: UAV detection , Middleware Communication Interface , and Position Control actuator. The Detection element of the UAV Detection module uses a single-stream RGB image and a weights file as input and returns the corresponding bounding boxes for each instance. In the Middleware Interface , the Landing Target Computation section computes the relative distance from the UAV to the target landing site using the pixel to meter ratio. Then the Overlap section, overlaps the data and outputs a single-stream RGB image with the detected UAV and useful data. In the UAV Position Controller actuator commands are sent to the UAV depending on the nav landing waypoint assessment to perform the landing.	38

3.3	YOLO detection system (taken from [55]).	40
3.4	The Detection Model.	42
3.5	The Overlapping processes. The bounding boxes, classes, and scores arrays are sent along with the input image that has been modified for visualisation. The output of the model shows these arrays over the image.	42
3.6	Bounding Box coordinates in relation with the frame.	44
3.7	Calculating an object's 3D location from a camera image (adapted from [1]).	46
3.8	The Overlapping processes.	47
3.9	UAV location and control orders are given using a 3D coordinate system (adapted from [1]).	48
3.10	3D view of the approach zone.	49
3.11	Estimation of position controller error. An example of velocity commands mentioned in 3.11 from the target posture, shown in blue, and the current posture of the UAV, shown in grey. These commands are then subjected to error calculations in order to determine the offset of the drone in relation to the target, in each coordinate.	51
4.1	An overview of the YOLOv3 [60].	53
4.2	Network unit for backbone network base on Darknet-53.	54
4.3	Building a feature pyramid using an image pyramid (adapted from [61]).	55
4.4	Workflow for general machine learning that was employed for this dissertation.	58
4.5	Examples of images taken under various atmospheric conditions. Both sunny and extremely cloudy weather could be recorded. Different cloud shapes, sizes, and formations. Images 1, 2, and 5 (from right to left) also show the effects of a solar lens flare.	59
4.6	Representative frames from the constructed data set. Each image depicts a separate scenario that was captured after the limitations for the data collection step. To make viewing easier, the UAV was marked in several photos.	61
4.7	Examples of how the Data Augmentation techniques applied to the dataset.	67
4.8	Training phases implemented for the proposed model.	70
4.9	Example of the Helipad camera's Preliminary Training behaviour utilising the YOLOv3 Tiny and Darknet framework.	71
5.1	Helipad.	73
5.2	Real world flight tests setup.	73
5.3	Various frameworks accuracy, adapted from [55].	75
5.4	RGB channel training behaviour of the Helipad's camera using YOLOv3 Tiny with Darknet framework. With Validation mAP (%) in red and total Training loss in blue.	80
5.5	IoU calculation, adapted from [65]	82

5.6	A heatmap showing the distribution of the UAV class object in the image frame. The yellow diamond shape in the figure represents the area where the UAV is most likely to be found in the dataset.	83
5.7	Parrot Sphinx + Gazebo 9 Simulation Environment. Ground plane and coordinate axis in a simple universe The global file was simplified to conserve processing power. The UAV may be seen flying in images B and C. A close-up of the Parrot Bebop 2 model is seen in Image D. Rotor interference and movement are modulated in the surrounding area.	87
5.8	Results of simulation tests performed on the Position Controller module to evaluate its functionality and suitability for use in the real world.	90
5.9	3D Trajectory of real world flight tests.	91
5.10	Flight coordinates detail.	92
5.11	Flight coordinates detail X and Y coordinates.	92

LIST OF TABLES

2.1	Dynamical Parameters for Parrot Bebop 2 drone (adapted from [20]). . . .	20
2.2	Principal control commands of bebop_autonomy for Parrot Bebop 2 drone.	20
2.3	Typical phases of the landing process in a multirotor UAV.	21
2.4	Comparison between BebopS and Sphinx Simulator.	35
4.1	Network structure for the convolutional layer in backbone.	56
4.2	Values for the parameters used to train the network.	69
5.1	Mean Average Precision for both frameworks with IoU=0.5.	76
5.2	Training Loss metrics for each training phase. The Preliminary Train is covered in the previous chapter's section 4.3.5, while this chapter's prior section 5.2.1.2.1 describes the three final phases.	81
5.3	Number of images in the data set from each category.	83
5.4	Validation Mean Average Precision (mAP) values for each train phase with IoU=0.5 and Inference Times in frames per second. The Preliminary Train is covered in the previous chapter's section 4.3.5, while this chapter's prior section 5.2.1.2.1 presents the three final train phases results.	84

ACRONYMS

This document is incomplete. The external file associated with the glossary ‘acronym’ (which should be called `output.acr`) hasn’t been created.

Check the contents of the file `output.acn`. If it’s empty, that means you haven’t indexed any of your entries in this glossary (using commands like `\gls` or `\glsadd`) so this list can’t be generated. If the file isn’t empty, the document build process hasn’t been completed.

Try one of the following:

- Add `automake` to your package option list when you load `glossaries-extra.sty`.
For example:

```
\usepackage[automake]{glossaries-extra}
```

- Run the external (Lua) application:
`makeglossaries-lite.lua "output"`
- Run the external (Perl) application:
`makeglossaries "output"`

Then rerun \LaTeX on this document.

This message will be removed once the problem has been fixed.

INTRODUCTION

Technological improvements in robotic research have resulted in low-cost hardware with sophisticated sensors and adaptable motor capabilities. Robots are already capable of executing sophisticated procedures, although it appears that their primary restrictions are due to software. [Unmanned Aerial Vehicle \(UAV\)](#), like most technical developments and inventions in human history, were initially designed for military purposes. [UAVs](#) can relay real-time intelligence, surveillance, and reconnaissance data from hostile regions to the combat commander. All of this can be conducted without ever putting a human aircrew's life in jeopardy. That is the beauty of unmanned aircraft. For all of these reasons, they quickly became a valuable tool in the hands of armed forces all over the world. [UAVs](#) have been a key technology in recent years, not just for the military but also for robotics and the general public. Multi-rotor vehicles, in particular those capable of [Multi Rotor Vertical Takeoff and Landing \(MR-VTOL\)](#), are more versatile, have a lower production cost, and are easier to teleoperate than fixed-wing equivalents. They have become a desirable asset in a wide range of scientific, civic, and recreational uses. [UAVs](#) are being utilised in a wide range of applications, including search and rescue in natural disaster scenarios, deliveries, agricultural and agribusiness applications, and monitoring. These are only a few examples of [UAV](#) uses. Accurate landings on a moving vehicle are becoming increasingly important in today's environment. Improved landing performance would assist applications like as package delivery, docking and recharging, and surveillance. Accurate relative estimate is required for high-accuracy landings, which is not possible with the [Global Positioning System \(GPS\)](#) alone. The purpose of this research is to create a solution that is especially suitable for the sea. With the ongoing advancement of technology and the growing processing capability of our computers, the possibilities for [UAV](#) applications become nearly unlimited. It may not be too far-fetched to imagine that in the future, our skies will be filled with [UAVs](#) built to do the most diverse jobs in

our daily lives. The automatic UAV detection module that will correctly detect and determine the position of the drone, as well as compute the landing path for a safe landing, will be the main topic of this dissertation. It will receive data from the nearby UAVs and the stationary helipad in real time.

1.1 Dissertation Outline

This dissertation will be structured as follows:

- Chapter 1 - **Introduction**: Provides an introduction to the document, its inspiration, the challenge of UAV object detection and categorization, and a description of the suggested solution and its integration into the situation at hand.
- Chapter 2 - **State of the Art**: Displays the various systems that have already been implemented to solve this and similar problems and explains the various technologies that will be used to implement the proposed solution, primarily in the areas of high-precision autonomous landing, multirotor UAV control theory, and gazebo simulation environment.
- Chapter 3 - **Proposed Model**: Explains the broad model under consideration and the considerations that went into the decisions that were adopted.
- Chapter 4 - **Implementation**: Contains a full overview of the tools used to implement the suggested model and how they operated.
- Chapter 5 - **Experimental Results**: Examines the experimental outcomes obtained using the suggested approach.
- Chapter 6 - **Conclusions**: Offers a summary of the work done in this scope of this dissertation, as well as the possible further research topics, and will present the conclusions findings of this document.

1.2 The problem

The goal of this document is to create an autonomous detection and landing system using data from a Ground Control Station (GCS) and an airborne UAV to detect appropriate landing trajectories, classify them, and extract position controller parameters from the previously collected data. The UAV is equipped with remote vision-based sensing capabilities that ensure the detection of the landing pad and the classification is to be done based

on their relative **Position and Orientation (POSE)**. UAVs have piqued the interest of the scientific community in recent years, particularly those capable of **Vertical Takeoff and Landing (VTOL)**. One of the most serious difficulties for UAVs is autonomous landing. Landing UAVs on boats in choppy waves or in unfamiliar situations requires autonomous landing on an inclined surface. The capacity to land independently, particularly on a ship's deck, is still not solved. For example, a ship may have a level landing platform, yet its orientation shifts with time. Also, given the maritime circumstances, which include adverse wind and sea currents, normally paired with adverse wind gusts that affect the UAV, the calculation of ship movements might be influenced to the point where landing is not always feasible.

The system to be constructed should be able to monitor the UAV by calculating its **Position and Orientation** in relation to the landing-pad ship, as well as compute the relevant landing parameters that must be sent into the control loop in order to assess the likelihood of a successful landing. Despite their widespread use in the past, and despite their benefits, **Inertial Navigation Systems (INS)** and **Global Navigation Satellite Systems (GNSS)** are not appropriate for a system where the aim is to achieve high-precision landing on not just a moving platform, but a marine heaving platform. Vision-based solutions are becoming more popular since they are passive and require no additional equipment other than a camera and a processing unit. So, in order to effectively identify the problem and develop a structured solution, it must be broken down into its major components:

1. Focusing on the **autonomous detection** of the landing pad, it is important to develop a model and correspondent sensor array capable of receiving in real time images from the UAV and the autonomous ship.
2. Since the data has been acquired, the **autonomous tracking** of the UAV is now possible, therefore this module is tasked with analysing it to calculate the pose measurements of the UAV and the GCS, or more specifically the ship landing pad, as well as the metrics of their location with relation to each other.
3. Now that the all the modules have the necessary information about one another, the UAV can start it's **autonomous descent** over target. It's here that the proposed model needs to be able to, with the pose data collected, first compute the necessary adjustments to the longitudinal position control to align with the landing pad (lateral distance), and second to perform altitude position control (vertical distance). This data needs to be delivered in a near real-time seamless communication manner between the UAV and the GCS.
4. Assuming that the atmospheric and maritime circumstances are moderate enough for the control loop to determine that a positive landing is feasible, the final issue is **landing confirmation**. In addition, the deployment of a UAV securing system may be required to guarantee that the UAV has safely landed, has been disarmed, and

is securely attached to the platform. This guarantees that any operation the UAV requires at home base may be satisfied.

1.3 Proposed Solution

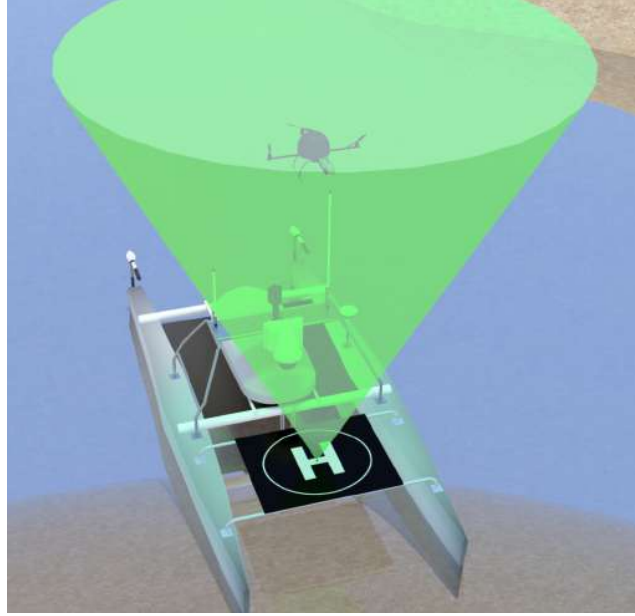


Figure 1.1: Proposed solution based on vision-based GCS method for landing MR-VTOL UAVs autonomously (adapted from [1]). The green cone represented depicts the camera Field of View (FOV) of the proposed robotic landing platform stationed on the autonomous boat.

This dissertation offers a cooperative vision-based landing system for MR-VTOL UAVs, with the goal of addressing some of the challenges that typical pattern-based systems confront. Rather of landing on its own, the UAV receives and relies on information provided by the helipad. The helipad is effectively transformed into a smart element capable of sensory data collecting and information processing, which may or may not be linked to a mobile platform. A camera is installed in the centre of the helipad, facing upwards towards the sky, with the goal of identifying the UAV during the landing, as depicted in Figure 1.1. As a result, a large portion of the calculation is transferred away from the UAV and to the helipad landing system, allowing the UAV to preserve computing resources and battery power. The detecting method makes use of the the research work done by Prates et al. (2018) [1] in the scope of his dissertation.

A high-level control technique will be designed that takes into account the relative position and orientation between the UAV and the helipad. The most difficult aspect of computing the pose will be solved by utilising a 1 camera running on a Convolutional Neural Network (CNN) capable of detecting the UAV position in the frame for later computing the pose vector appropriately. It is possible them to create pose estimates for the picture by using focal matching techniques in conjunction with the camera's

known intrinsic and extrinsic characteristics. When compared to other depth-camera-sensors, these often utilise less power. With the data from this sensor, the control module creates an approach zone above the helipad airspace, with each divergence from the centre position representing a different velocity instruction. The position controller node has the benefit of having the intended goal position as the centre of the reference, or in other words the coordinates point (0,0,0) if we use XYZ nomenclature. A velocity command with a fixed orientation and magnitude is given for each relative position of the UAV regarding the landing pad vessel inside the zone. It is worth noting that there is potential of making alterations to the landing platform in order to ease the process of landing stabilisation that unfortunately was not addressed. Not just to make things easier, but also to allow for considerably harsher marine and atmospheric unfavourable situations. The proposed platform solutions are dependent on the trial findings, nonetheless it is reasonable to say that some type of servo actuators can be put beside the landing pad to guarantee an added level of stability. It is also helpful for security reasons to consider the existence of a security net that encircles the landing pad to guarantee that the UAV does not damage the autonomous vessel.

The provided solution is general and modular, meant to act as a stand-alone system, conduct the landing on its own, or be part of a larger system. For example, the system and a traditional technique can work together to overcome or minimise each other's faults by combining numerous cues to conduct a more reliable and robust autonomous landing algorithm. It is assumed that our solution will not operate in every environment, whether atmospheric or other, but thorough testing will be done to verify that the capabilities of the intended solution are not under or overestimated.

STATE OF THE ART

2.1 Model and Control of a multi-rotor UAV

2.1.1 UAV Physics and Model Theory

2.1.1.1 General Remarks on multi-rotor UAVs

Unmanned Aerial Vehicle (UAV) [2] are aircraft that can fly without the presence of a pilot on board, this feature has been increasingly popular in recent years due to the capacity of UAVs to do complicated jobs including surveillance, catastrophe monitoring, wildfire management, and tracking. The UAV, the Ground Control Station (GCS), and the communication links are all part of the UAV system, a simple diagram is provided in 2.1. Recent research in the technology of sensors, microprocessors, and aerodynamics have allowed multi-rotor UAVs such as quadcopters to flourish. Quadcopters are aircraft that have four independently controllable rotors installed at the extremities of a rigid planar-shaped cross structure. Each rotor shaft is attached to a propeller. This arrangement has the principal benefit of enabling vertical take-off and allowing the aircraft to hover without effort. Because the UAV is intrinsically unstable, multiple sensors, as well as a rapid and reliable control system, are required for a steady flight.

The multi-rotor UAV, unlike traditional helicopters, lacks a tail rotor for direct yaw control, demanding a more sophisticated manoeuvre involving torque generated by the engines. The general principle behind this says that when two motors revolve clockwise (*cw*) and two revolve counterclockwise (*ccw*) the total torque is zero as long as each pair of motors revolves at the same speed, this enables the UAV to achieve a stable hovering state. All these features make multi-rotor, especially quadrotors, extremely popular for applications in commercial aircraft activities once they are suitable for the case study. This entails the use of precise dynamic models, high-performance and precise controllers,

low-latency status estimators, obstacle avoidance and route planning, as well as the ability to land on mobile platforms and pick up payloads [3]. Li et al. (2020) [4] and Moon et al. (2019) [5] applied this to the more complex and less commercial task of autonomous drone racing.

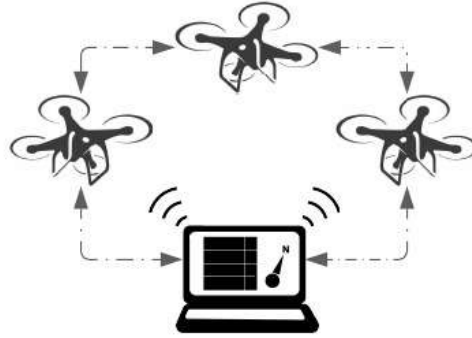


Figure 2.1: Concept of UAV system architecture.

The following constitute the most critical components of a multi-rotor UAV [6], for simplicity's sake the example will be about a quadrotor UAV. For better understanding of the architecture of a quadrotor UAV a diagram is also provided below in 2.2.

- The **frame** is the bearing structure of a UAV. To withstand collisions and stress, it must be light and strong. It's usually composed of plastic or carbon fibre, and it's shaped like an \otimes or a \oplus .
- The brushless **motors** power the propellers, supporting and maintaining the drone in the air.
- The **propellers** might be built of plastic or carbon fibre and must be as light and strong as the frame.
- The **Electronic Speed Controls (ESC)** regulate the rotation speed of the electric motor and command its rotation direction by varying the supply voltage supplied by the battery. Each engine is equipped with a separate **Electronic Speed Controls (ESC)**.
- The **Lithium Polymer (LiPo) battery** is the main energy source, the only one in most UAVs, and its capacity dictates flying autonomy and the maximum power generated by the engines.
- The **microcontroller** collects sensor data and, after processing it with a specialised **CPU**, regulates the rotational speed of motors through **ESCs**. The paramount sensors on a drone are the **Inertial Measurement Unit (IMU)**, which measures angular acceleration and speed, but also the gyroscope, **GPS** and altimeter.

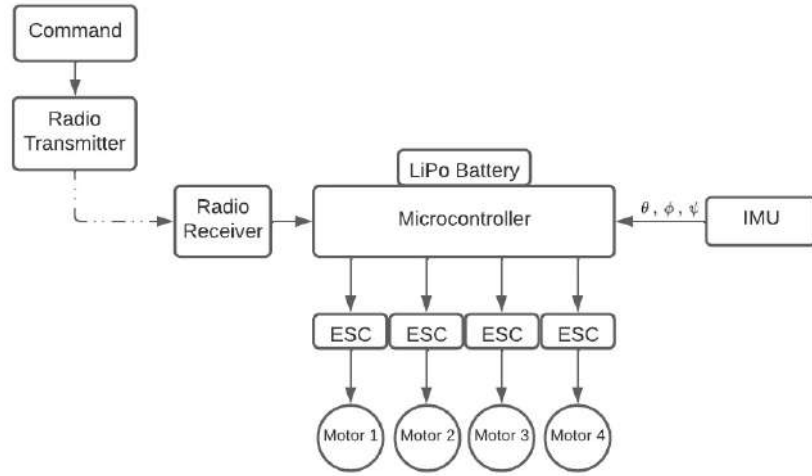


Figure 2.2: Simple diagram of a quadrotor UAV architecture (adapted from [3]).

2.1.1.2 Quadrotor UAV Mathematical Model

The quadcopter's location and orientation may be controlled by adjusting the speed of its four motors. These are separated into two groups based on the direction of rotation, with Motor 1 and 3 rotating *ccw* and Motor 2 and 4 rotating *cw*, according to 2.3. Whenever a quadrotor is in a hovering state, we assume that all of its propellers revolve at the same speed to counteract gravity's acceleration. As a result, the quadrotor conducts stationary flight, with no forces or torques causing it to move from its position [7].

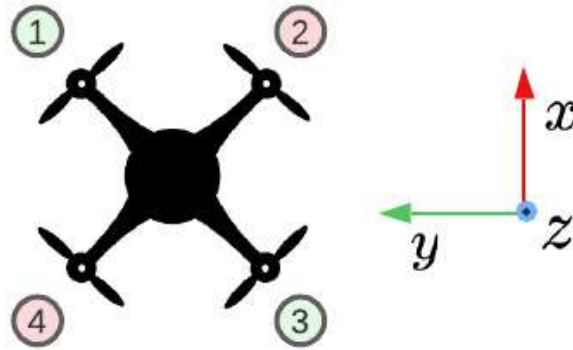


Figure 2.3: Structural model of a quadrotor UAV.

In the model in 2.3, it is also possible to see that the quadrotor has 6 **Degrees of Freedom (DoF)** despite the fact that it only has four propellers, therefore it is not feasible to attain the desired set-point for all of the DoF, but only for a maximum of four. Nevertheless, because of its structure, it is reasonably straightforward to select the four best controllable variables, as the quadrotor objectives are thus tied to the four primary movements that allow the helicopter to attain a specific height and attitude. A brief description of those motions is provided to the reader below [6]. Note that the description follows the same logic of propeller identification depicted in 2.3.

- Throttle U_1

This operation is implemented by increasing (or reducing) the speeds of all propellers by the same amount. It generates a vertical force (Z-axis) concerning the body-fixed frame, which raises or lowers the quadrotor. When the helicopter is horizontal, the vertical direction of the inertial frame and one of the body-fixed frames coincide. Otherwise, the given thrust causes inertial accelerations in both the vertical and horizontal directions.

- Roll U_2

This operation is implemented by increasing (or reducing) the speed of propellers 1 and 4 and decreasing (or increasing) the speed of propellers 2 and 3. It produces torque concerning the X-axis, which causes the quadrotor to revolve. Because the overall vertical thrust is the same as when hovering, this operation simply results in a roll angle acceleration.

- Pitch U_3

This operation is analogous to the roll and is implemented by increasing (or reducing) the speed of propellers 3 and 4 and decreasing (or increasing) the speed of propellers 1 and 2. It produces torque concerning the Y-axis, which causes the quadrotor to revolve. Because the overall vertical thrust is the same as when hovering, this instruction only results in a pitch angle acceleration.

- Yaw U_4

This operation is implemented by increasing (or reducing) the speed of propellers 1 and 2 and decreasing (or increasing) the speed of propellers 3 and 4. It produces a torque concerning the Z-axis, which causes the quadrotor to turn. The yaw movement is caused by the fact that propellers 2 and 4 revolve *cw* while propellers 1 and 3 revolve *ccw*. As a result, if the entire torque is uneven, the helicopter will flip on itself around the Z-axis. So because the overall vertical thrust is the same as when hovering, this operation only results in a yaw angle displacement.

2.1.1.2.1 Coordinate System and General Assumptions

In order to create the equations that characterise the motion of a quadcopter, the reference coordinates in which the orientation and position of the drone are defined must be specified. In our scenario, two reference systems are used: one fixed, E-frame, also known as earth inertial frame, which represents the ground, and one mobile, B-frame, also known as body-fixed frame, which is attached to the drone. The next section focuses on the actual characterization of the frames as well as their formal mathematical definition (2.1.1.2.2). The earth inertial frame was utilised to describe the linear position and

velocity of the quadcopter's centre of mass, while the body-fixed frame tethered to the drone was used to describe the position and angular velocity.

Many assumptions have been taken in developing the model in order for the equations of motion, described in sections 2.1.1.2.2 and 2.1.1.2.3, to be more clearly expressed in the body-fixed frame [8]:

- The structure is regarded as perfectly rigid, which ensures symmetrical qualities to facilitate the computation of the diagonal inertial matrix.
- The E-frame is considered time-invariant.
- The propellers are assumed to be equivalent and rigid, not allowing for deformation.
- On-board measurements are readily translated to body-fixed frame, B-frame, measurements.
- To generalise the electric modelling, motors are almost identical.
- Virtually all control forces are delivered in the B-frame.
- Thrust and drag are maintained in ideal gas conditions with laminar stream velocity.

2.1.1.2.2 Newton-Euler Formalism

In order to identify the basic equations of a 6 DoF rigid body the Newton-Euler formulation has been adopted, a more in depth description and formulation is provided by Miguel and Martins (2019) [3]. Kinematics and dynamics are used to assess the quadrotor UAV model.

- *Kinematics*

For body kinematics, without taking into account any forces or torques imparted to the quadrotor, two frames were defined in section 2.1.1.2.1:

- **E-frame** - earth inertial related reference frame

$$E (O_E, x_E, y_E, z_E) \tag{2.1}$$

$$\xi = [\Gamma^E \quad \Theta^E]^T = [X \quad Y \quad Z \quad \phi \quad \theta \quad \psi]^T$$

- **B-frame** - body-rigid related reference frame

$$B (O_B, x_B, y_B, z_B) \quad (2.2)$$

$$\mathcal{V} = \begin{bmatrix} V^B & W^B \end{bmatrix}^T = \begin{bmatrix} u & v & w & p & q & r \end{bmatrix}^T$$

Euler angles are used to describe a rigid body's orientation in space. These angles, in particular, represent the location of a reference system (XYZ) linked to a rigid body via a sequence of rotations beginning with a fixed reference system (ZYX). The origins of the two reference systems are the same and resorting to Euler angles were employed to characterise the quadcopter's orientation in space. Combining the aforementioned simple rotations and the frames in Equations 2.1 and 2.2 yields the rotation matrix translating from B-frame coordinates to E-frame coordinates:

$$R(\phi, \theta, \psi) = \text{Rot}_z^T(\psi) \cdot \text{Rot}_y^T(\theta) \cdot \text{Rot}_x^T(\phi) \\ = \begin{bmatrix} c_\psi \cdot c_\theta & -s_\psi \cdot c_\phi + c_\psi \cdot s_\theta \cdot s_\phi & s_\psi \cdot s_\phi + c_\psi \cdot s_\theta \cdot c_\phi \\ s_\psi \cdot c_\theta & c_\psi \cdot c_\phi + s_\psi \cdot s_\theta \cdot s_\phi & -c_\psi \cdot s_\phi + s_\psi \cdot s_\theta \cdot c_\phi \\ -s_\theta & c_\theta \cdot s_\phi & c_\theta \cdot c_\phi \end{bmatrix} \quad (2.3)$$

According to equation 2.3, the rotation matrix R_Θ is involved in the relation between the linear velocity in the B-frame V^B and that in the E-frame $\dot{\Gamma}^E$:

$$V^E = \dot{\Gamma}^E = R_\Theta \cdot V^B \quad (2.4)$$

$$\text{with } R_\Theta = R(\phi, \theta, \psi)$$

Using the same approach, we can calculate the angular velocity in the E-frame using the following relation:

$$\dot{\Theta}^E = T_\Theta \cdot W^E \\ \text{with } T_\Theta = \begin{bmatrix} 1 & s_\phi + t_\theta & c_\phi \cdot t_\theta \\ 0 & c_\theta & s_\phi \\ 0 & \frac{s_\theta}{s_\phi} & c_\phi \cdot c_\theta \end{bmatrix} \quad (2.5)$$

Equations 2.1 and 2.2 may be described in a single equivalence, in equation 2.6, that relates the derivative of the generalised position in the E-frame $\dot{\xi}$ to the generalised velocity in the B-frame v . The transformation is made feasible by the generalised matrix J_Θ , which converts the velocity of the B-frame to the velocity of the E-frame [3]. The symbol $0_{3 \times 3}$ in this matrix denotes a sub-matrix of dimension 3×3 filled with zeros.

$$\begin{aligned}\dot{\xi}^E &= J_\Theta \cdot v \\ \text{with } J_\Theta &= \begin{bmatrix} R_\Theta & 0_{3 \times 3} \\ 0_{3 \times 3} & T_\Theta \end{bmatrix}\end{aligned}\quad (2.6)$$

• *Dynamics*

The dynamics of the quadrotor is derived from the Newton-Euler equation described by Martinez (2007) [9]. The influence of forces and torques on movements should be examined in the B-frame from a dynamics standpoint, but recalling the assumptions in section 2.1.1.2.1, due to the invariance of the inertial matrix in time and by exploiting the symmetry of the body, despite the explicit nature of applied forces. To ease the computation of the inertial moment's matrix, the origin of the B-frame O_B was carefully chosen as the centre of mass. The linear components of the body motion are derived from Euler's first axiom of Newton's second law, according to equation 2.7.

$$\begin{aligned}m \cdot \ddot{\Gamma}^E &= F^E \\ m \cdot \widehat{R_\Theta V^B} &= R_\Theta \cdot F^B \\ m \cdot (\dot{V}^B + \omega^B \times V^B) &= F^B\end{aligned}\quad (2.7)$$

where $m[kg]$ is the body's mass, $\ddot{\Gamma}^E[m s^2]$ is the quadrotor's vector of linear acceleration in the E-frame, $F^E[N]$ is generalised forces in the E-frame, and $\dot{V}^B[m s^2]$ is the vector of linear acceleration in the body frame. Similarly, the angular component of movements of the Euler axiom of Newton's second law is deduced [3].

$$\begin{aligned}I \cdot \ddot{\Theta}^E &= \tau^E \\ I \cdot \dot{\omega}^B + \omega^B \times (I \cdot \omega^B) &= T_\Theta \cdot \tau^B\end{aligned}\quad (2.8)$$

where $I[N m s^2]$ represents the body inertia matrix (in the B-frame), $\ddot{\Theta}^E[rad s^{-2}]$ represents the quadrotor angular acceleration vector with respect to E-frame, $\dot{\omega}^B[rad s^{-2}]$ represents the quadrotor torque vector with respect to B-frame, and $\tau^E[N m]$ represents the quadrotor torques vector with respect to E-frame. It is now possible to describe the motion of a 6 DoF rigid body by combining equations 2.7 and 2.8.

$$\begin{bmatrix} m \cdot I_{3 \times 3} & 0_{3 \times 3} \\ 0_{3 \times 3} & I \end{bmatrix} \begin{bmatrix} \dot{V}^B \\ \dot{\omega}^B \end{bmatrix} + \begin{bmatrix} \omega^B \times (m \cdot V^B) \\ \omega^B \times (I \cdot \omega^B) \end{bmatrix} = \begin{bmatrix} F^B \\ \tau^B \end{bmatrix}\quad (2.9)$$

The symbol $I_{3 \times 3}$ denotes an identity matrix of size 3×3 . This equation is completely general and applies to any rigid body that adheres to the assumptions stated in section 2.1.1.2.1.

A generalised force vector Λ is defined as follows in equation 2.10. To accomplish this, two assumptions must be made in order to simplify the dynamics of the body model:

- The first asserts that the origin of the body-fixed frame coincides with the body's centre of mass.
- The second stipulates that the B-axes frame corresponds with the body's primary axis of inertia. In this example, the inertia matrix I is diagonal, making the body equations easy to solve.

$$\Lambda = \begin{bmatrix} F^B & \tau^B \end{bmatrix}^T = \begin{bmatrix} F_x & F_y & F_z & \tau_z & \tau_y & \tau_x \end{bmatrix}^T \quad (2.10)$$

As a result, equation 2.9 may be rewritten in matrix form [3].

$$M_B \cdot \dot{v} + C_B(v) \cdot v = \Lambda \quad (2.11)$$

Where \dot{v} is the generalised acceleration vector with respect to the B-frame. For the B-frame, M_B is the system inertia matrix and $C_B(v)$ is the Coriolis-centripetal matrix. The system inertia matrix M_B is defined in equation 2.12:

$$M_B = \begin{bmatrix} m \cdot I_{3 \times 3} & 0_{3 \times 3} \\ 0_{3 \times 3} & I \end{bmatrix} \quad (2.12)$$

Because of the assumptions mentioned above, it's easy to demonstrate that M_B is diagonal and constant. The Coriolis-centripetal matrix is seen in equation 2.13.

$$C_B(v) = \begin{bmatrix} 0_{3 \times 3} & -m \cdot S \cdot (V^B) \\ 0_{3 \times 3} & -S \cdot (I \cdot \omega^B) \end{bmatrix} \quad (2.13)$$

Equation 2.11 is general, and it is true for any rigid body motion given previously discussed assumptions. Nevertheless, because it was used to mimic the quadrotor helicopter in this work, the last vector contains information on its dynamics. Depending on the nature of the quadrotor contributions, Λ can be broken down into three components.

