UNIVERSITAT POLITÈCNICA DE VALÈNCIA

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

School of Industrial Engineering

Developing enhanced Drone Operations: Enabling Gesture Recognition with Integration of a Custom Fleet Management System based on ROOSTER

Master's Thesis

Master's Degree in Industrial Engineering

AUTHOR: Haas, Patrick

Tutor: Blanes Noguera, Juan Francisco

Cotutor: Valera Fernández, Ángel

External cotutor: Dörfler, Florian

ACADEMIC YEAR: 2023/2024

Acknowledgements

First, I want to thank my supervising professor from UPV Prof. Dr. Juan Francisco Blanes Noguera, for having offered me this opportunity to write a master's thesis in his research group. Furthermore, I am extremely grateful for Lauras' great supervision, the many insightful discussions, the critical questions asked, and the relentless support I have experienced during the many hours spent in the laboratory and the fields. Thanks to Lauras' contagious affection for this project, I also discovered new interests, and I was able to broaden my horizons in this vast research topic. Furthermore, I appreciated Pacos' helpful inputs and insights and his valuable share of experience on that topic.

Secondly, I want to thank the whole *ai2* lab team, consisting of Maria, Emima, Pedro, Fran, Jorge, Antoine, and all the other team members from the laboratory across the hallway, not only for the warm welcome but also the excellent integration into the group, the culture and the fun time we spent. It was a pleasure to spend *almuerzo* with you. Moreover, I want to thank Marcos and Javier for their support in conducting real-world flights and for sharing their vast knowledge about drones and their applications in general.

Furthermore, I want to thank my supervising professor from my home university Prof. Dr. Florian Dörfler, for supervising this thesis. Moreover, I want to thank my home university, ETH Zürich, the national agency movetia, and the Heyning-Roelli Stiftung for enabling me to conduct my master's thesis abroad.

Finally, I want to thank all who supported me throughout this master's thesis. Mainly my family, for always believing in me and for the strong support I received even from abroad, and my wonderful girlfriend, Belinda, who thoughtfully supported me during my whole stay in Valencia.

Abstract

Industries are paying more attention to autonomous Unmanned Aircraft System (UAS) operations due to the latest advancements in sensor technology, Artificial Intelligence (AI), and the ease of employing them. UAS can help to reduce costs, enhance efficiency, and improve safety for personnel. The Fostering and Enabling AI, Data and Robotics Technologies for Supporting Human Workers in Harvesting Wild Food (FEROX) project aims to leverage these advantages to improve berry farming by using UAS to locate berries in terrain that is difficult to access and assisting berry pickers working on the ground.

This project focuses on integrating a Holybro X500 V2 Development Kit UAS into the Formiga Fleet Management System (FMS), which uses Robot Optimization, Scheduling, Task Execution and Routing (Ro.O.S.T.E.R.) as its base. The result can be used as a test bed to confirm the compatibility of future improvements for the Formiga FMS with real-world flights. Further, an obstacle avoidance algorithm is implemented and tested, utilizing a forward-facing Light Detection and Ranging (LiDAR) sensor. Finally, a hand gesture detector is deployed to improve the safety of ground personnel, allowing them to stop a bypassing UAS and trigger an emergency call only using hand gestures.

Keywords: UAS, fleet management, hand gesture recognition, Robot Operating System (ROS), Computer Vision (CV), AI

Declaration of Originality

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Patrick Haas,
Zürich, September 17, 2024

Contents

List of Acronyms

AI	Artificial Intelligence.
AMR	Autonomous Mobile Robot.
API	Application Programming Interface.
CNN	Convolutional Neural Network.
CV	Computer Vision.
CVPR	Conference on Computer Vision and Pattern Recognition.
EU	European Union.
FCU	Flight Controller Unit.
FEROX	Fostering and Enabling AI, Data and Robotics Technologies for Supporting Human Workers in Harvesting Wild Food.
FIFO	First In First Out.
FMS	Fleet Management System.
FTG	Follow The Gap.
GCS	Ground Control Station.
GPS	Global Positioning System.
GUI	Graphical User Interface.
HWD	Heavyweight Drone.
IMU	Inertial Measurement Unit.
LiDAR	Light Detection and Ranging.
LWD	Lightweight Drone.
MEx	Mobile Executer.
ML	Machine Learning.

NN	Neural Network.
PID	Porportional - Integral - Derivative.
QGC	QGroundControl.
RKT	Real-Time Kinematic.
Ro.O.S.T.E.R.	Robot Optimization, Scheduling, Task Execution and Routing.
ROS	Robot Operating System.
RRT	Rapidly exploring Random Tree.
SITL	Software In The Loop.
SSD	Single-Shot Detector.
UAS	Unmanned Aircraft System.
UAV	Unmanned Aircraft Vehicle.
VFH	Vector Field Histogram.
VPU	Video Processing Unit.
Wi-Fi	Wireless Fidelity.

Part I

Report

Chapter 1

Introduction

Nowadays, UAS, commonly known as drones, are increasingly gaining more attention for their ability to operate without an onboard human pilot. This raises the need to steer them remotely by an operator normally stationed on the ground or by an onboard computer. These UAS are highly configurable such that they can meet many needs of sophisticated systems, thus making them versatile in their use case.

With the help of advanced sensors, sophisticated algorithms, and AI, autonomous UAS operations can perform complicated missions autonomously without human intervention. Being able to conduct autonomous missions even in harsh environments thanks to being equipped with advanced navigation systems such as Global Positioning System (GPS) antennas, LiDAR sensors, and CV ready cameras enable UAS to operate in diverse and hazardous conditions with real-time data processing.

UAS are used in various sectors, spreading from agriculture and logistics to infrastructure and inspections and emergency calls, helping to cut down costs, improve efficiency, and enhance safety (Feraru et al., 2020; Merz et al., 2022). Over the recent years, more industries have recognized the potential benefits of integrating UAS in their operations, which, in return, will further stimulate innovation (Bhat et al., 2024; Perez-Segui et al., 2023).

1.1 The FEROX Project

The FEROX project aims to leverage the power of AI, data, and robotics to significantly facilitate wild berry harvesting in Nordic countries. Wild berries and mushrooms are considered national treasures that grow naturally in the forests and require no additional cultivating resources. According to Hamunen et al. (2019), less than 10% of the annual wild berry crop is harvested due to the challenging manual labor involved and the poor working conditions of pickers. The short harvest season results in much of the work being done by foreign laborers with limited knowledge of the local language, culture, and terrain.

The FEROX project aims to improve the working conditions of wild berry pickers by using drones equipped with advanced sensors to acquire data to build 3D models of the forests. The project seeks to estimate berries' locations, amounts, and types accurately. The collected data will be used to build AI models to help workers locate the berries and optimize their procedures. Additionally, FEROX will provide wild berry pickers with navigation and localization services and physical support to improve their working conditions.

The FEROX project is expected to contribute significantly to the overall safety of the workers by automatically monitoring the berry pickers and providing assistance when needed. As a result, the amount of harvested berries in the area is expected to increase, encouraging local people and hikers to engage in harvesting during the summer months. These results will open new business opportunities for European companies to adopt solutions developed for industrialized cultivation and support global sustainable berry harvesting. As a proof of concept, the outcomes of the FEROX project will be tested in the forests.

The FEROX project is funded by the Horizon Europe fund of the European Union (EU) and started on 1st of September in 2022 and has a fixed duration of three years.

1.2 Motivation

Especially in recent years, numerous industries have shown interest in drone technology, leading to unprecedented opportunities for innovation and efficiency improvements (Emimi et al., 2023). This thesis seeks to improve drone operations not only by integrating a real-world drone into an existing FMS based on Ro.O.S.T.E.R. but also by leveraging advanced Machine Learning (ML) and CV algorithms for hand gesture controlling the drone to enhance the safety of berry pickers. These enhancements are expected to significantly improve user interaction and operational efficiency for various applications.

1.3 Objectives

The thesis' goals are presented in the following enumeration.

1. Integrate the open-source Holybro X500 V2 development kit into the existing Formiga FMS.
2. Enhance drone operations by leveraging an onboard camera for hand gesture recognition.
3. Study the feasibility of running CV and ML algorithms on board of the UAS and quantify its reliability.
4. Incorporate a basic obstacle avoidance algorithm to ensure a safe, autonomous drone operation.

1.3.1 Methodology

In order to fulfill the thesis' goals, custom software has been developed for integrating an open-source Holybro X500 V2 development kit drone equipped with a Pixhawk 6C serving as Flight Controller Unit (FCU) into an already implemented FMS based on Ro.O.S.T.E.R.. With this integration step, this platform can serve as a proof of concept to showcase further capabilities that could be realized. Additionally, the platform can be used as a test bed to validate its functionality if further improvements in hardware or software have been made. Moreover, implementing hand gesture recognition extends the functionality of autonomous UAS operations. This enhancement will broaden operational capabilities and enhance interaction between end users and the system.

Chapter 2

Background

This chapter conveys the concepts compiled and used in this thesis. First, an outline about UAS in general is presented in section 2.1. In order to enable autonomous drone missions, a capable flight controller is employed, as explained in section 2.2. Furthermore, some light is shed on the Ro.O.S.T.E.R. FMS in section 2.3. Afterward, the employed ML pipeline for detecting hand gestures is addressed in section 2.4. Last but not least, some theory about obstacle avoidance strategies is presented in section 2.5.

2.1 Unmanned Aircraft System

An UAS encompasses the entire system of unmanned aerial operations, including the following three core components, which namely are a Unmanned Aircraft Vehicle (UAV), a Ground Control Station (GCS), and a data link.

Being the airborne component of the UAS, UAVs come in different sizes and shapes. Their size can differ from small, hand-held devices, which can safely be used indoors, up to large, sophisticated aircraft. It is important to note that the term UAV only encompasses the bare frame without additional sensors or controllers.

For safe operations of a UAS, a GCS plays a crucial role in a UAS. It is responsible for many vital services needed to operate a UAS, such as mission planning, real-time flight control, telemetry monitoring, and maintaining communication with the UAV. In the frame of this thesis, QGroundControl (QGC) has been used, which is an open-source GCS software that is widely used in industry and research due to its compatibility with various UAS and its intuitive user interface (Dardoize et al., 2019; Ramirez-Atencia & Camacho, 2018). QGC employs the MAVLink protocol, a lightweight and efficient communication standard to exchange commands and telemetry data with UAS.

Last but not least, telemetry is essential to every UAS because it provides a real-time data link between the UAV and the GCS. It sends and receives vital information to the GCS such as UAV's state. Thanks to this continuous data link, operators can monitor the UAV's performance, make

informed decisions in order to ensure safe and efficient operation, and, in a worst-case scenario, operators can intervene to restore safety.

In the future, UAS will proceed to possess a critical position in society. Future advancements may be improved payload capacities for conveyance administration, expanded independence, and other progressions. UAS are expected to be more profoundly integrated into everyday operations as legislation alters to address safety concerns. This will revolutionize business processes and probably will greatly impact human lives.

2.2 Flight Controller

A flight controller is one of the foremost vital components of a UAS. Ordinarily, it comprises either a microcontroller or microprocessing unit, together with different sensors such as an Inertial Measurement Unit (IMU), indicators, magnetometers, or numerous more. Its primary responsibility is stabilizing the airplane in real time and guaranteeing exact flight maneuvers by specifically connecting with sensors and actuators.

The flight controller receives its input from a pilot through a remote control transmitter or an independent control framework, such as a mission planner or other autonomous navigation software. It processes these inputs to stabilize the UAS, control its orientation (yaw, roll, and pitch), maintain elevation, and oversee flight modes.

On the other hand, a FCU houses a flight controller and handles broader flight control tasks such as mission planning, navigation, and communication. A FCU refers to a module within the overall flight control system, enclosing hardware and software components, whereas a flight controller typically is a specific hardware or software unit.

2.2.1 PX4 Autopilot Firmware

Originating in 2009, PX4 autopilot is a powerful open-source software running on the NuttX RTOS and was originally designed for affordable autonomous drones by members of the *Computer Vision and Geometry Lab* at ETH Zürich. The project provides flexible tools for drone developers to share technologies and create tailored solutions for drone applications.

Some of the main characteristics offered by PX4 are:

- Support for many vehicle frames/types, including multicopters, fixed-wing aircraft, etc.
- Support for three main flight modes: fully manual, partially assisted, and fully autonomous.
- Vehicle stabilization
- Waypoint navigation
- Integration of position, speed, altitude, and rotation sensors.
- Automatic triggering of cameras and external actuators such as, for example, a gripper mechanism.
- Robust and deep integration with companion computers and robotics APIs such as ROS2 and MAVSDK.

PX4 is a core part of a broader drone platform that includes QGC as GCS, Pixhawk hardware acting as FCU, and MAVSDK for integration with companion computers using the potential of cameras and other hardware using the MAVLink protocol. MAVROS is part of the MAVSDK Application Programming Interface (API) and ensures seamless integration of the MAVLink protocol into the ROS environment. For further explanations about ROS, please refer to section 4.2. PX4 is hosted by Dronecode, a non-profit Linux Foundation.

PX4 consists of two main layers: the *flight stack* and the *middleware*. The *flight stack* consists of an estimator and a controller system encompassing a collection of guidance, navigation, and control algorithms for autonomous drones. On the other hand, the *middleware* consists primarily of device drivers for embedded sensors, communication with the external world such as companion computers, GCS, etc., and the uORB publish-subscribe message bus. Moreover, it also contains a simulation layer that allows PX4 flight code to run on a desktop operating system in a Software In The Loop (SITL) setup.

Employed Controller

For stabilizing a multicopter, the type of UAV used in this thesis and within the FEROX project, the PX4 flight stack implements a standard cascaded control architecture. Cascade control is a hierarchical control strategy in which multiple control loops are arranged in series, where the first controller adjusts the set point of the second controller (Idres et al., 2017). According to Tan et al. (2005), this control strategy is especially beneficial for controlling multicopters since it divides the control problem into different layers or loops, each responsible for a specific aspect of the system's dynamics, thus making it robust.

The overall control architecture can be partitioned into two fundamental parts as shown in Figure 2.1. The primary part on the left deals with position and velocity control, which are dealt with in an Inertial Frame. An Inertial Frame is a stationary or uniformly moving frame and is, hence, not experiencing any acceleration. The second part on the right in Figure 2.1 houses the innermost control loops, which handle the angular control and angular rate control, referenced in a Body Frame.

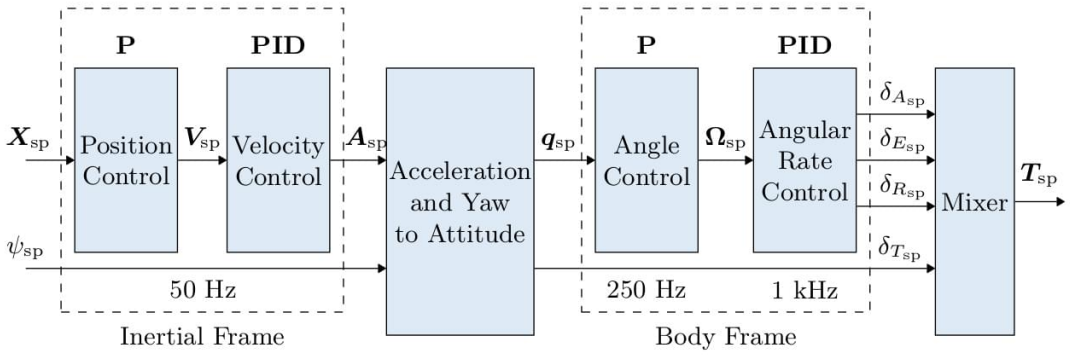

Figure 2.1: Overview of the general control structure (PX4-Development-Team, 2024). The left side shows the position and velocity controllers, whereas the right side indicates the angular and angular rate control.

The above-mentioned controllers are introduced in the following parts:

- Position and Velocity Control:
  The employed position controller is a simple proportional controller that takes as input X_{sp}, corresponding to the position set point, and calculates the error signal by comparing it against the current position X. This error gets multiplied by a proportional gain P. To obtain the first part of the velocity set point v_{sp}, the multiplied error signal is truncated to the minimum and maximum velocities v_{min} and v_{max} which the user can define. This architecture can be observed in the Figure 2.2 below, where the left part corresponds to the position controller.

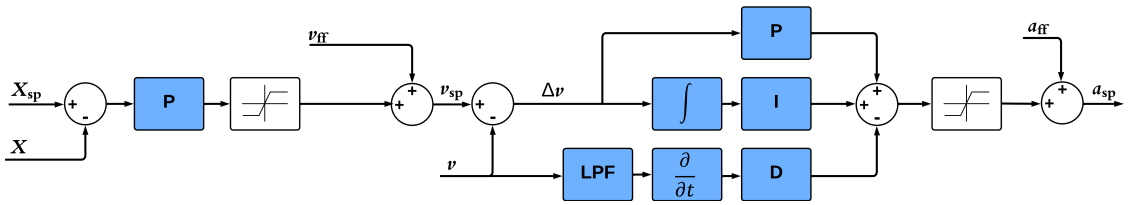

Figure 2.2: Position and velocity control architectures combined with a feed-forward mechanism (PX4-Development-Team, 2024).

The control architecture on the right side of Figure 2.2 then corresponds to the velocity controller where the first part of the beforehand calculated velocity set point adds up with a feedforward velocity v_{ff}. This feedforward mechanism translates into a velocity offset for the velocity set point v_{sp}, significantly improving positional tracking performance due to the lack of perfect positional set point tracking. After obtaining the velocity set point v_{sp}, the velocity error signal Δv is calculated by comparison with the velocity measurement v. This error signal is then, on the one hand, multiplied with the proportional part P and, on the other side, integrated and multiplied by the respective integral part I of the PID controller.

Furthermore, the integral authority is featured with an anti-reset windup mechanism, ensuring that the integral component does not dominate control when a constant error is faced. In addition, the derivative part of the PID controller is applied to the low pass filtered velocity measurement and not as usual to the deviation between the measured velocity v and the set point velocity v_{sp}. This measure helps mitigate the so-called derivative kicks, which reveal high-frequency components when transformed into the spectral domain and whose effects can be reduced by applying a low-pass filter. The resulting weighted sum is subsequently constrained to either the minimum allowable acceleration a_{min} or the maximum allowable acceleration a_{max}.

The desired acceleration set point a_{sp} is ultimately determined by adding an acceleration feedforward value a_{ff}, following the same logic as explained for the velocity set point.

- Attitude Control:
  A common way to describe the attitude of a UAS is via Euler angles as they are intuitive to understand (Alaimo et al., 2013). This approach causes a major problem, commonly referred to as "gimbal lock" or singularity. Therefore, quaternions were employed to describe the 3D attitude.

The implemented attitude controller needs as inputs a set point quaternion q_{sp} and the measured attitude encoded as a quaternion q. The set point quaternion is obtained through a mapping of the yaw set point ψ_{sp} and the acceleration set point a_{sp}, represented by the block in the middle in Figure 2.1. A more detailed explanation will not be presented in this thesis. The overall implementation for this attitude controller originates from a paper published by Brescianini et al. (2013). Although this controller includes only a proportional component, it remains highly complex due to the operations and handling of quaternions. However, tuning this controller is fairly simple since only one appropriate value for P has to be found. Finally, the resulting angular rate set point Ω_{sp} gets saturated. This explained controller architecture is depicted in Figure 2.3 below.

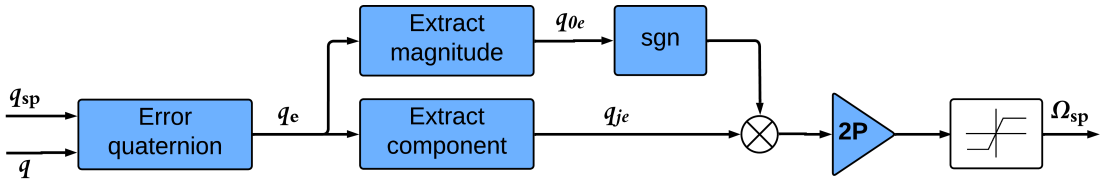

Figure 2.3: Control architecture of the attitude controller (PX4-Development-Team, 2024).