The first contribution is the gravitational vector $G_B(\xi)$ derived from the gravitational acceleration g [m s⁻²]. Since it is a force rather than a torque, it is obvious that it only impacts linear equations and not angular equations. The adjustments used to get $G_B(\xi)$ are shown in equation 2.14.

$$\mathbf{G}_B(\xi) = \begin{bmatrix} \mathbf{F}_G^B \\ \mathbf{0}_{3 \times 1} \end{bmatrix} = \begin{bmatrix} \mathbf{R}_\Theta^{-1} \cdot \mathbf{F}_G^E \\ \mathbf{0}_{3 \times 1} \end{bmatrix} \quad (2.14)$$

Where $\mathbf{F}_G^B[N]$ is the gravitational force vector in the B-frame and $\mathbf{F}_G^E[N]$ is the same in the E-frame. $\mathbf{0}_{3 \times 1}$ is a vertical vector having three zeros in it. In addition, because \mathbf{R}_Θ is an orthogonal normalised matrix, its inverted \mathbf{R}_Θ^{-1} equals its transposed \mathbf{R}_Θ^T .

The second component addresses the gyroscopic effects caused by the propeller rotation. When the algebraic total of the rotor speeds is not equal to zero, there is an overall imbalance because two of them rotate cw and the other two ccw. If the roll or pitch rates are also greater than zero, the quadrotor suffers gyroscopic torque, as calculated by equation 2.15.

$$\mathbf{O}_B(\boldsymbol{\nu}) \cdot \boldsymbol{\Omega} = J_{TP} \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ q & -q & q & -q \\ -p & p & -p & p \\ 0 & 0 & 0 & 0 \end{bmatrix} \cdot \boldsymbol{\Omega} \quad (2.15)$$

The gyroscopic propeller matrix is $\mathbf{O}_B(\boldsymbol{\nu})$, and the total rotational moment of inertia around the propeller axis is $J_{TP} [N \ m \ s^2]$. The overall's propeller speed $\Omega [rad \ s^{-1}]$ is defined by equation 2.16. Each propeller speed Ω_i follows the same logic as Figure 2.3.

$$\Omega = -\Omega_1 + \Omega_2 - \Omega_3 + \Omega_4 \quad (2.16)$$

The third contribution considers the forces and torques produced directly by the primary movement inputs. According to aerodynamics, both are proportional to the squared speed of the propellers. As a result, the movement vector \mathbf{U}_B is obtained. Martins (2019) [3] discusses in depth the derivation of the aerodynamic contributions (thrust $b [N \ s^2]$ and drag $d [N \ m \ s^2]$ components). We can define the vector \mathbf{U}_B in equation 2.17 based on our understanding of the nature of quadcopter movements.

$$\mathbf{U}_B(\boldsymbol{\Omega}) = \mathbf{E}_B \cdot \boldsymbol{\Omega}^2 = \begin{bmatrix} 0 \\ 0 \\ U_1 \\ U_2 \\ U_3 \\ U_4 \end{bmatrix} \quad (2.17)$$

Where U_1 , U_2 , U_3 and U_4 are the movement vector components described in equation 2.16. The correlation between their speeds and the speeds of the propellers is derived from aerodynamic calculus.

2.1.1.2.3 Dynamic Model and Notes on Linearization

Six-second order differential equations derived from balancing forces and momenta operating on the drone comprise the simplified dynamic model. The previous sections illustrated the formulation of the kinematics and dynamic equations that define a 6 DoF rigid body, or a quadrotor UAV, and are used to formulate the model. From the last section equation 2.17 is used to explain the quadrotor dynamics when these contributions, mentioned above, are taken into account. These contributions can be defined according to equation 2.18 and it is possible to isolate the acceleration vector $\dot{\mathbf{v}}$ of the B-frame by rearranging them, as in equation 2.19.

$$\mathbf{M}_B \cdot \dot{\mathbf{v}} + \mathbf{C}_B(\mathbf{v}) \cdot \mathbf{v} = \mathbf{G}_B(\boldsymbol{\xi}) + \mathbf{O}_B(\mathbf{v}) \cdot \boldsymbol{\Omega} + \mathbf{E}_B \cdot \boldsymbol{\Omega}^2 \quad (2.18)$$

$$\dot{\mathbf{v}} = \mathbf{M}_B^{-1} \cdot \left(-\mathbf{C}_B(\mathbf{v}) \cdot \mathbf{v} + \mathbf{G}_B(\boldsymbol{\xi}) + \mathbf{O}_B(\mathbf{v}) \cdot \boldsymbol{\Omega} + \mathbf{E}_B \cdot \boldsymbol{\Omega}^2 \right) \quad (2.19)$$

Representing equation 2.19 now in a system of equations [7]:

$$\begin{cases} \dot{u} = (v \cdot r - w \cdot q) + g \cdot s_\theta \\ \dot{v} = (w \cdot p - u \cdot r) - g \cdot c_\theta \cdot s_\phi \\ \dot{w} = (u \cdot q - v \cdot p) - g \cdot c_\theta \cdot s_\phi + \frac{U_1}{m} \\ \dot{p} = \frac{I_{YY} - I_{ZZ}}{I_{XX}} \cdot q \cdot r - \frac{J_{TP}}{I_{XX}} \cdot q \cdot \Omega + \frac{U_2}{I_{XX}} \\ \dot{q} = \frac{I_{ZZ} - I_{XX}}{I_{YY}} \cdot p \cdot r + \frac{J_{TP}}{I_{YY}} \cdot p \cdot \Omega + \frac{U_3}{I_{YY}} \\ \dot{r} = \frac{I_{XX} - I_{YY}}{I_{ZZ}} \cdot p \cdot q + \frac{U_4}{I_{ZZ}} \end{cases} \quad (2.20)$$

Where the speed inputs for the propellers are:

$$\begin{cases} U_1 = b \cdot (\Omega_1^2 + \Omega_2^2 + \Omega_3^2 + \Omega_4^2) \\ U_2 = l \cdot b \cdot (-\Omega_2^2 + \Omega_4^2) \\ U_3 = l \cdot b \cdot (-\Omega_1^2 + \Omega_3^2) \\ U_4 = d \cdot (\Omega_1^2 + \Omega_2^2 \Omega_3^2 + \Omega_4^2) \\ \Omega = \Omega_1 + \Omega_2 \Omega_3 + \Omega_4 \end{cases} \quad (2.21)$$

The quadrotor dynamic system is stated in the B-frame. It is preferable to be able to manage the quadrotor's location in relation to the E-frame as well as its orientation in relation to the B-frame. As a result, in this new reference, a hybrid frame with its position in the Earth's frame and its orientation in the B-frame is developed to make it simpler to depict the dynamics in combination with the control (especially for the vertical position in the E-frame). A new set of equations is needed to define the system in the new H-frame, analogously in the previous section, the same operations are performed and depicted in the equations below [7]. The quadrotor generalised velocity vector ζ with respect to the H-frame:

$$\zeta = \begin{bmatrix} \dot{\mathbf{r}}^E & \boldsymbol{\omega}^B \end{bmatrix}^T = \begin{bmatrix} \dot{X} & \dot{Y} & \dot{Z} & p & q & r \end{bmatrix}^T \quad (2.22)$$

According to equation 2.23, the dynamics of the system in the H-frame may be recast in matrix form.

$$\mathbf{M}_H \cdot \ddot{\zeta} + \mathbf{C}_H(\zeta) \cdot \dot{\zeta} = \mathbf{G}_H + \mathbf{O}_H(\zeta) \cdot \boldsymbol{\Omega} + \mathbf{E}_H(\xi) \cdot \boldsymbol{\Omega}^2 \quad (2.23)$$

The same can be done to the definitions of all the matrices and vectors used in equation 2.23.

$$\mathbf{M}_H = \mathbf{M}_B = \begin{bmatrix} m \cdot \mathbf{I}_{3 \times 3} & \mathbf{0}_{3 \times 3} \\ \mathbf{0}_{3 \times 3} & \mathbf{I} \end{bmatrix} \quad (2.24)$$

$$\mathbf{C}_H(\xi) = \begin{bmatrix} \mathbf{0}_{3 \times 3} & \mathbf{0}_{3 \times 3} \\ \mathbf{0}_{3 \times 3} & -\mathbf{S} \cdot (\mathbf{I} \cdot \boldsymbol{\omega}^B) \end{bmatrix} \quad (2.25)$$

$$\mathbf{G}_H(\xi) = \begin{bmatrix} \mathbf{F}_G^E \\ \mathbf{0}_{3 \times 1} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ -m \cdot g \\ \mathbf{0}_{3 \times 1} \end{bmatrix} \quad (2.26)$$

$$\mathbf{O}_H(\xi) = \mathbf{O}_B(\boldsymbol{\nu}) \cdot \boldsymbol{\Omega} = \mathbf{J}_{TP} \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ q & -q & q & -q \\ -p & p & -p & p \\ 0 & 0 & 0 & 0 \end{bmatrix} \cdot \boldsymbol{\Omega} \quad (2.27)$$

It is possible to isolate the acceleration vector $\ddot{\xi}$ with respect to H-frame by rearranging equation 2.23.

$$\ddot{\xi} = M_H^{-1} \cdot (-C_H(\xi) \cdot \dot{\xi} + G_H + O_H(\xi) \cdot \Omega + E_H(\xi) \cdot \Omega^2) \quad (2.28)$$

Representing equation (28) now in a system of equations:

$$\begin{cases} \ddot{X} = (\sin \psi \cdot \sin \phi + \cos \psi \cdot \sin \theta \cdot \cos \phi) \cdot \frac{U_1}{m} \\ \ddot{Y} = (-\cos \psi \cdot \sin \phi + \sin \psi \cdot \sin \theta \cdot \cos \phi) \cdot \frac{U_1}{m} \\ \ddot{Z} = -g + (\cos \theta \cdot \cos \phi) \cdot \frac{U_1}{m} \\ \dot{p} = \frac{I_{YY} - I_{ZZ}}{I_{XX}} \cdot q \cdot r - \frac{J_{TP}}{I_{XX}} \cdot q \cdot \Omega + \frac{U_2}{I_{XX}} \\ \dot{q} = \frac{I_{ZZ} - I_{XX}}{I_{YY}} \cdot p \cdot r + \frac{J_{TP}}{I_{YY}} \cdot p \cdot \Omega + \frac{U_3}{I_{YY}} \\ \dot{r} = \frac{I_{XX} - I_{YY}}{I_{ZZ}} \cdot p \cdot q + \frac{U_4}{I_{ZZ}} \end{cases} \quad (2.29)$$

Where the propeller speed inputs are the same as in the system with regard to B-frame and are provided in equation 2.21.

While the nonlinear model described above may successfully represent the dynamic behaviour of a quadrotor aircraft in generic configuration and during generic flights, it is well known that a linearized model is reasonably accurate during semi-hovering. A linear dynamic system is one whose evolution is guided by a linear equation and so adheres to the concept of superimposing effects. The work in [10] has recently reinforced the relevance and accuracy of employing linearized models in quadrotors. The process of converting a non-linear dynamic model to a linear one is also known as linearization, and it is based on the analysis of the non-linear model around an equilibrium point while taking only minor deviations into account.

2.1.2 Control Theory

Controlling the UAV is a critical component in the development of an autonomous landing system. The aim is to place the UAV on top of the landing platform as safely as possible, for the purpose of simplicity, it is assumed that the helipad is parallel to the ground and that there are no obstacles above it. The UAV takes off and is led to the helipad by navigation technologies, for e.g GPS-based ones. After spotting the features, patterns, or landmarks, it attempts to align itself with the centre of the helipad, gently beginning its descent while attempting to maintain the required alignment until it securely lands. Control of a multirotor UAV can be divided into low-level, which states how the motors are to be actuated to perform a certain movement in a certain direction, and high-level, which determines what movement to perform in a given scenario [1].

2.1.2.1 Low-level Control

Now that the dynamical model of the multirotor UAV system has been developed, it is possible to consider a control loop into which it may be integrated. Because of their simplicity and ease of use, a **Proportional Integrative Derivative (PID)** controller is a traditional option. PID controllers are one of the most prevalent methods for performing UAV control, yet effective tuning of a PID controller may be a difficult task. Because each PID controller has three main characteristics that must be tweaked, tuning numerous PIDs can take a long time, and in most cases, a large number of test flights are required to attain the desired performance. Typically, multirotor UAV PID controllers are set to work under specified conditions. However, if these conditions change, extra tuning may be required, such as if the payload carried by the UAV changes, because various weights might affect the UAV's flight dynamics in different ways, if the payload of a specific UAV has to be changed, the PID parameters may need to be re-tuned. A great deal of study has gone into implementing a PID controller to multirotor UAVs, ranging from simple textbook control systems to more complicated self-tuning. Li and Li (2011) [11] is an example of a simple PID control scheme that concludes that a quadrotor can achieve attitude stabilisation if the PID parameters are appropriate for the task at hand. Babu et al. (2017) [12], on the other hand, proposes a self-tuning PID controller based on gradient descent to perform waypoint navigation and leader-follower formation control. Giernacki (2019) [13] proposes a new real-time auto-tuning approach for fixed-parameter controllers based on a modified golden-search (zero-order) optimization algorithm and bootstrapping methodology. Feng et al. (2018) [14], on the other hand, proposes a different approach to the task of landing a quadrotor UAV, using a non-linear **Model Predictive Control (MPC)** and a Kalman filter for landing platform position estimation. Another alternative for the control loop is to use a **Neural Network (NN)**, which has learning capabilities that are suitable for quick reactions and flexibility in unknown situations. Maturana and Scherer (2015) [15] created a system that uses 3D Convolution Neural Networks to analyse data from the onboard sensors, the **LiDAR**, and determine the best landing zone for a UAV allow with estimation for controller parameters for the high-level behaviour of landing.

2.1.2.2 High-level Control

The majority of high-level controls consist of a set of behaviours whose nature is determined by the job at hand. Only two behaviours are required in a landing manoeuvre, first the altitude control to conduct the descent and secondly positioning control to perform the alignment with the helipad. Once over the helipad, the UAV should attempt to align itself with the helipad's centre while maintaining height. It should begin its controlled "fall" when the relative positions have been precisely determined while making modifications to the horizontal position in real-time whenever possible. Feng et al. (2018) [14] proposes that during the landing manoeuvre, the position controller attempts to maintain alignment with the helipad by making necessary modifications, while the altitude

controller progressively descends. Cabrera-Ponce and Martinez-Carranza (2017) [16] offer a similar design but implement two proportional control to separate the positioning controller into yaw and altitude controllers, allowing for more granular control of the UAV.

2.1.3 Parrot Bebop 2 Overview

From among different commercial UAVs that can be effectively used in conditions of safe educational practice (while adhering to minimum safety rules) and research (e.g., in the field of methods of autonomous control, techniques of estimation and fusion of sensory data, machine learning, optimization algorithms, or path planning methods), three very popular and cost-effective constructions deserve special attention: AR.Drone 2.0 [17], Crazyflie 2.0, and Bebop 2 [18]. The focus of this section is on the latter one, the Parrot Bebop 2 drone, which in contrast to the others, provides significantly longer flying periods and a medium-quality camera. Many proofs of the versatility of the Bebop have become available over recent years, for e.g autonomous drone racing [4], 3D modelling and image analysis [19]. The Parrot company's Bebop 2 is a low-cost, small-scale unmanned aerial vehicle. It is the next generation of ready-to-use, stable, and safe micro aerial class UAVs, which has piqued the curiosity of the UAV community following AR. Drone 2.0. All of this research work done on this drone is feasible because of its unique features and robust open-source software. Bebop 2 is a relatively small drone with just 0.328×0.382 m of diameter. The UAV weighs 0.5 kg during takeoff and has four motor units (4 KV BLDC Motor, 7500 – 12000 rpm), 6" propellers, and a 2700 mAh LiPo battery, allowing it to fly for up to 25 minutes without cargo. Offers a maximum horizontal speed of 60 km/h and a maximum vertical speed of 21 km/h. Bebop 2 can withstand roughly 63 km/h kilometres of headwind. The BusyBox Linux operating system is installed on the UAV, which has a P7 dual-core CPU Cortex 9 processor, 1 GB of RAM, and 8 GB of flash memory. An Extended Kalman Filter (EKF) is used to estimate the state of the Bebop 2 based on current data from the 9-DoF IMU and onboard sensors. 2.1 summarises, below, some of the dynamical parameters of the Parrot Bebop 2 drone, as well as their corresponding values, as discussed in 2.1.1.2.3. Controlling the drone can be done with the Parrot SkyController 2 device that can be used in autonomous flights using a GCS equipped with Wi-Fi. The most common options at the moment are based on the open-source driver bebop_autonomy, which was created natively for Linux Ubuntu 16.04 LTS and Robot Operating System (ROS) in the Kinetic Kame version, by Monajjemi et al. (2015) [21]. The package allows for communication and programming control commands for the Bebop 2 from the GCS. Some of these commands, illustrated in Table 2.2, are provided in ROS syntax.

	Symbol	Value	Unit
Total quadrotor mass	m	0.5	kg
Body inertia along x-axis	I_x	0.00389	$kg\ m^2$
Body inertia along y-axis	I_y	0.00389	$kg\ m^2$
Body inertia along z-axis	I_z	0.0078	$kg\ m^2$
Time motor constant	T_m	0.0125	s
Thrust motor constant	b_f	8.549×10^{-6}	$k\ g\ m$
Motor moment constant	b_m	0.016	m
Distance of propellers	l	0.129	m
Maximum propellers speed	Ω_{\max}	1475	$rad\ s^{-1}$

Table 2.1: Dynamical Parameters for Parrot Bebop 2 drone (adapted from [20]).

Action	Command
Takeoff	rostopic pub -once /bebop/takeoff std_msgs/Empty
Landing	rostopic pub -once /bebop/land std_msgs/Empty
Emergency	rostopic pub -once /bebop/reset std_msgs/Empty
Piloting	rostopic pub -once /bebop/cmd_vel geometry_msgs/Twist

Table 2.2: Principal control commands of bebop_autonomy for Parrot Bebop 2 drone.

2.2 High-Precision multi-rotor UAV Landing Systems

2.2.1 Landing Routine of multi-rotor UAVs

As civilization progresses towards the intelligent era, multirotor UAVs have a growing number of application possibilities. The UAV must be able to take off, manoeuvre, and land without the direct direction of a human operator in order to achieve the acceptable amount of autonomy needed by mission scenarios. While autonomous waypoint navigation works well when Global Positioning System (GPS) is available, and autonomous takeoff is not difficult, autonomous landing remains a tricky operation for all types of UAVs. Fully autonomous takeoff and landing would even allow for massive fleets of UAVs by making deployment and recovery more efficient without the need for human intervention. Positioning and navigation are two critical components of UAV landing, and usually the majority of current precision landing research focuses on landing in a known organised environment, such as a helipad or runway. Another consideration is the UAV limited payload, which affects the number of batteries it can carry, therefore the window of flight to perform the mission at hand. As a result, the vehicle must land frequently during brief periods of operation for battery replacement or recharging. Further to that, landing is one of the key goals of the quadcopter in applications such as delivery services, environmental research, and surveillance. These insights underscore the importance of the researcher's work in UAV autonomous landing, despite its extraordinary complexities.

2.2.1.1 High-Level Routine Overview

From a high-level perspective, a variety of elements must be addressed for a smooth landing, including the kind of landing, visibility, terrain type, wind disturbances. A conventional landing system incorporates a [Global Positioning System \(GPS\)](#) and an [Inertial Navigation Systems \(INS\)](#). Because [GPS](#) height measurement is incorrect, a close range sensor such as a radar altimeter or barometric pressure sensor is employed. There are two elements to landing that form its core, namely sensing and control. Camera vision seems to be the prominent sensing technology used by, specifically designed, landing controllers to determine the [Position and Orientation \(POSE\)](#) of the [UAV](#). This type of controllers can be of various types, ranging from simple linear control to complicated approaches including intelligent and hybrid control systems. In order to complete the landing routine in a safe and exact way, a landing planner is added to the landing control loop and once the landing job is initiated, the planner ensures that the vehicle follows a landing process that typically consists of the following phases [22]:

Phase	Description
1. Landing Area approach	The vehicle is driven horizontally to the landing place while maintaining its height.
2. Landing Target approach	Once the target has been identified, the planner will direct the UAV to approach it horizontally until the horizontal distance between the vehicle and the target meets a predefined criteria.
3. Descent over Target	When the UAV gets near enough to the target horizontally, it begins descending while maintaining its horizontal distance from the target.
4. Touchdown approach	When the UAV is vertically near to the target, it will initiate by lowering the vehicle throttle to allow for a rapid touchdown on the landing platform and disabling stabilised control.
5. Landing complete	The UAV will be disarmed once it is motionless on the landing platform, and the landing task will be finished.

Table 2.3: Typical phases of the landing process in a multirotor [UAV](#).

For simplicity's sake, the figure 2.4 below illustrates a block schematic of a general basic linear landing control system. The system is divided into four components: sensors & navigation system, guidance controller, flight controller, and [UAV](#). The [POSE](#) of the [UAV](#) is mostly determined by the sensors & navigation system. This data is combined for the flight and guidance controllers. To follow a desired trajectory, the guidance controller generates guiding orders such as changes in velocity, acceleration, and rotation. The guiding command is used by the flight controller to create the proper actuation instructions

based on the type of **UAV**, such as multirotor or fixed wing [23].

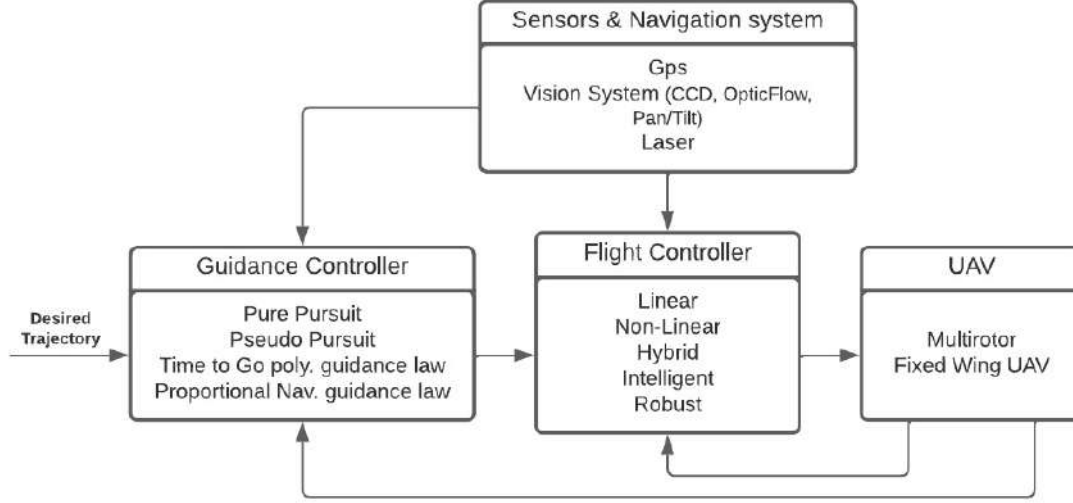


Figure 2.4: General basic linear landing control system (adapted from [24]).

2.2.1.2 The Problem with **UAV** Landing

The process of flying a **UAV** consists of several phases, including takeoff, ascent, cruise, descent, and landing. Due to significant hazards and dependability difficulties, most **UAV** autopilots feature autonomous take-off, for e.g catapult and manually launched, and cruise capabilities but limited autonomous landing capabilities. Accurate measurements, or optimal approximations, of the positions of the landing platform and the **UAV**, as well as robust trajectory tracking in the face of disturbances and uncertainty, are the key hurdles in autonomous landing [25]. As a result, landing precision is critical; otherwise, the **UAV** may crash. Aside from that, the manoeuvre must be completed in a restricted amount of time and space, demanding precision sensing and control approaches. The majority of **UAVs** rely on **GPS**-based state estimates for landing, however drift accumulates over time, rendering these systems useless in **GPS**-denied environments. As a result, a drone cannot consistently land at the same position and may seek to land in an unforgiving location, such as a neighbouring tree [26]. Nowadays, **GPS**-based positioning and navigation technologies clearly do not function effectively, since **GPS** precision is only approximately 10 metres. The **GPS** signal can also be interfered with and hindered for certain high structures in cities, causing inaccuracy rise and even signal loss, and it is incredibly easy to crash, which plainly cannot fulfil the demands of exact landing. The majority of the work conducted by peers and researched for this dissertation are vision-based techniques that also limit their application at night and/or in low-light conditions.

2.2.1.3 The Challenge of UAV Landing on Mobile Platforms

When the platforms are placed in motion, the landing algorithms for stationary platforms are unable to meet the landing performance criteria. Autonomous landing on a moveable and/or inclined surface is required for landing UAVs on boats in open waters or in unfamiliar situations, where an uninclined and/or stationary landing platform may not be accessible. In addition, quadrotors are load bearing, therefore landings must be as gentle as possible to ensure the safety of the load. When compared to other mobile landing platforms, a ship deck setting makes landing much more difficult. A ship's landing deck may be flat, but its orientation shifts with time. A multirotor UAV should be able to measure the inclination as it varies over time and securely lands. Scorpion [27] is a remotely controlled UAV with **Extreme Short Take-Off and Landing (ESTOL)** capabilities in use by the US Navy, with the ability to tilt the thrust vector without affecting the attitude of the aerodynamic surfaces. This design provides increased flexibility and resistance to turbulence and stall, which is especially useful for ship launch and recovery in severe seas. Das et al. (2012) [28] using four PING sensors, placed under each rotor, detects the orientation of the landing surface. Assuming the surface is inclined along the roll axis, in this scenario the average distance of the two sensors on the left side of the rotor will differ from the average distance on the right. To align the quadrotor with the landing surface, the algorithm alters the roll in incremental stages until the distances on both sides of the quadrotor are the same, indicating that the quadrotor is aligned. The simultaneous operation of the PING sensor and throttle reduction guarantees that the quadrotor lands on the landing surface while changing orientation. Rodriguez-Ramos et al. (2018) [29] focussed is research in **Deep Deterministic Policy Gradients (DDPG)** algorithm and created a flexible Gazebo-based reinforcement learning framework.

2.2.2 Detection Methods for Autonomous Landing

In this section it will be provided documentation regarding the researched autonomous landing control systems for this dissertation. The methods studied will be divided into two main groups, vision-based and sensor-based systems. A brief description of a traditional detection method, more easily illustrated in the case of vision-based systems. The UAV flies to the landing zone using a GPS-based navigation system, for example. It begins looking for the pre-determined pattern, in the example depicted in Figure 2.5, a "H" written in a circle, using its camera, the field of view of which is indicated by the red cone. The landing operation can begin after the respective locations are determined. On the other hand, the operation can be solemnly done by the use of sensor-based technologies. The usage of different sensors is always a decision that the researcher needs to make in order to complete the mission in the given scenario. Usually vision-based is paired with sensor-based, or vice-versa, in order to have a more robust and complete landing control system, but also have fail safe systems to ensure redundancy for UAV security.

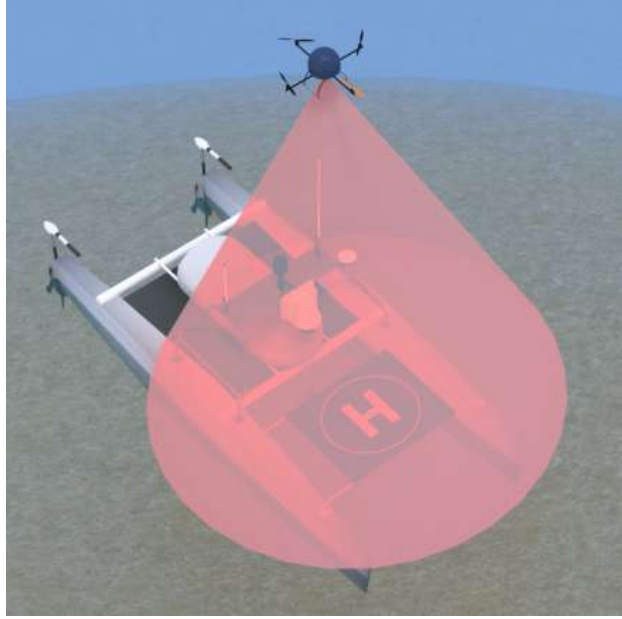


Figure 2.5: Vision-based most common method for landing MR-VTOL UAVs autonomously (adapted from [1]). The red cone represented depicts the camera mounted on the quadrotor Field of View (FOV)

2.2.2.1 Related's Work Methodology

Possible solutions to the problem highlighted in the previous section are presented from the perspective of a sensor-based autonomous landing system solution. Some researchers used altitude sensors [30] to assess the distance between the landing pad and the aerial vehicle. Ultrasonic sensors, used by Das et al. (2012) [28], are sensitive to the composition of the targeted surface and can, especially, provide incorrect readings for sound-absorbing materials. Infrared sensors are also commonly employed for range finding, however they can be incorrect when used outside or in areas with direct sunlight. Furthermore, as the relative angle of the targeted surface gets significant, the accuracy of these sorts of sensors tends to diminish.

Rydalch et al. (2021) [31] developed two methods for precise maritime landing of an autonomous multirotor aircraft based on Real-Time Kinematic (RTK) Global Navigation Satellite Systems (GNSS). The purpose of his study was to create a solution that is particularly suitable for the sea and does not require computer vision or sophisticated control and estimation systems. The first approach, known as the RTK-Localized Method (RLM), use RTK GNSS data to locate a maritime vessel and carry out the landing. RLM was demonstrated in hardware outside and landed on a physically recreated boat known as a mock-boat with an average landing inaccuracy of 9.7 cm. The landing-boat was triggered to move like a boat and have a forward velocity of 2 m/s. This approach demonstrated that precise landings are achievable when RTKGNSS is used as the primary way of locating a marine vessel. The localization was performed without the assistance of non-RTK sensors or an estimator, but lacked complete attitude estimation and measurement

smoothing. The second technique, known as the [RTK-Estimation Method \(REM\)](#), gives a more thorough and resilient solution, especially at sea. It has a landing pad estimator that combines readings with a dynamic model of a marine vessel. The base estimator is made up of an [EKF](#) and a complementary filter that calculates the relative location, attitude, and velocity of a moving object. Because the only sensors used to track the marine vessel were [RTK](#) receivers and inertial measurement units, the procedures outlined differ from standard methods. Most current solutions rely on computer vision, which can fail in bad lighting, in the presence of ocean spray, and in other situations. Under such situations, the proposed solutions by Rydalch and Kent [31] do not fail. Using readings from satellites thousands of kilometres away, the two approaches can land on quite small landing pads at sea, on the scale of 1 m by 1 m.

Possible solutions to the problem highlighted in the previous section are presented from the perspective of a vision-based autonomous landing system solution. The vehicle must locate the landing platform or landing zone, which is generally identifiable from the rest of the ground, that's why cameras in this context stand out. A variety of vision-based control algorithms are available for helipad detection, tracking, and landing, both indoor and outdoor. Cameras are sensors with the ability to perceive the surroundings and are often lightweight, so autonomous flight of a [UAV](#) can be accomplished by employing computational methods based on visual analysis of video footage collected from the onboard camera [16]. In the feedback control loop of an autonomous landing system, computer vision is employed. The use of vision in the control loop is particularly appropriate for circumstances when the landing pad is in an uncertain position or is non-stationary. Traditional vision-based target tracking and landing approaches are based on object recognition using edge detection algorithms. Although vision is a natural sense for object recognition and landing, it can only detect changes caused by applied forces rather than the forces themselves. Visual processing is really highly effective for solving autonomous navigation and landing systems at a predefined place, but it must be paired with sensors to measure height above ground. As a result, vision-based approaches are combined with traditional control techniques to provide a good and strong landing design [32].

In [23] and [33], for example, the authors suggest an autonomous landing system based on a mix of visual data processing and [GPS](#). The [GPS](#) gives location, velocity, and a compass to determine direction, while the vision system uses the camera to locate the landing zone. This combination achieves automated landing with great precision. When just a monocular camera is utilised, visual tracking cannot be used directly to retrieve the 3D location of a marker. However, if the size of the marker is known, depth may be estimated and used inside a controller for autonomous landing [32]. Pluckter and Scherer (2020) [25] proposed technique eliminates these issues by following the established take-off route back to its starting point with a monocular fisheye camera, allowing them to land the drone properly. When compared to outdoor landing, indoor landing implies landing in a controlled setting with less external disturbances. For Indoor landing using

visual infrared tracking, Xuan-Mung et al. (2020) [22] employ a fixed infrared camera, with the control algorithm running on an inbuilt microprocessor, to stare downwards and track a pattern of infrared dots. At a high frequency, the integrated circuit supplies the pixel position of each spot and the calculated posture is utilised to control the vehicle motion in various integrated PID control loops.

2.3 Gazebo Simulation Environment for multi-rotors UAV

2.3.1 Gazebo Overview

2.3.1.1 Description

Gazebo [34] began to be developed in 2002 at the University of Southern California as part of Nate Koenig's Ph.D. under the wing of professor Dr Andrew Howard under the Apache 2.0 licence. Curiously, most Gazebo' users use it to simulate indoor environments in a reliable and resource-friendly way, despite the intended goal of answering the need for a high-fidelity simulator for robots in an outdoor environment. Development continued and in 2009, John Hsu, integrated Robot Operating System (ROS) and the PR2 (robot designed by Willow Garage) into Gazebo, branding it to become one of the primary tools in the ROS community user's arsenal. In 2012, the Open Source Robotics Foundation (OSRF) made the last move and transformed Gazebo into an independent project that continues to develop with support from the community. One of the reasons for its great success is the open-source core, enabling the source code to be available to all users and be developed in a collaborative public form.

2.3.1.2 Architecture and ROS Integration

Gazebo uses a distributed architecture with separate libraries for physics simulation, rendering, user interface, communication, and sensor generation. The package includes two main executables that allow the user to run the simulator: **GzServer**, the core, and **GzClient**, the user interface and simulation controls, depicted in Figure 2.6.

From the start, Gazebo's major feature was the ability to easily create new elements in the simulation, so they maintain a simple Application Programming Interface (API) that resides on third-party libraries that handles the simulation physics and rendering at a low-level (Figure 2.7). The World depicts the set of all models and simulation factors, from gravity and friction to lighting and wind. The Model, in this case, the quadrotor UAV, is composed of body and joints, as for all robots, paired with all sensors, flight controllers and simulated virtual camera feeds, packed together with visual meshes for graphical purposes. This makes Gazebo perfect for integration with ROS.

Robotic Operating System, more commonly known as ROS, is a licensed robotics system designed to control and design robotic components from a Linux-running machine. The basic architecture relies on the publish/subscribe messaging paradigm, used widely

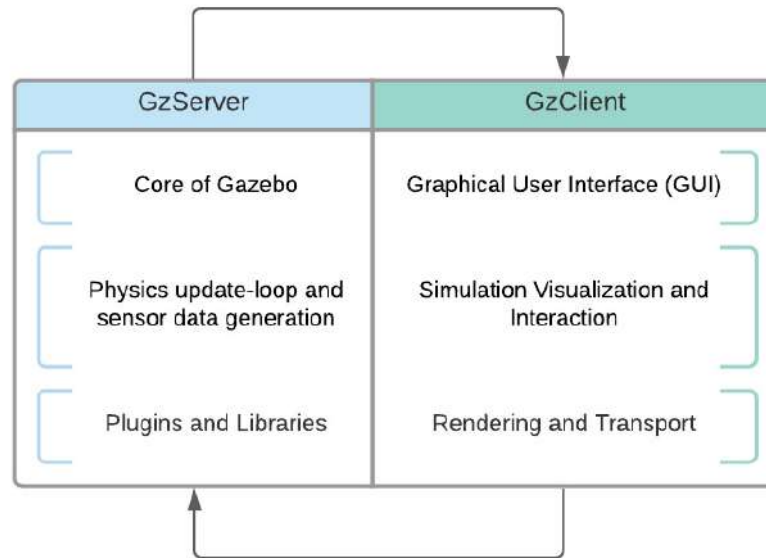


Figure 2.6: Illustration of the main processes of Gazebo (adapted from [35]).

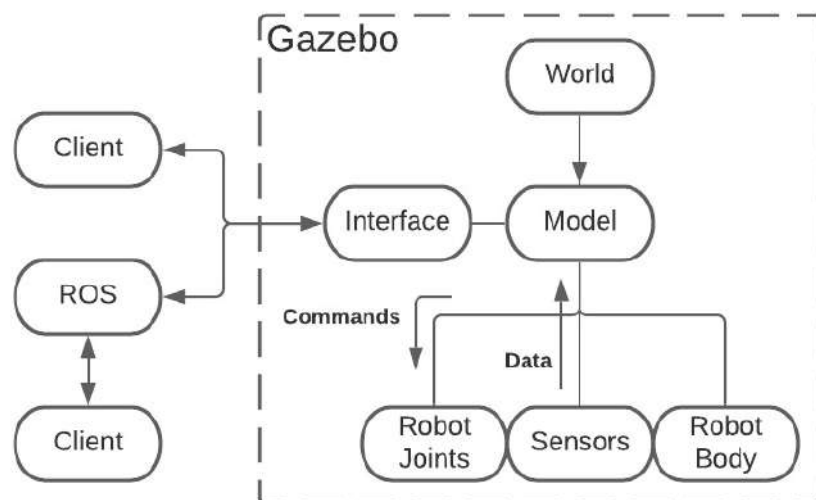


Figure 2.7: General environment structure in integration with ROS (simplified from [34]).

in our modern world, that forms the communicating interface between the independent nodes that assemble [ROS](#). Those messages are then delivered to a special inbox, a topic, that can be opened by all and any number of other nodes. This way the [ROS](#) nodes don't need to be running on the same machine or even written/designed with the same architecture, this feature is what makes it the go-to [Operating System \(OS\)](#) for robotics applications.

Looking at a simple example, in Figure 2.8, of a quadrotor [UAV](#) with a camera facing downwards, composed of two nodes and one topic.

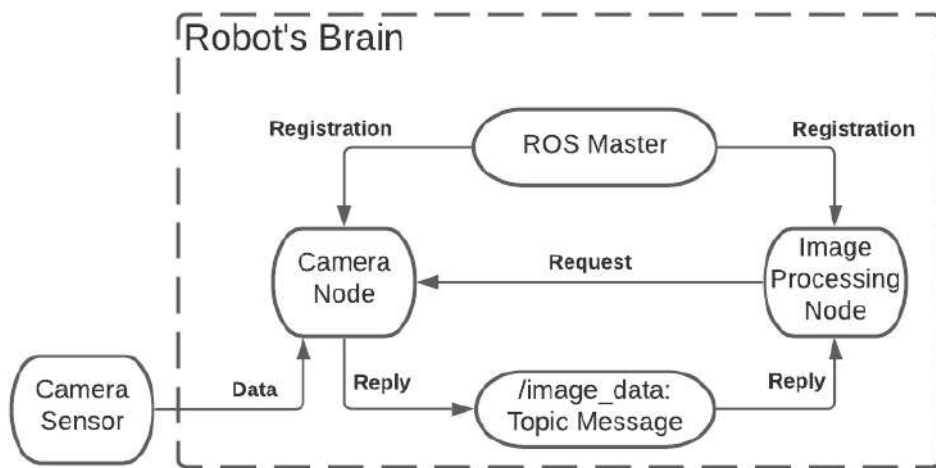


Figure 2.8: Schematic of a simple Drone + Camera communication over [ROS](#).

Segmenting the drone we have:

- Master, the core of [ROS](#), allows for all other [ROS](#) Nodes to find and talk to each other.
- Camera Node, that handles all the communication with the camera sensor.
- Image Processing Node, which handles image data processing.

Turning the system on, makes all Nodes register with Master as if Master is a yellow-pages book where all Nodes come to search for where to and what messages to send. While registering with Master, the Camera Node states that it will publish to the Topic, in our drone `/image_data`, some message of type, in our case `image_data`. On the other hand, the Image Processing Node states that it will subscribe to the Topic `/image_data`, and messages of type `image_data`. This way every time the Camera Node receives some information from the Camera Sensor, it sends a message to `/image_data`, making the information available to the Image Processing Node that receives the message, this communication is made essentially over TCP/IP. Note that it is also possible for the Image Processing Node to request information at any time from Camera Node, that's why [ROS](#) implemented Services. These services are registered at startup by the Node when registering the messages and Topics it will publish/subscribe to [35].

2.3.1.3 Context of simulation in multi-rotors UAV experiments

Conducting multi-rotors UAV outdoor experiments in the context of this dissertation requires ideal weather conditions and compliance with local aviation regulations, the autonomous vessel needs to be ready and deployed at sea in a safe controlled area, this uses a significant amount of resources, not to mention the risk of testing with equipment and platforms in the physical world [36]. It is, therefore, preferable to evaluate scenarios in simulation prior to experimentation whenever possible.

“In the relatively short life span of Gazebo, we have seen adaptation and contributions from other universities and creative uses of Gazebo as not just a simulator but also a safety device.” [34]

In order to do so, the quadrotor simulator needs to be able to provide realistic models for dynamics, wind, and sensor noise, and to balance accuracy with speed, enabling (near) real-time performance. Not only this but It needs to verify component integration and evaluate their performance under lots of different scenarios. The value of simulation UAVs in a 3D environment to test algorithms stands out as an indispensable stage in designing multirotor applications. The array of quadrotor simulators available today are huge and Gazebo may not include as many features as other engines available, but it fits the requirements that make it a top-level robotic simulator, that in conjunction with the right libraries and plug-in's, courtesy of ROS integration, becomes a high-fidelity multi-rotor UAV simulator.

2.3.2 Software-in-the-loop (SIL) Simulation

2.3.2.1 Fundamental Concepts and Major Features

Software-in-the-Loop (SIL) testing methodology is part of the verification phase of the Model-Based Design approach [37]. Comes opposing Hardware-in-the-Loop (HIL) and consists of the technique of testing a component of a system, or even the complete system itself, relying only on software. The need to have a real piece of hardware in your control and feedback loop is removed by adding software simulating physical hardware parts in the loop. SIL gives engineers the utmost control of their system by enabling detection of system-level defects or bugs before other testing methodologies, significantly reducing the costs of later stage troubleshooting. As an example, SIL provides the freedom to add a gust of wind, or in an extreme case rotor malfunction, to the precise moment where the drone is just centimetres away from the ground, allowing it to monitor the outputs and behaviour of the system given those input signals.

2.3.2.2 RotorS - SIL Simulation ROS packages for multi-rotor UAVs

ROS, relying on the integration with Gazebo, provides a modular Micro Aerial Vehicle (MAV) simulation framework, the RotorS. This allows developers to design and test

applications that can then be run on the real platform without any changes. A simple architecture of the RotorS simulator and package structure is shown in Figure 2.9 and Figure 2.10, respectively. Gazebo handles the simulation of UAVs components through

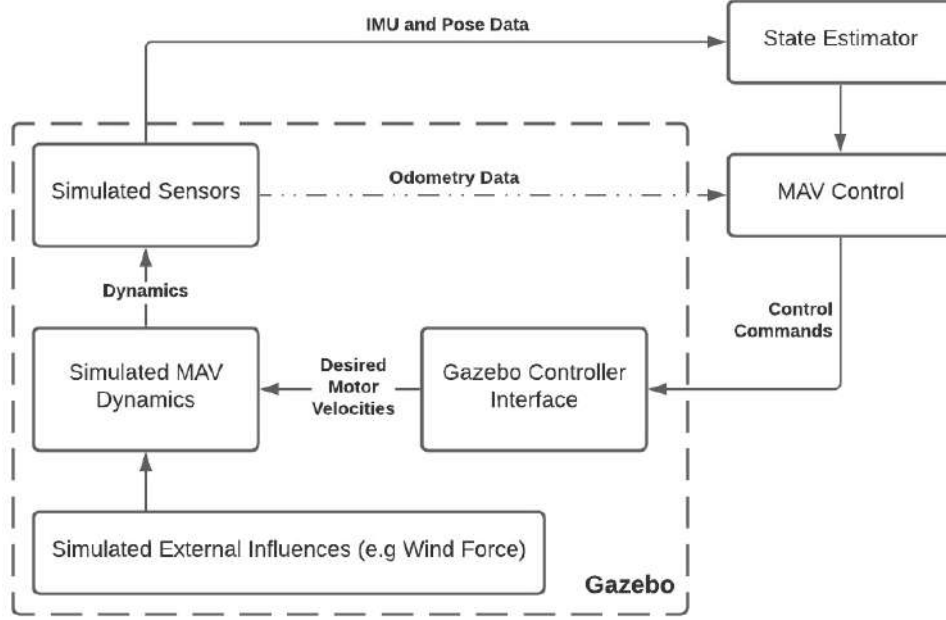


Figure 2.9: Simple Architecture of RotorS Simulator (adapted from [38]).

Gazebo plugins and physics engine, but to be able to simulate a UAV it is necessary to put together a modular way of depicting it with accuracy. So according to RotorS developers, a MAV consists of a body equipped with a fixed number of rotors in specific locations, and sensors attached to it. All rotors have their individual motor dynamics and aerodynamic effects. Sensors that form the core of almost all robots are attached to the body and simulated by Gazebo, either it be for e.g. an IMU, an odometry sensor or a camera sensor. Noise models for the sensors were implemented to ensure near real-world approximation. Regarding the control strategies, a simple implementation of a geometric controller is provided, allowing several degrees of control, e.g. angular rates, altitude, or position. Keep in mind that the implementation of more advanced and case-specific control strategies is encouraged by the team [38]. In order to get stable and robust multi-rotor UAV flights, the key element is knowing the UAV state. To tackle this a state estimation block was added to the loop to ensure that information about the state of the UAV is delivered at a high rate. Another advantage of using SIL is that we replace the crucial process of state estimation in real UAVs, with a generic (idyllic) odometry sensor. This is done by fusing IMU measurements with generic 6 DoF pose measurements, obtained from, for e.g. visual-odometry, visual SLAM systems (Intel RealSense Camera Sensor) or laser rangefinder techniques. By combining both types of measurements, the RotorS team developed an almost drift-free, high-rate and low delay estimation of state of a UAV, or MAV. All of this can be accomplished because position, orientation, linear and angular

velocities are provided by a Gazebo plugin. From a filesystem point-of-view (Figure

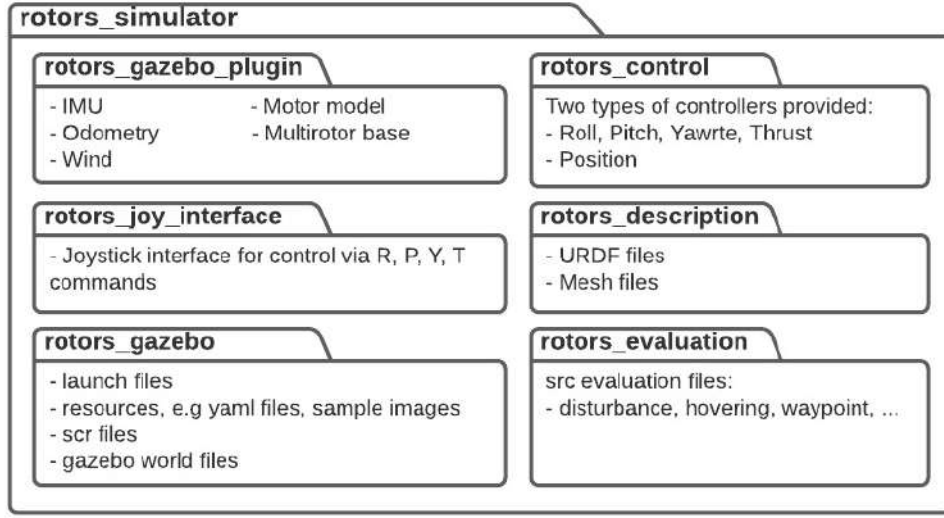


Figure 2.10: Package Structure of RotorS Simulator. (adapted from [38]).

2.10), the RotorS simulator is divided into six packages, all with a specific purpose for the overall simulation environment. Two packages for UAV control, `rotors_control` the actual simulated multi-rotor controller, and `rotor_joy_interface` an interface that enables user-control based on a joystick. Two packages for Gazebo simulation integration, the `rotors_gazebo` and `rotors_gazebo_plugin`. And finally, `rotors_description` package for UAV model description, including body and rotor dynamics and sensor configuration, and `rotors_evaluation` a simple package with validation files.

2.3.3 BebopS - Parrot Bebop 2 + RotorS

BebopS [20] is an extension of the ROS package that we discussed in the section 2.3.2.2, RotorS. It was developed by Giuseppe Silano, Pasquale Oppido, and Luigi Iannelli from the University of Sannio in Benevento, Italy, under the Apache 2.0 licence. The objective is to model, develop and integrate the existing ROS package with the Parrot Bebop 2 [18] and the Gazebo simulation environment. The repository [39] was developed with the aim of designing complex control systems for the Bebop 2 drone, although the author states that it can be used as a template for any other multi-rotor UAV controller implementation. Regarding the flight control system, depicted in Figure 2.11, a common cascaded control architecture was chosen with standard **Proportional Integrative Derivative (PID)** controllers, the literature standard solution for quadrotor UAVs controller designs [40]. The position controller computes thrust and attitude needed to go from the measured UAV position to the desired position, with the desired heading orientation. Since the program is designed following the SIL methodology the platform allows us to detect and manage instabilities of the drone that otherwise might not arise when following a HIL methodology (Matlab/Simulink simulation).

“In the relatively short life span of Gazebo, we have seen adaptation and contributions from other universities and creative uses of Gazebo as not just a simulator but also a safety device.” [34]

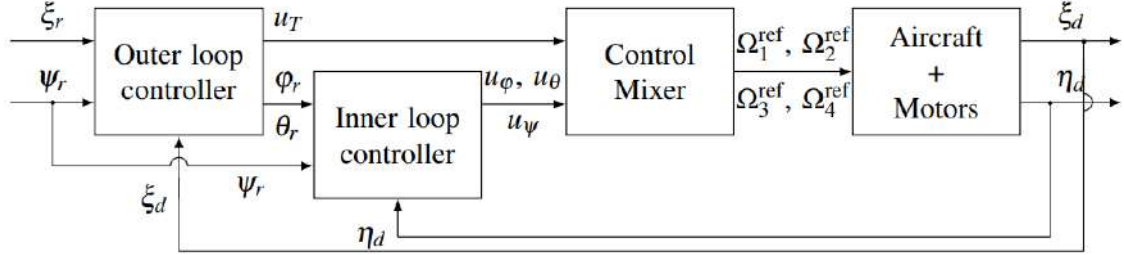


Figure 2.11: BebopS Control Scheme (obtained from [20]).

The outer loop controller, or the position controller, uses the UAV measured position to compute the desired thrust and the attitude, $\eta = \begin{pmatrix} \varphi & \theta & \psi \end{pmatrix}^T$ (Euler angles roll, pitch, and yaw) used to describe the drone’s body orientation using the ZYX convention [41]. On the other hand, the inner loop controller, or the attitude controller, uses the UAV measured attitude to compute the model inputs that should be actuated to achieve the desired attitude. Finally, the control mixer obtains the commanded motor velocities by inverting the controller outputs, which will be later used as inputs for the UAV dynamical model, discussed in section 2.1.1.2.3.

2.3.4 Parrot’s Sphinx Simulation

Sphinx [42] is a simulation tool designed to meet the demands of Parrot engineers as an internal tool for the development and automatic testing of working drones. It offers SIL simulation in which the drone and its surroundings are fully virtualized. The drone’s physical and virtual surroundings are simulated using the open-source Gazebo multi-robot simulator. The firmware for the Parrot drones is executed in a separate environment from the host system. The simulator is built on a Gazebo version 7 completely modified by the team at Parrot to run exclusively for Parrot drones that’s why it supports various commercial products, e.g Bebop, Bebop 2, Disco and ANAFI series. Sphinx is capable of multiple drones in the same simulation and can be used to fine-tune control algorithms and perform manual validation tests.

The simulator relies on a simple three-part architecture, represented in Figure 2.12. The Gazebo 7 custom-build, with plugins designed exclusively for Parrot drones. GzServer, previously discussed in section 2.3.1.2, simulates a world packed with models and their physical interactions, system’s world, model and GUI plugins, and an optional GzClient for real-time OpenGL visualisation. The drone firmware is Parrot’s firmware that is loaded on the go with no need for previously installed packages, this feature permits to lower the resources needed in the host system to simulate while bringing the need for a stable internet connection each time the simulator is run [42]. Parrot made it possible

simulation. Several studies have described examples of this kind of simulator, such as AirSim [46], Morse [47], jMAVSim [48], X-Plane [49]. Regarding these kinds of software, a formal comparison between them is presented in a few noteworthy and relevant studies.

Pitonakova et al. (2018) [43] provides a ranked evaluation of V-REP, Gazebo, and ARGoS, on the simulator characteristics and performance on a vast array of comparison criteria. The authors compare the characteristics of the simulator in regards to their built-in capabilities, robot and model availability, programming methods, and [User Interface \(UI\)](#), while the performance metrics stay simple and the evaluated criteria are the real-time factor, the amount of [CPU](#) usage, and the amount of memory usage. However, this study does not provide a formal comparison between the multi-rotor [UAV](#) simulators presented in sections 2.3.3 and 2.3.4, nevertheless providing useful insight on Gazebo's characteristics and performance metrics, the base of BebopS simulator. In conclusion, the research states that Gazebo has many of the features of a more complex simulation, for e.g multiple physics engines, the ability to interact with the world during simulation, and, most crucially, visual modification and optimization, but at the same time [UI](#) and default robot models are substantially more straightforward making it a lower resource-consuming option. However, the trials revealed that Gazebo's usability is lacking. Another concern is the interface, which has a lot of flaws and does not adhere to standard practices. Finally, problems with installing dependencies for Gazebo and several of its third-party models have been reported. In conclusion, the findings suggest that while these difficulties are not necessarily serious on their own, when combined, they might have a detrimental influence on a study effort.

Ebeid et al. (2018) [48] presents a survey on open-source flight controllers designed for [UAVs](#) based on a web-based survey of 20+ flight controllers. The authors evaluate and contrast the features, specs, licencing kinds, and other characteristics of each flight controller and open-source simulation system, including Gazebo.

Hentati et al. (2018) [50] presents a formal comparison of specifically designed multi-rotor [UAV](#) simulators, including Gazebo on criteria relevant to this dissertation case study. Besides that, the authors also provide an extensive comparison of ground control station software applications that run on a ground-based computer. The team concludes that for their case study, which consists of exchanging MAVLink messages, FlightGear [51] and QGroundControl [52] prove to be the focus of future research. It is important to note that [ROS](#) Integration is valued as a lower requirement criterion.

To the extent of my knowledge, there has never been a systematic and meaningful comparative assessment of BebopS and Parrot-Sphinx multi-rotor [UAV](#) simulators. Nonetheless, the Table 2.4 below provides an informal comparison of the simulators. Remember that this is the culmination of the study conducted as part of this dissertation.

Criterion Collection	Advantage	Explanation	References
ROS Integration	BebopS	Based on RotorS, plus Sphinx offers zero integration.	[20]
World Modelling	BebopS + Sphinx	Both rely upon Gazebo system and world engines (plugins).	[20, 42]
Robot Model	BebopS	Based on a more generic and open-source engine. No custom builds to create incompatibility, unlike Sphinx.	Based on Previous Work
Customization	BebopS	Both offer customization, but because BebopS is built into OS+Gazebo the possible range of tweaks becomes endless.	No reference
Multi-UAV Support	BebopS	Both offer support, but BebopS multi-UAV simulation is much more complete.	[20]
CPU Use	Sphinx	In this case, only the simulation is run on the host machine because the firmware service takes care of the rest.	No reference; Testing conducted in Previous Work
Real-world proximity	Sphinx	Uses framework that comes from the real drone framework.	[42]
Ease of Development	BebopS + Sphinx	Both have strong points for different scenario missions.	Based on Previous Work

Table 2.4: Comparison between BebopS and Sphinx Simulator.

PROPOSED MODEL

This dissertation offers a cooperative vision-based landing system for **MR-VTOL UAVs**, with the goal of addressing some of the challenges that typical pattern-based systems confront. Rather than landing on its own, the **UAV** receives and relies on information provided by the helipad. The approach and algorithms created were developed with the aim of constructing an autonomous detection and landing system using machine learning algorithms. Section 3.1 presents a basic overview of the developed software architecture as well as the physical system that was used as the foundation. Section 3.2 explores all the techniques and general procedures used to find and identify the **UAV** in imagery captured by the helipad's ground-to-sky camera. The tracking algorithm used to enhance the real-time **UAV** position computations together with the calculations in relation to the desired target position are described in Section 3.3. To conclude this chapter, section 3.4 offers a potential high-level strategy for directing the **UAV** to its goal while ensuring a steady descent and successful landing.

3.1 General Overview

Two essential components of the system are an **MR-VTOL UAV** and a helipad. The helipad may be a stationary platform or it may be put on top of a moving robot that serves as a mobile base for the **UAV**. For the purposes of this dissertation, the system was created to accommodate a marine heaving platform. Both the helipad and the **UAV** are active elements that may gather and analyse sensory data. The **UAV** has **GPS** and an on-board **Inertial Measurement Unit (IMU)** for estimating outdoor pose. An upward-looking camera with its optical axis perpendicular to the ground is located in the middle of the helipad, as depicted in Figure 3.1. Thanks to the camera sensor array, the helipad also features an **IMU** that effectively transforms it into a smart element capable of sensory

data collecting and information processing. As a result, a large portion of the calculation is transferred away from the UAV and to the helipad landing system, allowing the UAV to preserve computing resources and battery power. Both modules have a high degree of reliability in their capacity to connect and interact with each other.

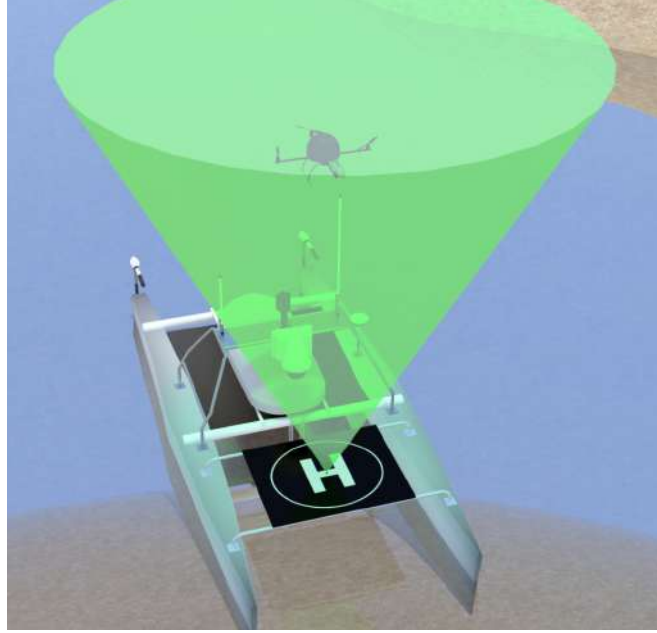


Figure 3.1: Proposed technique physical system, based on vision-based GCS method for landing MR-VTOL UAVs autonomously (adapted from [1]).

The overall proposed model structure is depicted in 3.2 is organised into three primary processes that run entirely on their own: **UAV detection**, **Middleware Communication Interface**, and **Position Control** actuator. When detecting UAV instances, the **Detection** element of the UAV Detection module uses a single-stream RGB image and a weights file as input and returns the corresponding bounding boxes for each instance. This information and the input image from the model are both given to the **Overlap** section, which overlaps the data and outputs a single-stream RGB image with the starting image already filled with coloured boxes surrounding the detected UAV. In the **Middleware Communication Interface**, the **Landing Target Computation** section computes the relative distance from the UAV to the target landing site (the center of the image) using the detected UAV bounding box position, in pixels relative to the image, and estimates the UAV centre of mass and size to compute the pixel to meter ratio. This data, together with the annotated input image, are sent to the **Overlap** section, which overlaps the current bounding boxes with the UAV's centre of mass, the target's location, and other crucial data for administration and control of the UAV, like the battery and rotor states.

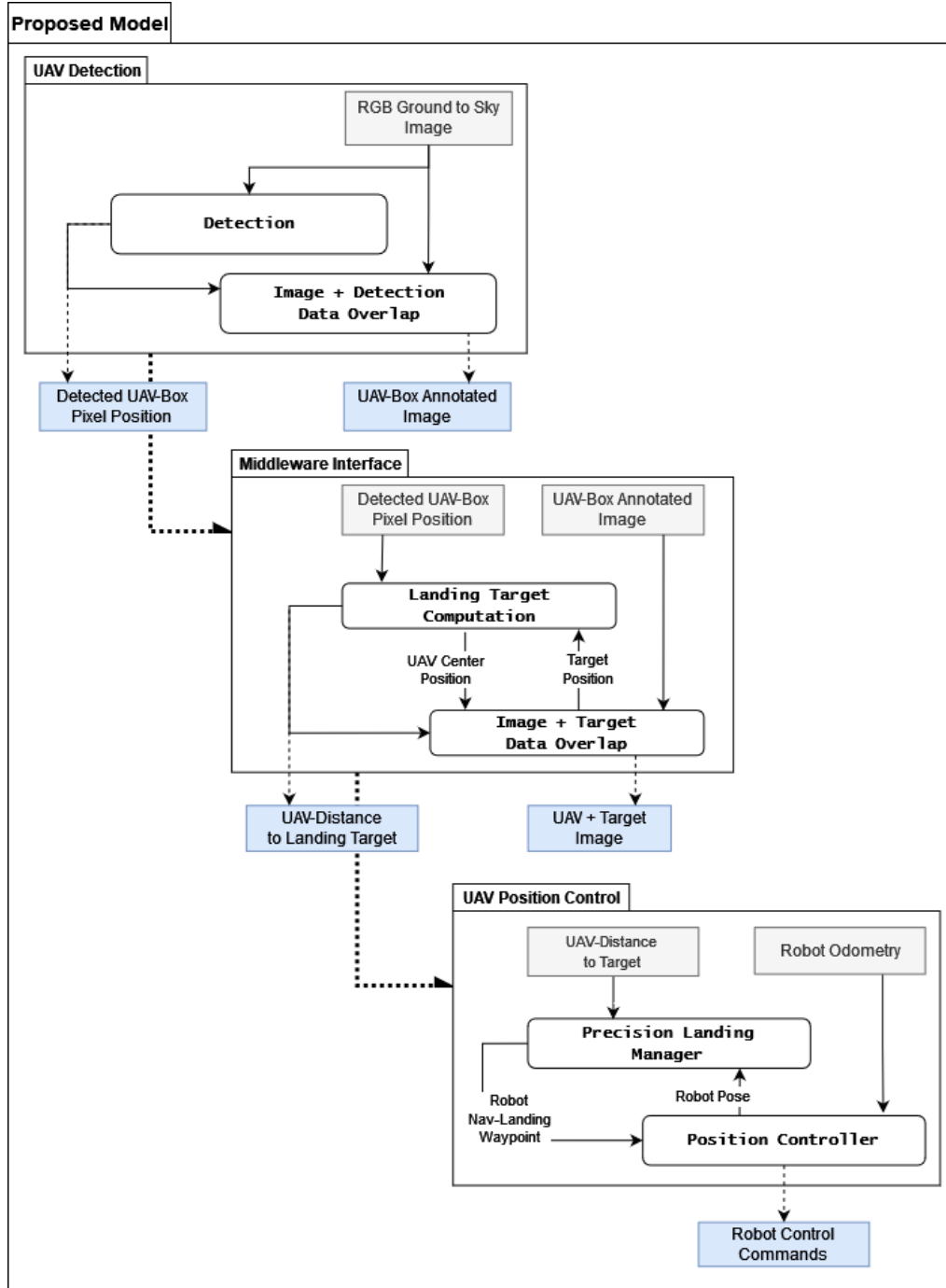


Figure 3.2: The proposed model structure is organised into three primary processes: **UAV detection**, **Middleware Communication Interface**, and **Position Control** actuator. The Detection element of the **UAV Detection** module uses a single-stream RGB image and a weights file as input and returns the corresponding bounding boxes for each instance. In the **Middleware Interface**, the Landing Target Computation section computes the relative distance from the **UAV** to the target landing site using the pixel to meter ratio. Then the Overlap section, overlaps the data and outputs a single-stream RGB image with the detected **UAV** and useful data. In the **UAV Position Controller** actuator commands are sent to the **UAV** depending on the nav landing waypoint assessment to perform the landing.