- Angular Rate Control:
  The inner loop controller of this cascaded control architecture handles the angular rate in order to obtain the actuator inputs δ_{sp} which correspond to throttle command and angle command in the physical system. For this purpose, it expects an angular rate set point Ω_{sp} and the corresponding measurement of the angular rate Ω. Given the very similar structure of this controller compared to the previously presented velocity controller above (see Figure 2.2), a detailed explanation is deemed unnecessary.

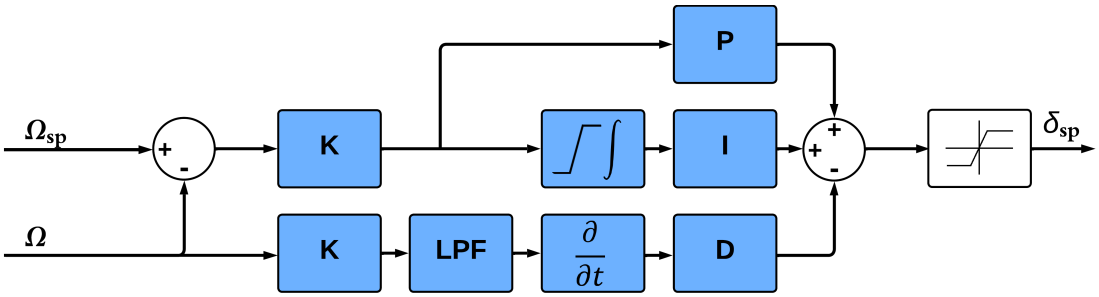

Figure 2.4: Control architecture of the Angular Rate controller (PX4-Development-Team, 2024).

2.3 Fleet Management System

A Fleet Management System within the context of robotics is a software platform that oversees and facilitates the operations of a fleet of Autonomous Mobile Robot (AMR)s working together, guaranteeing optimal task allocation and empowering navigation and communication capabilities among the AMRs. This software is vital since it makes these AMR operate seamlessly, avoids clashes, maintains optimal performance, and executes their tasks effectively ("Guide to Mobile Robot Fleet Management", n.d.). The following sections explain the core functionalities of a FMS according to Saghaei (2016b).

To coordinate tasks along multiple AMRs, the FMS delegates tasks based on the accessibility, area, and current workload of an AMR. Therefore, the work is spread evenly, reducing downtime and ensuring that important tasks are done first.

Likewise, the FMS needs to always keep track of all AMRs. Therefore, it constantly checks their current status, location, and how efficient they are. This gives the operator a full picture of what is going on. Getting information in real-time helps to detect and fix problems quickly.

In this context, FMSs use advanced algorithms to make AMRs' routes more efficient. They consult previously obtained maps and the AMRs' current locations to find the best routes that save time and energy. These systems can also change routes on the fly to avoid traffic and keep the route smooth.

Detailed performance analysis allows for an improvement in performance over time. FMSs collect data on how long tasks take to complete, how far AMR move, how much battery is used, and when an AMR needs maintenance. This information helps find patterns and opportunities for improvement, making it easier to make smart decisions.

2.3.1 Robot Optimization, Scheduling, Task Execution and Routing

Ro.O.S.T.E.R. was created as part of the research conducted on the *Collaborating and coupled AGV swarms with extended environment recognition* project funded by EIT Manufacturing at the *Center of Design for Advanced Manufacturing* lab of TU Delft in the Netherlands.

Ro.O.S.T.E.R., in detail, is an open-source project based on ROS for developing a heterogeneous FMS solution with task allocation, scheduling, and autonomous navigation capabilities for so-called Mobile Executer (MEx). In this context, MEx are the AMRs which make up the fleet for performing tasks. The FMS consists of the four core components Front End Graphical User Interface (GUI), Job Manager, MEx Sentinel, and Marker Array. In the following sections, each of the components will be discussed.

Front End GUI

The front end mainly serves as a point of interaction between the operator (User) and the actual FMS software. Here, an operator can get an overview of the current fleet status. Furthermore, orders can be defined, altered, or deleted. If the operator decides so, they can place orders, which then get assigned to an available MEx and turn into jobs. Once a MEx starts working on a job, the operator can overview the status of ongoing jobs and cancel them if necessary.

MEx Sentinel

The MEx Sentinel module runs in the background and keeps track of the MEx's ID, its status, and the assigned job ID. Thus, this module is crucial for providing information that helps coordinate jobs in the MEx fleet. Furthermore, it provides callable services to retrieve a list of all registered MEx, to retrieve and change the status of a MEx, and to assign or unassign a job to a MEx.

Job Manager

The Job Manager is responsible for appending orders received through the FMS Front End to an order list. This order list is regularly checked, and orders are translated into jobs and tasks. Those jobs are prioritized in the *Pending Job* list. The Job Manager then checks regularly the *Pending Job* list and picks the first pending job with the highest priority. If pending jobs have the same priority, the next pending job that is assigned is determined by the First In First Out (FIFO) method.

Once a job is active, they regularly update the status in MEx Sentinel. A job is counted as completed once its status is either successful, canceled, or aborted. If a job is completed, it is removed from the active jobs list, and the MEx employed with the job is freed, and its status is set to STANDBY in the MEx Sentinel.

Marker Array

The Marker Array module visualizes the fleet's order locations in Rviz, a 3D visualization tool used to inspect the robot's perception of its environment, whether simulated or real, within the ROS framework.

2.4 Gesture Recognition

Gesture recognition is a popular research topic in the field of CV and ML since it combines the cutting-edge technologies of both branches of research. To achieve reliable communication between the UAS and humans, even without having a direct physical communication link such as a remote control or similar, gesture recognition appears to be a suitable solution since it only requires a direct line of sight to a human. Especially in the context of the FEROX project where UAS are equipped with a downwards-facing camera to detect berries, this setup comes in handy for simultaneously detecting hand gestures of humans underneath the UAS while it is performing its task.

2.4.1 Introduction to Convolutional Neural Networks for Image Classification

Nowadays, many ML pipelines are based on Convolutional Neural Network (CNN)s, a special type of deep learning model designed for analyzing and processing different types of data, including images. CNNs are particularly strong in recognizing image patterns and features.

CNNs take an input image and assign importance to various aspects or objects in the images, represented by the learnable weights and biases. For this purpose, CNNs use a technique called

convolution, a linear operation typically used in signal processing, which differentiates them from normal feed-forward Neural Network (NN).

An RGB image is a 3D matrix of pixel values. Figure 2.5 below shows a pixel value representation of an RGB image with an exemplary width and height of only four pixels. The three matrices represent the three color channels: red, green, and blue. CNNs then apply kernels, which are small rectangular filters, to the input image to detect features like edges or shapes. Those filters slide over the width and height of the image and compute dot products between the filter and the input to obtain a so-called activation map. This overall operation is called a kernel convolution. Formally spoken, the activation map $\mathcal{G}$ at a given row m and column n is given by the convolution of f, which is the input image, and h, which represents the kernel applied, as defined in Equation 2.1. j and k are indices for the row and column index of the kernel h.

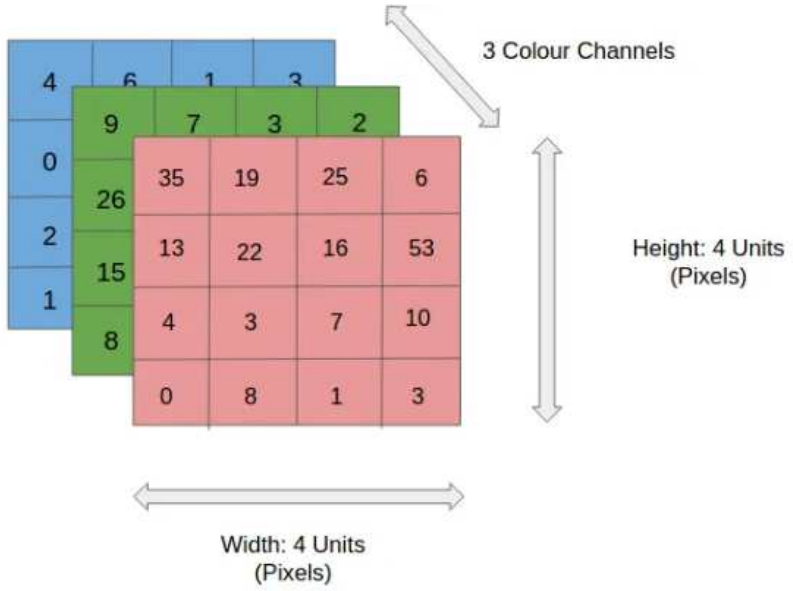

Figure 2.5: Pixel value representation of an RGB image (Saha, 2018).

$$\mathcal{G}[m,n] = (f * h)[m,n] = \sum_j \sum_k h[j,k] f[m-j, n-k] \tag{2.1}$$

Figure 2.6 shows a kernel convolution's first and second steps, which are repeated for each color channel matrix. Applying the convolution to the input image makes the convolved feature size smaller than the input image, reducing the computational resources needed to process that image.

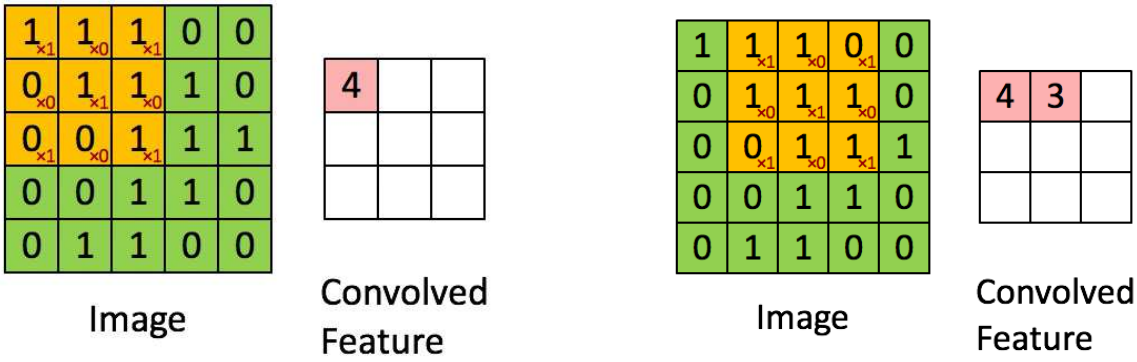

Figure 2.6: The left figure shows the first step of the convolution operation where the black numbers indicate the pixel values and the red numbers indicate the kernel values. The orange area indicates the area of the image to which the kernel is currently applied during that step. The right figure shows the next step of the kernel convolution (Saha, 2018).

Similar to the previously mentioned kernel convolution, a pooling layer is applied, which reduces the size of the convolved activation map. It drastically reduces the computational power needed to process the data and extracts dominant features. There are two main pooling methods: max and average pooling. As their names indicate, they either retrieve the maximum value of the area on which the pooling is applied or the average value. Figure 2.7 shows the results by applying a maximum or average pooling to the activation map on the left.

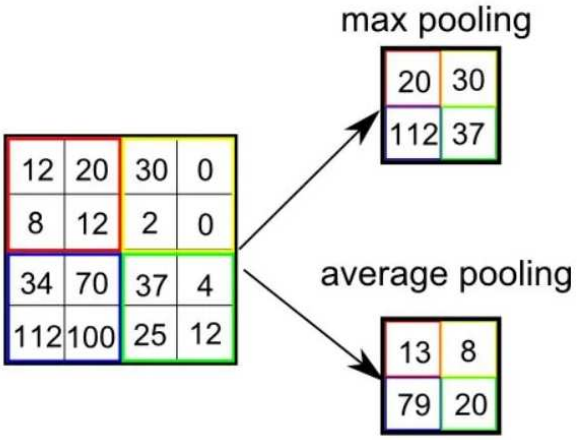

Figure 2.7: Example of a maximum or average pooling (Saha, 2018).

After applying the kernel convolution and the pooling, the model successfully understands the image's features. In the next step, these features are classified. A fully connected layer is used to learn a non-linear classification function of the high-level features, requiring that the pooled data be flattened into a vector form as indicated in Figure 2.8. The x_1 to x_d correspond to the input, which is the flattened result matrix after pooling. Since the NN is fully connected, every node from the previous layer is connected to every node of the next layer. Every connection has a weight w. The shown NN has two hidden layers $v^{(1)}$ and $v^{(2)}$. To obtain the prediction y for a given input x_1 to x_d, for every node in every layer, its value is calculated dependent on the

node values of the previous layer and the weights that connect the previous layer nodes with the current node.

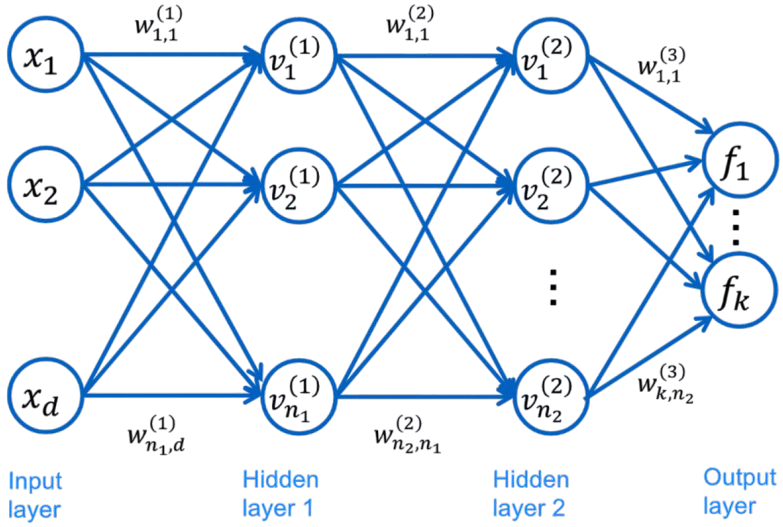

Figure 2.8: Example architecture of a fully connected NN. indicate the inputs, whereas f_1 to f_k are the resulting classifications. The depth is one bigger than the amount of hidden layers. Here, the depth is three since there are two hidden layers $v^{(1)}$ and $v^{(2)}$.

For a node $v_j^{(l)}$ the following is calculated as indicated in Equation 2.2

$$v_j^{(l)} = \varphi(z_j^{(l)}) \tag{2.2}$$

, where $z_j^{(l)}$ is obtained as a sum of the previous node values and the connecting edge weights as in Equation 2.3

$$z_j^{(l)} = \sum_{i=1}^{n_{l-1}} w_{j,i}^{(l)} v_i^{(l-1)} \tag{2.3}$$

, where j is the index of the node at level l, n_l is the amount of nodes at l and $w_{j,i}$ is the weight assigned to the connection from node v_i^{l-1} to node v_j^l. Furthermore, $\varphi(\cdot)$ indicates a non-linear activation function. Popular choices here are either the sigmoid function as in Equation 2.4

$$\varphi(z) = \frac{1}{1 + \exp(-z)} \tag{2.4}$$

or the ReLU function as in Equation 2.5

$$\varphi(z) = \max(z, 0) \tag{2.5}$$

By calculating $v_j^{(l)}$ for every layer, one can, in the end, obtain the probabilities for the classes f_1 to f_k. Those are calculated by plugging in the obtained values into Equation 2.6

$$f_j = \sum_{i=1}^{n_{L-1}} w_{j,i}^{(L)} v_i^{(L-1)} \tag{2.6}$$

, where f_j are the probabilities for classifying an object to be in class j, and L is the depth of the NN.

During the training phase, the NN predicts a class j for every input. This classification is then compared against the true label of that object. A loss function measures how well the NN predicts the right label and thus must be minimized. Therefore, for every prediction, a loss is calculated, which is used for backpropagation to learn new weights and biases to minimize a loss function $\ell(\cdot)$. Backpropagation solves the following problem depicted in Equation 2.7.

$$w^* = \arg\min_w \sum_{i=1}^{n_{L-1}} \ell \left(f_i; \sum_{j=1}^{m} w_j z_j^{(i)} \right) \tag{2.7}$$

w^* contains the new weights for all edges, minimizing the cost function $\ell(\cdot)$, f_i are the predicted labels as explained in Equation 2.6, w_j denote the old weights and $z_j^{(i)}$ indicates the calculated nodal value as in Equation 2.3. The updated weights are then used to classify the next input, hopefully improving the accuracy of the predictions made by the CNN.

Furthermore, various methods exist for backpropagation on the loss derived from the loss function. For this reason, a detailed explanation of the loss function and backpropagation is omitted here.

2.4.2 Introduction to Single-Shot Detector Models for Object Detection

Single-Shot Detector (SSD) models, in contrast to the beforehand discussed CNNs (see subsection 2.4.1), predict the object position and size in an image. To rapidly detect the position and size of an object, SSD models leverage two components: a backbone and an SSD head. The backbone usually is a pre-trained image classification network such as a CNN for feature extraction. Normally, only the fully connected NN from the CNN is removed since this part is responsible for classification. Thus, a feature representation of the image is left, which can be used by the SSD head.

In the first step, a grid is laid over the input image. Figure 2.9 shows a 4x4 example grid layover. Now, the SSD model predicts a class of an object for every cell in that grid. Classifying an object means that an object is present in that cell; hence, it will be considered for object localization later. There might be a case where multiple objects of different shapes are detected in one cell. Anchor boxes are used to overcome this challenge.

SSD models use a so-called matching phase while training to match the appropriate anchor box with every object's ground truth bounding box. The anchor box with the highest degree of overlap with the ground truth bounding box predicts the object's class and location. However, sometimes, a pre-defined anchor box does not fit the object. Hence, an aspect ratio and a zoom level are needed.

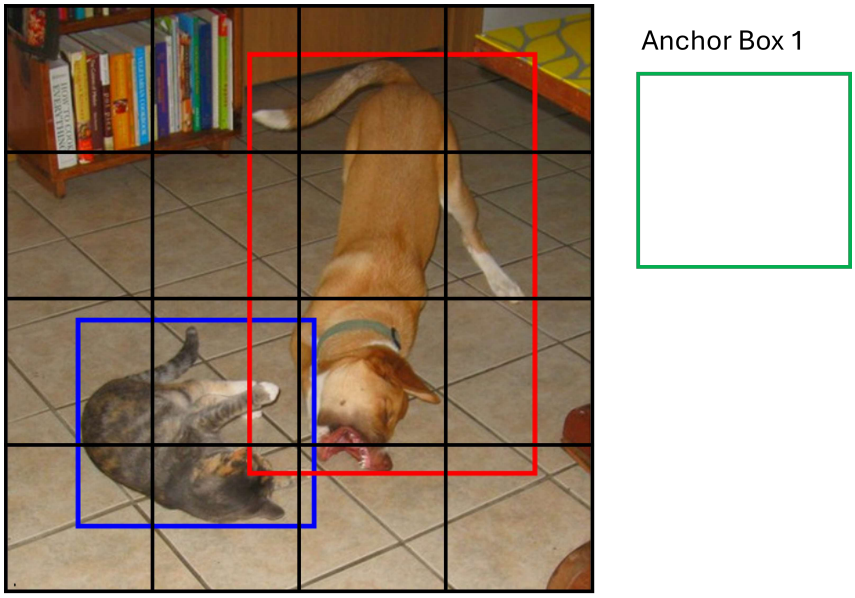

Figure 2.9: Example image with a 4x4 grid layover. The blue and red boxes indicate the ground truth boxes whereas the green box indicates a default anchor box (Liu et al., 2016).