For user visualisation, the final image is published in a single-stream RGB topic. The **Precision Landing Manager** component in the **UAV Position Controller** gets the real-time robot (drone) pose and the **UAV** distance to landing target to ensure that the desired behaviour is obtained. To evaluate all the data and generate a conclusive and more accurate estimate of the **UAV**'s pose, the data from these sources are combined and a Nav landing waypoint is then returned after validating the altitude and ensuring that the target's distance in 2D coordinates is within the precision landing criteria. In order to conduct the landing, the **Position Controller** section then receives this information along with robot (drone) odometry and sends actuator commands for the **UAV** depending on the nav landing waypoint. In order to guarantee that the calculated desired behaviour and the actual behaviour are identical, the **Precision Landing Manager monitors** the actuator commands and responses.

A file holding the weights for the network trained using the Darknet framework and **YOLO**, a real-time object detection system, must also be provided as input to the model. This weights file is crucial to these operations because it gives the network the values for the connections between neurons, effectively dictating how the network will behave. For instance, the weights file created by using the Parrot Drone to train the network is only suitable for detecting the Parrot Drone. It would be essential to use a different weights file to categorise an image using a different **UAV**. Section 4.3 goes into further detail about the procedure used to collect these input files. The duration of the process should be considered because the suggested system is meant to assist in an autonomous landing. Therefore, the proposed algorithms should be frugal to avoid adding detection delays that could affect the system's overall robustness.

3.2 UAV Detection

The **UAV Detection** module utilises a single image as its input along with a single file containing the weights of the relevant network. The **Detection** part and the **Overlapping Data** section are two additional sections that can be used to structure this process. The detection section is in charge of locating potential **UAV** occurrences in the input image that was received. On the other hand, the overlapping section is in charge of rearranging all the components that were delivered to the preceding section with the image and outputs it. The detection of possible Parrot Bebop 2 drones in the sky, is performed using an open source neural network framework and a real-time classification network.

In the Darknet detection phase, the **YOLO** neural network model runs over the input image in inference mode using the specified weights to identify all pixels that belong to the **UAV** instance and create bounding boxes around the area of interest.

3.2.1 Detection

Darknet [53] is an open source **Convolutional Neural Network (CNN)** framework written in C and CUDA that acts as a backbone for the **You Only Look Once (YOLO)** [54], a state-of-the-art, real-time object detection approach used in this dissertation. Humans can quickly identify the objects in an image, their locations, and their relationships just by looking at it. Human vision is quick and precise, enabling us to carry out difficult activities like driving with little conscious effort. Fast, precise object detection algorithms would open the door to general-purpose, responsive robotic systems, and the ability for assistive devices to transmit real-time scene information to human users. The complexity of current detection systems necessitates the training of each component separately, making them slow and difficult to optimise.

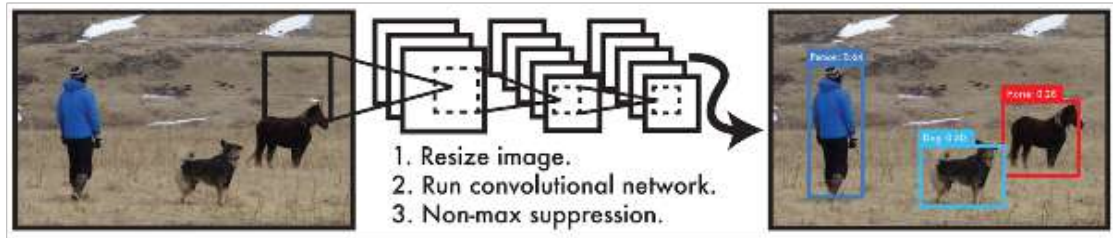


Figure 3.3: **YOLO** detection system (taken from [55]).

YOLOv3 is a real-time object detection algorithm based on neural networks, illustrated in 3.3. **YOLO** stands for "You Look Once" because object detection only needs to happen after one forward cycle over the network. Using **YOLO** to process pictures is easy and uncomplicated. The system first reduces the size of the input image to 448x448, then employs a single convolutional network on the image, followed by a thresholding step based on the model's confidence. The following chapter's section 4.2 will demonstrate how **YOLOv3 CNN** uses Darknet-53 as a backbone network for feature extraction. Along with class predictions, it also uses a bounding box to pinpoint where an object is located within an image. The term "class" in this context refers to the object's label in the output, for as the label "car" for an object that was identified. The object in the image is enclosed by a rectangular shape known as the bounding box. It is faster and more accurate because it only uses one deep **CNN**, making it a better choice for real-time object detection. **YOLOv3** is also a generalizable model, which means that it functions well with new types of images inside a class.

Darknet is a convolutional neural network with 53 convolutional layers. A **CNN** is a type of deep neural network that is mostly utilised for tasks like object detection and picture categorization. **CNN** has a focus on finding patterns in data. In particular, Darknet-53 is beneficial for studying image data. Convolutional layers exist in **CNN**. Each

convolutional layer filters the image differently. These filters take the image's features and extract them. The filters in the first layer identify edges, curves, and other geometric elements in the image. The deeper layers and convolutional layers recognise features specific to an object and whole objects, respectively. Darknet-53 is a combination of [Residual Network \(ResNet\)](#) and deep Darknet-19 (19 levels). [Residual Network \(ResNet\)](#) introduced skip connections, where the output of one layer is provided as an input to the following layers by bypassing some layers, to improve the training of a deeper neural network. The architecture of Darknet-53 is further explained in section 4.2.1. Darknet-53 serves as the backbone network for [YOLOv3 CNN](#). There are 106 completely convolutional layers in the [YOLOv3 CNN](#), 53 of which are stacked over the Darknet-53's 53 layers. This block outputs four arrays:

- **Class Name** with class string label identifiers;
- **Class IDs** with class integer identifiers;
- **Confidence Scores** with float probability percentage of detected instance belonging to certain class;
- **Bounding Boxes** regarding data that give information on the position and size of the bounding box in pixel coordinates surrounding the detected [UAVs](#).

The confidence score's value ranges from 0—which indicates the absence of an object—to 1—which indicates the presence of an object with absolute certainty. As seen in 3.4, [YOLOv3](#) divides the input image into $S \times S$ grid cells for object detection. It functions by predicting the class and bounding box for items that are present in each grid cell. The algorithm determines the bounding box coordinates and class probability for each cell. The confidence score for each estimated bounding box is then determined. Two step filtering is used once the confidence score has been computed. It starts by eliminating bounding-boxes whose confidence score is below the threshold. Then, predictions with lower-class probabilities are removed using non-max suppression. Five predictions make up each bounding box: x , y , w , h , and confidence. The centroid of the box in relation to the boundaries of the grid cell is represented by the (x, y) coordinates. Relative to the entire image, the width and height are predicted. In the developed system, detection is modelled as a regression issue, depicted in 3.4. It creates a $S \times S$ grid out of the image and forecasts B bounding boxes, confidence in those boxes, and C class probabilities for each grid cell. These forecasts are represented by the tensor $S \times S \times (B * 5 + C)$.

3.2.2 Image + Detection Data Overlap

The Image + Detection Data Overlapping module is in charge of overlaying the image and the bounding boxes to arrange them in a way that the user can understand after receiving them. It is feasible to convert the pixel sizes to metres and determine the regions of the identified occurrences by using the [UAV's](#) altitude and the helipad's camera focus

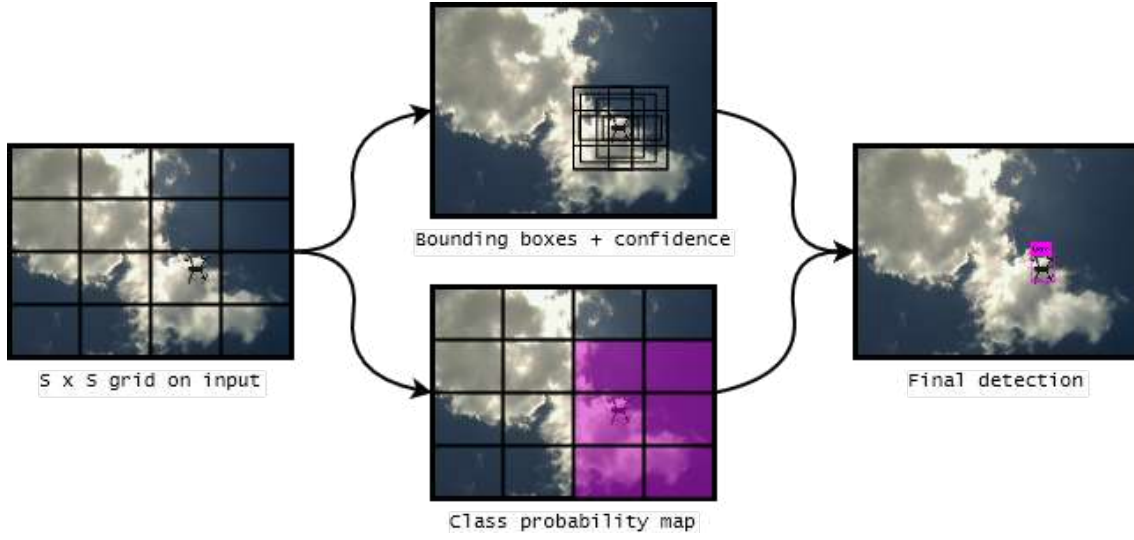


Figure 3.4: The Detection Model.

length. This section uses the input image and overlaps the computed bounding boxes with it. Every box in the image is marked, and inside each one, every pixel identified as a UAV object is given a different random colour. Additionally, the class name *uav* and the percentage of detection confidence are printed next to each occurrence.

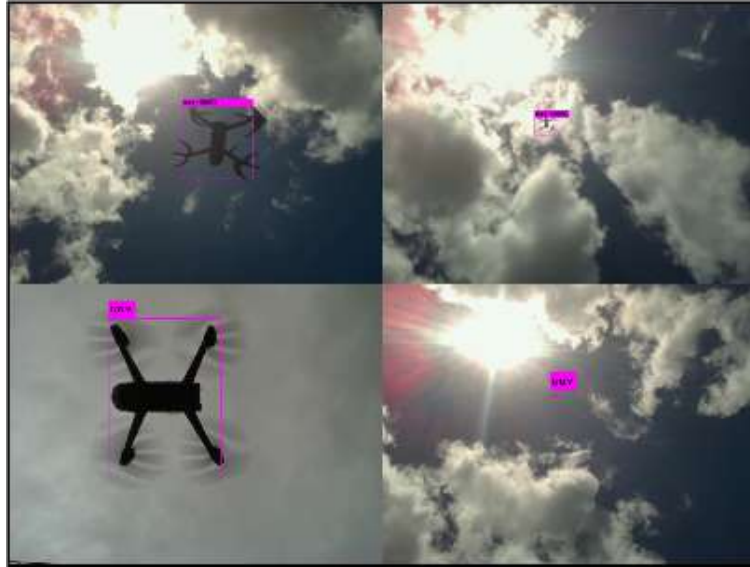


Figure 3.5: The Overlapping processes. The bounding boxes, classes, and scores arrays are sent along with the input image that has been modified for visualisation. The output of the model shows these arrays over the image.

3.3 Middleware Communication Interface

The developed algorithm to determine and track the UAV's actual position based on the location of the retrieved bounding box and calculate the distance from the target is

described in this section.

3.3.1 Landing Target Computation

The following guidelines served as a foundation for the computation of landing target system's development:

1. The UAV's size remains a constant while it's position and size in the image may vary noticeably between consecutive frames;
2. The UAV can only begin to appear and disappear in the image from its borders or by descending/ascending from/to very high altitudes
3. The size of the UAV in the image of an undistorted camera is directly related to its altitude;
4. The Detection module's fast refresh rate enables this method to continuously update the UAV's size and position. The target position is recalculated as a result of the Detected UAV box's changing dimensions.

These presumptions serve as backing for the creation of a heuristic that will be incorporated into the relative landing target estimation algorithm. In order to maximise the system's portability and adaptability, calculations made to get the necessary result were done with the idea of using the most generalised and straightforward algorithm possible. Allowing for simple hardware swaps between UAVs and helipad's cameras, such as switching to a larger UAV with a payload or a different camera for landing. The Middleware module is intended to serve as the communication component for the other two modules, allowing them to be switched out as more sophisticated or superior detection or control modules are required and developed. Another benefit is that by using this module in the system the UAV requires less processing power. Battery economy and conservation are important because the UAV will land on a maritime autonomous vehicle. By making the algorithm simpler and requiring less processing power, the battery consumption can be reduced. Allowing for the module's resource-friendly design. The proposed method does not rely on any mark or pattern established in the UAV to detect its centre. Frame-by-frame analysis based on the object size can become unreliable but even with this the computation is based on a ratio between the bounding box size and the actual UAV size. Given that the camera is positioned in the middle of the helipad and calculations are conducted from the point of view of the camera, the target landing point is simplified as the center of the image.

$$img_landing_target_{pixel} = (h_{img} \times 0.5 \quad w_{img} \times 0.5) \quad (3.1)$$

where h_{img} and w_{img} are the height and width, of the image received from the camera in pixels.

A message with the determined bounding box dimensions — x_{min} , x_{max} , y_{min} , and y_{max} — is received from the preceding model. All the data received by this module are relative to dimensions of the image in pixels. The assumption is that the upper right corner of the image corresponds to the starting point, or in other words pixel (0,0). A diagram is presented in 3.6 to illustrate the behaviour. In figure 3.6, the chosen image

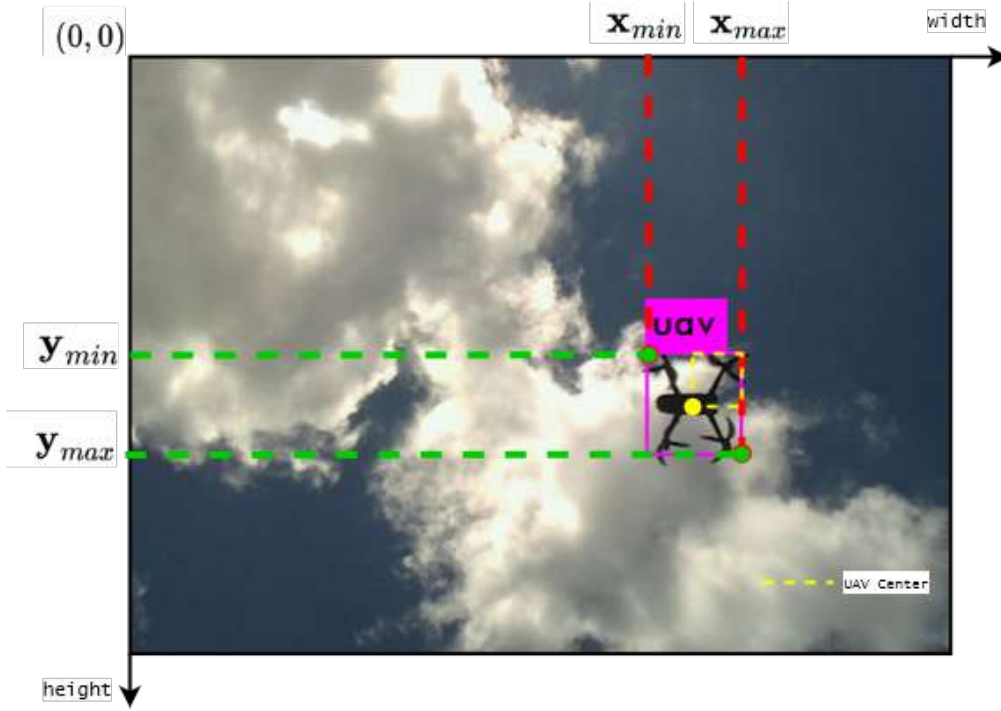


Figure 3.6: Bounding Box coordinates in relation with the frame.

size is 640×480 . The message from the Detection module outputs the coordinates for the bounding box containing the detected UAV, X and Y are assigned red and green, respectively. Additional information about the center of the UAV is presented in yellow.

These values, which reflect the UAV with the highest confidence score and are updated every millisecond, are the basis of this module. Then the UAV Detected Box size in pixels relative to the full image size is obtained.

$$uav_box_size_{pixel} = (x_{max} - x_{min} \quad y_{max} - y_{min}) \quad (3.2)$$

After some measurement and observation of the actual drone, it is determined that its centre of mass is in the centre of the square with the dimensions of the UAV. The final calculations are the centre of the detected box. An approximation is then obtained, further illustrated in 3.6.

$$uav_center_position_{pixel} = ((x_{max} + x_{min}) \times 0.5 \quad (y_{max} + y_{min}) \times 0.5) \quad (3.3)$$

Frame-by-frame analysis based on object size can become error prone because the proposed method does not rely on any mark or pattern established in the UAV to determine its centre. The propellers' uneven shape and detection abnormalities causes mistakes in the calculation of the UAV's size and position from frame to frame. Additionally, the position of the UAV cannot change significantly between frames. Consequently, a Kalman Filter (KF) was used to enhance the results, with the displayed state matrix:

$$X_{kf} = \begin{bmatrix} x & y & v_x & v_y & w & h \end{bmatrix} \quad (3.4)$$

The UAV's pixel coordinates are represented by \mathbf{x} and \mathbf{y} , its speed components are represented by \mathbf{v}_x and \mathbf{v}_y , and its width and height in the frame are represented by \mathbf{h} and \mathbf{w} , which, given that the UAV is assumed to have a square shape, are equal to \mathbf{l}_{uav} .

Estimation is done solemnly in 2D coordinates, as later on the z-axis coordinate will be addressed. Some computations are necessary because the UAV actuators require the relative distance from the UAV to the Landing Target in meters to operate. The number of pixels between the drone and the landing destination, either on the \mathbf{x} and \mathbf{y} coordinates, is determined using a straightforward decomposed Euclidean distance.

$$uav_landing_target_{pixel} = img_landing_target_{pixel} - uav_center_pos_{pixel} \quad (3.5)$$

Next we compute the pixel to meter ratio that allows for the conversion to a value to feed the following module.

$$pixel_meter_ratio = \frac{length_{uav}}{img_size} \quad (3.6)$$

This is the most error-prone statistic in this module, although thanks to the detection module's refresh rate, this effect is minimised. As the UAV path tends to the centre of the image, where there is less distortion from the camera, the position values are updated and the length is corrected in each iteration. The distance from the UAV to the landing target can then be output when a conversion between pixels and metres has taken place.

$$uav_target_waypoint = uav_landing_target_{pixel} \times pixel_meter_ratio \quad (3.7)$$

Knowing the UAV's 2D position in the image makes it possible to estimate its position with respect to the camera and, in turn, the helipad in a 3D axis system [56]. Additionally, the camera parameters are used to confirm the 3D coordinate estimation. Being aware of the ideal focal length of the camera without taking into account any distortion-related errors.

$$z_{uav} = \frac{L_{uav} * f}{l_{uav}} \quad (3.8)$$

where z_{uav} is the object's distance from the camera in metres, f is the lens's focal length in metres, L_{uav} is the object's actual dimensions in metres, and l_{uav} is the object's size in the image (pixels). The UAV will be projected in the image plane from 3.7 parallel to the camera optical axis since the camera is pointing up, and the distance to the camera equals how high the UAV is above the landing pad. As in figure 3.7, it becomes possible

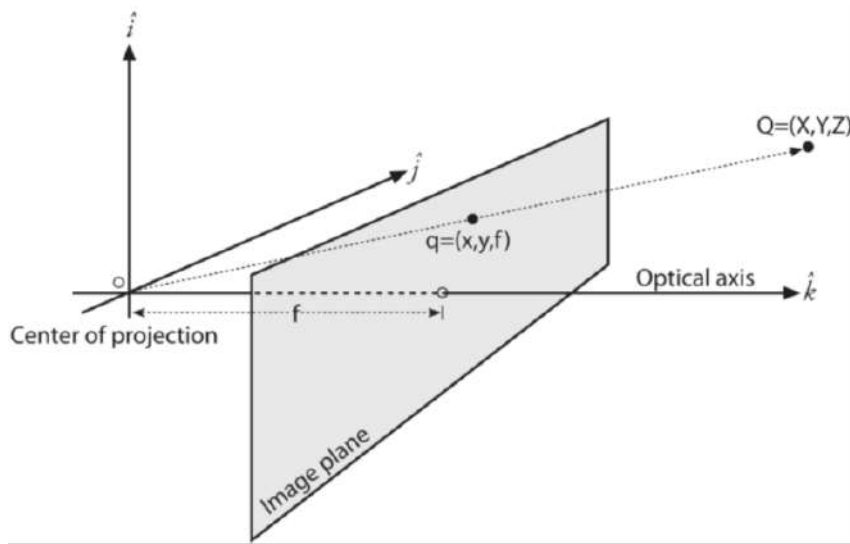


Figure 3.7: Calculating an object's 3D location from a camera image (adapted from [1]).

to calculate an approximated Z by computing their ratio given the camera's intrinsic f , at least the distance between two locations q , in pixels, and the actual distance between their correspondent projections Q .

This allows one to calculate the relative coordinates of X and Y from Z .

$$x_{uav} = d_x * \frac{z_{uav}}{f} \quad (3.9)$$

where d_x is the object's distance from the picture centre in pixels and x_{uav} is the object's estimated position in the X axis in meters. The computation of y_{uav} can be done using the same procedure. This makes it possible to calculate the final UAV position, $\mathbf{P}_{uav} = (x_{uav}; y_{uav}; z_{uav})$.

3.3.2 Image + Target Data Overlay

The image and the variables calculated in this module must be overlaid in order for the user to understand them after receiving them, and this is the responsibility of the Image + Detection Data Overlapping module. The computed UAV centre point and the target landing location (the image's centre) are overlaid on the input image in this part. Additionally the more information about the UAV state, such as battery state, is overlaid. To make the process simpler, the user is also given a rough estimate of the movement vector. In figure 3.8 the centre, target, and battery arrays are sent along with the input

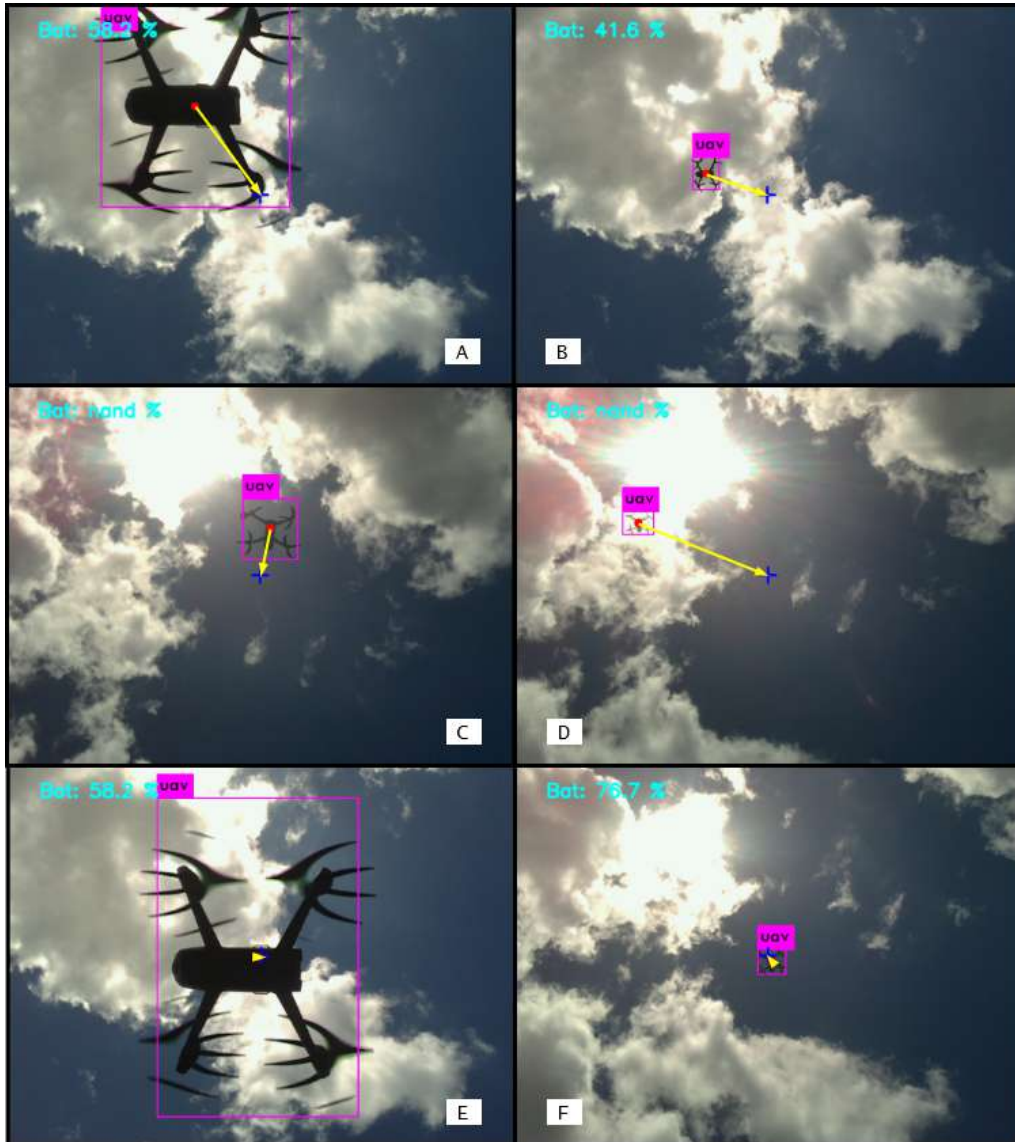


Figure 3.8: The Overlapping processes.

image that has been modified for visualisation. The output of the model shows these arrays over the image. Images A through D show several overlapping module outputs obtained under various atmospheric conditions.

3.4 UAV Position Controller

3.4.1 Precision Landing Manager

The high-level control mechanism for the autonomous landing is described in this section. The system's objective is to safely guide the UAV from its flying position to the landing pad's surface. An estimate of the UAV's 3D position with respect to the helipad is supplied from the previous module in order to assess the landing properly. It makes use of a three-dimensional Cartesian coordinate system as reference. The centre of the helipad is positioned at the origin of the referential, with the z axis parallel to its surface and aligned with the optical axis of the camera. From the perspective of this module, we are just interested in the high-level control, or the direction that the UAV should move in when it is at a specific 3D P_{uav} coordinate in relation to the helipad.

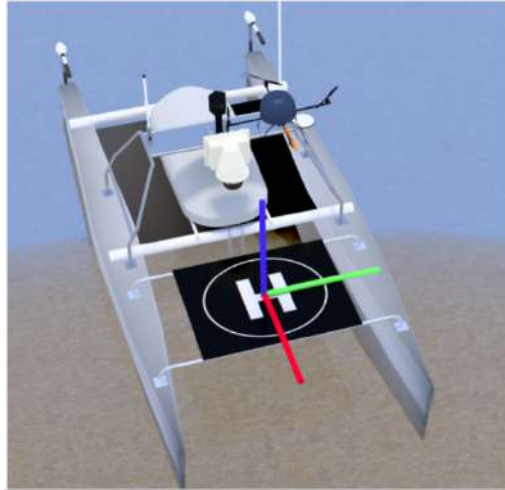


Figure 3.9: UAV location and control orders are given using a 3D coordinate system (adapted from [1]).

In figure 3.9 the red, green, and blue lines of the referential frame on the image are, respectively, the X, Y, and Z axes. Its origin, which corresponds to the coordinate, lies in the helipad's centre (0,0,0).

The direction and strength of the commands should change depending on where the UAV is in relation to the helipad, with smoother commands being issued the lower the UAV is and the closer it is to the helipad's centre. On top of the helipad's surface, an approach zone is created. This zone designates the region where the UAV can safely manoeuvre without threatening the nearby sensor array of the vessel, all the while maintaining the necessary alignment with the helipad. It was also anticipated that there are no obstructions in the airspace above the helipad. The UAV begins or resumes the landing when it enters the approach zone, with guidance commands directing it to descend and get closer to the target. However, if the UAV is outside the zone, the objective is to enter it so that the landing manoeuvre can begin. A multiplicative inverse function is used

to model the approach zone [1]. If the UAV is above the curve, it is regarded as being inside the zone, represented in 3.10. This function was selected because to its remarkable resemblance to a funnel which directs the UAV from a broad area at a high altitude to the centre of the helipad in a relatively smooth and progressive method. In order to better meet our demands, the approach zone is approximated using a multiplicative inverse function, adding a factor of scale a_{az} and an offset b_{az} we get the approach zone $az(x)$:

$$az(x) = -\frac{a_{az}}{x} - b_{az} \quad (3.10)$$

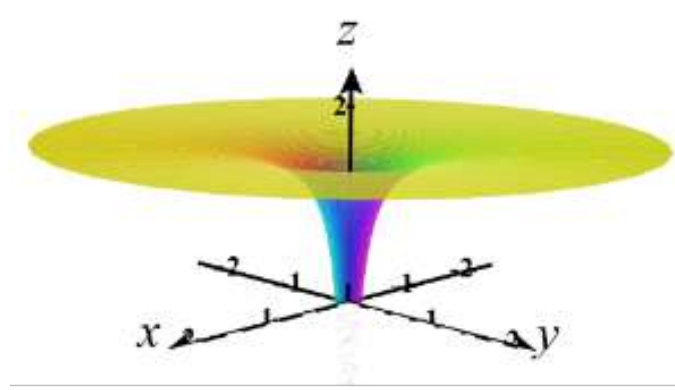


Figure 3.10: 3D view of the approach zone.

The UAV is deemed to be "within" the zone when it is above the surface and "outside" when it is below. The colours depict the velocity command's intensity. Cooler colours like blue have a smaller magnitude than warmer ones like yellow.

3.4.2 Position Controller

In this dissertation, a controller was created for the Parrot Bebop II drone so that it could fly to the target selected locations on its own. Due to the creation of the ROS + Parrot integration module, the controller process was adapted from the controller process was modified from [57]. A PID controller was low-level control, or how much effort should be put into each of the motors attached to the propellers, is outside the purview of this dissertation. Since the procedure in this part assumes a low-level controller is already in place, it will be presumed that it is. In this case study, a PID controller is used to ensure that the UAV complies with the high-level orders that are given to it. The controller calculates the discrepancy between the required location and the odometry, and a PID controller is used to fix the discrepancy. The Parrot Bebop's Bebop-Autonomy driver activates the drone to move in the chosen direction. For each P_{uav} , a velocity command is supplied. A vector with the following parameters can be used to represent the velocity command, v_c :

$$\mathbf{v}_c = (x, y, z)(\phi_{v_c}, \theta_{v_c}, \psi_{v_c}, m_{v_c}) \quad (3.11)$$

where (x, y, z) denotes its origin point, m_{v_c} denotes its magnitude, and ϕ_{v_c} , θ_{v_c} , ψ_{v_c} stand for roll, pitch, and yaw in regard to the 3D coordinates frame, respectively. The origin of the vector will always match to the present position of the UAV because we are controlling it. Additionally, roll is always set to a fixed position because it has no effect whatsoever on the direction of the velocity vector. By fixing the UAV yaw to a constant value it is possible to land the UAV always in line with the vessel.

The distance that the UAV needs travel in order to get at the target landing location, or the target pose, is read from the distance waypoint that the preceding model reported and which is a distance measurement in 2D coordinates. The bebop autonomy driver's on-board sensory data is used to compute the drone's current Pose. The desired target for navigation is then represented by a point in 2D coordinates that is created by adding the target pose to the current UAV pose. The difference between the desired position and the present position is then used to calculate the error vector. In order to decrease the error, a PID controller constructs the correction vector using the input error as a starting point, further illustrated in 3.11. Finally, using bebop autonomy, the correction vector is given to the drone by publishing a new rotor velocity command.

This approach of performing the controller process on the distance to target message yields the velocity commands. Implementing the modular methodology suggested by the proposed system requires having the inputs and outputs of each module properly defined. This condition needs to be taken care of either in the development stage or the implementation stage, both of which are thoroughly covered in chapter 4, in order to comply with the model requirements.

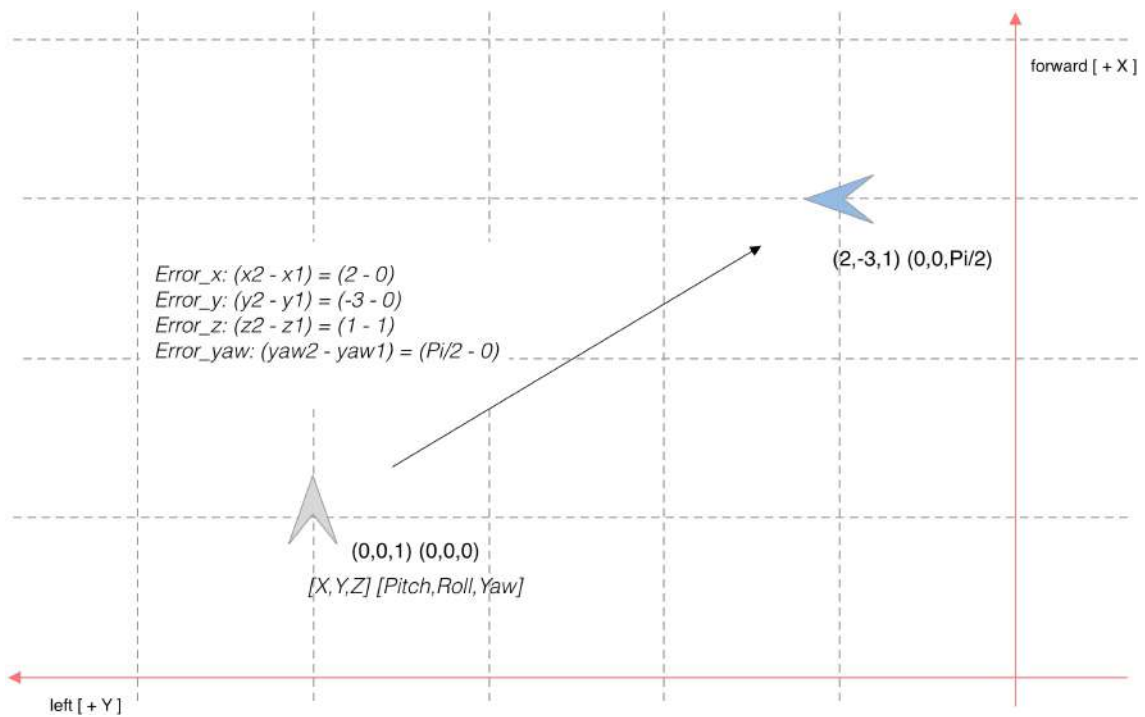


Figure 3.11: Estimation of position controller error. An example of velocity commands mentioned in 3.11 from the target posture, shown in blue, and the current posture of the UAV, shown in grey. These commands are then subjected to error calculations in order to determine the offset of the drone in relation to the target, in each coordinate.

IMPLEMENTATION

The implementation of the model described in the preceding section will be covered in this chapter, along with an evaluation. The Darknet-53 + [YOLOv3](#) framework, and Training Procedures are the primary components. The process of setting up the hardware and software simulations and the work done to integrate the system so that it complies with [ROS](#) are the key themes of the experimental setup section. The section on the Darknet-53 + [YOLOv3](#) framework covers the structure of this framework, which was chosen to be a key component of the suggested model. The training processes section, which concludes the chapter, describes the preliminary work required to create the ideal environment for training as well as the training itself.

4.1 Experimental Setup

The suggested approach was fully developed in Python and made fully compatible with the Robot Operating System ([ROS](#)). Different [OS](#) configurations were used, one for system deployment and the other for simulation, further described in [5.2.2.0.1](#). For testing and development convenience, simulation was carried out on a different system inside a [Virtual Machine \(VM\)](#) environment. When configuring with more recent versions of Ubuntu and [ROS](#), integration issues with the Parrot Sphinx simulation environment and Gazebo surfaced. The implementation workflow required to be viewed as a [ROS](#) ecosystem in order to completely comply with [ROS](#), hence all suggested methods had to be modified to fit this methodology.

4.2 Darknet-53 + YOLOv3 CNN Framework

The Darknet-53 + YOLOv3 framework is integrated into the model to carry out the needed object identification, as specified in the proposed model chapter. Despite its complexity, this framework is essential to the performance of the implemented model, therefore its fundamentals must be grasped. The YOLOv3 has an open source implementation and is accessible on this GitHub repository [58]. It was created and released in 2018 by Joseph Redmon [59]. It is a framework built on Python 3 and Darknet, with all of its code documented for quick access. Darknet is an open source neural network framework written in C and CUDA. It enables GPU computing and is quick and simple to setup. Its extensive and adaptable toolkit makes it simpler to design and deploy apps based on deep learning algorithms. It uses Python and has discrete modules that can be included into users' projects making it very user-friendly. This section will concentrate on describing its general practise, its organisation, and its key components.

4.2.1 Architecture

This architecture method extracts features from the input image using the backbone and the Multi Scale Feature Pyramid Network (MSFPN), predicts bounding boxes based on the learned features, and then utilises Non-Maximum Suppression (NMS) to obtain the results. Figure 4.1 illustrates it in full. Three modules make up the MSFPN: the Concat module, in charge of linking each backbone network feature, the Encoder-Decoder module, in charge of producing multi-scale features, and the Feature Fusion module, in charge of combining features. The various network modules will be described in more depth below. The Multi Scale Feature Pyramid Network (MSFPN) and the lightweight backbone are

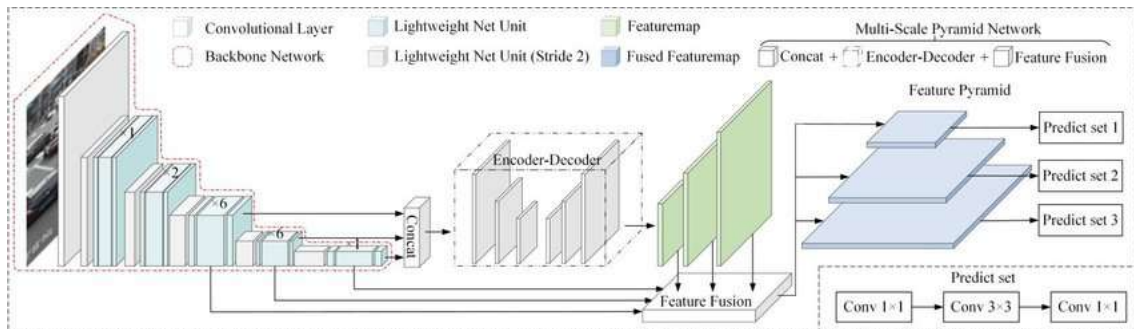


Figure 4.1: An overview of the YOLOv3 [60].

used by YOLOv3 to extract features from the input image. To create the basic feature in MSFPN, the Concat model merges three feature maps of the backbone. Encoder-Decoder creates a collection of multi-scale features, and the Feature Fusion model combines the collection of multi-scale features with three feature maps of the backbone to create a feature pyramid, as in figure 4.1.

4.2.1.1 Backbone Architecture

On Darknet-53, the backbone network is built. Figure 4.2 displays the network unit. A backbone is a **CNN** whose main objective is to extract characteristics from unprocessed images. The early layers identify the simpler elements, such as forms and edges, whereas the last layers identify the more intricate ones. Darknet-53 employs 1×1 and 3×3 convolutional layers in succession but is now much larger and includes some shortcut connections. As the name implies, there are 53 convolutional layers in it. In the network, standard convolutions are factorised into two layers using depthwise separable convolutions, a type of factorised convolution. A single filter is applied to each input channel in the first layer's depthwise convolution. The depthwise convolution's outputs are combined with an 1×1 convolution in the second layer, referred as as the pointwise convolution. This is divided into two layers by the depthwise separable convolution: a layer for combining and a layer for filtering. This indicates that the network structure makes better use of the **GPU**, which results in faster evaluation. When complete, this method outputs the obtained feature maps.

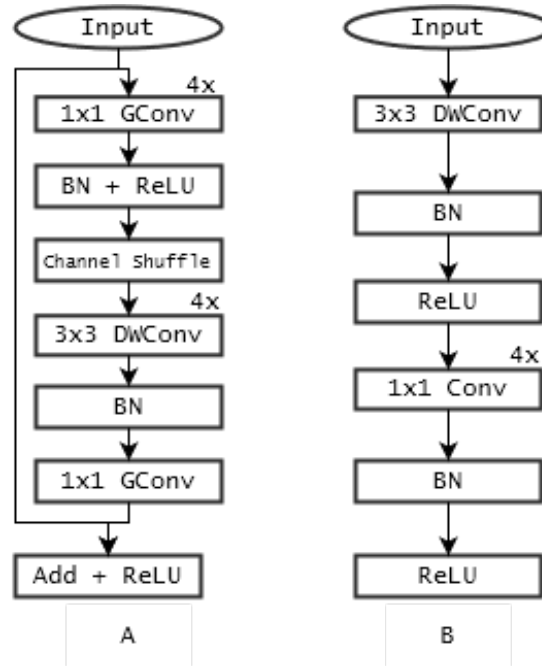


Figure 4.2: Network unit for backbone network base on Darknet-53.

4.2.1.2 Multi-Scale Feature Pyramid Network

The Multi-Scale Feature Pyramid Network [61] is designed to enhance the backbone network's features and provide a more effective multi-scale feature pyramid. An illustration is depicted in Figure 4.3. To create the basis feature, the concatenation model merges three feature maps of the backbone. Encoder-Decoder creates a collection of multi-scale features, and the Feature Fusion model combines the collection of multi-scale features

with three feature maps of the backbone to create a feature pyramid. We go into great detail on the three modules below.

1. Concat Module

Backbone features from three separate levels are combined by the Concat model, which is essential for building the final feature pyramid. The three feature maps from the backbone network are the input to the Concat module, and before concatenating, we up sample them to re scale the depth features to the same scale.

2. Encoder-Decoder Module

The Encoder-responsibility Decoder's is to produce feature maps with three scales. The input feature map is down sampled using a series of 33 convolutional layers in the encoder to create a reference-set of feature maps. A set of 33 convolutional layers make up the decoder. Of course, there are element wise sum operations and up sampling. Finally, to improve feature representation and maintain feature smoothness, we add an 11 convolutional layer at the branch.

3. Feature Fusion Module

The goal of the feature fusion module is to create a feature pyramid out of the multi scale features produced by the encoder-decoder and the features from the three backbone levels. The multi scale features produced by the encoder-decoder and features with equivalent scales throughout the channel dimension are concatenated in the first stage of the feature fusion module. All of the boxes, including small, medium, and big, became more accurate after a Feature Fusion module is included.

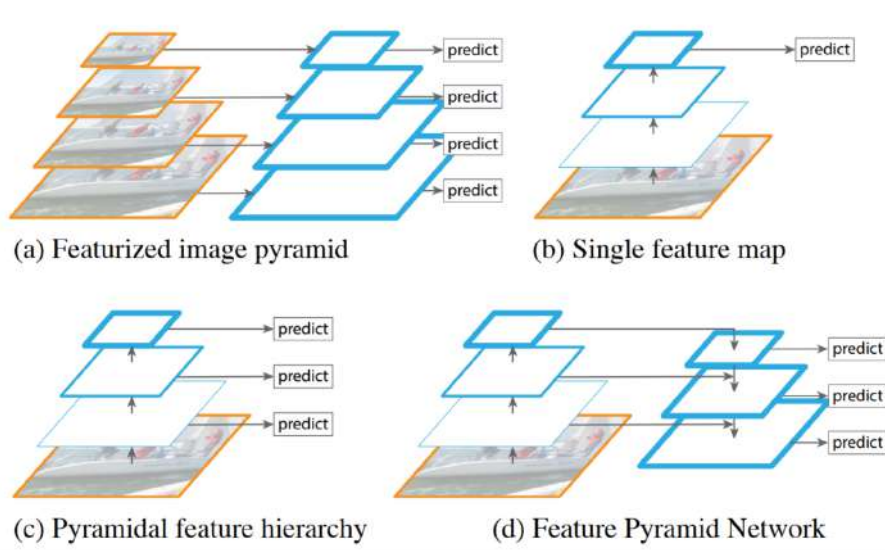


Figure 4.3: Building a feature pyramid using an image pyramid (adapted from [61]).

It takes time to compute features separately for each of the image scales. In order to speed up detection, recent detection systems have chosen to simply use single scale

features. An alternative is to utilise the ConvNet-generated pyramidal feature hierarchy as though it were a feature-rich image pyramid. The feature maps in picture 4.3 are shown as blue outlines, and thicker outlines signify semantically more significant features.

4.2.1.3 YOLOv3

You Only Look Once: version 3 (YOLOv3) [54] is an object detector proposed by Joseph et al., and it treats detection as a regression job. This technique accelerates detection and takes input images of various sizes. Table 4.1 displays the network structure. Because

	Type	Filter	Channel	Stride	Featuremap
	Conv 2d	3×3	32	1	416×416
	Conv DW	3×3	16	2	208×208
1×	G Conv	1×1	16	1	
	Conv DW	3×3	16	1	
	G Conv	1×1	16	1	208×208
	Conv DW	3×3	32	2	104×104
2×	G Conv	1×1	128	1	
	Conv DW	3×3	128	1	
	G Conv	1×1	32	1	104×104
	Conv DW	3×3	64	2	52×52
6×	G Conv	1×1	256	1	
	Conv DW	3×3	256	1	
	G Conv	1×1	64	1	52×52
	Conv DW	3×3	128	2	26×26
6×	G Conv	1×1	512	1	
	Conv DW	3×3	512	1	
	G Conv	1×1	128	1	26×26
	Conv DW	3×3	256	2	13×13
1×	G Conv	1×1	1024	1	
	Conv DW	3×3	1024	1	
	G Conv	1×1	256	1	13×13
	Conv 2d	1×1	1024	1	13×13

Table 4.1: Network structure for the convolutional layer in backbone.

YOLOv3 employs multi-scale prediction, it can be found on feature maps of many scales. Target detection becomes more precise as a result. On a high performance computer, **YOLOv3** can achieve real-time detection thanks to the GPU's potent computing capacity. Real-time applications are frequently not practicable due to the embedded devices' performance, which is significantly lower than that of high performance computers. Because it enables the model to view the entire image during testing, its predictions are influenced by the image's overall context. Convolutional neural network methods like **YOLO** "rank" regions according to how closely they resemble predetermined classes. Regions that score highly are reported as positive detections of the class that they most closely match. Applying detection kernels to feature maps of three different sizes at three different locations throughout the network is how **YOLOv3** detects objects.

4.2.2 Framework Files

The three primary folders of the [YOLOv3](#) architecture are called logs, databases, and data. The network maintains the files containing its output darknet weights in the logs folder while it is still in training. The database folder is divided into various folders, each containing training data for various detection outcomes. In this instance, a folder called uav was made to hold all the data needed to train the uav detection network. The crucial Python files for package training and inference, as well as the required libraries for [ROS](#) integration, are located in the data folder. Brief descriptions of the remaining relevant Python files are provided below:

1. **model.py**

The primary [YOLOv3](#) model implementation for both training and detection is contained in this file. It is organised according to network layers and includes methods for determining loss values, generating data, and formatting data. Additionally, it contains the darknet class, which unifies the train, detect, and [ROS](#) ecosystem aspects of the [YOLO](#) model.

2. **config.py**

The base configurations class and all significant parameters that define the training and inference behaviours of the network are contained in the config.py file. The default values for these parameters are set, and they are typically changed by modifying a distinct Python file.

3. **visualise.py**

The display and visualisation routines are contained in the visualize.py file, as its name suggests. It is a very significant file since it contains the tools used to apply the acquired mask and display the resultant images. It has tools for masking and painting bounding areas and regions of interest in arbitrary colours. Additionally, it has tools for assisting with network performance visualisation, such as precision plots and statistics for weights obtained. The suggested model relies heavily on the overlapping section, and this file lacks some tools that can be utilised directly in the model. The only original framework file that was directly altered as a result was visualize.py.

4.3 Training Procedures

A well-written problem statement, data collection and preparation, model training and improvement, and inference are the common steps in solving any given machine learning problem. The method used in the context of this dissertation is not strictly linear. For instance, we might discover that our model performs terribly on a particular type of image label, in which case we should go back and collect further information.



Figure 4.4: Workflow for general machine learning that was employed for this dissertation.

4.3.1 Data Gathering

A solid data set is essential for training a network that can recognise objects with an accuracy and precision appropriate for use in real-world scenarios. In order to accurately depict the range of conceivable scenarios that could be met on the field, this data set must be made up of numerous photos with a wide diversity of examples. The ideal data set would include depth-cloud information as well as ground-to-sky images of the Parrot Bebop 2 taken by a multi band sensor with a channel for each each wavelength. However, it is quite difficult to find this type of particular data online because it is typically utilised sky-to-ground, or aerial imagery for study. As a result, the first data set used in this dissertation was made up entirely of information gathered from a simulation that employed the Unreal Engine software to automatically create and categorise images. Although the photos were accurate, preliminary findings showed that the network model trained with these images was unable to identify the real drone while it was in flight, necessitating the use of a different strategy to finish the assignment.

The same Parrot® Bebop 2 UAV was captured a number of different batches of frames. While some of these batches of UAV occurrences were recorded on the same day under similar cloud and lighting conditions, others were recorded under quite distinct meteorological circumstances. The photographs with the UAV instances were taken looking up at the sky from the ground, with the UAVs hovering between the ground and a height of 40 metres. Since the UAV is very modest in size, it loses visibility at heights of about 45 metres. The conditions needed for the images collected were defined under some restrictions:

1. Since the UAV would be flying above the ocean, no man-made items, including background elements like buildings or even people, would be visible;
2. Lighting effects are affected by many atmospheric circumstances, such as clear and overcast weather;
3. Due to the very slow moving nature of these elements, cloud formations and solar flares would generate the most background object interference;
4. Moving objects that could interfere in the UAV detection because of it's resemblance, i.e birds, planes, helicopters.

The recording sessions were set up to account for these limits, some examples of the many scenarios that were recorded are shown in Figure. 4.5. To achieve the most realistic

representation of the potential UAV attitude and attitude during flight, various phases of the landing procedure were also captured. Images captured in various air circumstances included drone images that were landed on top of the camera at the ideal target position. From this point until the camera's maximum detection height, the landing and takeoff procedures were also captured on camera. Figure 4.6 shows examples of photographs taken under these circumstances and during these manoeuvres.



Figure 4.5: Examples of images taken under various atmospheric conditions. Both sunny and extremely cloudy weather could be recorded. Different cloud shapes, sizes, and formations. Images 1, 2, and 5 (from right to left) also show the effects of a solar lens flare.

All of the data sets cited in this dissertation's references were compiled using multi-spectral images taken with an Intel® RealSense™ L515 LiDAR camera mounted on a sheet of metal. LiDAR readings are only possible in a 9 top meters range. Each image that is taken has the potential to be split into three channels, producing three separate files,

each of which contains information on the RGB, near-IR, and depth spectrum. Only the RGB channel was processed in order to create this dataset, which is used with the [YOLO](#) framework. Changes to use the depth frames were largely completed, but additional work is still required. Each RGB image file is a 640×480 pixel 0.31MP jpg file, it was necessary to scale down from the 1024×768 allowed by the camera because of processing problems that arose while gathering the frames.

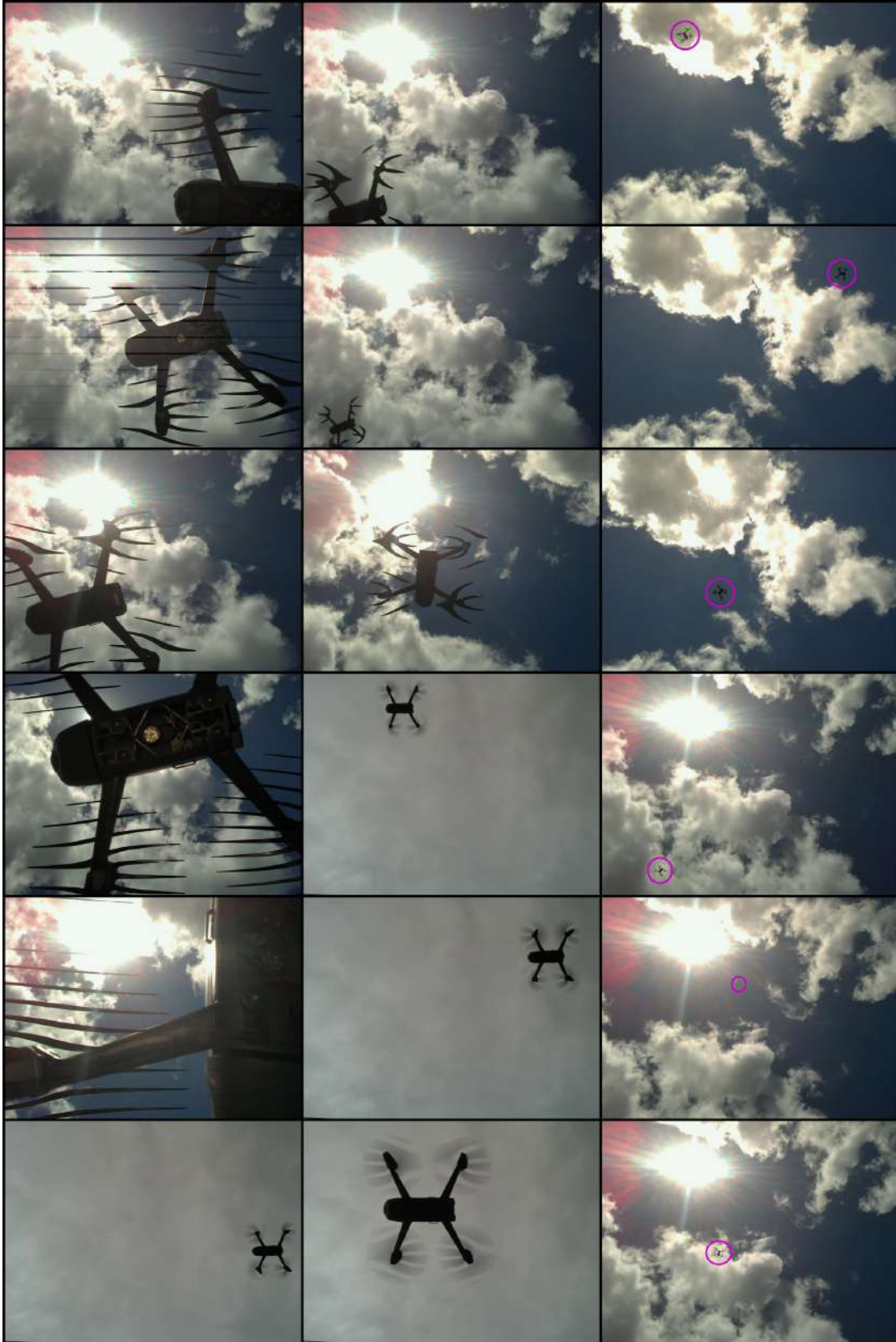


Figure 4.6: Representative frames from the constructed data set. Each image depicts a separate scenario that was captured after the limitations for the data collection step. To make viewing easier, the UAV was marked in several photos.

4.3.2 Image Annotation

Every data set must have the relevant details, including the pixel-wise area where objects are situated within each image, while training an image classification network. Depending on the framework, this information can be delivered in a variety of ways, and in order to acquire it, an annotation procedure must be carried out. Each image must be tagged with the location of UAV instances after the data sets are organised by their classes. On YOLO Darknet, each collection of images' annotations must be represented by a single text file with a text object for each image. Example of a image text annotation of a single UAV detected instance:

Darknet *labelmap* file for the model in this dissertation as a single label system:

This format includes a *labelmap* that converts the numeric IDs to human readable strings and one text file per image that contains the annotations and a numerical representation of the label. The annotations are easier to deal with even after resizing or stretching photos because they are normalised to lie within the range $[0, 1]$. It has grown in popularity as a result of adopting different YOLO model implementations from the Darknet framework.

Given this, RoboFlow [62], a programme that allows for the extraction of annotations as a file in any framework format, was selected to assist in the process of picture annotation for this framework. It makes available all the resources required to turn raw photos into a specially trained computer vision model and apply it in apps using their training and deploying API. You may quickly convert YOLO Darknet files to or from any other object detection annotation format because Roboflow can read and write YOLO Darknet files. The process of transitioning from raw photos to a trained and deployed computer vision model is substantially streamlined by the self-serve annotation tool called Roboflow Annotate. The AI-assisted labelling for bounding boxes, polygons, and instance segmentation, is the feature that considerably aided in the tiresome effort of categorising thousands of photos. Using an existing neural pre-trained model to automatically annotate images by identifying objects in the images and applying labels in a quick labelling workflow. To detect frequent items in your dataset, such as dogs and humans, there are models that have already been trained on the COCO database. However, because there is no link between any uav class and the COCO database, these models were unable to detect the UAV. In order to combat this, a small batch of 300 photographs was manually annotated in order to train a model with the Roboflow API to assist with the labelling of the remaining images.

4.3.3 Data Augmentation

By modifying the training data already available, image augmentation allows you to expand your dataset. Fundamentally, it is the creation of phoney data that looks authentic.

A model can generalise to a wider range of scenarios more effectively using image augmentation. To ensure that learning and inference happen on the same picture attributes, preprocessing should be applied to your training, validation, and testing set. In order to display images the same way they are saved on disc, auto-orient strips your images of their EXIF data. The orientation of a given image is determined by EXIF information. The images are also downsized to 416×416 pixels since we need smaller images than the 640×480 dimensions that were originally taken for speedier training and because the YOLOv3 architecture performs best with multiples of 32. By extending the variety of learning examples for your model, data augmentation can improve the generalizability of its performance. In order to add additional variations and more images to your dataset, augmentation conducts transformations on your current photographs. In the end, this improves model accuracy across a wider range of use situations. A couple major advantages come from doing your augmentations through Roboflow as opposed to during training:

1. Reproducibility of the model is improved. Each augmented image has a duplicate of how it was stored. For instance, you might discover that your model works better with bright photographs than with dark images, in which case you should gather more low-light training data.
2. Less time is spent training. Augmentations are limited by the CPU. Your GPU frequently waits for your CPU to supply enhanced data at each epoch when you are training on your GPU and doing augmentations on-the-fly.
3. Costs for training are reduced. GPUs frequently wait to be fed images for training since augmentations are CPU-constrained processes, wasting processing power.

The data augmentation steps selected for this case study were defined by the previously stated limitations and deployed on the Roboflow platform. Some experiments were done with other augmentation strategies to make sure these steps were the best. In order to evaluate the effects, examples of the various stages applied are shown in Figure 4.7 beside an original frame.

The first was **Hue** augmentation, a method that randomly modifies an input image's colour channels to prompt a model to evaluate other colour schemes for objects and settings. This method can help prevent a model from memorising the colours of an object or scene. Hue augmentation enables a model to take into account both the edges and geometry of objects as well as their colours, even though output image colours may appear strange or even abnormal to human perception. Hue is measured radially since it has colour circle origins, which means input images are changed relative to a certain number of degrees plus or minus their starting point. The number of degrees for the hue augmentation chosen was 60° . The platform randomly selects the number of degrees between 0° and 60° and whether to apply the hue shift positively or negatively. The second technique was **Saturation** augmentation, which is comparable to hue but modifies how bright the image appears. A picture that is completely desaturated turns monochrome,

one that is partially desaturated has subdued colours, and one that is positively saturated moves colors closer to the main colours. When colours in the real world are different (for instance, when a different white-balance is established, when different lighting is present from sun flares, or even when it's misty outdoors), changing the saturation of an image helps the model perform better. The next step was **Exposure** augmentation, which increases image brightness variability to make the model more tolerant to variations in lighting and camera settings. The camera is stationary, but the objects it is detecting are frequently in motion, therefore a small amount of **Blur** has been injected into the image. By just changing the **Bounding Box (BB)** of a source image, bounding box level augmentations create fresh training data. Systematic improvements are produced by bounding box changes, particularly for models with small data sets. **BB Rotate** improves rotational diversity to make the model more resistant to camera roll. **BB Shear** changes the model's angle of view to make the model more resilient to camera and subject pitch and yaw. **BB Flip** contains horizontal or vertical flips to help the model become insensitive to subject orientation. **BB Noise** incorporates noise into the model to increase its resistance to camera artefacts.



(a) Original frame. No augmentation steps.



(b) Blur up to 20px.



(c) Hue, -60° .



(d) Hue, $+60^\circ$.



(e) Saturation, -70% .



(f) Saturation, $+70\%$.



(g) Exposure, -30%.



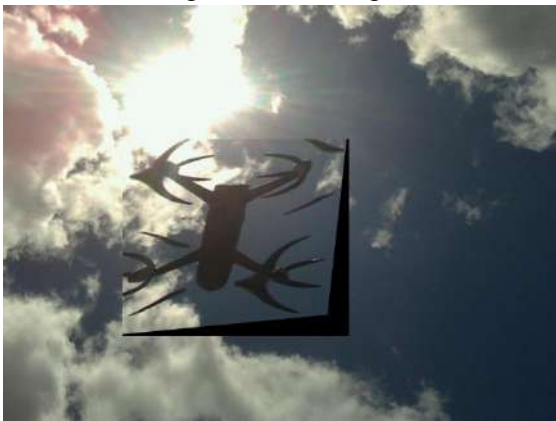
(h) Exposure, +30%.



(i) Bounding Box Rotate augmentation.



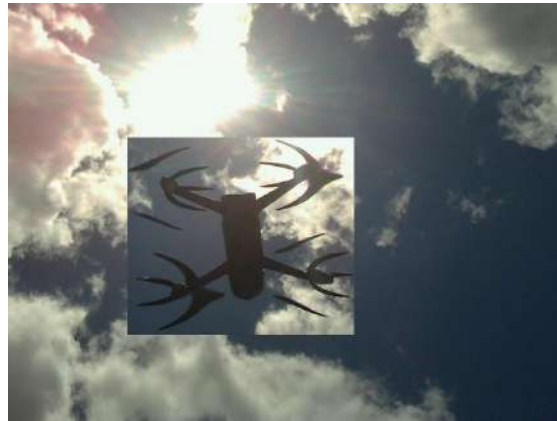
(j) Bounding Box Rotate augmentation.



(k) Bounding Box Shear augmentation, $\pm 15\%$ Horizontal.



(l) Bounding Box Shear augmentation, $\pm 15\%$ Vertical.



(m) Bounding Box Flip augmentation, Horizontal + Vertical.



(n) Bounding Box Noise augmentation, up to 20% px.



(o) Bounding Box Noise augmentation, up to 20% px.

Figure 4.7: Examples of how the Data Augmentation techniques applied to the dataset.

4.3.4 Parameters

The data sets and annotation files were imported into Google CoLab, where the training was carried out. The user can run Python code in the web using CoLab, a hosted Jupyter notebook. It is excellent for machine learning since it offers access to [GPU](#) processing. Users of CoLab Pro have access to an NVIDIA Tesla T4 [GPU](#). Additionally, it has a direct connection to Google Drive, where all of the data sets were kept. The network can never be trained for more than 12 hours straight on the CoLab [GPU](#) Virtual Machine because to its 12-hour maximum run time. But it can restart from the most recent finished period without losing any of the earlier work. Based on existing training files already present in the framework, the python Jupyter file *uav_detection.py* was developed to train the network. This file replaced the majority of the framework's adjusted parameters rather than replacing their default values directly. Below is a list of the network's trained values as well as the tweaked parameters. The majority of these parameters were changed numerous times, however the values shown were the best ones that produced the best results. All of these values are shown in table [4.2](#) below.

- **num_classes:** The number of classes the network can identify. The value of this parameter is always 1 for the uav class, which corresponds to the number of trained classes.
- **batch:** The quantity of image samples to be processed in a single batch.
- **subdivisions:** the quantity of mini-batches in a batch. The weights will be adjusted for batch samples as the number subdivision samples are processed simultaneously by the [GPU](#).
- **max_batch:** This many iterations of processing will be applied to the training (batches).
- **steps:** The learning rate will be multiplied by the scaling factor during these iterations.
- **width:** During Training and Detection, every image will be downsized to the network size (width).
- **height:** During Training and Detection, every image will be downsized to the network size (height).
- **channels:** Every image will be translated to this amount of channels during Training and Detection due to the network size (channels).
- **decay:** It removes dataset dysbalance by optimising for a weaker updating of the weights for typical features.