For a default anchor box, its current center (c_x^d, c_y^d), width w^d, and height h^d are needed to obtain the ground truth adjustments needed to be performed on the default anchor box as presented in Equation 2.8 below

$$t_x = \frac{c_x - c_x^d}{w^d} \quad t_y = \frac{c_y - c_y^d}{h^d} \quad t_w = \log\left(\frac{w}{w^d}\right) \quad t_h = \log\left(\frac{h}{h^d}\right) \tag{2.8}$$

, where c_x and c_y denote the ground truth center, w the ground truth width an h the ground truth height for that anchor box. t_x, t_y, t_w and t_h correspond to the ground truth adjustments needed to the default anchor box centered at (c_x^d, c_y^d). To learn the correct adjustments needed for the anchor boxes, training is performed with a smooth L1 loss function to obtain the current performance of the SSD model on predicting the adjustments for each anchor box and then backpropagating the loss to improve future predictions. The loss function is defined in Equation 2.9

$$L_{\text{loc}} = \sum_{m \in N} \sum_{i \in \{x,y,w,h\}} \text{smooth}_{L_1}\left(\hat{t}_i^m - t_i^m\right) \tag{2.9}$$

, where $\hat{t}_x$, $\hat{t}_y$, $\hat{t}_w$ and $\hat{t}_h$ indicate the adjustments predicted by the SSD model on the x- and y-coordinates of the center, the width and height of the anchor box. N indicates the number of default boxes available to the SSD model. The smooth L1 function is defined in Equation 2.10

$$\text{smooth}_{L_1}(x) = \begin{cases} 0.5x^2 & \text{if } |x| < 1 \\ |x| - 0.5 & \text{otherwise} \end{cases} \tag{2.10}$$

Hence, the SSD does not directly learn the correct center coordinates, width, and height for each anchor box but rather learns to make good predictions about adjustments of default anchor boxes that need to be applied to overlap with the ground truth box as much as possible.

Later, when the SSD model has been trained, the predicted adjustments $\hat{t}_x$, $\hat{t}_y$, $\hat{t}_w$ and $\hat{t}_h$ to each default anchor box are used to obtain the adjusted anchor box. The predictions are used in Equation 2.11

$$\begin{aligned} c_x &= \hat{t}_x \cdot w^d + c_x^d \\ c_y &= \hat{t}_y \cdot h^d + c_y^d \\ w &= \exp\left(\hat{t}_w\right) \cdot w^d \\ h &= \exp\left(\hat{t}_h\right) \cdot h^d \end{aligned} \tag{2.11}$$

, where the new center of the altered anchor box (c_x, c_y) and its width w and height h is obtained.

Normally, for classification of the detected object, SSD are trained with a softmax loss. This loss measures how well the model classifies the presence of a given object in a given default anchor box. It is defined as

$$L_{\text{conf}} = -\sum_{i \in \text{positive}} \log\left(\hat{p}_i^c\right) - \sum_{i \in \text{negative}} \log\left(\hat{p}_i^0\right) \tag{2.12}$$

, where $\hat{p}_i^c$ indicates the predicted probability that the i-th default anchor box contains an object of class c. On the other hand, $\hat{p}_i^0$ is the predicted probability of the i-th default anchor box containing no object. As a reminder, the predicted probabilities $\hat{p}_i^c$ and $\hat{p}_i^0$ are derived from the softmax output of the classifier network. The softmax function outputs a probability distribution over all possible classes (including the background class). So for a given default anchor box i, the softmax output is calculated through equation 2.13

$$\hat{p}_i^k = \frac{\exp(z_i^k)}{\sum_j \exp(z_i^j)} \tag{2.13}$$

, where z_i^k is the raw score predicted by the model for a class k for the i-th default anchor box and $\hat{p}_i^k$ is the probability that the i-th default anchor box contains an object of class k.

2.4.3 Hand Gesture Recognition

For the scope of this thesis, the focus lies on employing a hand gesture recognition algorithm that runs onboard the UAS. Hand gesture recognition appeared to be a more suitable solution than body gestures for the FEROX project since the UAS are going to fly at low altitudes above ground, and making hand gestures involves less effort than body pose gestures, especially in case of emergency when an injured person cannot properly move anymore.

To make the hand gesture recognition less dependent on the Wireless Fidelity (Wi-Fi) network and reduce reaction latency, the employed algorithm should run onboard the UAS. This poses major challenges to the mission computer since the computational resources are limited. Furthermore, the mission computer would be responsible for demanding the tasks performed by the UAS and running CV algorithms for detecting hand gestures. Hence, hand gesture detection should run apart on a device other than the mission computer.

The Luxonis DepthAI OAK-D S2 camera shows impressive capabilities in running complex CV algorithms or even NNs onboard. For this purpose, geaxgx (2023) has published a GitHub repository that leverages a sophisticated ML and CV pipeline that can be run directly on the hardware of the Luxonis camera. This codebase marked the foundation for the hand gesture software stack, which has been adapted to serve its purpose in this thesis. The beforehand mentioned ML / CV pipeline will be discussed in the following subsection 2.4.4.

2.4.4 On-Device, Real-Time Hand Tracking with Google MediaPipe

In 2019, Google introduced a new approach to hand perception through MediaPipe, an open-source, cross-platform framework designed for building pipelines to process perceptual data, including video and audio. Previewed at Conference on Computer Vision and Pattern Recognition (CVPR) 2019, this method offers high-fidelity hand and finger tracking by leveraging ML to extract 21 3D key points of a hand from a single frame. Unlike state-of-the-art techniques that depend on powerful desktop environments, this approach achieves real-time performance on mobile phones and can be scaled to detect multiple hands.

The developed ML pipeline for hand tracking and gesture recognition leverages several models working together, which will be explained in further detail in the following subsections.

However, the employed ML pipeline, which is running on the Luxonis OAK-D S2 camera, has a prior so-called Body Pre Focusing estimator, which is needed to overcome the challenge that Google's MediaPipe framework reliably detects hand gestures only up to two meters away from the camera. Nevertheless, the implementation should also work for a distance greater than two meters. Thus, a Body Pre Focusing estimator is employed. Figure 2.10 illustrates the different steps exemplary on an image. The first image on the right depicts the detection of the body pose (in blue) used for Body Pre Focusing. The green square in the image in the middle shows the result of the Body Pre Focusing process, which is fed into the MediaPipe hand gesture detection pipeline. The last image on the right displays a smaller, yellow square containing the hand that the hand gesture detection is processing.

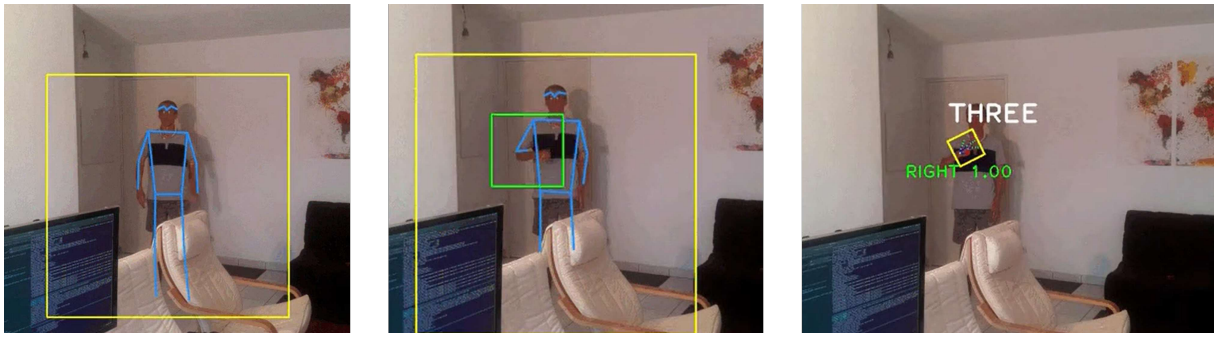

Figure 2.10: Exemplary visualization of the ML pipeline applied on a video feed (geaxgx, 2023).

Body Pre Focusing

The palm detector introduced in the Google MediaPipe framework was trained to detect hand gestures less than two meters away from the camera. To overcome this challenge, an additional body pose estimator can help focus on the image's zone that probably contains a hand. Hence, instead of the whole image, only a cropped image of the zone around the hand is fed to the palm detector.

To achieve this, a body pose estimator is used to provide information about the wrist key points, which can then extract the region of interest that contains a hand. One widely used natural body pose estimator is the MoveNet Single pose proposed by Goyal et al. (2023). This body pose estimator can detect and track human body poses from images. The model is built upon TensorFlow, a widely used tool for developing and training ML models.

The model takes images or frames as input and predicts the coordinates of the body key points as depicted in Figure 2.11. At the end, MoveNet predicts the wrist key points, which are then used as the center of the zones where hands are expected. Hence, the Palm detector only has to check for palms in the region around the wrist key point, which significantly facilitates its job.

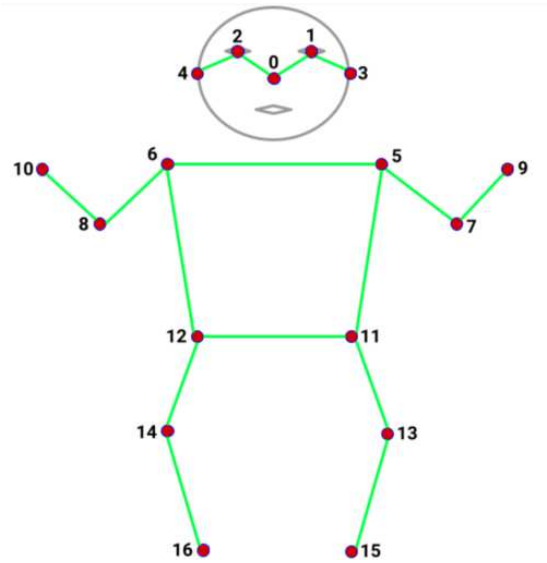

Figure 2.11: Extracted body key points by the TensorFlow MoveNet model (TensorFlow, 2022). The key points of interest are numbers 9 and 10.

Palm Detector

To initially find the location of the hand in a frame, an SSD model, proposed by Liu et al. (2016), called BlazePalm, has been employed. Any other SSD relies on a set of default boxes, also called anchor boxes of different aspect ratios and scales, placed uniformly across the image. BlazePalm, however, abuses the fact that hand palms can be modeled as square boxes, which helps ignore other aspect ratios and, therefore, further reduces the number of anchors, making detection faster.

Since the employed SSD for detecting palms is facing class imbalance because there are many more background (negative) examples compared to palm (positive) examples, a Focal Loss was

used to train the classification capabilities of the SSD (Lin et al., 2017). The Focal Loss addresses the class imbalance issue by down-weighting the loss assigned to the well-classified examples, thus helping the model focus more on the harder, misclassified examples. The Focal Loss is defined as in equation Equation 2.14

$$FL\left(p_t\right) = -\alpha_t \left(1 - p_t\right)^{\gamma} \log\left(p_t\right) \tag{2.14}$$

, where p_t is the model's estimated probability of having detected a palm, α is a weighting factor for the palm class to balance the importance of positive/negative examples, and γ is the focusing parameter which adjusts the rate at which easy examples are down-weighted.

In the context of object detection with a SSD, the Focal Loss can be applied to the confidence loss part as in equation 2.15, which defines the locational error

$$L_{\text{loc}} = \frac{1}{N} \sum_{i} FL\left(p_t^i\right) \tag{2.15}$$

, where N is the number of positive matches and $FL\left(p_t^i\right)$ is the focal loss of the i-th predicition. With this knowledge, the adjusted total loss can be obtained as presented in Equation 2.16

$$L = \frac{1}{N} \left(L_{\text{loc}} + \alpha \sum_{i} FL\left(p_t^i\right) \right) \tag{2.16}$$

, where α is a balancing weight and L_{loc} is the in Equation 2.15 presented loss over the predicted adjustments.

Hand Landmark Detector

After applying the palm detector to the whole image, a direct regression model is used to predict the coordinates of the 21 3D hand-knuckle coordinates inside the detected hand regions (Zhang et al., 2020). The regression loss L_{landmark} used to train the direct regression model is defined in Equation 2.17

$$L_{\text{landmark}} = \frac{1}{21} \sum_{j=1}^{21} \left\| \overrightarrow{x_j} - \overrightarrow{x_{\text{gt}}^j} \right\|_2^2 \tag{2.17}$$

where $\overrightarrow{x_j}$ are the predicted 3D coordinates of the j-th hand landmark and $\overrightarrow{x_{\text{gt}}^j}$ are the corresponding ground truth 3D coordinates of the j-th hand landmark. $\|\cdot\|_2^2$ denotes the squared Euclidean distance (L2 norm).

Gesture Recognition

For recognizing gestures, a simple algorithm, which is applied on top of the predicted hand skeleton, accumulates the angles of joints. The angle of joints is calculated via Equation 2.18

$$\cos(\theta) = \frac{(\overrightarrow{p_j} - \overrightarrow{p_i}) \cdot (\overrightarrow{p_j} - \overrightarrow{p_k})}{\|\overrightarrow{p_j} - \overrightarrow{p_i}\| \|\overrightarrow{p_j} - \overrightarrow{p_k}\|} \tag{2.18}$$

, where the landmarks $\overrightarrow{p_i}$, $\overrightarrow{p_j}$, and $\overrightarrow{p_k}$ are the finger's base, middle, and tip. With the help of this calculation, a state can be derived for each finger, such as active or inactive, in case of stretching the finger. This set of finger states is then mapped to pre-defined gestures. This straightforward solution has proven effective and robust with reasonable quality (Zhang et al., 2020).

2.5 Obstacle avoidance

Collision avoidance is an important requirement for autonomous flights since no human interference is desired during the flight. Especially in the context of the FEROX project, the UAS must show capabilities to detect obstacles and successfully avoid them since the UAS has to fly in forests close to the ground to detect berries or personnel. In general, obstacle avoidance can be divided into local/reactive or global obstacle avoidance.

Local obstacle avoidance algorithms do not depend on building a map of the environment and do not save obstacles' positions. The appropriate reaction is derived online from the current sensor data. One of the main advantages of employing a local algorithm is that they come at a low computational cost, which is often a prerequisite for onboard computing when utilizing weight-critical robots. However, they will not find the optimal path to the goal since they cannot compute the whole path to the goal.

Contrarily, global approaches have access to a precomputed map of the environment to encounter the optimal path by leveraging path finding algorithms such as Dijkstra's or A* algorithm. Regardless, they are computationally more demanding, such that global algorithms are often outsourced to a more powerful ground control computer, relying on a working network connection and adding an extra layer of delay to the process of obtaining an optimal path to the goal.

For the sake of this thesis, a reactive approach has been chosen due to the tight constraints on weight, power consumption, and available onboard computing power.

2.5.1 Basic local Obstacle Avoidance

For this thesis, a simple range finder such as a short-range LiDAR sensor was used to measure the distance to an object right in front of the UAS. A LiDAR sensor was employed due to its lightweight, low cost, and ease of integration into the already existing UAS stack. However, a major issue when using those range finders is that they are limited in their sensing range and can only sense the distance to an object just straight in front of the sensor. Thus, no point cloud or array of range points was available to the obstacle avoidance algorithm.

A straightforward idea is to sense obstacles right in front of the UAS during missions. When a certain threshold is violated, evasive maneuvers are performed. This basic idea was already employed by Bouabdallah et al. (2007) when dealing with a heavily constrained flight stack.

They tested different shapes of trajectories when evading the obstacle. In contrast to this thesis, four range finder sensors, each mounted on each side of the UAS, were used, which enabled them to gather information about when the UAS has passed the obstacle. Due to this difference, the employed obstacle avoidance algorithm in this thesis (see section 4.5) is a modified version of the second option that was presented in the previously mentioned paper of Bouabdallah et al. (2007).

If an obstacle has been detected, the UAS turns first by -90° to determine if it is safe to sidestep to the corresponding side of the UAS. If no obstacle is detected for the next two meters, the UAS will move two meters forward in this direction and get the original goal as a waypoint for its mission. However, if the UAS detects an obstacle within two meters after turning -90°, it will turn an additional 180° to check if it is safe to evade the obstacle on the left side of the original orientation of the UAS. There, the procedure is the same as explained after having turned -90° in the first place. However, if there is also an obstacle on the left side of the UAS it will not find a path to overcome it.

Chapter 3

Related Work

In section 3.1, some of the most widely known applications of FMSs and its related technologies in industry will be covered. Furthermore, state-of-the-art gesture recognition algorithms are presented in section 3.2, which are more capable of running on low-power devices. Last but not least, relevant research done in obstacle avoidance in the area of UAS is introduced.

3.1 Fleet Management System

Recent research has been focusing on deploying customized FMSs in production systems. Especially in manufacturing plants producing low-weight products face significant logistical issues due to the high numbers of products that need to be transported within the facility. Cordova and Olivares (2016) proposed a fleet management model for UAS that optimizes delivery and pickup operations. Furthermore, Cordova and Olivares (2016) developed a tool to determine the optimal number of UAS needed for various logistic operations, taking into account factors like product demand, drone specifications, and the plant layout.

Rinaldi and Primatesta (2024) explored the usage of UAS in urban environments for package delivery, traffic monitoring, and emergency management. Their paper sheds some light on the task allocation system for a heterogenous fleet of UAS with a strong focus on optimizing energy efficiency and task completion times. Furthermore, a market-based allocation algorithm in order to balance the UAS utilization dynamically is introduced, which is particularly beneficial for delivery and recharging tasks. Rinaldi and Primatesta, 2024 claim that their implementation efficiently allocates heterogeneous tasks with time deadline constraints to a heterogenous fleet, showcasing good robustness with respect to lossy communication scenarios.

Saghaei (2016a) focuses on a novel FMS especially suited for monitoring using automatic vehicle locators and a web-based software for tracking and reporting. Furthermore, they leveraged the usage of the cellular network for communication purposes. Their proposed FMS is characterized by high positional accuracy, its user-friendly approach. and the efficient energy usage.

Saghaei (2016a) installed their proposed sensors on trucks and conducted real-world experiments, claiming a significant improvement in monitoring capabilities.

3.2 Gesture Control

The task of gesture control has seen significant advancements in both academic research and industry applications over the past few years, leading to new applications in the area of UAS. Those applications, however, greatly vary in their implementation and complexity. Thus, first, some light is shed on body pose gesture control in subsection 3.2.1, and more specifically, advancements in hand gesture control are presented in subsection 3.2.2.

3.2.1 Body Pose Gesture Control

Yam-Viramontes and Mercado-Ravell (2020) present a gesture-controlled UAS. However, the employed autonomous platform is not powerful enough to run a pose estimation model. Thus, the video stream of the embedded camera was processed on a ground station that runs OpenPose and a pose classifier. After classification, the ground station returns the orders to the UAS. The pose classification system is efficient as it is not a trainable model and only uses four features drawn from the key points, such as the angles between the torso and both arms and the distances between the hands and the torso. Orders are then deducted based on those values using a threshold look-up table. The system's scalability is limited, while a new class can be defined without a dataset. Furthermore, a not-discussed delay is introduced by outsourcing the pose estimation and classification onto a ground station computer.

Contrary to the beforehand presented project, Findelair (2021) developed a Python package named Pose-Classification-Kit for facilitating dataset creation, model evaluation, and the deployment of a processing pipeline for a human body pose classification onto a UAS. The project emphasizes the intermediate representation of the human body poses as key points to ensure efficient and accurate gesture recognition, even with partial inputs. Overall, the project aimed to identify the most effective approach for robust performance by comparing different classification models. The ultimate objective was optimizing and deploying this system on a UASs' companion computer, providing a proof of concept for an embedded gesture control interface.

3.2.2 Hand Gesture Control

In the domain of hand gesture control, Kiselov (2021) employed Google's MediaPipe for hand gesture recognition to control a small form factor consumer UAS equipped with a high-resolution action camera. By utilizing MediaPipe Hands, the project mainly aimed to create a more user-friendly control system. The team used a Ryze Tello quadcopter for the proof of concept, utilizing its open Python SDK for development. However, it has been reported that even though the used UAS supports deploying a custom-developed code base, running the hand recognition model on board was impossible. Thus, the video stream was redirected to a ground station computer, which ran the classification model. The system's effectiveness has been quantified through the precision of gesture recognition. They claim to achieve over 97% accuracy across most classes, proving that their system can reliably control a drone (Kiselov, 2021).

3.3 Obstacle avoidance algorithms

Obstacle avoidance algorithms have received much attention over the past few years, especially with the emerging availability of UAS for mainstream consumers. The foundations for obstacle avoidance research started in the 90s when the first algorithms were designed for robots moving in a 2D environment. Nevertheless, with the evolution of robots, adopted algorithms were developed to consider 3D environments.

3.3.1 Basic Obstacle Avoidance

One of the most basic reactive obstacle avoidance algorithms is to employ a range-finding sensor that measures the distance to the object right in front of the robot. Bouabdallah et al. (2007) used a heavily constrained UAS with four ultrasound sensors for measuring the distance to objects at every side of the UAS. They defined zones around the UAS, which act as fallback options to evade the encountered obstacle. When measuring a distance to the closest object, it got classified, and reactive measures were taken by checking which fallback option was available. Due to onboard sensor limitations, they could not implement their obstacle avoidance controller in such a way that it could be validated at cruise speed but rather when hovering.