- **momentum:** Optimises for the accumulation of movement and the degree to which the history will influence future weight changes.

The Darknet [YOLOv3](#) framework is designed to save training time by requesting a file with weights that have already been trained, like those from the [COCO](#) data set. In order to prevent conflict between the weights created and the channel input images, the first layer, `conv_1`, is removed from the weight loading process.

Parameters	Value
<code>num_classes</code>	1
<code>batch</code>	64
<code>subdivisions</code>	16
<code>max_batch</code>	4000
<code>steps</code>	3200, 3600
<code>width</code>	416
<code>height</code>	416
<code>channels</code>	3
<code>decay</code>	0.0005
<code>momentum</code>	0.9

Table 4.2: Values for the parameters used to train the network.

4.3.5 Training

With the path to the training and validation data sets, the function *train* inside of *uav_detection.py* was called to begin training. With the following input parameters: the data sets, the random image augmentations effects, the learning rate value, the number of batches, and the layers to train, this function executed the train method from *model.py*. Every batch, the [YOLOv3](#) model covers the full training set before proceeding to the validation phases. The Darknet network calculates and returns the training loss values after each step as well as the validation [mAP](#) values after each batch while the model is training, making it possible to track its development.

The network train shown in Figure 4.8, which would eventually integrate the proposed model system, was composed of four smaller training stages with distinct aims. In the **Preliminary Train** phase, the [COCO](#) dataset is used to compare pre-trained convolutional weights with computed comparison metrics that helped develop the [YOLOv3](#) network and Darknet framework. Although not done against a [UAV](#) illustrative object, this provides crucial information about the network's behaviour. The next phase is called **Parameter Pre-Train**, and it uses a hyperparameter flag to employ an integrated evolution optimizer to fine-tune the input parameters. A [Genetic Algorithm \(GA\)](#) approach for optimization is used in the Hyperparameter Evolution method of optimization. Because they control a wide range of training components, hyperparameters in machine learning can be challenging to optimise. Grid searches and other traditional approaches can quickly become ineffective due to the high-dimensional search space, unclear linkages among the

dimensions, and expensive nature of evaluating the fitness at each location, making GA the best candidate for hyperparameter searches in YOLO networks.

The full **Train** will be started after the best start parameters have been determined, which will take 4000 batches and the entire train time. The goal of this stage is to train the network weights entirely from scratch against the entire dataset, with random augmentation made using the techniques described in section 4.3.3. The **End-Spectrum Train** phase marks the end of the network's training process. Using the same dataset and another randomised seed for the augmentation steps, the weights generated from the full train are then evolved. Since the batches in the Darknet framework serve as a timetable, the train is run for 6000 batches in this phase, an additional 2000 batches.

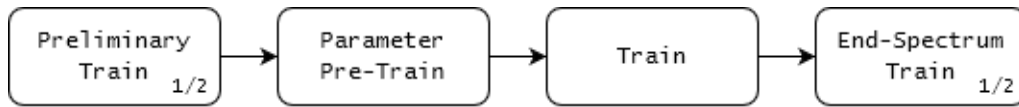


Figure 4.8: Training phases implemented for the proposed model.

Figure 4.9 represents the train loss graph for the **Preliminary Train** phase illustrating how the training process evolved for later iterations. Total validation loss is shown in blue and mAP (%) is shown in red. Because the pre-trained network uses pre-trained weights, the training is only visible after a few iterations. This process was obtained by training the network on top of the pre-trained convolutional weights. The total of the mask training loss and bounding box training loss is reflected by both of the loss values. Each of these loss metrics is also the total of all the individual loss values that were calculated for each region of interest.

The weights file from the train with the lowest learning rate was retrieved and afterwards added in the detection phase since it had the overall most satisfactory loss values. Once the best and most effective strategy for training the network had been determined, the entire data set was subjected to this training method once more, this time in full spectrum, and the results were also assessed using validation loss graphs and the best weights files that had been downloaded. The process is described in more detail in 5.2.1.2.

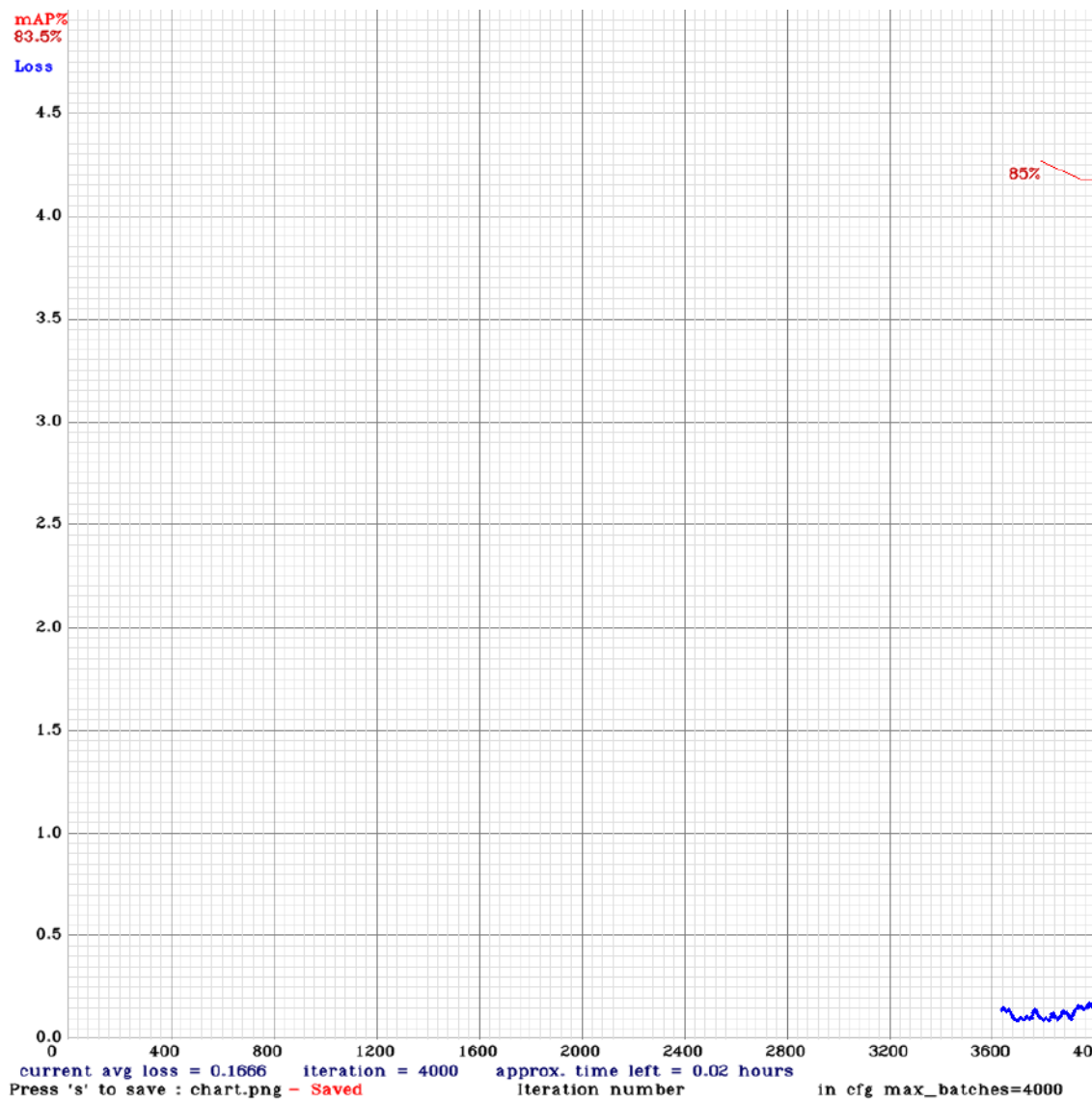


Figure 4.9: Example of the Helipad camera's **Preliminary Training** behaviour utilising the YOLOv3 Tiny and Darknet framework.

EXPERIMENTAL RESULTS

The developed techniques for gathering data are presented in detail in this chapter, enabling a thorough assessment of the model's performance. A description of the hardware and software utilised as the experimental setup for the implementation and testing of the suggested model is provided in the first section. There is also a description of the procedures followed to integrate the software and hardware with the tests created for the modules, either independently or in conjunction with one another. In the second section, it begins with the UAV detection by outlining the findings that back up the decision to use the Darknet + YOLOv3 framework for this model. This chapter's last section primarily focuses on the training done using various data sets, while the third section gives the model's inference results. The quality and influence of the training and inference findings on the model's final output are analysed, compared, and debated. The Controller simulation findings will be covered in the following section.

5.1 Experimental Setup

The suggested approach was fully developed in Python and made fully compatible with the Robot Operating System (ROS). A Jetson Nano with an ARM quad-core running at 1.43GHz, 4GB of LPDDR4 RAM, and the Nvidia CUDA toolkit + cuDNN library was used to deploy the system. A Maxwell 128-core graphic processing unit is used for image inference. Different OS configurations were used, one for system deployment and the other for simulation, further described in 5.2.2.0.1. For testing and development convenience, simulation was carried out on a different system inside a Virtual Machine (VM) environment. For the virtual simulation environment Gazebo 9, the 64-bit Linux distribution running Ubuntu 18.04 (Bionic Beaver) with ROS Melodic Morelia was chosen as the operating system. The OS selected for system deployment was the 64-bit Linux

distribution running Ubuntu 20.04 (Focal Fossa) and [ROS](#) Noetic Ninjemys. All low-level computer vision operations were performed using [Open Computer Vision \(OpenCV\)](#) 3.2.

The setup for the experimental tests in the real world consists of 3 components. The Drone, a Parrot Bebop 2 equipped with a front camera to perform visual odometry. The drone approaches a 32 cm square. The camera, an Intel RealSense L515 to capture 640x480 size images at 30 fps. This camera is installed in the centre of the helipad so that it faces the platform. The GCS, my computer, which has an AMD Ryzen 7, as a processing unit, and an NVIDIA RTX 3060, as a graphics unit. An illustration in figure 5.2 of the experimental setup for testing.



Figure 5.1: Helipad.

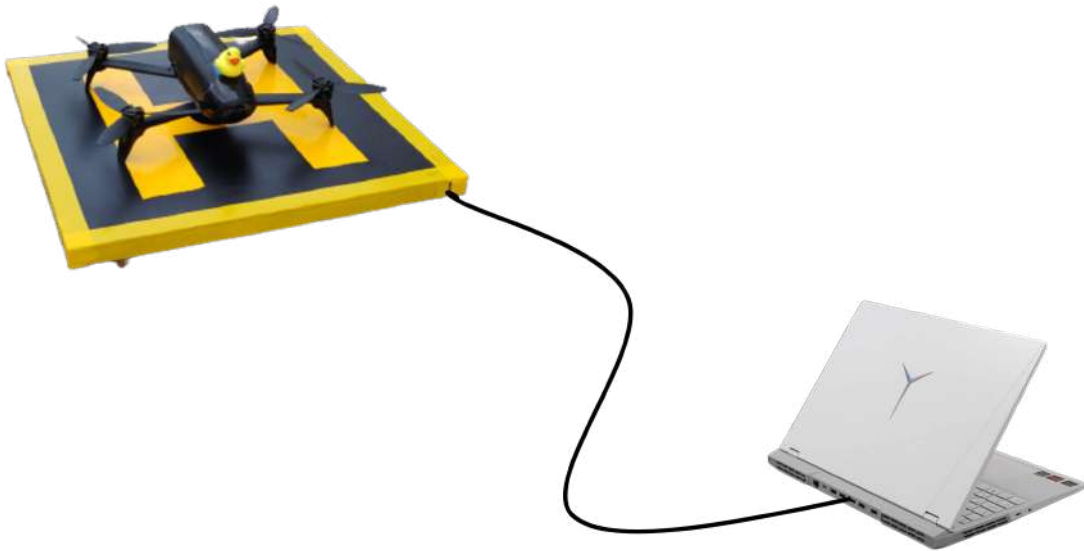


Figure 5.2: Real world flight tests setup.

5.1.1 ROS Integration

The experimental setup workflow required to be completely compliant with [ROS](#), hence all suggested methods had to be modified to fit this methodology. Each component, or

node, in the system functions independently and exchanges data asynchronously using the ROS framework. In the ROS system, data is shared in the form of "messages" by publishing and subscribing via asynchronous network connection. Four distinct nodes make up our entire system, these are detection, communication interface, position controller and *bebop_autonomy*. The Legged Robotics [63] team created *darknet_ros* [64], a ROS package for object identification in camera pictures that relies on the YOLO framework and the darknet backbone, used in the Darknet detection module. The configuration of the darknet inference process to rely on the GPU and CUDA + cuDNN, as well as the package integration, went off without a hitch. Because the Middleware communication interface, in section 3.3 was built from the ground up to function as a node, no extra integration was required. The position controller node required a number of modifications to function with ROS's publisher-subscriber technique. It was necessary to first adapt the module to subscribe to the detection output messages and then to publish the right message to the velocity commands for the Parrot Bebop 2 drivers once all the node had been configured to comply with the library's requirements and the node workflow had been designed.

5.2 Preliminary Results

The results of the preliminary tests conducted on the dissertation's suggested model are presented in this part. Since it was not feasible to test the entire system in operation simultaneously with more force, the modules were tested separately. Military vacations prevented any workers from entering the facility, and construction work made the use of the Alfeite laboratory exceedingly precarious. As a result, the tests could only be performed the previous recorded data in simulated environments. The implemented techniques for gathering the initial data are presented in detail, allowing for a thorough assessment of the model's performance. The preliminary results that support the UAV detection are presented at the beginning of the first part. This chapter's second section focuses mostly on the Position Controller Module's initial findings.

5.2.1 UAV Detection

The UAV detection vision system aims to accurately recognise and identify the UAV in every frame while also computing its three-dimensional position in relation to the camera by analysing the data collected from the helipad's camera.

5.2.1.1 YOLOv3 vs Other Frameworks Benchmark

An object detection framework must be integrated into the model, as was previously discussed in chapter 3, in order to construct a UAV detection model. There are, however, a number other frameworks with related goals. The crucial traits that make a detection framework applicable to this model are discussed in this part, along with the reasons the YOLOv3 was chosen among the other candidates.

In theory, framework comparison studies might be used to determine which framework is most suitable in terms of accuracy and speed. Unfortunately, the majority of studies that have been found use a variety of frameworks and are frequently carried out in various settings using unique test devices and test sets. In order to determine the best frameworks, a number of studies had to be taken into account and compared. YOLOv3 and Faster R-CNN are two examples of the most recent and sophisticated object identification frameworks, and the table 5.3 is an example taken from [55] and shows the average precision from both frameworks. The RetinaNet offers the most accurate results, but it also performs at the slowest speeds, incapable of real-time detection. Despite showing precision values that are comparable to those from YOLOv3, the SSD variations are also shown in [55] to be 3 times slower. The best Average Precision (AP) results are presented by faster R-CNN and YOLOv3. On the other hand, Mask R-CNN adds the mask to the output image while showing results with almost the same speed and accuracy as Faster R-CNN with Feature Pyramid Network. For the purposes of this dissertation, object segmentation is crucial since it can pinpoint the exact location of the UAV region, rather than just its general shape. This is crucial, especially when the UAV is visible in the acquired images towards the edge where there is the maximum lens distortion, rendering the large created bounding box useless. The fastest speed rates, nearly twice as fast as those of Mask R-CNN, are provided by YOLOv3 [54], which, along with its great accuracy, may be the deciding factor.

	backbone	AP	AP ₅₀	AP ₇₅	AP _S	AP _M	AP _L
<i>Two-stage methods</i>							
Faster R-CNN+++	ResNet-101-C4	34.9	55.7	37.4	15.6	38.7	50.9
Faster R-CNN w FPN	ResNet-101-FPN	36.2	59.1	39.0	18.2	39.0	48.2
Faster R-CNN by G-RMI	Inception-ResNet-v2	34.7	55.5	36.7	13.5	38.1	52.0
Faster R-CNN w TDM	Inception-ResNet-v2-TDM	36.8	57.7	39.2	16.2	39.8	52.1
<i>One-stage methods</i>							
YOLOv2	DarkNet-19	21.6	44.0	19.2	5.0	22.4	35.5
SSD513	ResNet-101-SSD	31.2	50.4	33.3	10.2	34.5	49.8
DSSD513	ResNet-101-DSSD	33.2	53.3	35.2	13.0	35.4	51.1
RetinaNet	ResNet-101-FPN	39.1	59.1	42.3	21.8	42.7	50.2
RetinaNet	ResNeXt-101-FPN	40.8	61.1	44.1	24.1	44.2	51.2
YOLOv3 608 × 608	Darknet-53	33.0	57.9	34.4	18.3	35.4	41.9

Figure 5.3: Various frameworks accuracy, adapted from [55].

These were the frameworks that were originally examined to incorporate the implemented model, since YOLOv3 and Mask R-CNN produced the two frameworks with the best overall results for the COCO data set. To determine whether model network is better suited, both were trained on the particular topic from this dissertation. Both utilised the same data set (explored in 4.3.1) and Google CoLab notebooks for training. The entire data set needs to be transformed to either PNG or JPG before being used with the YOLOv3 darknet architecture. For this paradigm, each image's annotation format is essentially a text file, with each file's structure consisting of one integer and four float values per object. The float values denote the region in the image where the object is located, and the integer designates the class to which the object belongs (the text file contains all classes

Framework	mAP
Mask R-CNN	63.2
YOLOv3	89.1

Table 5.1: Mean Average Precision for both frameworks with IoU=0.5.

to be taught).

RoboFlow Annotate, a graphical picture annotation tool, was utilised to make it easier to extract the right float values for each instance of an object [62]. With the help of this web-based application, you can graphically place a box around each region where an item is inserted and give it a class that has already been defined. Utilizing a network that has already been taught is another option. This entire process was completed in a fair amount of time thanks to the ability to save the annotation file afterwards in the YOLO format. Following training, the inference procedure was carried out by both networks on a test set; this process is covered in more detail in section 5.2.1.3. The Mean Average Precision (mAP) (section 5.2.1.3) of a trained model can be calculated using the YOLO framework, just like with Mask R-CNN. The results of computing both networks' mAP are shown in table 5.1.

The Mask R-CNN framework, even with a lower level of complexity and no segmentation process, delivers inferior detection results, as shown by a comparison of the two values. The YOLOv3 inference times are more than enough for a real-time UAV detection model that requires faster than 1 ms answers. As a result, despite its faster detection and subpar precision, the Mask R-CNN network does not exhibit a significant benefit. These facts suggest that the suggested model is not as well suited for integration into this framework, which is unable to outweigh the benefits of YOLOv3 instance segmentation. In order to integrate the paradigm suggested in this dissertation, the YOLOv3 framework was adopted.

5.2.1.2 Training Evaluation

All loss values were calculated and plotted during the training procedure, as was previously indicated in 4.3.5. To guarantee the appropriate augmentation steps were used, the network was trained numerous times using the parameter values provided in section 4.3.4 against the same data set.

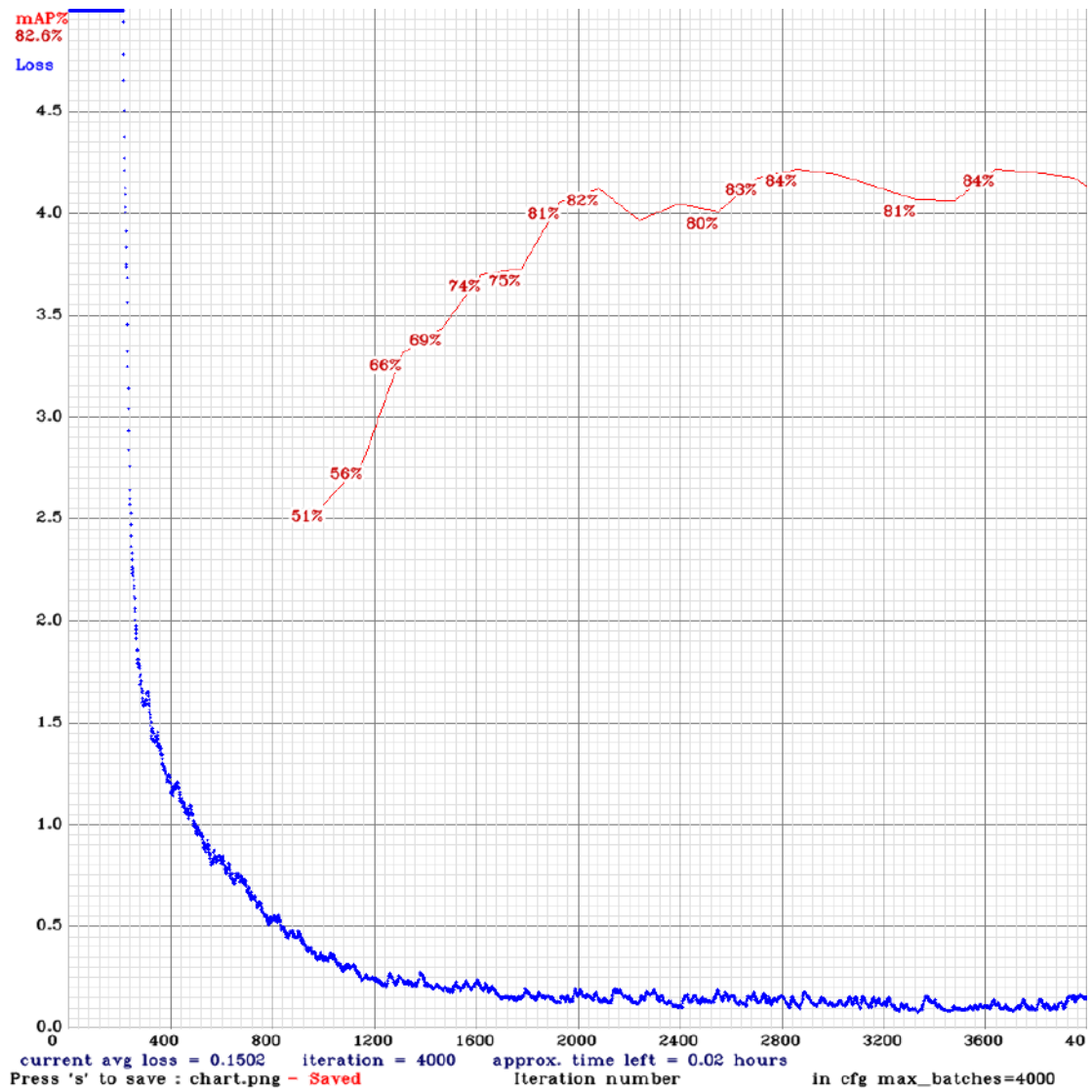
The behaviour depicted in each of the graphs in the following subsections is in respect to the training methods that produced the best results, not to those that produced the most typical behaviour. Every training resulted in a reduction in learning rate, and subsequent training sessions included several repetitions of this value. Most of the time, this drop was practised during and after the over fitting behaviour of the training. None of the training procedures that were visible took place for more than 4000 batches.

5.2.1.2.1 Training Results

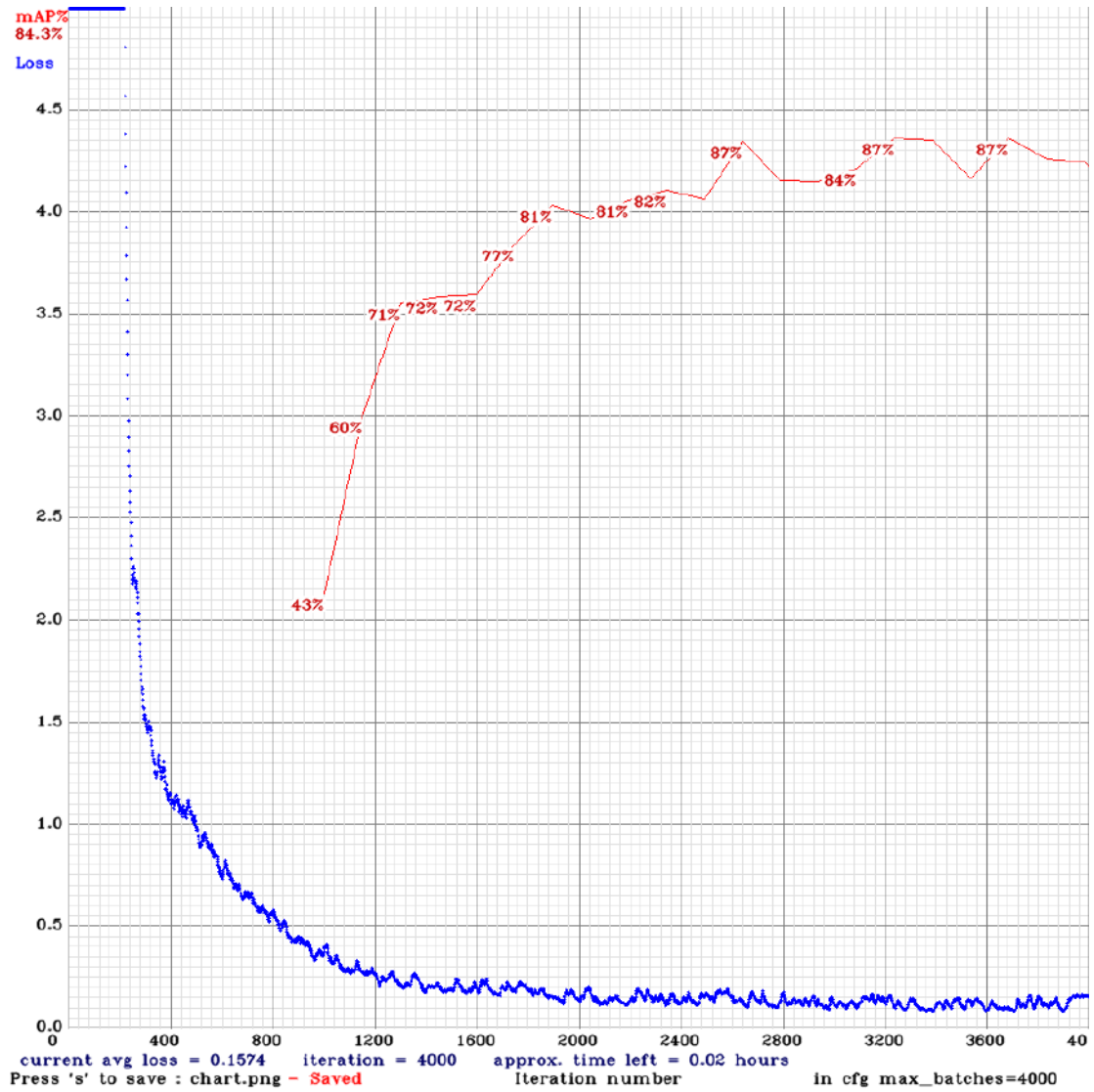
The **Training Loss** for the general training behaviour, which is comprised of the training losses for the Masks and **Bounding Box**, is depicted in figure 5.4. Figure 5.4a, the first training phase, **Parameter Pre-Train**, revealed a previously unrecorded insight into the appropriate start parameters. The *hyper-parameterization* flag that acts as an optimizer throughout the training phase produced noticeably worse results from an inference standpoint, but it allowed the network to recalculate the parameters in accordance with its requirements using our new data set. The second training phase, **Full Train**, figure 5.4b, highlighted the significance of the parameters from the first training method. The model received the proper micro-parameter modifications from the *hyper-parameterization* flag, greatly improving, training wise, this phase results. With the exception of a 0.01 loss decrease, the final training phase, **End-Spectrum Train**, figure 5.4c, demonstrated no discernible impact on the metrics used to record the train results. However, it had remarkable performance in the validation area.

The **Bounding Box** surrounding the region of interest and the masks overlapping the objects are the main emphasis of the presented model, so even if there are additional loss measures, these two were chosen to be the determining ones. For ease of analysis, the metrics are merged into the total training loss. Reaffirming the reason the Darknet + YOLOv3 architecture was chosen for this dissertation is because the Loss values reduce early in the training, around the 800 iterations. Any of the graphs can be used to determine that the over fitting behaviour started about batch 2000, which is also the point at which the training loss ceased to decrease. Both Mask and **Bounding Box** would offer comparable courses where the slower learning rate was advantageous. Even though it could not totally stop the over fitting behaviour, it was still able to obtain lesser loss values on average.

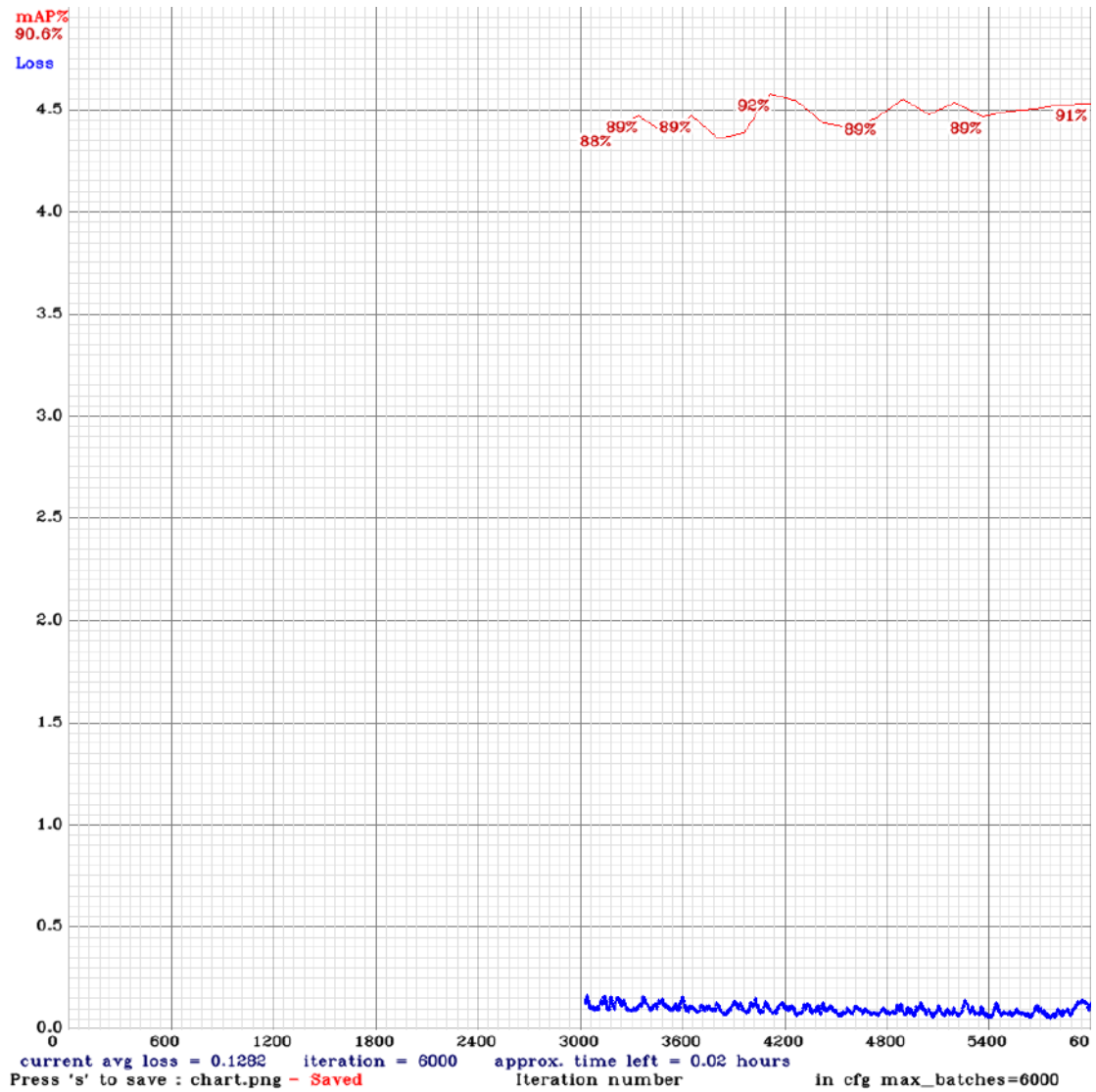
Each RGB Darknet YOLOv3 network training takes an average of 7 hours to complete, with iterations lasting between 2 and 7 seconds each.



(a) Training with the entire data set. In comparison to the initial start parameters calculated in the previous chapter, the training from this network provided improved start parameters at the conclusion.