3.3.2 Follow The Gap

Another purely reactive approach commonly known in autonomous racing is Follow The Gap (FTG). It uses a LiDAR, which provides 2D point cloud data of distances to objects in its field of view (Sezer & Gokasan, 2012). With this information, a suitable gap can be identified to successfully evade obstacles in the field of view of the LiDAR. One major advantage of this strategy is the simple implementation. Furthermore, it is relatively computationally inexpensive, making it a suitable choice for onboard applications. FTG was originally employed and tested on cars (Sezer & Gokasan, 2012). However, the recipe could also be commissioned for obstacle avoidance using UAS. One of the main differences is that a UAS generally has three degrees of freedom for its movements, whereas cars or MEx are constrained to two degrees of freedom in their movement. Indeed, Simpson and Sabo (2016) proposed a similar algorithm to FTG which, instead of employing a LiDAR sensor, is using an optical flow sensor for detecting gaps in a similar way as in FTG. Extending the FTG algorithm to be able to use the third dimension of movements might be a difficult task. Thus, one loses a degree of freedom by employing the initial algorithm.

3.3.3 Vector Field Histogram Methods

Similar to the beforehand discussed Basic Obstacle Avoidance and FTG algorithms (see subsection 3.3.1 and subsection 3.3.2), Vector Field Histogram (VFH) belongs to the reactive obstacle avoidance algorithms. Initially, its foundations have been laid in the 90s. The original VFH algorithm has been extended for the emerging possibilities offered by robots by employing constraints to find the adequate gap (Borenstein & Koren, 1991; Ulrich & Borenstein, 1998).

The underlying algorithm leverages information gained through a sonar sensor to generate a 2D occupancy grid of the environment, which is, in a further step, then mapped to a 1D polar histogram (Borenstein & Koren, 1991). This histogram is used to extract the available free direc-

tions. The enhanced VFH+ algorithm further considers constraints on the robot's maximum turn radius and safety margins (Ulrich & Borenstein, 1998). However, since UAS have one additional degree of freedom, this poses major challenges, which researchers tried to tackle by mapping the original 3D map to a 2D map or creating a 2D map for every height level. However, each technique showed drastic drawbacks in oversimplifying the environment's huge computational overhead (Vanneste et al., 2014). Thus, Vanneste et al. (2014) proposed a capable algorithm for overcoming the posed challenges, which then have been implemented and tested on a real-world UAS using a single forward-faced depth camera (Baumann et al., 2018).

Chapter 4

Implementation

This chapter explains the steps taken towards the implementation for integrating the Holybro X500 V2 UAS into the Formiga FMS. For this purpose, ROS action servers have been developed as depicted in section 4.3. Furthermore, the detailed implementation of the employed Hand Gesture Detector will be explained in section 4.4. Lastly, the approach towards an obstacle avoidance implementation using a range finder LiDAR is explained in section 4.5.

4.1 Development and Simulation Setup

A development and testing framework has been established to ensure a seamless integration of the Holybro X500 V2 drone into the Formiga FMS. For this purpose, new pieces of software have been planned, developed, and undergone rigorous testing in the simulation environment. This process ensures that premature errors can be resolved quickly and efficiently without risking crashes with real drones, which can be costly. The developed software for integration uses the PX4 firmware (see subsection 2.2.1) and the corresponding MAVROS interface. This code base has been selected due to its open-source nature, extensive expandability, and broad compatibility with various hardware platforms.

In order to use the provided functionalities of the PX4 firmware, MAVROS has been employed, which is a ROS package offering an interface to communicate in ROS with the PX4 firmware via the MAVLink protocol. Furthermore, an open-source simulation environment named Gazebo has been used for safely testing the developed software on a simulated UAS.

Development and simulation process has been carried out on a Framework Laptop 13 with an Intel® Core™ i7-1360P and 32 GB of RAM running a Linux Ubuntu 20.04 LTS as its operating system. The main constraint for using the older Ubuntu LTS version was the usage of ROS noetic, which depends on Ubuntu 20.04 LTS. This overall setup ensured that software could be developed efficiently and tested safely before carrying out tests on the real-world drone platform.

4.2 ROS

Writing software for robots can be challenging, especially with robotics's growing scale and scope. On top of this, the sheer size of the required code for integrating a large variety of hardware can be daunting. Despite its misleading name, ROS is not an operating system in the traditional sense of process management and scheduling sitting directly on the hardware level. It rather provides a structured communications layer on top of the host operating system, which normally is a Linux Ubuntu distribution of a heterogeneous compute cluster. Those communication layers then greatly facilitate data transport from different software and hardware by abstraction, thus ensuring reliable communication that stays in sync.

In ROS, there exist the following core concepts, which will be explained in the following subsections: nodes, topics, services, and actions.

A ROS node represents a computational unit that can perform specific tasks such as sensing and processing data or controlling actuators by running a sophisticated control algorithm. Nodes enable robotics software developers to structure their code base in a modular and distributed manner. To further ease the implementation, every node in ROS can be implemented in a different supported programming language such as C++, Python, and others, thus providing the freedom to choose the most suitable programming language for the to-be implemented task. Since ROS nodes may depend on data provided by other nodes, efficient communication to share data is needed. ROS topics provide this functionality by implementing a publish/subscribe paradigm, which allows nodes to exchange messages asynchronously. Therefore, nodes can subscribe to topics to receive messages or publish topics to share messages. For this, topics use predefined message types to structure data that needs to be transmitted. A wide range of standard message types can be used to construct custom messages to meet specific needs. Last but not least, ROS topics enable asynchronous communication between nodes so they do not need to wait for each other to send or receive messages, reducing dependencies between nodes and enhancing the system's responsiveness.

As mentioned, ROS topics are a simple way to share information. Some tasks need more precise communication, like setting goals, giving feedback, and reporting results. ROS actions are created to perform tasks that may take a while to finish. To implement this, a ROS action updates how the task is going and what the result is when it is finished. The ROS action has four main parts: goal, feedback, result, and cancellation. In a typical process, a client sends a request to the action server. The server then works on the request and keeps the client updated on the progress of the action. While doing the task, the action server node will also allow you to cancel or replace a task that is currently happening. After finishing the task, the action server returns the result to the client.

Unlike ROS topics and actions, which work asynchronously, ROS services directly communicate where a request is made and a response is given right after. So, the service handles requests from other nodes waiting for a reply. This example is like using a function in regular programming. In this case, the code calling the function waits for it to give back an answer before moving on with the rest of the process. This model is good for tasks that need to be done before moving to the next step, like checking the status of a component or getting a setting. Like ROS topics,

ROS services have set types, including a request and a response. Some regular service types are already pre-defined or can be customized to the needs of the use case.

4.3 Formiga Action Servers

The Formiga FMS is based on Ro.O.S.T.E.R., which has been discussed in detail in subsection 2.3.1. Formiga is a further developed FMS especially specialized for drone operations capable of coordinating a fleet of UAS at the same time. It was developed by Ingeniarus Lda., which also maintains the software. Outstanding is the efficient management of Lightweight Drone (LWD) and Heavyweight Drone (HWD), ensuring that predefined tasks such as item pickup, delivery, and emergency response are handled seamlessly by the correct type of drone. Furthermore, Formiga stands out for integrating with ROS.

The key part of integrating the Formiga FMS using a real UAS is done by implementing ROS action servers, which translate the action goal given by the FMS to commands which the UAS needs to perform to reach that goal successfully. Furthermore, the Formiga action server is running a ROS node to translate the by the UAS firmware obtained state information and republishes that topic for making the state of the UAS accessible for the Formiga FMS monitor panel.

4.3.1 TakeOff and LandIn

The TakeOff and, respectively, LandIn ROS action servers simply implement the functionality to take off or land the UAS. For this purpose, the action severs publish feedback on the current local height of the UAS to publish the progress of the action efficiently. Those actions can also be canceled, resulting in the UAS hovering at the altitude when it received the canceling command.

4.3.2 GoTo

The GoTo action guides the UAS to a given target GPS coordinate *goal*. While flying to the target coordinate, the camera remains active to detect hand gestures from personnel wanting to interact with the UAS. Thanks to this, personnel can stop the UAS by showing a STOP hand gesture and can even call the emergency service by performing the international SOS hand gesture. Showing an OK hand gesture releases the UAS to follow again its GoTo action. Further explanations on hand gesture detection can be found in section 4.4.

The GoTo action implementation only needs the name of the drone *droneName*, and the target point specified in local Cartesian coordinates with respect to a Real-Time Kinematic (RKT) base station. This approach is used since the UAS must be capable of operating in remote areas under the tree canopies, where the GPS signals may be weak or unreliable. First, the ROS action sever subscribes to all the ROS topics that provide important information, such as, for example, the GPS position of the UAS X_{GPS} and many more. After that, the goal given in local Cartesian coordinates needs to be converted to GPS coordinates since the MAVROS interface is expecting GPS coordinates for its waypoint missions. Next, the target point will then be uploaded as a waypoint mission onto the FCU using the MAVROS interface. While executing the waypoint mission, the ROS action server constantly checks for a STOP hand gesture in case personnel on the ground want to interact with the UAS. If a STOP hand gesture was detected, the UAS stops

and holds its current position. It will hover at its place until the person makes an SOS hand gesture, which then could launch an emergency call providing the current position of the UAS to facilitate the rescue mission or until an OK hand gesture is recognized, which then resumes the waypoint mission. Next, it is then checked whether there is an obstacle in front of the UAS. If so, the waypoint mission gets paused, and the obstacle avoidance algorithm will be executed (see section 4.5). Ultimately, the ROS action server publishes feedback on the remaining distance to the goal in meters.

Algorithm 1: GoTo Action Algorithm

Data: *droneName*, *goal*

Result: UAS *droneName* flies to *goal*

```
x_UAS ← subscribe to all relevant topics in ROS;
goal_GPS ← convert goal from local Cartesian coordinates to GPS
current waypoint mission ← goal_GPS;
while ROS node is running do
    if STOP hand gesture detected then
        current waypoint mission ← pause mission and hover at the current location;
        while not OK hand gesture detected do
            if SOS hand gesture detected then
                call emergency service ← current GPS coordinates of the UAS;
            end
            current waypoint mission ← continue before paused mission;
        end
    end
    if Check if an obstacle is present then
        current waypoint mission ← pause mission and hover at the current location;
        current waypoint mission ← run obstacle avoidance algorithm (see algorithm 6);
        current waypoint mission ← continue before paused mission;
    end
    ROS action server feedback ← distance left to the goal in meters;
end
```

4.3.3 Explore

The explore action is more sophisticated since it involves exploring a region defined by its corner points. In this pre-defined area, a photo must be taken every six meters. Thus, a complete flight route must be pre-computed and connects every point that needs to be reached. Figure 4.1 below shows a schematic representation of the explore action.

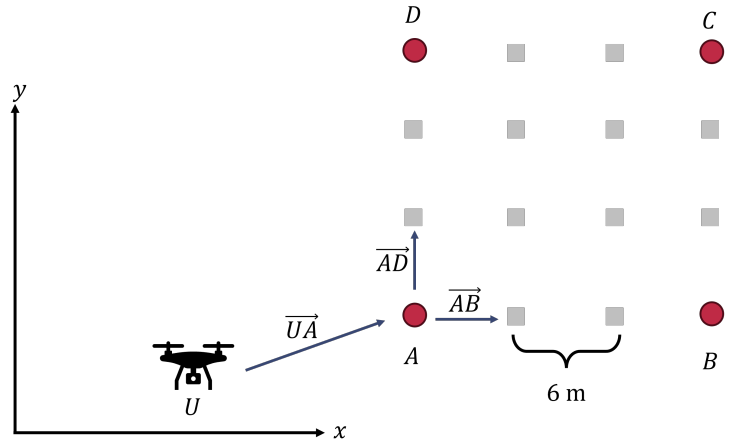

Figure 4.1: Schematic representation of the explore action. U denotes the current GPS position of the UAS whereas A, B, C, D are the GPS coordinates of the corner points of the to be explored region. In addition to the corner points, the UAS should also take an image at the grey points placed every six meters.

In addition, while exploring, the camera of the UAS is constantly ready to detect hand gestures of personnel operating on the ground. Thanks to this, personnel can stop the UAS by showing a STOP hand gesture and can even call the emergency service by performing the international SOS hand gesture. Showing an OK hand gesture releases the UAS to follow again its explore action. Further explanations on the hand gesture detection can be found in section 4.4.

Correctly planned flight routes are essential, as suboptimal routes drastically reduce the UAS's range. Hence, the route should follow a zick-zack pattern, which includes all the to-be-visited points. It has been assumed that when starting the explore action, the UAS should always start at one of the four corner points A, B, C, D. From there, a direction can be chosen, which can either be going first in the direction of corner B, which would correspond to the direction $\overrightarrow{AB}$ or to the other corner D with the direction $\overrightarrow{AD}$. The resulting flight routes are shown in Figure 4.2 below.

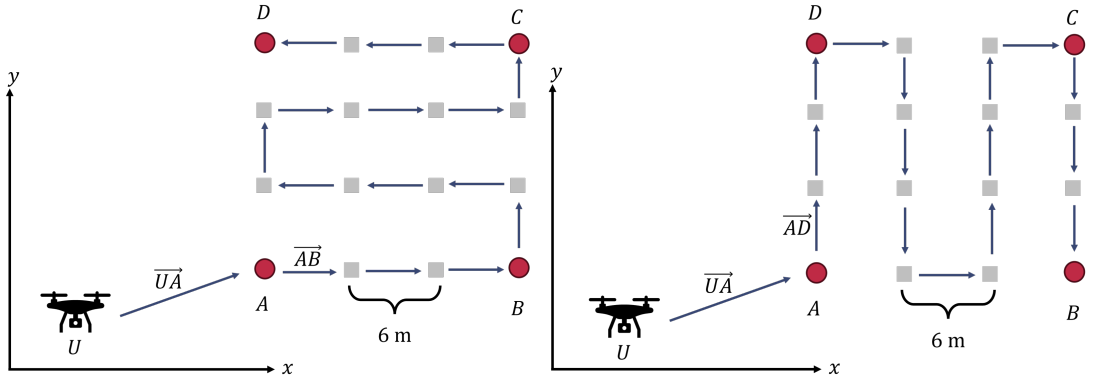

Figure 4.2: Schematic representations of the possible flight route patterns. U denotes the current GPS position of the UAS whereas A, B, C, D are the GPS coordinates of the corner points of the to be explored region. In addition to the corner points, the UAS should also take an image at the grey points placed every six meters. The left figure shows the flight route when $\overrightarrow{AB}$ is chosen as the best direction, whereas the right figure shows the flight route when $\overrightarrow{AD}$ was chosen as starting direction.

This choice of starting directions can minimize vibrations on the UAS and avoid abrupt changes in the direction in which the UAS is flying. Hence, choosing the starting direction vector of the explore action is desirable to align as much as possible with the direction that the UAS is heading $\overrightarrow{UV}$.

A practical measure of how much a starting direction $\overrightarrow{AB}$ or $\overrightarrow{AD}$ coincides with the direction $\overrightarrow{UA}$ is the dot product of the vectors. Hence, to find the optimal starting direction, one seeks the maximum of the two dot products of the two possible starting directions $\overrightarrow{AB}$ or $\overrightarrow{AD}$ with the current heading direction of the UAS $\overrightarrow{UV}$ as stated below:

$$\max\left(\left\langle \overrightarrow{UA}, \overrightarrow{AB} \right\rangle, \left\langle \overrightarrow{UA}, \overrightarrow{AD} \right\rangle\right) \tag{4.1}$$

Knowing this, the algorithm 2 that generates the mesh grid that connects all the to-be-visited points can be introduced. As input, the current position of the UAS X_{GPS} and the distance between the to-be-visited points *stepSize*, which is by default set to 6 meters, is needed. The algorithm then returns a list *path*, which contains all the points in the order in which they should be visited.

First, a basic check needs to be performed whether four points are given and if they form a rectangle. If so, the points will be sorted in counter-clockwise order such that the first corner point is the closest to the UAS's current position. After that, the starting direction of the path is determined according to the sense of optimality discussed in Equation 4.1. To cover the case where the mesh grid might be rotated, and its sides are not parallelly aligned to the coordinate axis, a rotation angle is calculated to simply rotate all the corner points to form a rectangle with sides parallel to the coordinate axis. The actual width and height of the rectangle are determined to compute the number of points necessary to visit on each side such that an image at every six meters can be taken. Then, x and y coordinates of evenly spaced points are generated according to the number of points required per axis. By putting all the combinations of x and y coordinates together, the final *path* is generated. For generating the path when $\overrightarrow{AD}$ is the starting direction, the *traverseClockwiese* flag is set to `True`, which leads to the loop, where the y coordinates are looped through.

Algorithm 2: Generate Mesh Grid Algorithm

Data: X_{GPS}, $[A, B, C, D]$, $stepSize$
Result: List $path$

if List $cornerPoints$ forms not a rectangle **then**
  raise Exception
end
$[A, B, C, D] \leftarrow$ reorder $[A, B, C, D]$ to counter-clockwise and to first point is the closest to X_{GPS};
List $cornerPoints \leftarrow$ rotate $cornerPoints$ by $\arctan\left(\frac{B.\text{y}-A.\text{y}}{B.\text{x}-A.\text{x}}\right)$;
$width \leftarrow$ get geographic distance between A and B;
$height \leftarrow$ get geographic distance between A and D;
$numPointsX \leftarrow \lceil \frac{width}{stepSize} \rceil$;
$numPointsY \leftarrow \lceil \frac{height}{stepSize} \rceil$;
$pointsX \leftarrow$ evenly spaced points between $\min([A, B, C, D].\text{x})$ and $\max([A, B, C, D].\text{x})$ with $numPointsX$ points;
$pointsY \leftarrow$ evenly spaced points between $\min([A, B, C, D].\text{y})$ and $\max([A, B, C, D].\text{y})$ with $numPointsX$ points;
List $path \leftarrow$ empty List;
$path \leftarrow$ put together $pointsX$ and $pointsY$;
return $path$

The overall explore action implementation depends on the beforehand discussed mesh grid generation algorithm (see algorithm 2). As inputs, only the name of the drone $droneName$ and the corner points $[A, B, C, D]$ in local Cartesian coordinates of the to-be-explored region need to be provided. Hence, the local Cartesian coordinates must first be converted into GPS coordinates for the same reason as explained in subsection 4.3.2. Then, the ROS action sever subscribes to all the ROS topics that provide important information, such as, for example, the GPS position of the UAS X_{GPS} and many more. After that, the flight path will be generated using the mesh grid generation algorithm as discussed in algorithm 2. This flight path will then be uploaded as a waypoint mission onto the FCU using the MAVROS interface. While executing the waypoint mission, the ROS action server constantly checks for a STOP hand gesture in case personnel on the ground want to interact with the UAS. If a STOP hand gesture was detected, the UAS stops and holds its current position. It will hover at its place until the person makes an SOS hand gesture, which then could launch an emergency call providing the current position of the UAS to facilitate the rescue mission or until an OK hand gesture is recognized, which then resumes the waypoint mission. Next, it is then checked whether there is an obstacle in front of the UAS. If so, the waypoint mission gets paused, and the obstacle avoidance algorithm will be executed (see section 4.5). After that, if a waypoint of the waypoint mission is reached, a photo will be taken. In this regard, the current GPS position of the UAS X_{GPS} is obtained and then written into the metadata of the photo that has been taken together with the drone name as the author of the photo and the current local time. In the end, the ROS action server publishes feedback on the current progress of the action measured in the covered area of the defined region in %.

Algorithm 3: Explore Action Algorithm

Data: $droneName$, $[A, B, C, D]$
Result: UAS $droneName$ explores region

$x_{\text{UAS}} \leftarrow$ subscribe to all relevant topics in ROS;
$[A, B, C, D]_{\text{GPS}} \leftarrow$ convert $[A, B, C, D]$ from local Cartesian coordinates to GPS
$path \leftarrow$ generate mesh grid with given corners $[A, B, C, D]_{\text{GPS}}$ (see algorithm 2);
current waypoint mission $\leftarrow$ $path$;
while ROS node is running **do**
  if STOP hand gesture detected **then**
    current waypoint mission $\leftarrow$ pause mission and hover at the current location;
    while `not` OK hand gesture detected **do**
      if SOS hand gesture detected **then**
        call emergency service $\leftarrow$ current GPS coordinates of the UAS;
      end
      current waypoint mission $\leftarrow$ continue before paused mission;
    end
  end
  if Check if an obstacle is present **then**
    current waypoint mission $\leftarrow$ pause mission and hover at the current location;
    current waypoint mission $\leftarrow$ run obstacle avoidance algorithm (see algorithm 6);
    current waypoint mission $\leftarrow$ continue before paused mission;
  end
  if a waypoint is reached **then**
    take a photo at the current location $\leftarrow X_{\text{GPS}}, droneName$ and current time;
  end
  ROS action server feedback $\leftarrow$ covered area of the region in %;
end

4.4 Hand Gesture Recognition

The main challenge in implementing the hand gesture detector is to run the entire detection pipeline onboard of the Luxonis OAK-D S2 camera and integrate it into the ROS environment. In this way, the valuable computing resources of the NVIDIA® Jetson Nano™ can be saved for other tasks that might be required in the future. The Luxonis OAK-D S2 stands out for its 3D vision capabilities, achieved by using multiple cameras that work together to sense depth and estimate the distance from an object. The camera is powered by an Intel® Movidius™ Myriad™ X Video Processing Unit (VPU) which is a processor designed for handling complex CV tasks and AI workloads efficiently. Despite its powerful capabilities, it is designed to be energy-efficient, making it suitable for portable and embedded applications.

To ensure seamless integration of hand gesture recognition based on Google's MediaPipe framework on the Luxonis OAK-D S2 camera with ROS, the existing code base of Geaxgx's GitHub repository has been extended since the provided code base fully implements the CV pipeline (geaxgx, 2023). Nevertheless, the data provided by the camera is not directly available in ROS.

To overcome this issue, an additional piece of software has been implemented acting as a bridge that transforms data received from the camera via the XLink interface into actual ROS messages, which are then published into the appropriate ROS topics, which is discussed in subsection 4.4.1. Additionally, to enhance the robustness of specific hand gesture detection, a buffering and threshold-based approach is used to accurately detect specific hand gestures, including SOS, STOP, and OK hand gestures, see subsection 4.4.2. This strategy ensures robust recognition by handling potential noise and reducing false positives.

4.4.1 XLink - ROS Bridge

The implementation of CV and ML pipelines on the Luxonis OAK-D S2 is organized into nodes, creating a modular and flexible architecture. This node-based structure is a key feature of the DepthAI framework. For our purpose, the pipeline starts with nodes running TensorFlow MoveNet and a SSD for initial wrist key point extraction and palm detection as previously discussed in subsubsection 2.4.4 and subsubsection 2.4.4. The resulting data is then fed into the MediaPipe framework node responsible for the actual hand gesture recognition. Finally, the processed data is sent to an XLinkOut node, which maintains a communication channel with the host device using the XLink protocol via a USB connection. On the side of the host device, the DepthAI Python API is used to establish the connection to the camera and for data handling. Last but not least, a custom ROS node converts the data and further processes it for further usage in the ROS environment. An additional Flask web application has been developed to display the camera video stream in a web browser, enabling further inspection when connected to the same Wi-Fi network. The beforehand nodal architecture is visualized below in Figure 4.3.

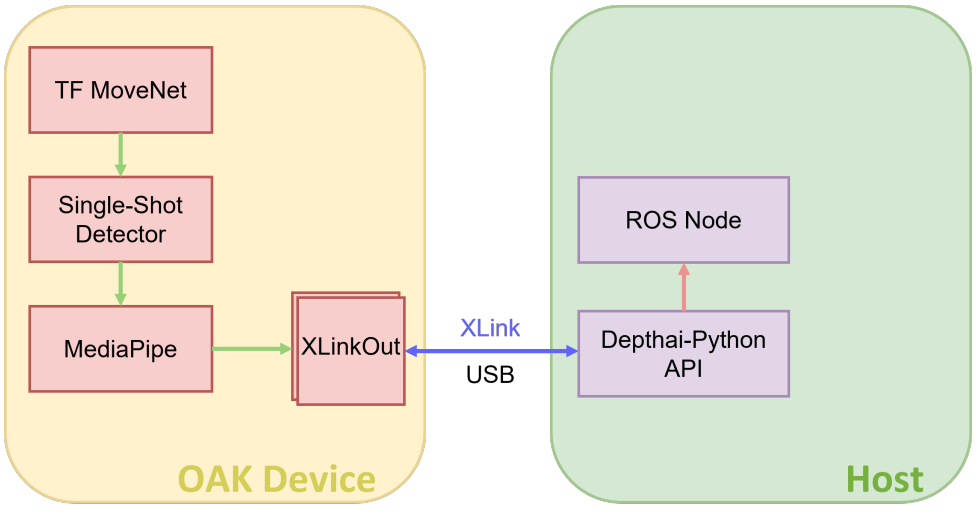

Figure 4.3: Illustation of the employed CV and ML pipeline on the Luxonis OAK-D S2 camera, showcasing node-based processing with TensorFlow MoveNet, SSD, and MediaPipe, connected via XLink to a host running the DepthAI Python API and ROS.

4.4.2 Hand Gesture Detector

In the previously described architecture, each frame is examined for hand gestures, detecting them if present. Nevertheless, directly using this unfiltered data stream without to trigger certain actions linked to hand gestures can be problematic due to the occurrence of false positives and negatives. To address this issue, a buffering and threshold-based approach is used to accurately detect specific hand gestures, including SOS, STOP, and OK gestures.

The STOP gesture detection algorithm performs checks to detect the presence of the FIVE gestures over a certain amount of frames, using a circular buffer. A real-world example for the STOP and OK hand gestures can be taken from Figure 4.4.

Figure 4.4: Figure on the left shows a FIVE hand gesture, which in this thesis was mapped to be a STOP hand gesture. On the right, the OK hand gesture is displayed (geaxgx, 2023).

For this purpose, algorithm 4 needs the XLink datastream originating from the camera, a value for the *threshold*, and a length of the sequence *sequenceLength*. *sequenceLength* corresponds to the number of consecutive FIVE gestures needed to be obtained while a certain amount of interruptions are tolerated, defined by the *threshold*. For every frame with a hand gesture present, the algorithm categorizes the hand gesture as a FIVE, OK, or OTHER gesture. After that, the sequence of shown FIVE gestures is identified as a STOP hand gesture if the FIVE gesture was detected a required amount of times adhering to the threshold. A schematic example is depicted in Figure 4.5. The same strategy has also been employed to detect the OK hand gesture.

Algorithm 4: STOP/OK Hand Gesture Detection Algorithm

Data: XLink Datastream, *threshold*, *sequenceLength*

Result: Detect if STOP/OK Hand Gesture is present

```
1  buffer ← empty queue;
2  currentGesture ← hand gesture from XLink datastream;
3  if currentGesture is FIVE or OK then
4  |  buffer ← append FIVE or OK to the queue;
5  else
6  |  buffer ← append OTHER to the queue;
7  end
8  correctGestureCount ← 0;
9  otherGesturesCount ← 0;
10 for gesture ∈ buffer do
11 |  if gesture is FIVE or OK then
12 |  |  correctGestureCount ← increase by 1;
13 |  |  otherGesturesCount ← reset to 0;
14 |  else
15 |  |  if gesture is OTHER and otherGesturesCount > 0 then
16 |  |  |  otherGesturesCount ← increase by 1;
17 |  |  |  if otherGesturesCount > threshold then
18 |  |  |  |  correctGestureCount ← reset to 0;
19 |  |  |  end
20 |  |  end
21 |  end
22 |  if correctGestureCount ≥ sequenceLength then
23 |  |  buffer ← clear buffer;
24 |  |  return True;
25 |  end
26 end
27 return False;
```

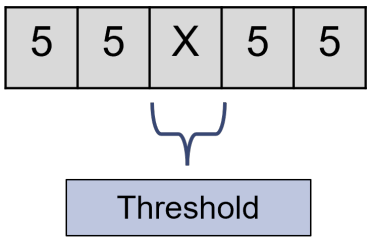

Figure 4.5: This illustration shows a buffer of length five, which gets filled with hand signs detected by the pipeline running on the camera. When the buffer is filled, and the threshold is not violated, a sequence of FIVE or OK hand signs is detected as STOP or OK hand gestures.

The signal for help, which is well established in the international community, is a single-handed gesture that shows the palm, tucking the thumb, and then trapping the thumb by folding down

the four fingers (“Signal for Help”, 2024). Figure 4.6 illustrates how to perform the SOS hand gesture.

Figure 4.6: This illustration shows how the SOS hand signal is executed (“Signal for Help”, 2024).

To identify this gesture, algorithm 5 needs an XLink data stream from the camera and a fault tolerance defined through *threshold*. Similar to the algorithm presented before, for every frame with a hand gesture present, the algorithm categorizes the hand gesture as a FOUR, FIST, or OTHER gesture. When a FOUR gesture appears, it is stored in the buffer, and the same happens with a FIST gesture. Then, the algorithm checks if a FIST follows a FOUR within a certain threshold. If so, the sequence is considered valid, and stored items in the buffer are deleted until the index where the sequence was evaluated as valid. Otherwise, if the sequence is interrupted by more malicious detections than tolerated by the threshold, the detection flag is reset.

Algorithm 5: SOS Hand Gesture Detection Algorithm

Data: XLink Datastream, *threshold*

Result: Detect if SOS Hand Gesture is present

```
buffer ← empty queue;
currentGesture ← hand gesture from XLink datastream;
if currentGesture is FOUR or FIST then
    buffer ← append currentGesture to the buffer;
else
    buffer ← append OTHER to the buffer;
end
consecutiveFistCount ← 0;
foundFour ← Flase;
for gesture ∈ buffer do
    if currentGesture is FOUR then
        foundFour ← True;
    else
        if currentGesture is FIST and foundFour then
            buffer ← deque up to current index;
            return True;
        else
            if foundFour is True then
                consecotiveFistCount ← increase by 1;
                if consecotiveFistCount > threshold then
                    foundFour ← reset to 0;
                end
            end
        end
    end
end
return False;
```

4.5 Obstacle Avoidance

As explained in section 2.5, obstacle avoidance plays a pivotal role in autonomous drone missions. Due to the lack of spare onboard compute power to run sophisticated obstacle avoidance algorithms such as presented in Baumann et al. (2018) and the lack of integration of a LiDAR sensor, which provides point cloud measurements, the choice fell on a simpler algorithm implementation as outlined in subsection 2.5.1.

The obstacle avoidance algorithm depicted in algorithm 6 will be explained in the following. First, it checks if an obstacle is located right in front of the UAS. For that, it relies on LiDAR sensor measurements, which provide data about the distance to an object right in front of the UAS. A simple rolling average has been employed to mitigate the effects of noisy measurement data. Measures will be initiated if the rolling average is evaluated below a certain threshold

value, for example, three meters, since an obstacle might be present. For this, the UAS will stop and first turn by -90° to determine whether it is safe to avoid onto the right side of the UAS. Therefore, after that turn, it will be checked if any obstacle is present in a range of two meters. If not, the UAS will move two meters forward and resume with the waypoint mission. However, there is also the case where after turning -90°, the UAS detects an obstacle within two meters. In that case, it will turn an additional 180° to check if it is safe to evade the obstacle on the left side of the original orientation of the UAS. There, the procedure is the same as explained after having turned -90° in the first place. Nevertheless, if there is also an obstacle on the left side of the UAS it will not find a path to overcome the obstacle. Figure 4.7 depicts the implemented obstacle avoidance procedure.

Algorithm 6: Obstacle Avoidance Algorithm

Data:

Result: Avoide Obstacle

```
x_uas ← subscribe to all relevant topics in ROS;
while ROS node is running do
    turn UAS by -90°;
    rolling average ← clear rolling average;
    wait for a second to get distance measurements;
    if rolling average ≤ 2 m then
        turn UAS by 180°;
        rolling average ← clear rolling average;
        wait for a second to get distance measurements;
        if rolling average ≤ 2 m then
            return False;
        end
        move forward by 2 m;
        return True;
    end
    move forward by 2 m;
    return True;
end
```

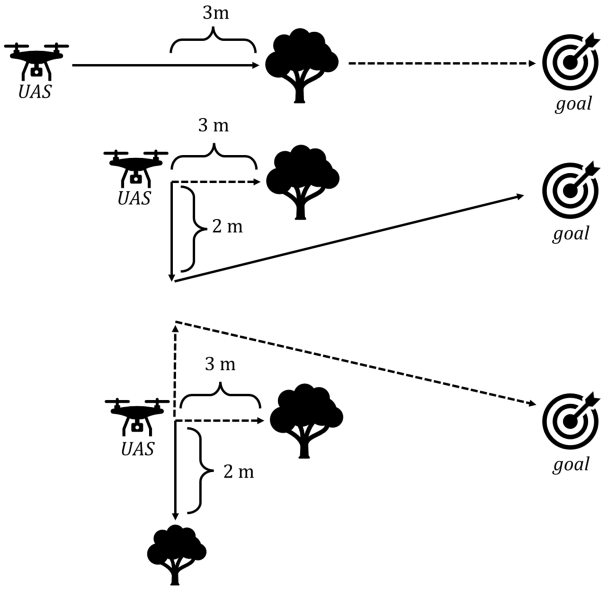

Figure 4.7: Exemplary illustration of the implemented obstacle avoidance procedure with the three base cases.

4.6 Experimental Setup

Validating the implemented software by deploying it on a real-world system is crucial to ensure that it does not only fulfill the requirements under perfect conditions as encountered in simulated environments. Therefore, an experimental setup was employed, which served as a test bed to thoughtfully test the software under real-world conditions and identify undetected issues when verified with simulations. The overall experimental setup can be divided into two parts. First, in subsection 4.6.1, the UAS setup is introduced with the additionally mounted sensors and camera. After that, in subsection 4.6.2, the ground station setup is presented.

4.6.1 UAS Setup

An open-source, pre-assembled, and ready-to-fly Holybro X500 V2 PX4 Development Kit is employed for a real system serving as a test bed for the developed software. This development kit comprises a Pixhawk 6C FCU compatible with the PX4 software stack and further sensors. Additionally, an NVIDIA® Jetson Nano™ is mounted to the back of the UAS and connected to the Pixhawk FCU via USB and a serial cable connection. Furthermore, two radio links are installed on the UAS and connected via cables to the UAS. One antenna establishes a data link to the ground station computer running QGC. The other antenna connects to the FCU for the remote control, which the operator can manually use.

For this thesis, additional LiDAR sensors have been acquired to test the implementation of the beforehand presented obstacle avoidance algorithm from section 4.5. First, the ST VL53L1X LiDAR sensor has been used. After that, the ST VL53L1X LiDAR sensor was replaced by a TFmini S LiDAR sensor due to encountered measurement noise and the limited range shown. Furthermore, a Luxonis OAK-D S2 autofocus camera has been mounted onto the UAS for hand

gesture recognition. The camera is connected to the onboard computer via a USB cable. The schematic overview of the components mounted onto the UAS is shown in Figure 4.8.

4.6.2 Ground Station Setup

The ground station setup is pivotal in the overall experimental setup since the UAS cannot be operated safely without it. It consists of a power source and a Wi-Fi router, which establishes the Wi-Fi network used by the ROS action servers running on the NVIDIA® Jetson Nano™ and the FMS running on the ground station computer. In detail, a lead battery is connected to an inverter to power a Wi-Fi router in the field since no electricity is available. Furthermore, a ground station computer on the ground is needed, which runs the QGC GCS to monitor the missions and, if necessary, intervene to maintain safety while operating the UAS. Lastly, a remote control maintains a third independent communication channel to the UAS to take control in an emergency. The schematic overview of the components used on the ground is shown in Figure 4.8.

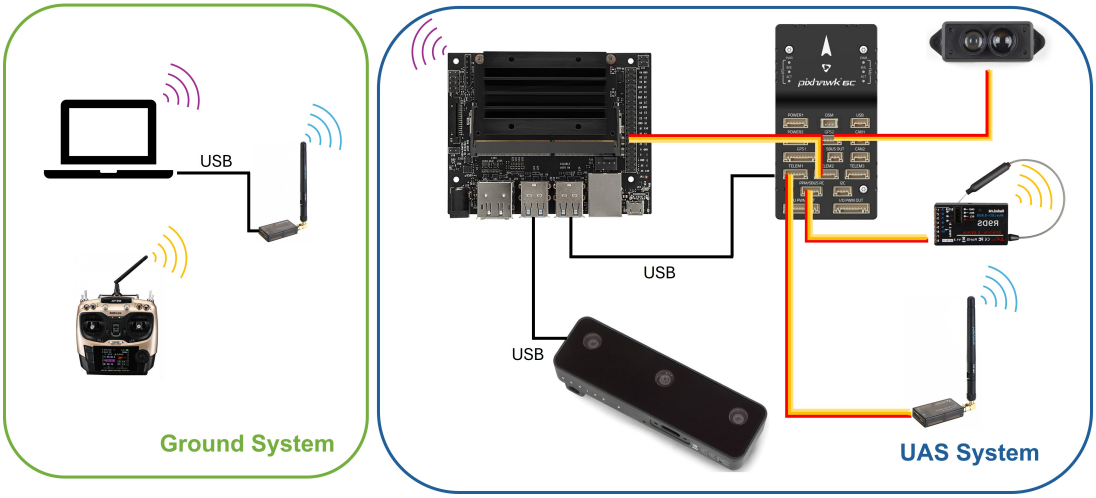

Figure 4.8: Schematic overview of the experimental setup. The green area indicates the ground system, whereas the area in blue shows the components mounted onto the UAS.

Chapter 5

Results

Some metrics need to be identified to measure and validate the implementation performance. These will be covered in the first section of this chapter; see section 5.1. Afterward, the results of the simulations conducted in Gazebo are presented in section 5.2. Last but not least, the results of real-world experiments will be presented in section 5.3, which have been conducted to back up the results obtained from simulation.

5.1 Metrics

To validate the system's implementation and measure its performance, the following metrics have been identified:

- **Quality Factor in [%]**
  An important metric to measure how well and reliably a system works is the quality factor, defined as the number of successful missions divided by the total number of conducted missions. The higher this percentage value is, the more trustworthy it is. Knowing its reliability is pivotal, especially when the implemented system might be deployed in the real world. However, this metric will be neglected in the results part since working with a development UAS platform came with significant drawbacks, which resulted in, for example, crashes where the propeller randomly went off during flight. Those incidents could be traced back to be caused by either the PX4 firmware or the hardware parts of the UAS. A limitation of this metric would be the development UAS platform itself; therefore, measuring the quality factor has been omitted in the results part.
- **Computational Load [%]**
  It is important to know if spare computational resources are still available on the onboard computing unit, particularly regarding potential system extensions. Nevertheless, it is also compelling to know the computational load to ensure system stability and thus mitigate overload situations.

- **Network Usage [$\frac{\text{MB}}{\text{s}}$]**
  The network usage is particularly interesting in environments with limited access to high-speed and high-bandwidth networks. Primarily in the case of the FEROX project, this metric reveals how practical the system's implementation is since the UAS should be able to autonomously act in remote forests where strong network coverage cannot be taken for granted.
- **System Latency [**ms**]**
  The system latency was measured to quantify the seamless integration of the Formiga FMS using the PX4 software stack on the Holybro X500 V2 UAS in ROS. Since ordered tasks have to route through a Wi-Fi network in the first place to get to the UAS where the demanded task faces multiple interfaces to translate it into actual commands that the UAS can perform, furthermore, this metric can also be understood as a metric of how efficient the code base concerning data transmission is since large overheads would further delay the task's start. Lastly, latency was also measured for hand gesture recognition, which is crucial for quickly reacting the UAS to certain hand gestures.
- **Hand Gesture Recognition Accuracy [%]**
  In order to measure how accurately one can interact with the UAS using hand gestures, the recognition accuracy needs to be determined. For CV and ML algorithms of this kind exist some well-established metrics such as Precision, Recall, and F1-Score, measured in % are introduced in detail in subsection 5.1.1. Those scores will also imply how certain an operator can be when a UAS claims to have recognized an emergency hand sign.
- **Obstacle Avoidance Rate [%]**
  This metric is used to quantify how reliably the implemented obstacle avoidance algorithm works by putting the number of successfully evaded obstacles in perspective of the number of encountered obstacles. The higher this fractional value is, the more reliable the obstacle avoidance algorithm is. This value is obtained through simulated flights since hardware issues arose when conducting the experiments in the real world when the LiDAR sensor was not publishing sensor data.