(b) Training with the entire data set. The start parameters established by the prior training of the network were used to train this one, providing improvement differences in the recorded behaviour.



(c) Training with the entire data set. Evolution of the weights derived from the entire train to enhance the validation procedure.

Figure 5.4: RGB channel training behaviour of the Helipad's camera using YOLOv3 Tiny with Darknet framework. With Validation mAP (%) in red and total Training loss in blue.

5.2.1.2.2 Training Results Analysis

The table 5.2 below gives a clear overview of these results by succinctly listing the train loss values, for each train phase.

Table 5.2: Training Loss metrics for each training phase. The Preliminary Train is covered in the previous chapter’s section 4.3.5, while this chapter’s prior section 5.2.1.2.1 describes the three final phases.

	Preliminary Train	Parameter Pre-Train	Train	End-Spectrum Train
Training Loss	0.179341	0.099600	0.070867	0.206791

The initial findings showed that the network was a great fit for this object detection application, with better than average loss results. Since the goal of the parameter pre-train phase is to effect validation, it was not surprising that the train loss metric was unaffected. Acceptable results in terms of loss values were shown during the Train phase. The table shows that the values of the end-spectrum train loss are those with the least influence. Its train was constructed over a previous one, therefore this behaviour was planned.

The table makes it evident that, despite the fact that the results from the parameter pre-train were significantly better than those from the preliminary train, the results from the simple train were by far the best, with the lowest values in every loss metric. The great outcomes of the Parameter Pre-Train phase are validated by the total validation loss, which represents a 10% drop from the preliminary train.

5.2.1.3 Inference Evaluation

A more practical method is necessary to gauge the effectiveness of the implemented model in real-world circumstances, even if it is crucial to examine the training process and compare loss values between the three employed channels. As a result, the [Mean Average Precision \(mAP\)](#) with [Intersection over Union \(IoU\)](#) of 0.5 was computed in order to assess the accuracy, effectiveness, and robustness of this model.

The average of all images’ [Average Precision](#) values, which are directly influenced by precision and recall measures, produces the [mAP](#) method. Precision is a metric that expresses how many predictions out of all predictions are accurate, and can be calculated with:

$$Precision = \frac{TP}{TP + FP} \quad (5.1)$$

On the other hand, recall counts the number of instances that were accurately identified and is determined by:

$$Recall = \frac{TP}{TP + FN} \quad (5.2)$$

The ideal **Mean Average Precision** for an image is 1, with all recall values matching solely to precision values of 1. The **AP** for an image corresponds to the area under the accuracy-recall curve.

The **Intersection over Union (IoU)** assessment metric is important because object detection is not a binary problem and its success cannot be determined by merely determining if the predicted detection agrees with the actual detection. The **IoU** is designed to reduce the evaluation's rigour, allowing for the acceptance of detections with minor discrepancies. This metric is determined using:

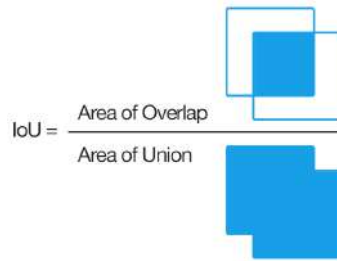

$$\text{IoU} = \frac{\text{Area of Overlap}}{\text{Area of Union}}$$

Figure 5.5: **IoU** calculation, adapted from [65]

A **True Positives** is deemed a success if the value obtained is higher than the specified threshold; on the other hand, a **False Positives** is deemed a failure. The threshold value was set to 0.5 for the purposes of evaluating the findings because this is the default value in the majority of apps and is sufficient for object detection in photographs shot from a distance of between 10 and 120 metres.

5.2.1.3.1 Test set Structuring

A test set was chosen that had a collection of 123 photographs that were all taken using the same camera in order to apply the **mAP** measure. Unfortunately, it was not possible to collect more data to create a fresh data set to be used as a test set because to the end limits of the Covid-19 epidemic and the time these experiments were completed. As a result, a collection of photographs from the validation set was chosen to make up the test set. Two subsets were taken from this testing group, just as the training set, to test the implemented model for various bands under the same circumstances. In order to create the most diverse and representative test set possible, the photographs were chosen based on their backgrounds, distinctive qualities, and differences from one another. By including the widest range of available examples, the data sets can be arranged according to environment types and the binary presence of items to clearly illustrate their diversity and streamline the results evaluation process. The Images are further categorised as having an *uav* or *Null* label. Table 5.3 shows their inclusion in the data set as well.

Depicts the regions of the train set image frame where **UAV** class objects are most likely to be detected, or, to put it another way, the image pixels where the drone object

Table 5.3: Number of images in the data set from each category.

Categories (%)			
Null	25.2	Cloudy	37.7
uav	74.8	Sunny	68.3

is displayed more often. The outcomes are overlapped, and the areas with more drone-related objects in the photos appear lighter. The 640×480 pixel image frame and the areas where the UAV is most active are shown in the figure below. The test set was designed to duplicate the same heatmap distribution behaviour of the train data because the heatmap of the train set is indicative of the complete dataset.

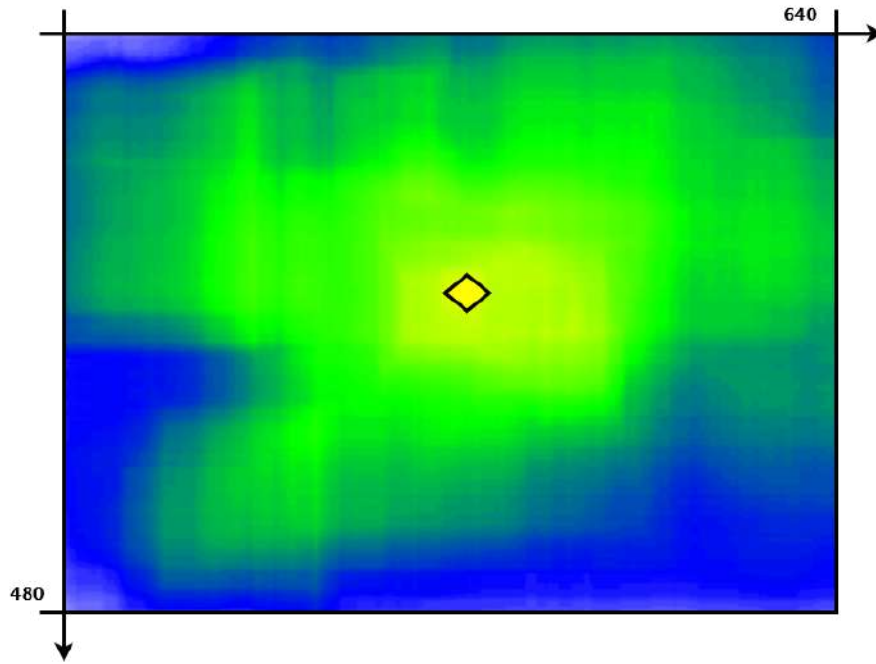


Figure 5.6: A heatmap showing the distribution of the UAV class object in the image frame. The yellow diamond shape in the figure represents the area where the UAV is most likely to be found in the dataset.

5.2.1.3.2 Inference Results

The detection confidence criterion for the inference tests that follow was set to 0.8 in a Google CoLab notebook. All of the various photos were fed into the model for each subset, and the number of True Positives (TP), False Positives (FP), and False Negatives (FN) was recorded. The table 5.4 below gives a clear overview of these results by succinctly listing the validation mAP, for each train phase.

It is evident from the table that the mAP computed values agree with the inferences made in the preceding section. Since there was no *uav* class defined in the COCO dataset and no inference time because this network train was never deployed, the Preliminary

Table 5.4: Validation **Mean Average Precision (mAP)** values for each train phase with **IoU=0.5** and Inference Times in frames per second. The Preliminary Train is covered in the previous chapter’s section 4.3.5, while this chapter’s prior section 5.2.1.2.1 presents the three final train phases results.

	Preliminary Train	Parameter Pre-Train	Train	End-Spectrum Train
Validation mAP (%)	45.32	74.31	86.91	91.57
Inference Time (fps)	-	-	17	55

Train produced subpar results. The Parameter Pre-train, as expected, has a very low value of 74.31%, and its difference from the others confirms that the computed parameters were not suitable for use in the final system in the real world on their own. Similar to the previous phase, no inference time measure was calculated in this one.

The Train phase proved to be more precise than other systems’ approaches and to be suitable in the field of uav detection. However, it clearly demonstrated fragility with a **mAP** value that was still below the intended level in several of the output instances given in the previous article. This is because the mildly specialised training set was smaller and less varied than expected, and the image’s background lacked sufficiently various items.

End-Spectrum Train, the last step, boosted the validation **mAP** significantly since it fed the network the whole spectrum of either augmentation or training method optimization in order to get the best, and final, results that would be used in the real-life model. Because a more potent **GPU** was used in this stage than in the previous ones, inference performed significantly more quickly.

When presented with higher altitude images, both Train and End-Spectrum results demonstrated an overall greater detection ability. Most of the time, these images were given with more confident confidence values and more assertive uav localization. Lower altitudes make the contrast observed surrounding most uav occurrences less noticeable, and both train phases performed worse with this kind of data. This demonstrates that the model works better with photographs taken at higher altitudes, and it would be ideal for the **UAV** to be flying at those heights when it is being utilised in a real-world scenario.

Additional changes must be made to the visual data detection, mostly to less-than-ideal detection conditions. The network needs more thorough training using bigger data sets with information from as many different situations as feasible. The data should be collected at lower altitudes and should contain many examples of items that may be present in marine settings frequently, such as cranes and ship equipment, as well as under bridges and aeroplanes, in order to provide the best improvement.

5.2.1.3.3 Inference Results Analysis

It is feasible to make a substantial inference about the model’s inference outcomes by taking into consideration the computed precision values, the analysed output instances, and the training results.

FP situations typically happen when there are abrupt brightness fluctuations in the image, which are brought on by changes in the camera's exposure because of the sun's motions. The network requires a specific number of frames to adjust to the new luminosity values because the majority of the image experiences a change in the intensity of its colours. The algorithm demonstrated its ability to recover and resume working appropriately after being given enough frames to adjust.

Other FPs are brought on by rapidly moving clouds. The Middleware Interface module should be able to counteract the negative effects of the blobs caused by clouds, so in general, this effect is timely and only produces a few sparse lone blobs in the bounding box.

Most FN occurrences are caused by the UAV loitering, with little translation movement, at extremely high altitudes, which lessens the visibility of the propeller movement. Poor luminosity conditions occasionally did not appear to be an issue; on the contrary, the best outcomes were demonstrated. The movement of the propeller can become extremely shaky, which causes the UAV bonding boxes to be included into the background model and become deformed. This could result in an increase in false positive cases, especially in overcast situations, although there will typically be a cost. Since the issue only arises in a very specific circumstance (such as when the UAV is hovering motionless at great heights), the module is able to track the UAV's location until its movements are once again discernible. The UAV should be deemed to have departed the frame when a predetermined length of time has passed.

When the sun is high in the sky and close to the centre of the picture, that presents yet another challenge. The saturation effect, or almost entirely white pixels in that area, caused by the sun and its glares on the camera lens can make it difficult or impossible to spot the UAV when it is flying over that area of the image. Although there isn't much that can be done in terms of picture processing to fix this issue, changes to the hardware can help to reduce it. For instance, changing the camera's aperture and other settings can be utilised to lessen the effect. Another intriguing option is to employ a luminosity sensor to make the modifications automatically. Similar to how solar glares cause areas of the image to have poor colour saturation and ultimately be perceived as clouds in the cloud mask, affecting the quality of the image as a whole. For these reasons, in order to reduce this issue, the cloud mask's parametrization must be fairly pessimistic.

5.2.2 Position Controller

The high-level control system in this part is designed to safely guide the UAV from an aerial position to the surface of the landing pad. Within the constraints of the data recording, the preliminary results of this module are reported in this section. The Alfeite laboratory underwent extensive construction work, making use of the area extremely precarious, and military vacations prevented any staff from entering the facilities, thus the experiments could only be carried out in a simulation.

5.2.2.0.1 Simulation Environment

Using a simulated environment, the Controller module was tested. It was decided upon the scenario that follows: The world's centre is occupied by a static, fictitious helipad with a fictitious camera pointed skyward at its centre. Due to issues integrating Intel RealSense libraries with the Sphinx modelling environment, the helipad is not an active component. Because of this, data interchange and communication between the UAV and the helipad are not possible. The UAV, on the other hand, is outfitted with a GPS in addition to a Parrot-provided IMU for pose estimation.

When configuring the simulator with more recent versions of Ubuntu and ROS, integration issues with the Parrot Sphinx simulator environment and Gazebo surfaced as well. To address this, it was necessary to downgrade Sphinx's version in order to ensure a stable and consistent environment, which compelled the system to do the same. This issue caused a tear in the information flow from the modules to the UAV simulation, which needed to be addressed. Setting up a Virtual Machine (VM) environment system that was totally isolated and tailored just for simulation was the answer. But because the VM couldn't handle real-time data transfer quickly enough for a UAV simulation to work, the information flow between the other modules wasn't fully resolved. Following the process for the model suggested in section 3, figure 3.2, the system was tested as individual modules, capturing the inputs and outputs of each model and feeding the desired ones to the following model. Utilizing a tool from the ROS ecosystem, these values are recorded in order to produce a bag file containing all of the released data for the topics.

As a result, we were able to record the real inputs/outputs, the module's code, as well as the data provided, in a more effective and secure manner than before utilising this test methodology.

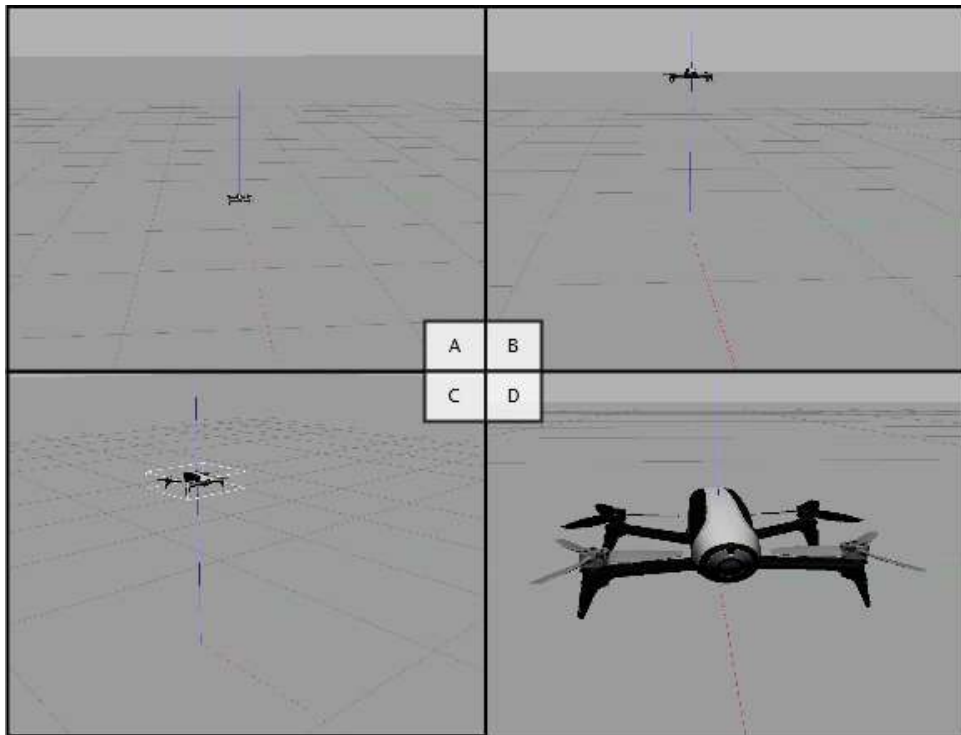


Figure 5.7: Parrot Sphinx + Gazebo 9 Simulation Environment. Ground plane and coordinate axis in a simple universe. The global file was simplified to conserve processing power. The UAV may be seen flying in images B and C. A close-up of the Parrot Bebop 2 model is seen in Image D. Rotor interference and movement are modulated in the surrounding area.

5.2.2.0.2 Simulation Results

A graphic illustration of the controller error deviation applied at various points throughout the landing manoeuvre is shown in Figure 5.8. This section describes the simulated environment and tests that were conducted in order to obtain the three different captured findings.

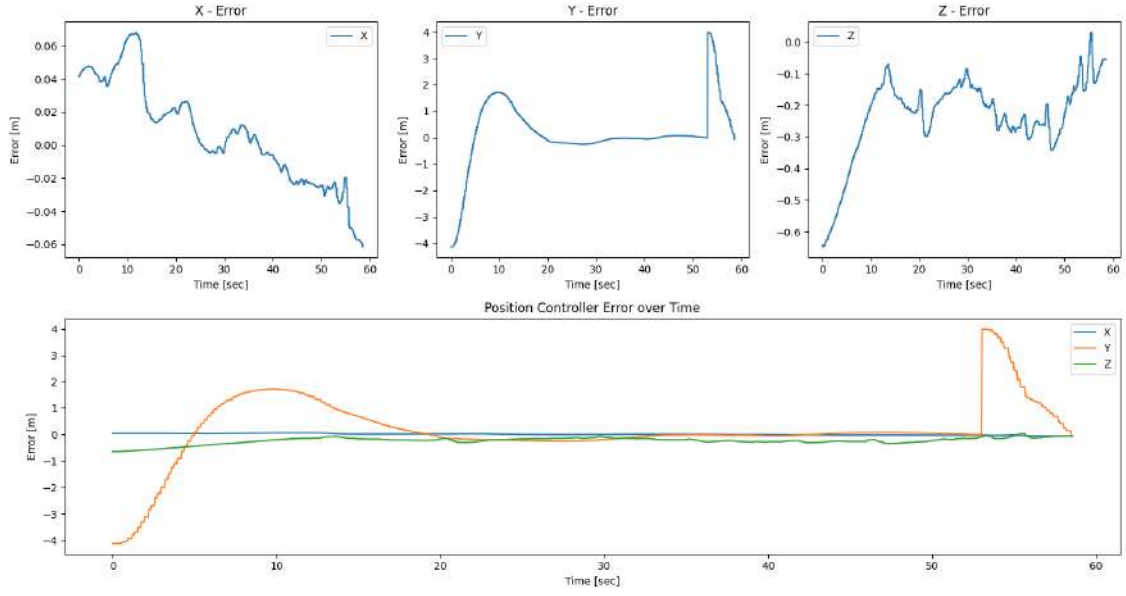
The 2D Position Alignment phase is being tested in the first scenario, depicted in figure 5.8a. The module receives an input stating that the target location is 4 metres away along the Y axis. When this is accomplished, the module receives another input indicating that the target point has returned to the initial start location, or, in terms of distance, negative 4 metres along the Y axis. The metrics that are recorded are the deviation errors of the three PID controllers for coordinates X, Y, and Z, each of which affects a different coordinate. Only one coordinate was impacted in order to make the experiments easier to post-evaluate and less demanding on computational resources. The X and Y coordinates are given the same cloned PID and tweaked gain parameters.

The second and final test scenarios are inside on the last precision descent of the Landing phase. This stage occurs after the UAV has been correctly positioned above the helipad's centre and descent along the Z axis can begin; as a result, there is very little inaccuracy recorded on the X and Y coordinates in the results of this test. In order to verify the approach function in the suggested model, Figure 5.8b illustrates the first through last stages of fall with the UAV starting 5 metres above ground. By looking at the figure, one can see the times when the approach constraints kick in and cause the deviation error to soar, indicating that a stage has been successfully finished. Another test was created to capture every aspect of the final and most significant step in the landing sequence—the last metre above the earth. During the scenario's creation, preliminary tests revealed that it was required to include an offset to the ground, which meant that the ground would appear to the UAV to be at a lower altitude.

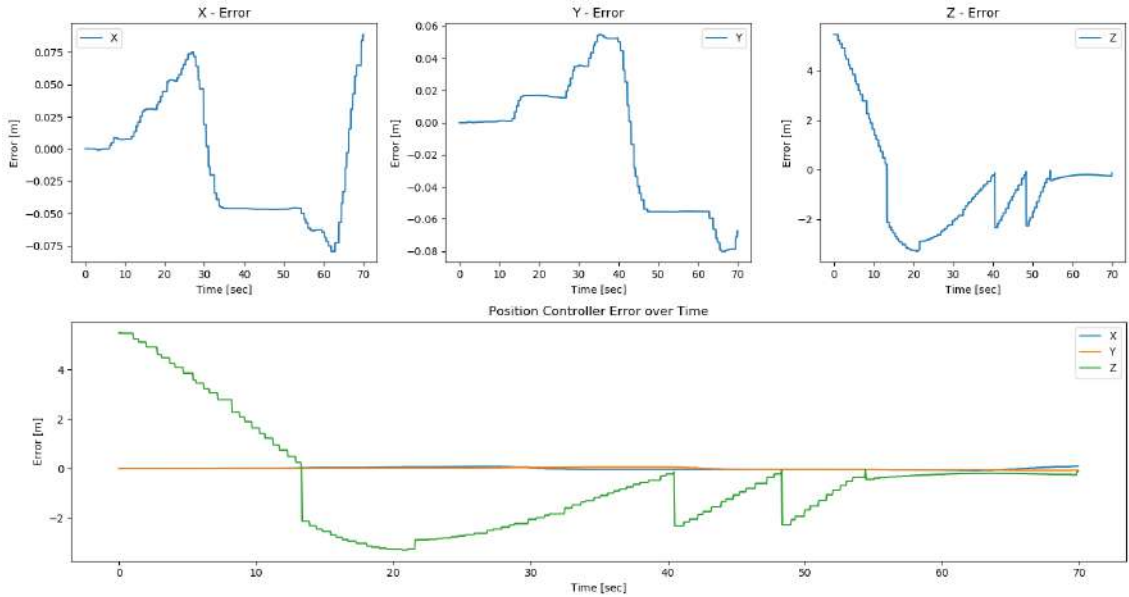
5.2.2.0.3 Simulation Results Analysis

The simulated UAV was safely landed by the control module in a respectable amount of time, with an average final distance of minus 6 cm from the helipad's centre. Although this modules results on precision deviation are astounding, it is crucial to fine-tune these parameters prior to the deployment of the model since too much precision can harm the PID controller's functionality.

As anticipated, the simulation experienced the same issue as the real-world footage and flight experience. It was determined via real-world pilot operating flights tests of the drone that the Parrot Bebop 2's on-board firmware contains built-in mechanisms to prevent it from colliding with the ground. This method has no effect whatsoever on the drone's 2D position because it only becomes apparent at lower altitudes. The consequences appear as the descent sequence begins and the UAV identified centre of mass



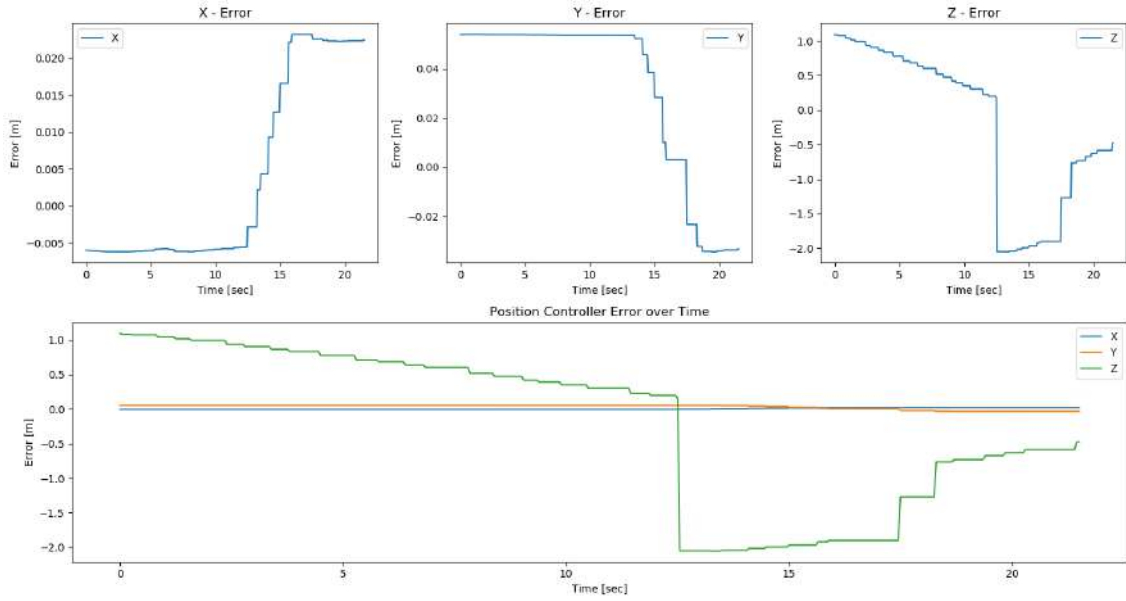
(a) Controller 2D Position Alignment of the Landing Phase simulation tests.



(b) Controller Final Landing Phase simulation tests. Stage immediately after the 2D Position Alignment done 5 meters above ground.

is aligned with the helipad. This translates to jitter in the controller velocity command received since the drone approximate pauses at a height that is unsuitable for landing. Nevertheless, some experiments demonstrated that the module was capable of reversing this. However, more studies are need to determine the exact nature of this erratic behaviour.

The PID controller as a stand-alone module performed the task successfully but wasn't exceptionally effective at it. The reaction took a very long time to get at the target and there was a significant amount of windup. Without responding to restrict or counteract



(c) Controller Final Landing Phase simulation tests. Detailed results on the last 1 meter descent above target.

Figure 5.8: Results of simulation tests performed on the Position Controller module to evaluate its functionality and suitability for use in the real world.

the effect, the **PID** continually overshoots the response. The results improved but were still not perfect after incorporating a threshold to the **PID** logic and tweaking the values. It's important to keep in mind that the controller's precision level at this point was on the millimetre scale, which meant that the controller would not tolerate any deviation from the calculated to target position. With these adjustments, a small-scale simulated test was run to accurately reproduce the overlapping impact of the control input module. This implies that the controller was continuously supplied new calculated position signals while the controller's reaction action was being monitored. The suggested technique, when used in this second case, dramatically improved the outcomes by allowing for a more precise and controlled response, greatly minimising overshoot, and allowing the UAV to land more steadily and closely to the centre of the helipad.

5.3 Real World Experimental Results

Here in figure 5.9 we can see the 3D trajectory graphs of the UAV during the test. Two position estimations are represented. The estimation performed by the drone itself is represented on the left while the estimation derived from the KAB detection is represented on the right. It is possible to make a correlation with the key moments. Although the flight was short, the error in the Bebop position estimation is evident. Since the test was performed in such a way that the UAV returns to the initial position to be easier to measure the error. We can verify that there is a deviation of approximately 1 metre on the X axis. In contrast, in the estimation performed from the detection we can see that there

was practically no error. Larger flights were performed to test the effect of the duration and also the drone's travel speed during the flight, on the Bebop Odometry error. It is concluded that the error increases a lot in larger flights, reaching an error of more than 7 meters. If the drone moves too fast, it leads to failures in the capture of video frames from the drone's own internal System, which produce odometry errors.

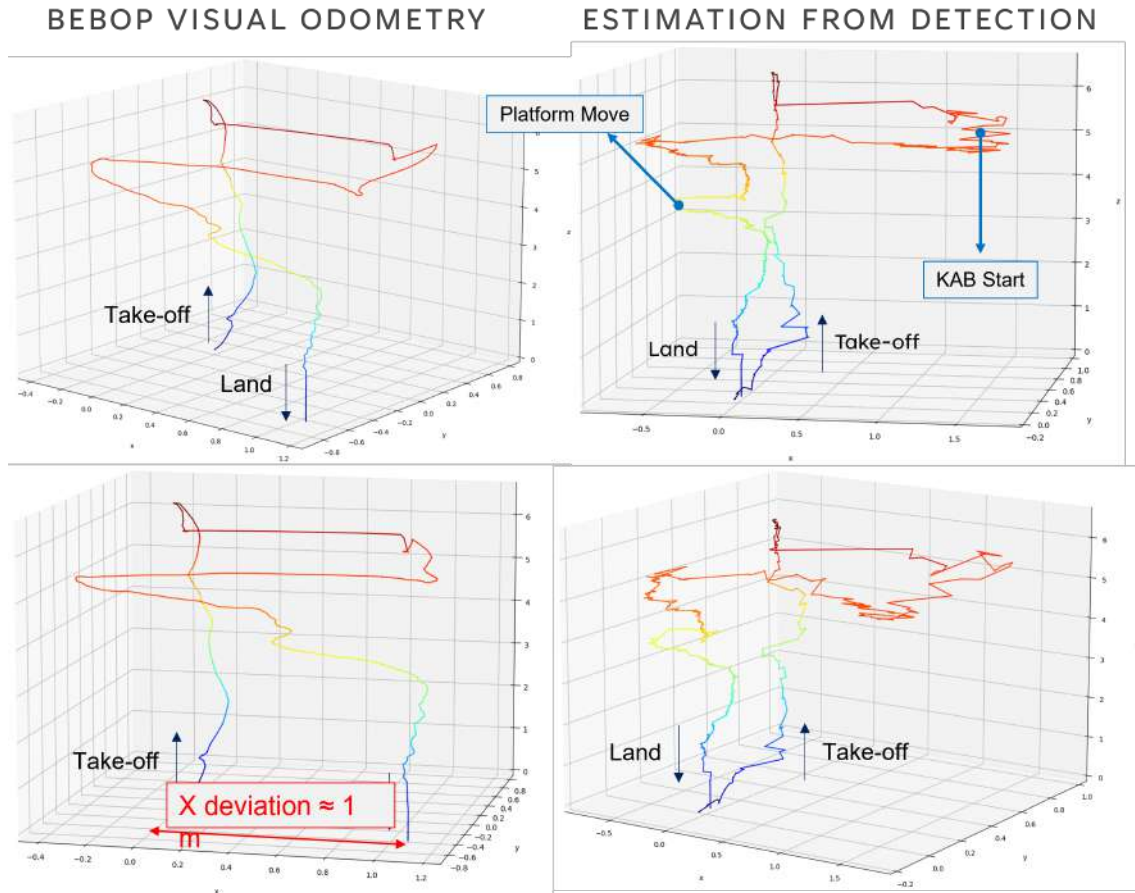


Figure 5.9: 3D Trajectory of real world flight tests.

Let's now analyze, in figure 5.10 in more detail and accuracy the landing precision and the controller behavior in this same flight.

Once again we have the same key moments identified in figure 5.10 for ease of reading. This graph shows in detail the difference between Bebop's Visual Odometry and the Estimation performed by Detection, separated by coordinates. X on top; Y in the middle; Z at the end. At the moment that the KAB is initialized the drone is in the position (X: 1.57 ; Y: 0.05; Z: 5.2), we can see that only 20 seconds after the flight starts already being visible error in the odometry of the Bebop. It is also possible to see the moment that the landing platform is moved. When the landing is complete we can see very different values. In relation to the X we have a deviation of about 1 meter. At the Y we have a deviation of 60 centimetres. The Z is done by reading the ultrasound sensor present in the UAV, a metric that is transmitted to GCS by default.