5.1.1 CV/ML Metrics

To quantify how accurately an employed CV/ML model is classifying objects, certain metrics have been established that are widely used and accepted in this area of research. First, the Precision metric is introduced. Afterward, the Recall metric is explained, and the F1-Score is presented.

In order to measure the accuracy of a classifier model such as the hand gesture detector used in this thesis, terms like true positive, false positive, true negative, and false negative need to be introduced. The columns in Figure 5.1 show the models' predictions, whereas the rows indicate the correct (true) classification. In this example, the object to be classified belongs to class b, which is considered the ground truth. So, if the model predicts that the object belongs to class b, that is counted as a true positive (TP) value when the model correctly predicts the object's class. However, if the model in this example predicted a, c, or d instead of the true class b, this would be counted as a false negative (FN). Otherwise, if the model classifies the object to be in class b the object belongs to any other class apart from b, that would correspond to a false

positive (FP) since the model considered the object belonging to b even though it should belong to any other class than b. Likewise, if the model did not predict b and the ground truth was also not b, this is considered a true negative (TN). Hence, the model correctly identified that the object does not belong to class b when it did not belong to class b.

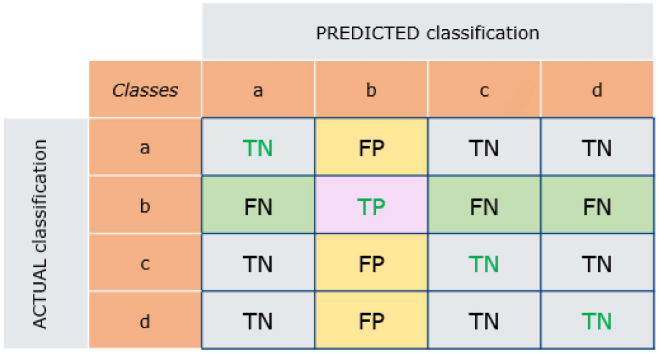

Figure 5.1: Example confusion matrix with classes a, b, c and d. The columns show the actual predicted label by the model, whereas the rows indicate the ground truth label (Mesuga & Bayanay, 2022).

Since the employed hand gesture model, similar to the previous example depicted in Figure 5.1, not only has two classes of choice but rather ten gestures, hence ten classes, a slightly modified version of the Precision, Recall, and F1-Score are presented.

Precision

The Precision for a multi-class classification model, see Equation 5.1, of a class i is a metric that measures how many of the in-total precited labels i, which is the sum of the true positives (TP) and the false positives (FP) of the class i, are true positives (TP) of this class i. Hence, to establish a connection to the previously presented example in subsection 5.1.1, the Precision indicates how many objects classified as i actually belong to class i.

$$\text{Precision}_i = \frac{\text{True Positives}_i \text{ (TP)}}{\text{True Positives}_i \text{ (TP)} + \text{False Positives}_i \text{ (FP)}} \tag{5.1}$$

Since a multi-class model has to be evaluated, the question arises of whether the Macro or Micro Precision should be considered for the overall performance assessment. The Macro Precision is defined as in Equation 5.2 below

$$\text{Macro Precision} = \frac{1}{C}\sum_{i=1}^{C} \text{Precision}_i \tag{5.2}$$

where Precision_i indicates the Precision obtained for a class i as in Equation 5.1 and C is the amount of different classes. The Macro Precision corresponds to the mean value of all Precisions for every class i. This implies that every class is equally weighted and should thus be equally often represented in the ground truth data. However, since some classes (hand gestures) have

been shown significantly more often than others, the Micro Precision appears to be a more suitable metric to not equally weight all the classes but rather on their appearances as shown in Equation 5.3.

$$\text{Micro Precision} = \frac{\sum_{i=1}^{C} \text{True Positives}_i}{\sum_{i=1}^{C} (\text{True Positives}_i + \text{False Positives}_i)} \tag{5.3}$$

Recall

In contrast to the beforehand presented Precision, the Recall measures how many of the ground truth class i labels, which is the sum of the true positives (TP) and the false negatives (FN), objects that the model did not capture as class i, are correctly identified as class i (TP). The exact formulation can be taken from Equation 5.4 below.

$$\text{Recall}_i = \frac{\text{True Positives}_i \text{ (TP)}}{\text{True Positives}_i \text{ (TP)} + \text{False Negatives}_i \text{ (FN)}} \tag{5.4}$$

Similar to the previously discussed precision, see subsubsection 5.1.1, there are two ways of calculating the overall Recall value. For the sake of completeness, the Macro Recall, according to Equation 5.5

$$\text{Macro Recall} = \frac{1}{C} \sum_{i=1}^{C} \text{Recall}_i \tag{5.5}$$

and the Micro Recall defined in Equation 5.6

$$\text{Micro Recall} = \frac{\sum_{i=1}^{C} \text{True Positives}_i}{\sum_{i=1}^{C} (\text{True Positives}_i + \text{False Negatives}_i)} \tag{5.6}$$

are presented, where C indicates the amount of classes. It is worth mentioning that if values are plugged into Equation 5.6, the obtained value will be the same as the calculated value in Equation 5.3 since both equations can be mathematically reformulated such that they are identical.

F1-Score

The F1-Score is the harmonic mean of Precision and Recall, which can be obtained separately for each class i. Hence, the F1-Score is defined as in Equation 5.7

$$\text{F1-Score}_i = \frac{2 \times \text{Precision}_i \times \text{Recall}_i}{\text{Precision}_i + \text{Recall}_i} \tag{5.7}$$

where Precision_i indicates the Precision value obtained through Equation 5.1 and likewise Recall_i by Equation 5.4.

Also, the Macro and a Micro F1-Score exist here. The Macro F1-Score is also defined to be the mean value of the F1-Scores across all classes i as indicated in Equation 5.8

$$\text{Macro F1-Score} = \frac{1}{C}\sum_{i=1}^{C}\text{F1-Score}_i \tag{5.8}$$

where the F1-Score_i is defined by Equation 5.7 and C is the amount of classes. Similar to the previous cases, the Micro F1-Score is defined as in Equation 5.9

$$\text{Micro F1-Score} = \frac{2 \times \text{Precision}_{\text{micro}} \times \text{Recall}_{\text{micro}}}{\text{Precision}_{\text{micro}} + \text{Recall}_{\text{micro}}} \tag{5.9}$$

where $\text{Precision}_{\text{micro}}$ is the value of the Micro Precision as in Equation 5.3 and $\text{Recall}_{\text{micro}}$ is the Micro Recall value as defined in Equation 5.6. Also here, it is worth mentioning that if values are plugged into Equation 5.9, the obtained value will be the same as the calculated value in Equation 5.3 and Equation 5.6 since the F1-Score is the harmonic mean of the two mentioned values.

5.2 In Gazebo Simulation

This section introduces the results of experiments conducted in the Gazebo Simulator. Two main types of experiments have been thoughtfully tested in simulation: the hand gesture recognition pipeline and the obstacle avoidance algorithm.

The UAS physics were simulated in a Gazebo environment, which was running on the ground station computer, while the ROS action servers were running on the NVIDIA® Jetson Nano™ Developer Kit connected to the same Wi-Fi network as the ground station computer. To make the experiment as similar as possible to the real-world experiments, the camera was connected through USB to the NVIDIA® Jetson Nano™ Developer Kit. Hence, the NVIDIA® Jetson Nano™ Developer Kit ran all the software as it would during a real-world flight. Only the sensor data readings are received through the Gazebo simulation environment connected via the Wi-Fi network.

5.2.1 Hand Gesture Recognition

This experiment was conducted in the laboratory where a person showed hand gestures in order to send commands to the simulated UAS in Gazebo. Figure 5.2 shows the number of frames where a hand gesture has been detected. The intensity of the blue color reveals how frequently a hand gesture was shown. One can easily observe that not all hand gestures are as often shown as others. For example, the gestures, ONE, TWO, THREE, PEACE, LET'S ROCK, CALL are not frequently shown since they are not required for the current implementation. Hence, other hand gestures such as FOUR, FIVE, FIST, OK are shown more often since they are used to trigger certain actions as explained in section 4.4.

It stands out that most hand gestures were classified correctly, as seen in Figure 5.2. Major confusion could be noticed when an OK hand gesture was shown, which then was detected as a FIST gesture.

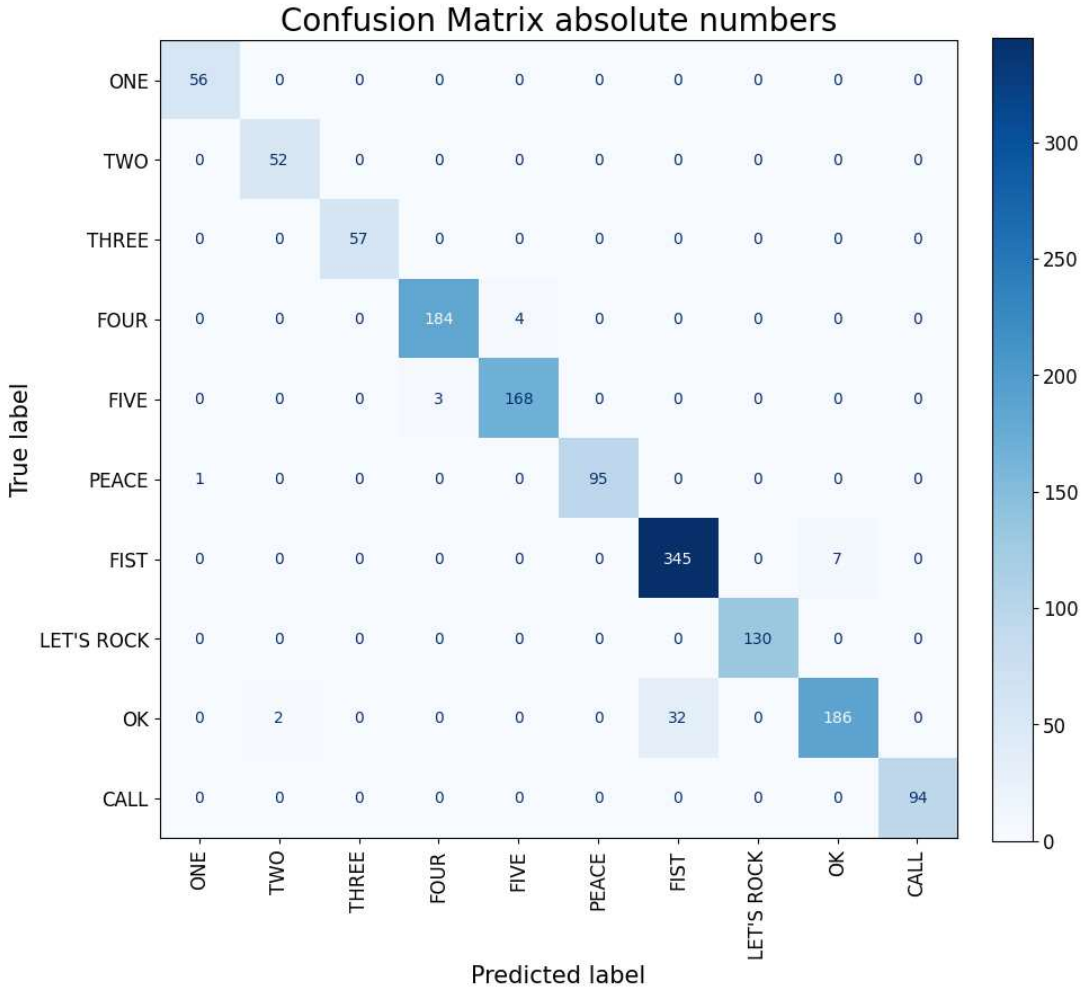

Figure 5.2: Confusion Matrix of the absolute number of frames where a hand gesture was detected. The rows indicate the ground truth labels, whereas the columns show the predicted label by the model.

This becomes more obvious in Figure 5.3 where the confusion matrix is normalized along its rows. Generally, anomalies can be detected for the FOUR, FIVE, PEACE, FIST, and OK classes. However, the misclassification errors for the FOUR, FIVE, PEACE, and FIST are small. They can often be explained by the fact that one finger was not correctly identified as being stretched out. Hence, the STOP hand gesture was mostly correctly identified, as the FIVE was detected reliably. Interestingly, the FIST gesture is mainly detected correctly. In contrast, the model shows difficulties detecting an OK hand gesture as it might not detect the thumb stretched out, which is the only difference to the FIST hand gesture.

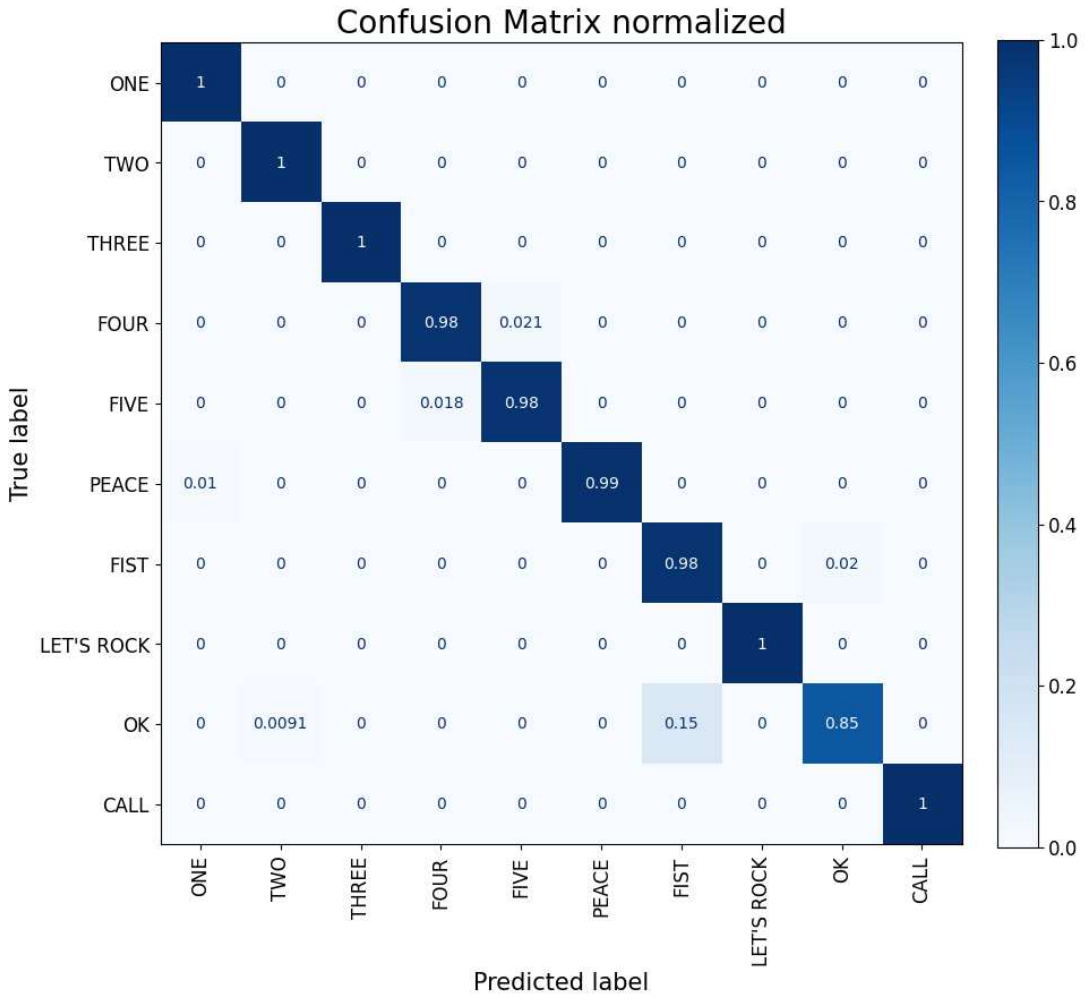

Figure 5.3: Confusion Matrix with normalized values along its rows. The rows indicate the ground truth labels, whereas the columns show the predicted label by the model.

Table 5.1 supports the abovementioned findings that the model recognizes most hand gestures sufficiently. The Precision value for the OK hand gesture indicates that the model sometimes wrongly classifies a different hand gesture, mainly the FIST hand gesture, as an OK hand gesture. This might be problematic when a person wants to call the emergency service as explained in subsection 4.4.2, and the FIST hand gesture is not identified as a fist but as an OK hand gesture. This would command the UAS to continue its mission. On the other hand, the low Recall value for the OK hand gesture indicates that a shown OK hand gesture is sometimes classified as a different hand gesture, mainly as a FIST. This misclassification is worrisome if a FOUR has been detected and followed up by an OK, which is misclassified as a FIST. This could trigger an unwanted emergency call.

	Precision	Recall	F1-Score
ONE	0.982	1.000	0.991
TWO	0.963	1.00	0.981
THREE	1.000	1.000	1.00
FOUR	0.984	0.979	0.981
FIVE	0.977	0.982	0.980
PEACE	1.00	0.990	0.995
FIST	0.915	0.980	0.947
LET'S ROCK	1.000	1.000	1.00
OK	0.964	0.845	0.901
CALL	1.000	1.000	1.00

Table 5.1: The obtained values for Precision, Recall, and F1-Scores separately for each class from experiments conducted in simulation.

As described in subsection 5.1.1, the Micro averaging rule is more suitable for this case since we mainly care about the important gestures such as the FOUR, FIVE, FIST, and OK, since those are crucial for the interaction with the UAS. For completeness, the Macro values of the Precision, Recall, and F1-Scores are also displayed in Table 5.2. The high Micro values indicate that the detection model accurately identifies the hand gestures.

	Macro	Micro
Precision	0.97850	0.978
Recall	0.97763	0.978
F1-Score	0.97752	0.978

Table 5.2: Precision, Recall, and F1-Score obtained across all hand gestures according to the Macro and Micro averaging rules from experiments conducted in simulation.

Lastly, Table 5.3 shows the metrics for the eleven showed SOS hand gestures. Eleven out of the eleven performed SOS hand gestures were detected correctly. In contrast, one false positive occurred when a FIVE got misclassified as a FOUR, followed by a FIST. Here, the Precision value is important to highlight since it puts more weight on classifying a hand gesture wrongly.

	Precision	Recall	F1-Score
SOS	0.917	1.00	0.957

Table 5.3: Precision, Recall, and F1-Score of the SOS hand gesture.

5.2.2 Obstacle Avoidance

The obstacle avoidance algorithm was tested in simulation first to evaluate if it works in theory. For this purpose, two boxes were placed in the Gazebo simulation environment. The UAS then got a command to go to a target point behind the boxes; hence, it faced obstacles during its flight. Figure 5.4 shows the taken flight path in global coordinates in blue. The starting positions were randomly changed every single time to create different situations. The red boxes show the obstacles placed in the simulated environment, whereas the green point shows the target point.

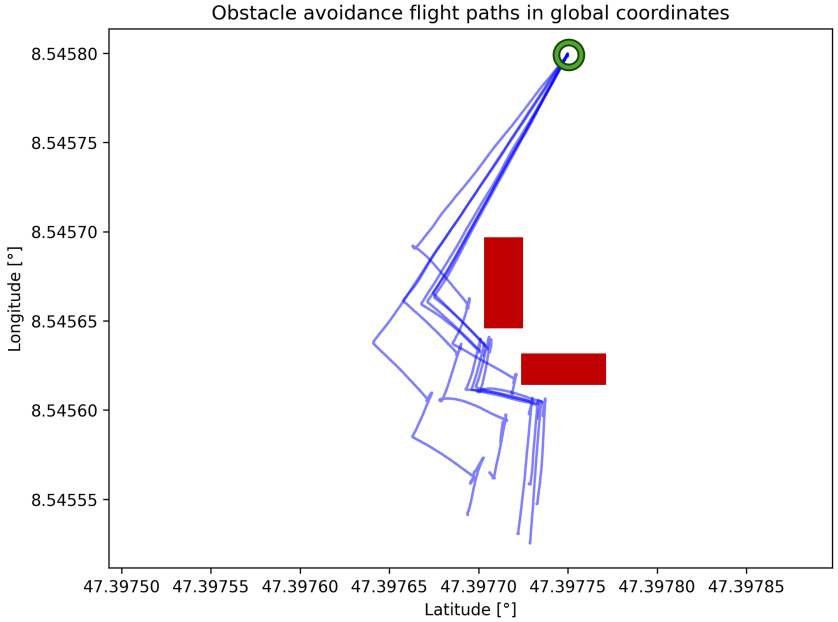

Figure 5.4: Flight paths in global coordinates of ten simulated UAS flights facing two obstacles, marked in red. The green point indicates the target point.

The UAS reached its goal in all ten flights without crashing or touching the obstacles. Hence, it was assumed that the obstacle avoidance algorithm should work during real-world flights.

5.3 Real-World Flight Experiments

5.3.1 Computational Load

Measuring the computational load the onboard computer faced during real-world flights is crucial in order to evaluate whether the onboard computer is operating at its capacity limit. This piece of information again helps to understand if there are free resources for further onboard software deployment and if it is safe to operate the UAS with this onboard computer.

The NVIDIA® Jetson Nano™ Developer Kit comes with a Quad-Core ARM® A57 CPU, 4 GB of LPDDR4 RAM and a 128-Core NVIDIA® Maxwell™ GPU. Only the load on the CPU has been measured since no CV/ML algorithms have been deployed to run on the GPU of the onboard computer. Figure 5.5 below shows the recorded CPU load during two consecutive flights where the first flight consisted of the following actions in this specific order: TakeOff, Explore, GoTo,

LandIn. Hence, all four implemented actions were used. Another flight was conducted directly connected to that flight, which explains the significant drop around the second 300 in Figure 5.5. The second flight only consisted of TakeOff, GoTo, and LandIn. During the GoTo action, the hand gesture recognition was tested. So the UAS was commanded to stop and trigger emergency responses several times by hand signs. As one can observe in Figure 5.5, the CPU load mainly stays below the 40% margin, which can be considered safe for real-time operations.

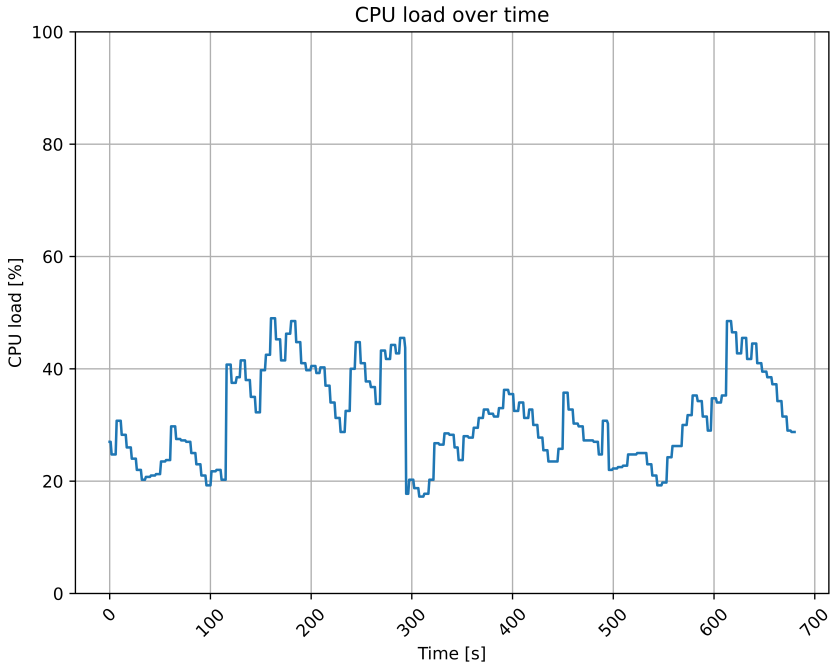

Figure 5.5: CPU load measurements on the NVIDIA® Jetson Nano™ Developer Kit during two consecutive flights. The first flight consisted of an Explore action, while the second flight consisted of multiple GoTo actions with hand gesture control.

The Table 5.4 below shows the CPU load's maximum, mean, median, and standard deviation during the two consecutive flights. Here, the CPU never exceeded 50% of the available CPU computing power. Furthermore, 50% of the time, the CPU load is even at 30.75% or below. Those insights reinforce the assumption that the onboard computer is not overwhelmed with its assigned tasks.

	CPU Load [%]
Maximum	49.00
Mean	31.39
Median	30.75
Standard Deviation	8.06

Table 5.4: CPU load's maximum, mean, median, and standard deviation in % during two consecutive flights.

5.3.2 Network Usage

Network usage plays a pivotal role, especially in the frame of the FEROX project, since network connectivity might be significantly limited in the intended operation areas, mainly dense, remote forests. Hence, some investigation has been done to determine the network requirements for operating the current state of the implementation. For this purpose, the onboard computer's down- and upload rates have been recorded. First, a more network-intensive setting was tested. Hand gestures were shown, and an implemented Flask web application streamed the camera image to a computer on the ground. The orange line in Figure 5.6 indicates the upload rate, consisting of data streams generated by the ROS action servers that send feedback, status, and results to the ground station computer. Furthermore, the UAS status has to be published to the ground station computer, including data about the current GPS position, battery level, and more. Last but not least, the camera image stream makes up the most significant part of the upload rate. On the other hand, the onboard computer uses the download rate in blue to receive goals or cancellations for the ROS action servers.

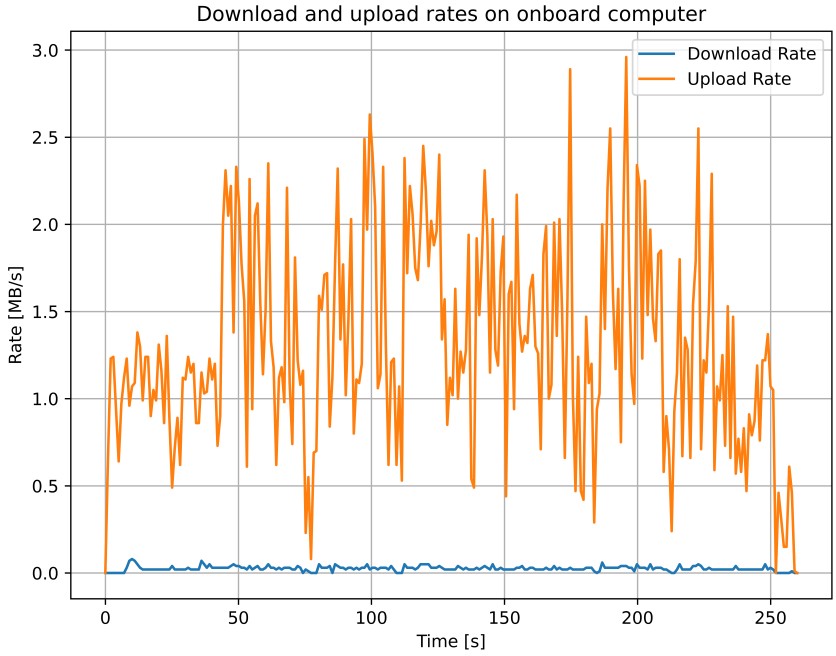

Figure 5.6: Band width measurements on the NVIDIA® Jetson Nano™ Developer Kit during a flight. The camera images have been streamed to a computer on the ground acting as a GCS.

The Table 5.5 below indicates the up- and download rate's maximum, mean, median, and standard deviation values during the flight mentioned above. The upload rate is much more prominent, especially in the setting when the camera images are streamed over the Wi-Fi network. The data traffic from the ground station computer to the onboard computer, considered in the download rate, is small since mainly ROS action goals must be sent.

	Upload Rate [MB/s]	Download Rate [MB/s]
Maximum	2.96	0.08
Mean	1.30	0.03
Median	1.22	0.02
Standard Deviation	0.59	0.02

Table 5.5: Used band width's maximum, mean, median (50 percentile), and standard deviation in MB/s for the upload and download rate.

The upload rate is expected to be significantly lower in a different setting, where the UAS operates without streaming the camera image to the ground station computer. This indeed comes closer to real-world operation since the main purpose of image streaming is debugging when testing hand gesture recognition. The downloading rate is omitted in the following analysis since sending ROS action goals remains the same, and thus, the download rate is not expected to alter much. Figure 5.7 shows the upload rate in KB/s while flying a mission consisting of TakeOff, Explore, GoTo, and LandIn actions. Without image streaming over the Wi-Fi network, the upload rate is particularly lower.

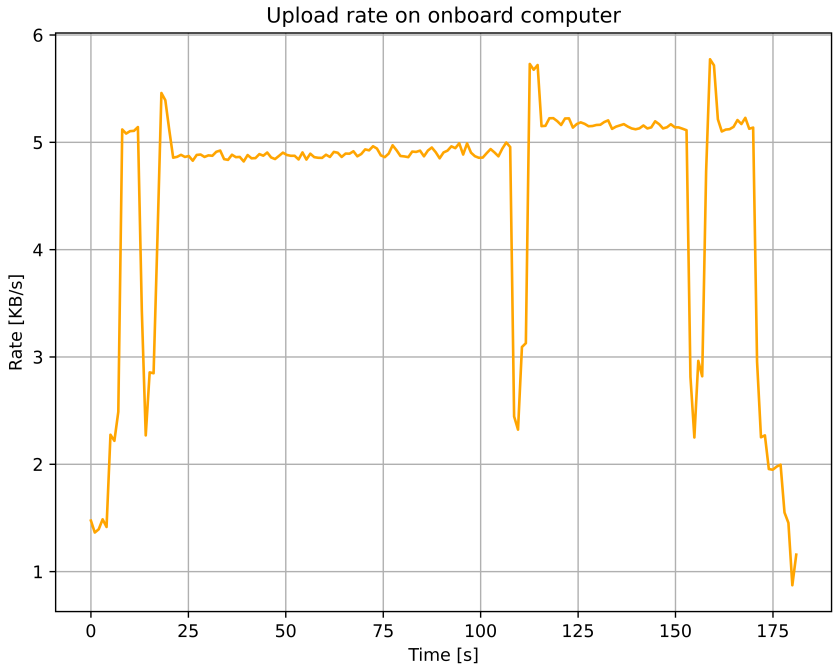

Figure 5.7: Band width measurements on the NVIDIA® Jetson Nano™ Developer Kit during a flight. No camera images have been streamed over the Wi-Fi network.

The Table 5.6 shows the download rate's maximum, mean, median, and standard deviation values. Compared to the upload rate values presented in Table 5.5, the values are by magnitudes lower, showcasing that the setup could operate with a low bandwidth Wi-Fi network.

	Upload Rate [KB/s]
Maximum	5.77
Mean	4.53
Median	4.90
Standard Deviation	1.12

Table 5.6: Used band width's maximum, mean, median (50 percentile), and standard deviation in KB/s for the upload rate without camera image streaming.

5.3.3 System Latency

To conduct reliable latency measurements, the system clock of the onboard computer and the ground station computer need to be synchronized since timestamps of data packets are compared to obtain the latency. This is achieved by employing the network time protocol, which synchronizes clocks of different computers in a network. By synchronizing the clocks of the onboard computer and the ground station computer every minute, the network time protocol estimated the clock difference between the two systems to be below a millisecond. Since this difference was much smaller than the measured latencies, the time difference between the systems was not further considered for this evaluation.

First, the reaction time caused by the communication link or the software running on the onboard computer was measured. So, the time difference between when the data frame left the ground station computer and when it was received at the onboard computer was measured. Figure 5.8 on the left shows a histogram of the measured latency caused by the communication link. As communication relies on Wi-Fi, which transmits signals almost at the speed of light, routing and other small delays caused by the processing of data frames could mainly be responsible for the small observed latencies. However, the average and mean values indicate that the communication link is reliable and fast.

The overall reaction latency is mainly caused by the ROS action servers that must process the received data and start the ROS action. The latency introduced by this procedure is depicted on the right in Figure 5.8. As one can observe, the median reaction latency stemming from software is five times bigger than the median of the communication latency. The latency caused by software is still considered to be low.

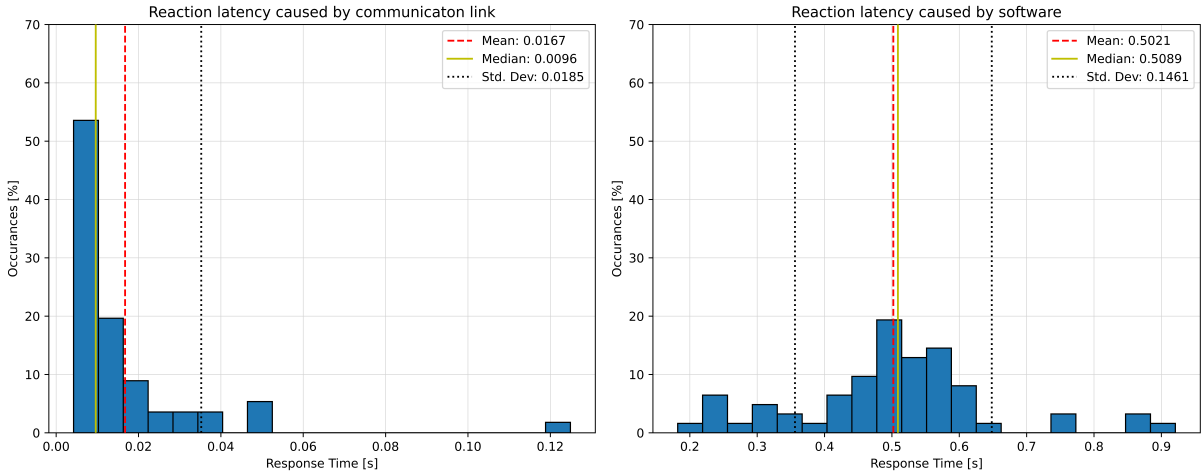

Figure 5.8: Figure on the left shows a histogram of the reaction latency caused by the communication link, including the mean, median, and standard deviation values, whereas the right figure shows the latency caused by the software.

Measuring the reaction times for recognizing the STOP and SOS hand gestures involves comparing the timestamp of the last frame needed to identify the hand gesture against the system time when the detection algorithm actually identifies the hand gesture.

The histogram in Figure 5.9 on the left shows the reaction times measured for detecting the STOP hand gesture, whereas the right figure shows the reaction times for detecting the SOS hand gestures. Detection of the SOS hand gesture takes more time since the detection algorithm is more sophisticated (for reference, see subsection 4.4.2). However, the standard deviation for detecting an SOS hand gesture implies that the reaction times can vary greatly, as observed in the histogram below.

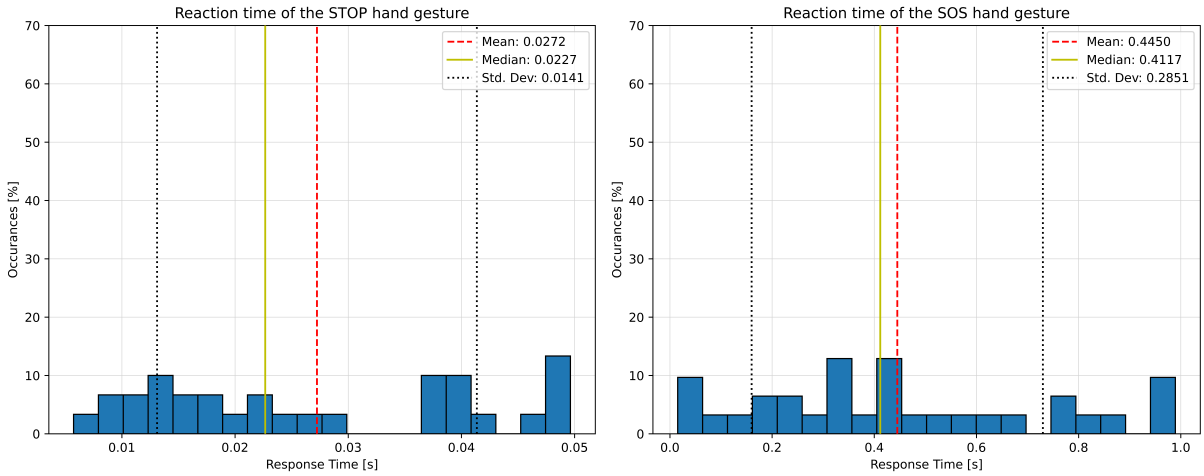

Figure 5.9: Figure on the left shows a histogram of the reaction times for recognizing the STOP hand gestures, including the mean, median, and standard deviation values, whereas the right figure shows the reaction times for recognizing the SOS hand gesture.

Even though the measured latencies can be considered low, the actual latency observed during real-world flights is slightly higher since it takes the UAS some time to perform a full stop,

for example. The duration between initiating the stopping command from the FCU and the subsequent loitering of UAS is challenging to quantify and has been excluded from the latency measurements discussed above.

5.3.4 Hand Gesture Recognition

The main difference to the setup explained in subsection 5.2.1 is that the hand gesture detection algorithm was tested during 15 real-world flights. For this purpose, the UAS received a GoTo action goal. On the way to its goal, a person stood underneath the UAS and showed hand gestures. The findings of those experiments are presented in the following sections.

Camera Image

One issue faced when conducting the hand gesture experiments was the significant blurriness of the camera image. This posed major challenges to the detection model as the hand could not be detected in the blurred regions of the image. Figure 5.10 below shows three images taken during a takeoff. The image on the left shows the camera image when the propellers were not spinning, whereas the rotational speed of the propellers gradually increased from left to right.

Figure 5.10: The images from left to right show the camera image evolution when propellers start spinning during a takeoff.

The observed blurriness most likely results from the vibration produced by the propellers. Hence, during the flight, the camera images show blurry regions. Innumerable times, the hand gesture recognition model showed difficulties detecting that there was a hand. Figure 5.11 shows a situation where a FIST was shown, but the model could not detect the hand gesture.

Figure 5.11: FIST hand gesture cannot be detected due to blurriness of the camera image.

The quality of the camera image got worse when the UAS was moving due to the effect known as motion blur. Motion blur appears if the camera shutter is opened longer while the object it captures is moving.

Hand Gesture Detection Results

Similar to the results part in subsection 5.2.1, some hand gestures were shown more often, leading to a class imbalance. Figure 5.12 shows the number of frames where a hand gesture has been detected. The intensity of the blue color reveals how frequently a hand gesture was shown. Conspicuous is, in contrast to the laboratory-conducted experiment, that more hand gestures were classified as a significantly different class. For example, the THREE was sometimes misclassified as a FIVE. However, if a hand gesture was detected, it was often correct.