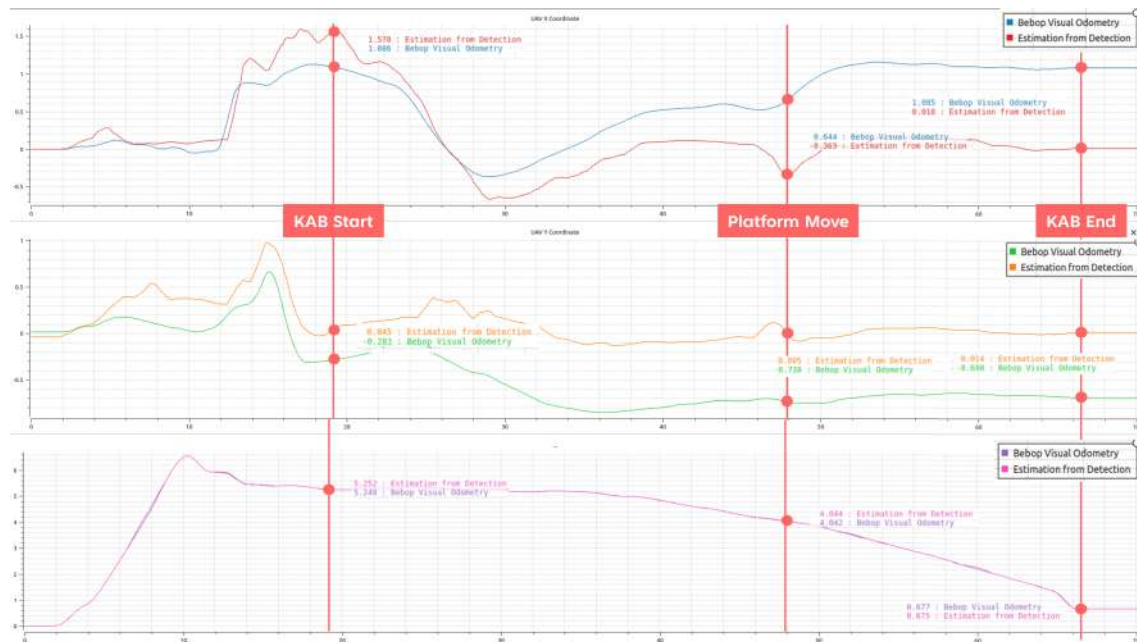


Figure 5.10: Flight coordinates detail.

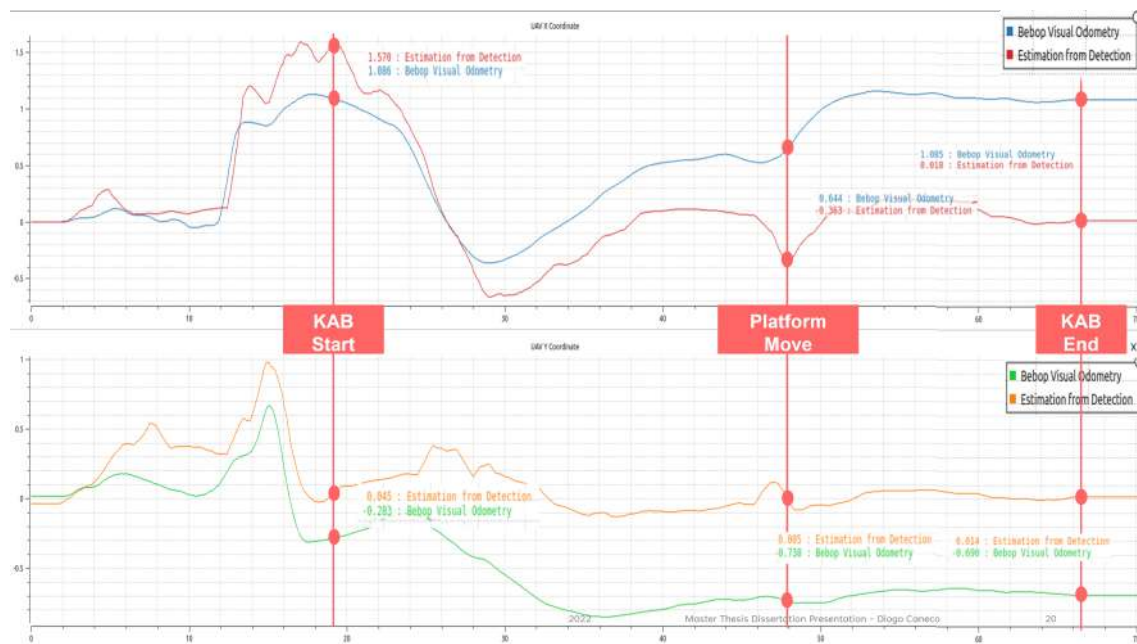


Figure 5.11: Flight coordinates detail X and Y coordinates.

We will now focus on the X and Y coordinates to analyse the Position Controller adapted for this dissertation in more detail in figure 5.11.

The controller is not perfect, since the parameters used for the PIDs that control the speed commands were not parameterized in detail for the situation. The parameterisation of a PID for a case with many degrees of freedom, such as drones, is an arduous and very difficult task, without using complex mathematical methods. We can see an overshoot around instant 30, of about half a meter, after the initial command is of a higher magnitude since the drone is further away from the target. To reduce the effect, the speed at which the UAV moves was limited to ensure greater stability of the System. The proposed solution was designed with mitigation of both odometry error and controller overshoot in mind. This is achieved by taking advantage of the high frequency in message transmission in ROS ecosystems. That is, the navigation waypoint sent to the controller is updated many times per second, and this causes the controller not to run on top of its own calculation. Instead a new control action is calculated completely free of cumulative error.

A video of the flight test analysed in this section is provided using the link in [KAB Real World Flight Test with Platform Moving](#).

CONCLUSIONS AND FUTURE WORK

This chapter includes a summary of the findings from the dissertation as well as recommendations for further study and potential system upgrades.

6.1 Conclusions

In this dissertation, a collaborative strategy is suggested for [Multi Rotor Vertical Takeoff and Landing \(MR-VTOL\) Unmanned Aerial Vehicle \(UAV\)](#)s to land on their own for the challenge at hand, which was a system meant to create a high-precision landing manoeuvre on a autonomous ship. A "smart"helipad, which is an active element capable of communication and data collection, is introduced as another robotic element into the scenario. The helipad uses a camera at its centre, with the optical axis pointing up towards the sky, to provide more details on the [UAV](#)'s position in relation to it. The system is intended to either function independently as the main means of guiding the [UAV](#) during its descent or as a component of a more complicated system with additional queues offering redundancy and enhancing overall robustness.

A State of the Art review was required to gain a better knowledge of the many technologies required for the proper execution of the solution suggested and to gain additional information on the work that has already been done in the area in issue. Initially, it concentrated on model, dynamics and kinematics, and control theory for multi-rotor [UAV](#)s, with a particular emphasis on quadrotor applications. To further suit the demands of this case study, the research subjects were broadened to cover high-level landing procedures, detection methods for autonomous landing and computer vision, simulation environments for multi rotor [UAV](#)s, and other more advanced control theories. Especially since the model must be developed concurrently with the experimental setting to prevent having to rework or recreate the model from start because the experimental testing

arrangement is too complex.

After that, a model to address the issue was suggested and thoroughly explained. The use of this paradigm is fully described in Chapter 3. The model was put into practise to receive a single image frame, identify the precise pixels where UAV instances were situated, overlap those pixels with coloured masks, and then record the resultant image and bounding box file. The detection portion of this model was carried out using the Darknet + YOLOv3 object detection framework. With its fine-tuned parameters, it was trained for various atmospheric and lighting conditions, and in Chapter 5 their inference outcomes were closely compared. This dissertation also includes a method for a position controller system, with the successful velocity commands provided.

The detection system's biggest drawback is when the sun is high in the sky because it causes saturated zones in the image that make it hard to extract any information. As a result, it may be virtually impossible to discern the UAV in the image. Adapting the camera lens to function in these circumstances is one technique to reduce the issue. Another issue can be that during the final centimetres of the fall, the UAV would not fit in the helipad camera's field of view.

It was suggested to use a high-level control system to direct the UAV to land on the landing pad. The control system issues sharper commands the farther the UAV is from the target and softer commands the closer it gets. It takes into account the relative location of the UAV and the helipad, their height differences, and their horizontal distances. The controller makes use of the multiplicative inverse curve-based idea of landing zone. The landing zone serves as a location where the UAV must be in order to make a secure landing. The movement's magnitude has a negative function, increasing at higher altitudes, decreasing gradually at lower altitudes, and eventually approaching zero in the final few centimetres of the drop. The simulation to test the results of the controller logic proved to be extremely useful to prepare the developed system to an application onto the real-world.

Unfortunately, outside forces intervened to abbreviate the experimental testing period, effectively nullifying it. The experimental laboratory facility chosen to test the UAV was located inside a military naval base, therefore their standards applied to how the space might be used. In addition, for about a month, the facility was closed due to military holidays, prohibiting access by non-military people. Simultaneously renovation work started on the upper floor of the laboratory producing great amount of dust and debris, typical of active construction sites, that made the use of the Alfeite laboratory exceedingly precarious. The designated flight test zone was the available location for storage when the RICS crew needed to transfer equipment to make room for the workers. As a result, there was less room available for the designed system to undergo controlled and safe flight tests. The dust produced settled to the ground, and when the drone would take flight the wind from the propellers would lift tremendous amounts of it, causing extreme discomfort on the human operators and bystanders. This adverse conditions proved harmful not only to the crew but also extremely deteriorating to the hardware equipment.

Despite the circumstances, attempts were made to test the system, but a straightforward tele-operated flight demonstrated that the system would be subjected to stress levels that would taint the outcomes.

The suggested approach further advances the study of autonomous landing systems, which are necessary for fully autonomous UAV operations. This dissertation aims to provide a fresh viewpoint on the overall issue by proposing an alternative strategy to conventional approaches. The proposed methodology can be used in conjunction with other established systems to overcome some of their shortcomings and provide a more dependable system that can manage a wider range of environmental conditions. A promising conclusion for future implementations was shown by the UAV Autonomous Landing system model using the Darknet + YOLOv3 framework. Even though the obtained findings might not be the ideal solution, they do demonstrate the model's promise in the area of autonomous drone landing, highlighting the want for additional study and improvement.

6.2 Future Work

There are some potential areas where the model could be improved. The first step is obviously testing the control system on a real UAV to confirm the outcomes of the simulation. The presence of wind, which can significantly affect the operation of the control system, is one of the additional issues that real-life settings bring. Using Light Detection and Ranging (LiDAR) devices is yet another potential solution to the problem of accurately calculating the UAV's centre while it is close to the camera. The Intel RealSense camera already has a pointcloud-capable LiDAR solution that is accurate to 9 metres. Another option is to place several LiDARs on the surface of the helipad and sweep the region directly above it. The data from the UAV can be cross-referenced when it is near the helipad, creating a point cloud that can be utilised to more accurately calculate the UAV's position during the final centimetres of the landing.

The data for the training procedure was gathered from two distinct groups of ground to sky photos, some of which had backgrounds that were comparable. With access to a larger collection of photographs collected from the most varied surroundings conceivable, this model would greatly benefit. More UAV flights would need to be scheduled in order to get this additional data, but that won't be possible given the constraints in place and the current weather. Additionally, it would be crucial to collect a sizable volume of data solely for testing. The inference tests conducted for this dissertation were satisfactory, however a new test set that the model has never encountered is required to ensure entirely impartial test results. More End-Spectrum training rounds were found to significantly improve the model during the network's training. According to calculations and network tests, an additional 4000 batches of end-spectrum training should improve the mAP outcomes by around 5%.

To further optimise the system, the controller module of the produced system can also be improved. First, a better inverse approximation function that fully utilises the

area around the landing space can be implemented. If the helipad is located on the bow of the ship, for example, modelling the ship to cross-reference the mutual space with the landing zone will ensure that the UAV performs the approximation from the bow of the ship, or the stern, if it is located on the stern. This allows the landing procedure to begin far from the central control tower, which houses the computer and sensor arrays. This guarantees that the UAV won't run into the ship's sensor arrays and cause damage. Modern control algorithms can also be utilised in place of the **Proportional Integrative Derivative (PID)** logic found in the controller module. According to the research conducted for this dissertation, applying Fuzzy Logic to the controller proved to be quite effective in UAV applications [66].

It would also be intriguing to construct a "smart helipad" that can direct UAVs during their landings. The helipad is entirely in charge of the landing because the suggested method is model-free and may function without requiring the UAV to carry out any special processing. The helipad might be a deployable component that can communicate with UAVs using a common protocol, making it the only asset required for the self-landing of any MR-VTOL UAV type.

BIBLIOGRAPHY

- [1] P. A. Prates et al. “Vision-based UAV detection and tracking using motion signatures”. In: *2018 IEEE Industrial Cyber-Physical Systems (ICPS)*. 2018, pp. 482–487. DOI: [10.1109/ICPHYS.2018.8390752](https://doi.org/10.1109/ICPHYS.2018.8390752) (cit. on pp. 4, 17, 24, 37, 46, 48, 49).
- [2] D. He, S. Chan, and M. Guizani. “Communication Security of Unmanned Aerial Vehicles”. In: *IEEE Wireless Communications* 24.4 (2017), pp. 134–139. DOI: [10.1109/MWC.2016.1600073WC](https://doi.org/10.1109/MWC.2016.1600073WC) (cit. on p. 6).
- [3] R. Miguel and F. Martins. “Controlo de Quadcopter”. In: (2019) (cit. on pp. 7, 8, 10–14).
- [4] S. Li et al. “Autonomous drone race: A computationally efficient vision-based navigation and control strategy”. In: *Robotics and Autonomous Systems* 133 (2020), p. 103621. ISSN: 09218890. DOI: [10.1016/j.robot.2020.103621](https://doi.org/10.1016/j.robot.2020.103621). arXiv: [1809.05958](https://arxiv.org/abs/1809.05958). URL: <https://doi.org/10.1016/j.robot.2020.103621> (cit. on pp. 7, 19).
- [5] H. Moon et al. “Challenges and implemented technologies used in autonomous drone racing”. In: *Intelligent Service Robotics* 12 (2019). DOI: [10.1007/s11370-018-00271-6](https://doi.org/10.1007/s11370-018-00271-6) (cit. on p. 7).
- [6] H. C. Fernando et al. “Modelling, simulation and implementation of a quadrotor UAV”. In: *2013 IEEE 8th International Conference on Industrial and Information Systems, ICIIS 2013 - Conference Proceedings*. IEEE, 2013-12, pp. 207–212. ISBN: 9781479909100. DOI: [10.1109/ICIInfS.2013.6731982](https://doi.org/10.1109/ICIInfS.2013.6731982). URL: <http://ieeexplore.ieee.org/document/6731982/> (cit. on pp. 7, 8).
- [7] M. Walid et al. “Modelling, identification and control of a quadrotor UAV”. In: *2018 15th International Multi-Conference on Systems, Signals and Devices, SSD 2018* October (2018), pp. 1017–1022. DOI: [10.1109/SSD.2018.8570512](https://doi.org/10.1109/SSD.2018.8570512) (cit. on pp. 8, 15, 16).
- [8] O. Bouaiss, R. Mechgoug, and A. Riadh. “Modeling, Control and Simulation of Quadrotor UAV”. In: 2020, pp. 340–345. DOI: [10.1109/CCSSP49278.2020.9151687](https://doi.org/10.1109/CCSSP49278.2020.9151687) (cit. on p. 10).
- [9] V. Martinez. “Modelling of the flight dynamics of a quadrotor helicopter”. In: (2007) (cit. on p. 12).

-
- [10] M. Bergamasco and M. Lovera. “Identification of Linear Models for the Dynamics of a Hovering Quadrotor”. In: *Control Systems Technology, IEEE Transactions on* 22 (2014), pp. 1696–1707. DOI: [10.1109/TCST.2014.2299555](https://doi.org/10.1109/TCST.2014.2299555) (cit. on p. 17).
 - [11] J. Li and Y. Li. “Dynamic analysis and PID control for a quadrotor”. In: *2011 IEEE International Conference on Mechatronics and Automation*. IEEE, 2011-08, pp. 573–578. ISBN: 978-1-4244-8113-2. DOI: [10.1109/ICMA.2011.5985724](https://doi.org/10.1109/ICMA.2011.5985724). URL: <http://ieeexplore.ieee.org/document/5985724/> (cit. on p. 18).
 - [12] V. M. Babu, K. Das, and S. Kumar. “Designing of self tuning PID controller for AR drone quadrotor”. In: *2017 18th International Conference on Advanced Robotics (ICAR)*. IEEE, 2017-07, pp. 167–172. ISBN: 978-1-5386-3157-7. DOI: [10.1109/ICAR.2017.8023513](https://doi.org/10.1109/ICAR.2017.8023513). URL: <http://ieeexplore.ieee.org/document/8023513/> (cit. on p. 18).
 - [13] W. Giernacki. “Iterative Learning Method for In-Flight Auto-Tuning of UAV Controllers Based on Basic Sensory Information”. In: *Applied Sciences* 9.4 (2019-02), p. 648. ISSN: 2076-3417. DOI: [10.3390/app9040648](https://doi.org/10.3390/app9040648). URL: <https://www.mdpi.com/2076-3417/9/4/648> (cit. on p. 18).
 - [14] Y. Feng et al. “Autonomous Landing of a UAV on a Moving Platform Using Model Predictive Control”. In: *Drones* 2.4 (2018-11), p. 34. ISSN: 2504-446X. DOI: [10.3390/drones2040034](https://doi.org/10.3390/drones2040034). URL: <http://www.mdpi.com/2504-446X/2/4/34> (cit. on p. 18).
 - [15] “3D Convolutional Neural Networks for landing zone detection from LiDAR”. In: *Proceedings - IEEE International Conference on Robotics and Automation* 2015-June. June (2015), pp. 3471–3478. ISSN: 10504729. DOI: [10.1109/ICRA.2015.7139679](https://doi.org/10.1109/ICRA.2015.7139679) (cit. on p. 18).
 - [16] A. A. Cabrera-Ponce and J. Martinez-Carranza. “A vision-based approach for autonomous landing”. In: *2017 Workshop on Research, Education and Development of Unmanned Aerial Systems (RED-UAS)*. IEEE, 2017-11, pp. 126–131. ISBN: 978-1-5386-0939-2. DOI: [10.1109/RED-UAS.2017.8101655](https://doi.org/10.1109/RED-UAS.2017.8101655). URL: <http://ieeexplore.ieee.org/document/8101655/> (cit. on pp. 19, 25).
 - [17] T. Krajník et al. “AR-Drone as a Platform for Robotic Research and Education”. In: *Communications in Computer and Information Science*. Vol. 161. 2011. ISBN: 978-3-642-21974-0. DOI: [10.1007/978-3-642-21975-7_16](https://doi.org/10.1007/978-3-642-21975-7_16) (cit. on p. 19).
 - [18] M. H. Weik. “Parrot Bebop 2 - User Guide”. In: *Computer Science and Communications Dictionary* (2000), pp. 1873–1873. DOI: [10.1007/1-4020-0613-6_20584](https://doi.org/10.1007/1-4020-0613-6_20584) (cit. on pp. 19, 31).
 - [19] D. Pagliari and L. Pinto. “Use of fisheye Parrot Bebop 2 images for 3D modelling using commercial photogrammetric software”. In: *International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences - ISPRS Archives* 42.2 (2018), pp. 813–820. ISSN: 16821750. DOI: [10.5194/isprs-archives-XLII-2-813-2018](https://doi.org/10.5194/isprs-archives-XLII-2-813-2018) (cit. on p. 19).

- [20] G. Silano, P. Oppido, and L. Iannelli. "Software-in-the-loop simulation for improving flight control system design: A quadrotor case study". In: *Conference Proceedings - IEEE International Conference on Systems, Man and Cybernetics* 2019-Octob (2019), pp. 466–471. ISSN: 1062922X. DOI: [10.1109/SMC.2019.8914154](https://doi.org/10.1109/SMC.2019.8914154) (cit. on pp. 20, 31, 32, 35).
- [21] M. Monajjemi. "bebop-autonomy - ROS Driver for Parrot Bebop Drone (quadrocopter) 1.0 and 2.0 - bebop-autonomy indigo-devel documentation". In: 2015. URL: <https://bebop-autonomy.readthedocs.io/en/latest/> (cit. on p. 19).
- [22] N. Xuan-Mung et al. "Autonomous Quadcopter Precision Landing Onto a Heaving Platform: New Method and Experiment". In: *IEEE Access* 8 (2020), pp. 167192–167202. ISSN: 2169-3536. DOI: [10.1109/ACCESS.2020.3022881](https://doi.org/10.1109/ACCESS.2020.3022881). URL: <https://ieeexplore.ieee.org/document/9189835/> (cit. on pp. 21, 26).
- [23] M. Laiacker et al. "Vision aided automatic landing system for fixed wing UAV". In: *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2013-11, pp. 2971–2976. ISBN: 978-1-4673-6358-7. DOI: [10.1109/IROS.2013.6696777](https://doi.org/10.1109/IROS.2013.6696777). URL: <http://ieeexplore.ieee.org/document/6696777/> (cit. on pp. 22, 25).
- [24] A. Gautam, P. B. Sujit, and S. Saripalli. "A survey of autonomous landing techniques for UAVs". In: *2014 International Conference on Unmanned Aircraft Systems, ICUAS 2014 - Conference Proceedings* May (2014), pp. 1210–1218. DOI: [10.1109/ICUAS.2014.6842377](https://doi.org/10.1109/ICUAS.2014.6842377) (cit. on p. 22).
- [25] K. Pluckter and S. Scherer. "Precision UAV Landing in Unstructured Environments". In: *Springer Proceedings in Advanced Robotics* 11 (2020), pp. 177–187. ISSN: 25111264. DOI: [10.1007/978-3-030-33950-0_16](https://doi.org/10.1007/978-3-030-33950-0_16) (cit. on pp. 22, 25).
- [26] J. Janousek and P. Marcon. "Precision landing options in unmanned aerial vehicles". In: *2018 International Interdisciplinary PhD Workshop, IIPhDW 2018* (2018), pp. 58–60. DOI: [10.1109/IIPHDW.2018.8388325](https://doi.org/10.1109/IIPHDW.2018.8388325) (cit. on p. 22).
- [27] K. Ro, K. Raghu, and J. B. Barlow. "Aerodynamic Characteristics of a Free-Wing Tilt-Body Unmanned Aerial Vehicle". In: *Journal of Aircraft* 44.5 (2007-07), pp. 1619–1629. ISSN: 0021-8669. DOI: [10.2514/1.27989](https://doi.org/10.2514/1.27989). URL: <https://doi.org/10.2514/1.27989> (cit. on p. 23).
- [28] P. I. T. M. Das, S. Swami, and J. M. Conrad. "An algorithm for landing a quadrotor unmanned aerial vehicle on an oscillating surface". In: *Conference Proceedings - IEEE SOUTHEASTCON* (2012), pp. 31–34. ISSN: 07347502. DOI: [10.1109/SECon.2012.6196934](https://doi.org/10.1109/SECon.2012.6196934) (cit. on pp. 23, 24).
- [29] A. Rodriguez-Ramos et al. "A Deep Reinforcement Learning Strategy for UAV Autonomous Landing on a Moving Platform". In: *Journal of Intelligent & Robotic Systems* 93.1-2 (2019-02), pp. 351–366. ISSN: 0921-0296. DOI: [10.1007/s10846-018-0891-8](https://doi.org/10.1007/s10846-018-0891-8). URL: <http://link.springer.com/10.1007/s10846-018-0891-8> (cit. on p. 23).

- [30] T. H. Nguyen et al. “Post-Mission Autonomous Return and Precision Landing of UAV”. In: *2018 15th International Conference on Control, Automation, Robotics and Vision, ICARCV 2018* (2018), pp. 1747–1752. DOI: [10.1109/ICARCV.2018.8581117](https://doi.org/10.1109/ICARCV.2018.8581117) (cit. on p. 24).
- [31] M. K. Rydalch. “Precision Maritime Landing of Autonomous Multirotor Aircraft with Real-Time Kinematic GNSS”. PhD thesis. Brigham Young University, 2021. URL: <http://hdl.lib.byu.edu/1877/etd11808> (cit. on pp. 24, 25).
- [32] X. Liu et al. “An onboard vision-based system for autonomous landing of a low-cost quadrotor on a novel landing pad”. In: *Sensors (Switzerland)* 19.21 (2019). ISSN: 14248220. DOI: [10.3390/s19214703](https://doi.org/10.3390/s19214703) (cit. on p. 25).
- [33] S. Lange, N. Sünderhauf, and P. Protzel. “A vision based onboard approach for landing and position control of an autonomous multirotor UAV in GPS-denied environments”. In: *2009 International Conference on Advanced Robotics, ICAR 2009* (2009) (cit. on p. 25).
- [34] N. Koenig and A. Howard. “Design and use paradigms for Gazebo, an open-source multi-robot simulator”. In: *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* 3 (2004), pp. 2149–2154. DOI: [10.1109/iros.2004.1389727](https://doi.org/10.1109/iros.2004.1389727) (cit. on pp. 26, 27, 29, 32).
- [35] M. Alajlan and A. Koubâa. *Writing global path planners plugins in ROS: A tutorial*. Vol. 625. Volume 1. 2016, pp. 73–97. ISBN: 9783319260525. DOI: [10.1007/978-3-319-26054-9_4](https://doi.org/10.1007/978-3-319-26054-9_4) (cit. on pp. 27, 28).
- [36] A. Symington et al. “Simulating quadrotor UAVs in outdoor scenarios”. In: *IEEE International Conference on Intelligent Robots and Systems Iros* (2014), pp. 3382–3388. ISSN: 21530866. DOI: [10.1109/IROS.2014.6943033](https://doi.org/10.1109/IROS.2014.6943033) (cit. on p. 29).
- [37] “Model Based Design Accelerates the Development of Mechanical Locomotive Controls”. In: *SAE 2010 Commercial Vehicle Engineering Congress*. SAE International, 2010. DOI: <https://doi.org/10.4271/2010-01-1999>. URL: <https://doi.org/10.4271/2010-01-1999> (cit. on p. 29).
- [38] F. Furrer et al. “RotorS—A Modular Gazebo MAV Simulator Framework”. In: *Robot Operating System (ROS): The Complete Reference (Volume 1)*. Ed. by A. Koubaa. Cham: Springer International Publishing, 2016, pp. 595–625. ISBN: 978-3-319-26054-9. DOI: [10.1007/978-3-319-26054-9_23](https://doi.org/10.1007/978-3-319-26054-9_23). URL: https://doi.org/10.1007/978-3-319-26054-9_23 (cit. on pp. 30, 31).
- [39] G. Silano. *BebopS GitHub repository*. 2019. URL: <https://github.com/gsilano/BebopS> (cit. on p. 31).
- [40] “Review: Modeling and classical controller of quad-rotor”. In: *arXiv* 5.August (2017), pp. 314–319. ISSN: 2331-8422 (cit. on p. 31).

- [41] B. L. Stevens, F. L. Lewis, and E. N. Johnson. *Aircraft Control and Simulation: Dynamics, Controls Design, and Autonomous Systems*. Wiley, 2015. ISBN: 9781118870976. URL: <https://books.google.pt/books?id=N4abCgAAQBAJ> (cit. on p. 32).
- [42] Parrot Drones SAS. *What is Parrot Sphinx - Parrot Sphinx 2.5.1*. URL: <https://developer.parrot.com/docs/sphinx/> (cit. on pp. 32, 35).
- [43] L. Pitonakova et al. "Feature and performance comparison of the V-REP, Gazebo and ARGoS robot simulators". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 10965 LNAI (2018), pp. 357–368. ISSN: 16113349. DOI: [10.1007/978-3-319-96728-8_30](https://doi.org/10.1007/978-3-319-96728-8_30) (cit. on pp. 33, 34).
- [44] A. Harris and J. M. Conrad. "Survey of popular robotics simulators, frameworks, and toolkits". In: *2011 Proceedings of IEEE Southeastcon*. 2011, pp. 243–249. DOI: [10.1109/SECON.2011.5752942](https://doi.org/10.1109/SECON.2011.5752942) (cit. on p. 33).
- [45] R. Brothers and D. Bevly. "a Comparison of Vehicle Handling Fidelity Between the Gazebo and Anvel Simulators". In: (2019) (cit. on p. 33).
- [46] S. Shah et al. "AirSim: High-Fidelity Visual and Physical Simulation for Autonomous Vehicles". In: *Field and Service Robotics*. Ed. by M. Hutter and R. Siegwart. Cham: Springer International Publishing, 2018, pp. 621–635. ISBN: 978-3-319-67361-5 (cit. on p. 34).
- [47] G. Echeverria et al. "Modular open robots simulation engine: MORSE". In: *Proceedings - IEEE International Conference on Robotics and Automation* (2011), pp. 46–51. ISSN: 10504729. DOI: [10.1109/ICRA.2011.5980252](https://doi.org/10.1109/ICRA.2011.5980252) (cit. on p. 34).
- [48] E. Ebeid et al. "A survey of Open-Source UAV flight controllers and flight simulators". In: *Microprocessors and Microsystems* 61 (2018), pp. 11–20. ISSN: 01419331. DOI: [10.1016/j.micpro.2018.05.002](https://doi.org/10.1016/j.micpro.2018.05.002). URL: <https://doi.org/10.1016/j.micpro.2018.05.002> (cit. on p. 34).
- [49] R. Garcia and L. E. Barnes. "Multi-UAV Simulator Utilizing X-Plane". In: *Journal of Intelligent and Robotic Systems* 57 (2010), pp. 393–406 (cit. on p. 34).
- [50] A. I. Hentati et al. "Simulation Tools, Environments and Frameworks for UAV Systems Performance Analysis". In: *2018 14th International Wireless Communications and Mobile Computing Conference, IWCMC 2018* (2018), pp. 1495–1500. DOI: [10.1109/IWCMC.2018.8450505](https://doi.org/10.1109/IWCMC.2018.8450505) (cit. on p. 34).
- [51] A. R. Perry. "The flightgear flight simulator". In: *Proceedings of the USENIX Annual Technical Conference*. Vol. 686. 2004 (cit. on p. 34).
- [52] Dronecode. *Overview QGroundControl User Guide*. URL: <https://docs.qgroundcontrol.com/master/en/index.html> (visited on 2022-02-11) (cit. on p. 34).
- [53] J. Redmon. *Darknet: Open Source Neural Networks in C*. <http://pjreddie.com/darknet/>. 2013–2016 (cit. on p. 40).

- [54] J. Redmon and A. Farhadi. “YOLOv3: An Incremental Improvement”. In: *arXiv* (2018) (cit. on pp. 40, 56, 75).
- [55] J. Redmon et al. “You Only Look Once: Unified, Real-Time Object Detection”. In: *CoRR* abs/1506.02640 (2015). arXiv: 1506.02640. URL: <http://arxiv.org/abs/1506.02640> (cit. on pp. 40, 75).
- [56] G. Bradski and A. Kaehler. *Learning OpenCV: Computer Vision in C++ with the OpenCV Library*. O’Reilly, 2012. ISBN: 9781449314651. URL: <https://books.google.pt/books?id=uSq2ngECAAJ> (cit. on p. 46).
- [57] S. Garivvani et al. *Coroush/bebop_control: ROS package to control Parrot Bebop 2 drone*. URL: https://github.com/Coroush/bebop_control%20https://www.iaacblog.com/programs/parrot-bebop-controller/ (cit. on p. 49).
- [58] J. Redmon. *Darknet: Convolutional Neural Networks*. <https://github.com/pjreddie/darknet>. 2018–2022 (cit. on p. 53).
- [59] J. Redmon. *Joseph Redmon Website*. URL: <https://pjreddie.com/> (cit. on p. 53).
- [60] Q.-C. Mao et al. “Mini-YOLOv3: Real-Time Object Detector for Embedded Applications”. In: *IEEE Access* 7 (2019), pp. 133529–133538. DOI: 10.1109/ACCESS.2019.2941547 (cit. on p. 53).
- [61] T.-Y. Lin et al. *Feature Pyramid Networks for Object Detection*. 2016. DOI: 10.48550/ARXIV.1612.03144. URL: <https://arxiv.org/abs/1612.03144> (cit. on pp. 54, 55).
- [62] RoboFlow. *Roboflow: Give your software the power to see objects in images and video*. 2020. URL: <https://roboflow.com/> (cit. on pp. 62, 76).
- [63] M. Bjelonic. *Robotic Systems Lab | ETH Zurich*. URL: <https://rsl.ethz.ch/> (cit. on p. 74).
- [64] M. Bjelonic. *YOLO ROS: Real-Time Object Detection for ROS*. https://github.com/leggedrobotics/darknet_ros. 2016–2018 (cit. on p. 74).
- [65] H. Kumar. *Evaluation metrics for object detection and segmentation: mAP*. URL: <https://kharshit.github.io/blog/2019/09/20/evaluation-metrics-for-object-detection-and-segmentation> (cit. on p. 82).
- [66] L. Doitsidis et al. “A framework for fuzzy logic based UAV navigation and control”. In: *IEEE International Conference on Robotics and Automation, 2004. Proceedings. ICRA ’04. 2004*. Vol. 4. 2004, 4041–4046 Vol.4. DOI: 10.1109/ROBOT.2004.1308903 (cit. on p. 97).





NOT A SCHOOL OF

NOT A SCHOOL OF