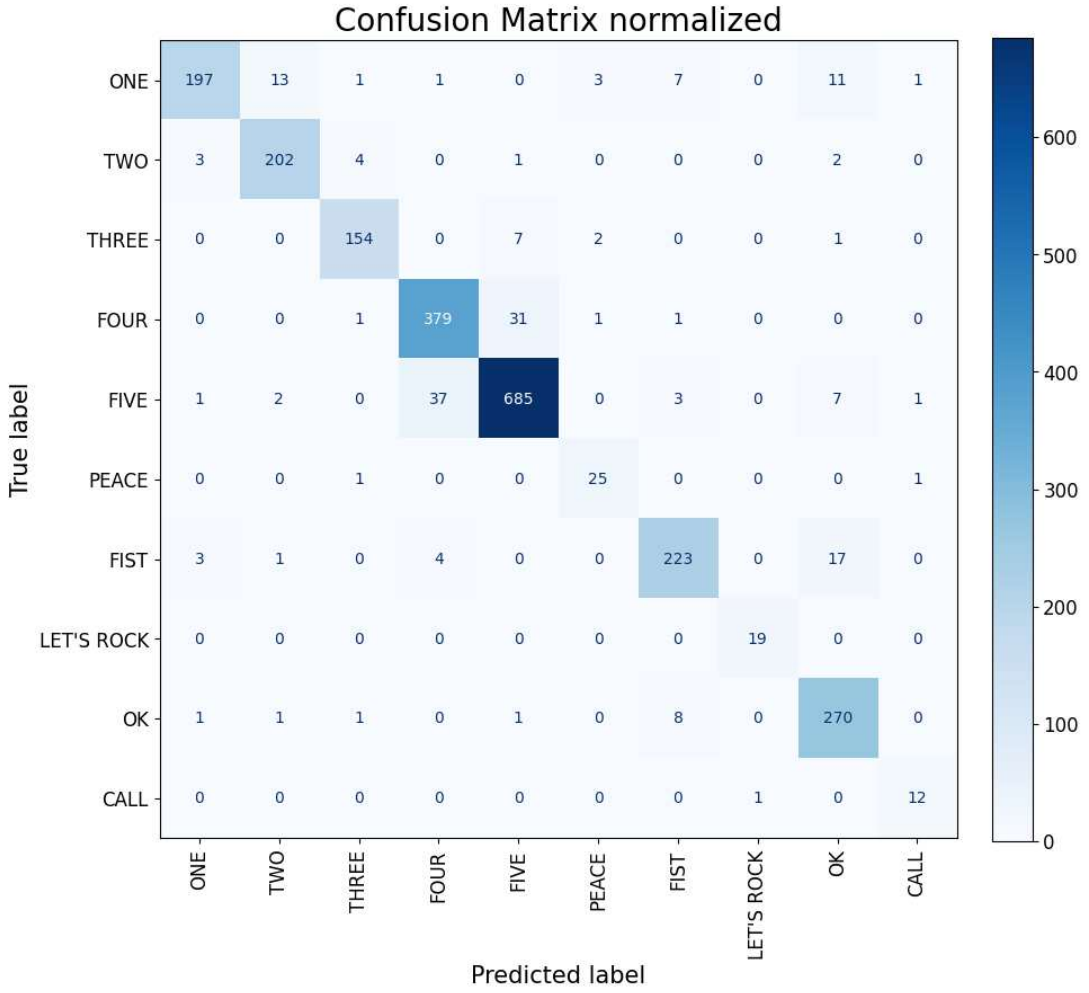

Figure 5.12: Confusion Matrix of the absolute number of frames where a hand gesture was detected. The rows indicate the ground truth labels, whereas the columns show the predicted label by the model.

This becomes more obvious in Figure 5.13 where the confusion matrix is normalized along its rows. Here, anomalies can be detected for the ONE, FOUR, FIVE, and FIST classes. Some of the misclassifications can be explained by the fact that one finger was not correctly identified as being stretched out. Hence, the STOP hand gesture was mostly correctly identified, as the FIVE was detected reliably. Interestingly, the OK gesture is mainly detected correctly. In contrast, the model shows difficulties when detecting a FIST. This could make the detection model prone to not correctly identifying the SOS hand gesture since a FOUR is followed by a FIST, which sometimes gets misclassified as an OK hand gesture.

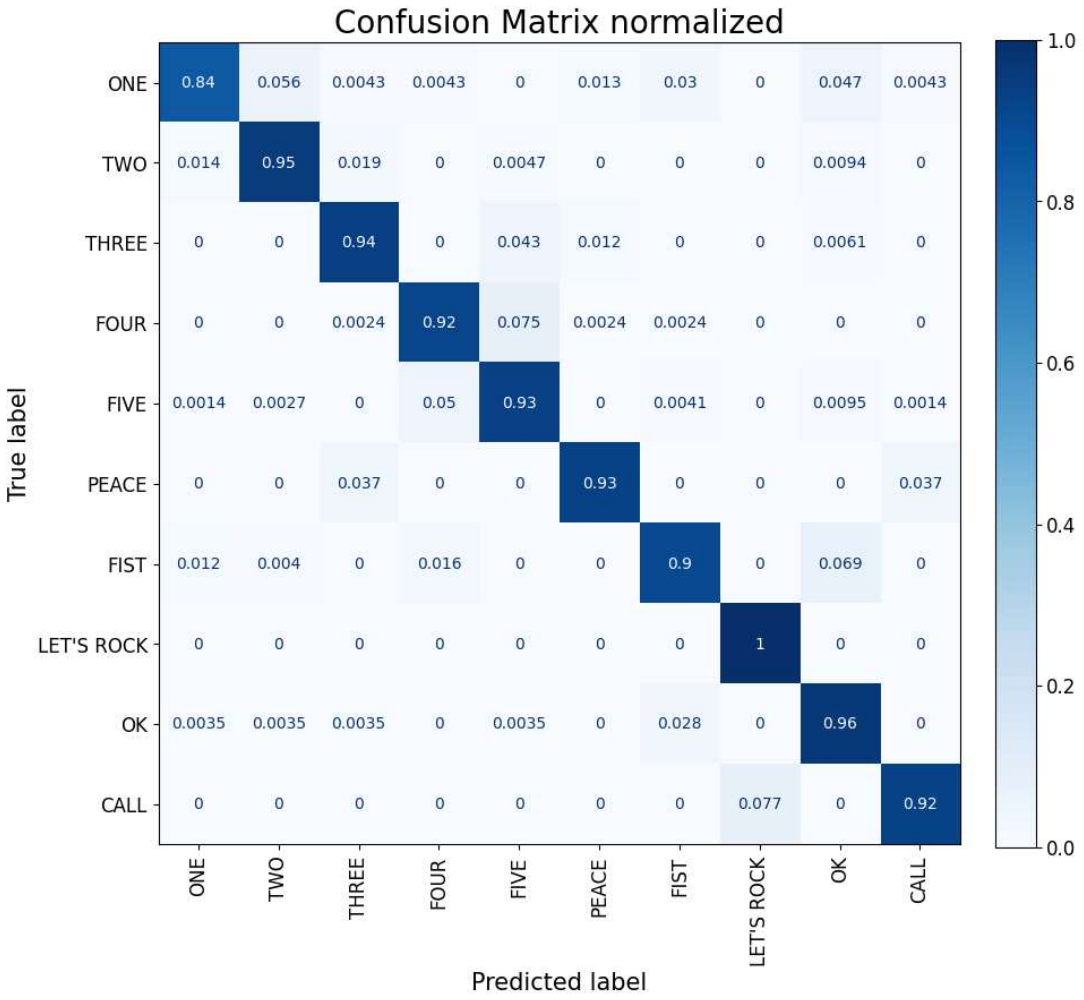

Figure 5.13: Confusion Matrix with normalized values along its rows. The rows indicate the ground truth labels, whereas the columns show the predicted label by the model

Table 5.7 supports the abovementioned findings that the model sufficiently recognizes most hand gestures if detected. The low Precision value for the FOUR hand gesture could lead to issues when detecting the SOS hand gesture, as false positives in detecting a FOUR could also decrease the Precision value for the SOS hand gesture. This could increase the likelihood that an unwanted emergency call is triggered.

	Precision	Recall	F1-Score
ONE	0.961	0.842	0.897
TWO	0.922	0.953	0.937
THREE	0.951	0.939	0.945
FOUR	0.900	0.918	0.909
FIVE	0.945	0.931	0.938
PEACE	0.806	0.926	0.862
FIST	0.921	0.899	0.910
LET'S ROCK	0.950	1.000	0.974
OK	0.877	0.957	0.915
CALL	0.800	0.923	0.857

Table 5.7: The obtained values for Precision, Recall, and F1-Scores separately for each class from real-world experiments.

As described in subsection 5.1.1, the Micro avereraging rule is more suitable for this case since we mainly care about the important gestures such as the FOUR, FIVE, FIST and OK, since those are crucial for the interaction with the UAS. For completeness, the Macro values of the Precision, Recall, and F1-Scores are also displayed in Table 5.2. The high Micro values indicate that the detection model mostly identifies the hand gestures correctly.

For completeness, Table 5.8 shows also the Precision, Recall, and F1-Scores according to the Macro rule. It is evident that the Micro values have significantly decreased compared to the lab setup. However, these metrics only reflect the performance of detected hand gestures. Undetected hand gestures are not accounted for in these results.

	Macro	Micro
Precision	0.903	0.922
Recall	0.929	0.922
F1-Score	0.915	0.922

Table 5.8: Precision, Recall, and F1-Score obtained across all hand gestures according to the Macro and Micro averaging rules from real-world experiments.

Lastly, Table 5.9 shows the metrics for the 32 showed SOS hand gestures during real-world flights. 30 were detected correctly, whereas six false positives and two false negatives occurred. This could be explained by the fact that a FIVE sometimes got misclassified as a FOUR followed by a FIST. Here, the Precision value indicates that the model sometimes triggers false positive emergency calls, whereas the Recall suggests that the model detected not all shown SOS hand gestures.

	Precision	Recall	F1-Score
SOS	0.833	0.938	0.870

Table 5.9: Precision, Recall, and F1-Score of the SOS hand gesture.

The hand gesture detection model works sufficiently well despite poor camera image quality. However, it is important to mention that the model often did not recognize that a hand gesture was shown. This is not properly reflected in the above-shown confusion matrix in Figure 5.13 where only the detected hand gestures are evaluated. Considering the undetected hand gestures is difficult as one would have to count the number of frames in which a hand gesture was shown, which the model did not detect to capture the error correctly. This is not covered in this thesis.

5.3.5 Obstacle Avoidance

Despite the results presented in subsection 5.2.2, the obstacle avoidance algorithm did not work during real-world flights. The main issue encountered was that the LiDAR sensor occasionally stops publishing sensor data. Figure 5.14 shows the recorded distance measurements obtained by the TFmini LiDAR sensor. At around the second 50, a first lack of data can be observed. Other data lacks can be found at around the second 80 and 100.

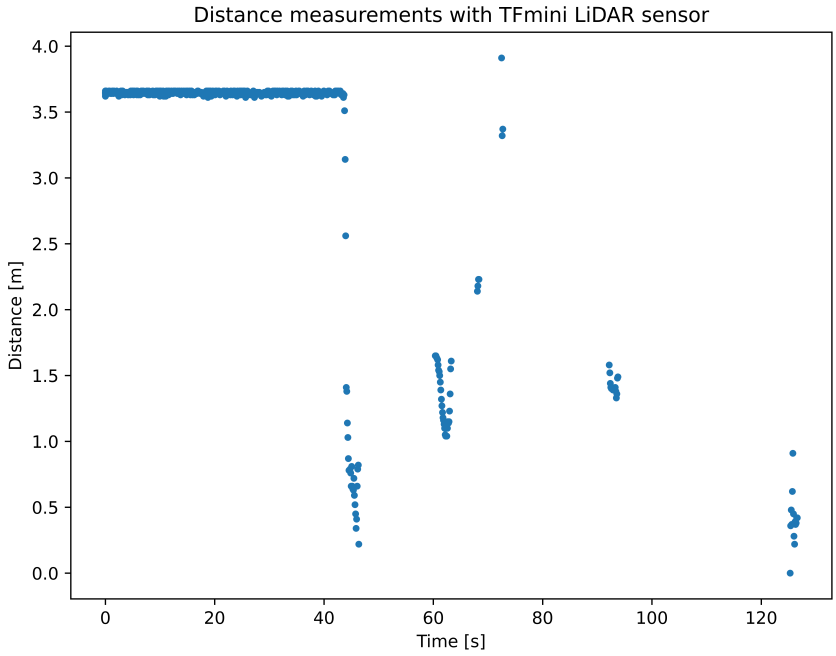

Figure 5.14: Distance measurements obtained by the TFmini LiDAR sensor during a real-world flight.

Also, similar observations were made during other flights using the TFmini LiDAR sensor. This is problematic as the obstacle avoidance algorithm relies on only one LiDAR sensor and needs to be fed with continuous distance measurements. Reference measurements were taken in the laboratory to reproduce this effect. However, the effect did not occur. Hence, the vibrations caused by the propellers during a flight are suspected to lead to this data loss. This assumption has not been backed up by data so far.

Chapter 6

Discussion

This chapter discusses the results presented in chapter 5.

6.1 Integration of the Formiga FMS

The first part of this thesis turned out to be performing at a sufficiently reliable level, as the results in subsection 5.3.1 to subsection 5.3.3 showcase the implementation is robust, and resources are left for further expansions. Furthermore, the setup is feasible as it could operate with sufficient network coverage. Overall, the software stack can be used as a test bed for future improvements on the Formiga FMS to verify its functionality in the real world. Major difficulties encountered during the project was that the Formiga FMS was still under development and hence under constant change, which influenced the integration implementation. Overall, the first goal, as introduced in section 1.3, was achieved.

6.2 Hand Gesture Recognition

The hand gesture recognition algorithm was tested in a laboratory and multiple times during real-world flights. As indicated in subsection 5.2.1, with good quality camera images, commanding a simulated UAS to stop, triggering an emergency call, and releasing the vehicle, is functioning well. All SOS hand gestures were detected, and only one false positive was observed. This observation changed when the same experiments were conducted under real-world conditions. The image quality drastically suffered due to vibrations detected by observations made during takeoff as described in subsubsection 5.3.4. The blurriness caused problems and inaccuracies in detecting hand gestures, which had a noticeable effect on the results, as seen in subsubsection 5.3.4. Hand gestures were mainly classified correctly when detected. Likewise, with the image quality, the detection of hand gestures suffered drastically. Further research revealed that the auto-focused version of the camera employed was unsuitable for this use case. Overall, the second goal, as described in section 1.3, was partially achieved since further improvements need to be done in

order to make the onboard hand gesture recognition more reliable. However, the third goal was accomplished. The presented results in subsection 5.3.4 can be interpreted as a proof of concept.

6.3 Obstacle Avoidance

Results presented in subsection 5.2.2 showed that the implemented obstacle avoidance algorithm can handle simple situations where there are not too many obstacles around the UAV. Contrary to expectations, real-world flights revealed a problem with the LiDAR sensor, which has not been rectified so far. According to the result presented in subsection 5.3.5, data loss was observed. Due to this unreliability, the obstacle avoidance algorithm could only be tested in simulation. Since no specific patterns could be identified, it is assumed that loose cable contact might be the cause.

Chapter 7

Conclusion and Future Work

This chapter contains some annotations regarding the current status of the implementation and the steps that need to be taken for further improvements.

Integrating the Formiga FMS onto the Holybro UAS platform works flawlessly. For this purpose, different metrics were captured and evaluated, which led to this conclusion. However, additional real-world flights should be performed to obtain the quality factor as explained in section 5.1. This metric is important to potential users to build trust in the system.

To improve hand gesture recognition during real-world flights, the fixed-focus version of the Luxonis OAK-D S2 should be used. It is advisable to decouple the frame, which is housing the camera, from the frame of the Holybro UAV to diminish vibrations as much as possible.

Last but not least, the obstacle avoidance algorithm needs to be thoughtfully tested during real-world flights. As seen in subsection 5.3.5, the underlying cause of the lack of published data of the LiDAR sensor must be correctly identified. So far, an assumption could be made that vibrations lead to a fragile cable connection. However, this assumption needs to be backed up with further inspections and data. Since the implemented obstacle avoidance algorithm could only be validated through simulation, the final objective of this thesis was only partially achieved.

Part II

Budget

Chapter 8

Budget

This chapter gives an overview of the costs faced during this thesis. First, the overall material cost is specified in section 8.1. Afterward, software license expenses are introduced in section 8.2. Then, the cost of the working hours is estimated in section 8.3. Last but not least, in section 8.4, the total project costs are presented by adding a profit margin, reserves for unforeseen costs, and taxes.

8.1 Material

The overall cost for the material used is broken down into two separate tables. The first table, Table 8.1, gives an overview of the cost caused by the laptop usage for this project. The Framework Laptop 13 has been used to develop software, conduct SITL experiments, conduct field experiments, and document the project. The Framework Laptop 13 is equipped with an Intel® Core™ i7-1360P, 32 GB of DDR4 RAM, and 1 TB of NVMe M.2 SSD memory. According to industry standards, the purchase cost of this laptop is fully depreciated over five years. Thus, the cost of using this laptop during this project is calculated as the fraction of the project duration divided by the five years multiplied by the purchase cost. This has indirectly been done in Table 8.1 where the hourly cost has been derived and multiplied by the time when the laptop was used for this project.

Item	Purchase Price [€]	Cost per hour [€/h]	Total cost [€]
Frame Laptop 13	1439.00	0.03285	143.90
Total cost of computer equipment.			**143.90**

Table 8.1: Total cost of computer equipment.

In Table 8.2, the total material cost is specified. The expenses include the purchase of a real-world drone, an onboard computer with an additional Wi-Fi module and antennas, a Wi-Fi router to establish a network for connectivity, batteries and chargers for the drone and for powering the Wi-Fi router, a Luxonis camera used for the implemented hand gesture control and two LiDAR sensors for obstacle avoidance.

Item	Units	Cost per unit [€]	Total cost [€]
Holybro X500 V2 Development Kit	1	468.25	468.25
Holybro X500 V2 Frame Kit	1	170.53	170.53
Radiolink AT9S remote control	1	147.00	147.00
NVIDIA® Jetson Nano™ Developer Kit	1	295.44	295.44
Intel® Dualband-Wireless-AC 8265 module	1	32.40	32.40
Wi-Fi antennas	4	1.78	7.12
Mercusys MR3000X Wi-Fi router	1	49.99	49.99
LiPo batteries	3	67.00	201.00
LiPo battery charger	1	319.00	319.00
Lead acid battery 20 A h	1	101.16	101.16
Ferve F-2520 lead battery charger	1	137.25	137.25
Genius Power 12 V - 230 V 300 W Inverter - G-12-300	1	34.95	34.95
Luxonis OAK-D S2 AF camera	1	280.20	280.20
TFmini S Uart LiDAR	2	40.83	81.66
ST VL53L1X LiDAR	1	13.83	13.83
Total material cost			**2339.78**
Total material cost (material + computer)			**2483.68**

Table 8.2: Total cost of the used material.

8.2 Software Licenses

Table 8.3 shows the expenses made for licenses. For developing the software, two operating systems were used. The Framework Laptop 13 came with a pre-installed Microsoft® Windows® 11 Pro. The other operating system used in this project was Linux Ubuntu 20.04 LTS, which was free of charge. The primary tool used to write code was Visual Studio Code, which is also free of charge. The whole PX4 software environment and Gazebo and QGC are fully open-source and thus free. The same applies to ROS and Google's MediaPipe framework. Overleaf has been used for documentation, which is also free for students.

Item	Type of License	Cost [€]
Microsoft® Windows® 11 Pro	perpetual	0.00
Linux Ubuntu 20.04 LTS	perpetual	0.00
PX4 firmware	perpetual	0.00
Gazebo	perpetual	0.00
QGroundControl	perpetual	0.00
Visual Studio Code	perpetual	0.00
Overleaf	perpetual	0.00
ROS	perpetual	0.00
Google MediaPipe framework	perpetual	0.00
Total license cost		**0.00**

Table 8.3: Total cost of the software licenses.

8.3 Labor Costs

Table 8.4 shows the costs for hours worked, derived from a standard hourly wage for a junior robotics engineer working in the Valencia region. Thus, a standard hourly wage of 15.19 €[1] was used.

Task	Duration [h]	Cost [€]
Bibliographic review	90	1367.10
Developing software for integration in Formiga FMS	360	5468.40
Developing software for integration of hand gesture control	150	2278.50
Validation in simulation	70	1063.30
Validation with field experiments	90	1367.10
Documentation of the project	140	2126.60
Total cost of working hours		**13671.00**

Table 8.4: Total cost of the working hours.

8.4 Total Costs

Table 8.5 contains the total gross cost. This includes all the expenses made for the material and the licenses. Furthermore, it considers the working hours spent on this project. According to industrial standards, an industrial profit margin of 6% and additional costs of 13% were added. These cost factors are reflected in the total gross cost.

[1]Based on the resolution from 17th of May 2024 published by the Valencian government. *Boletín Oficial de la Provincia de Valencia*, 17th of May 2024, number: 2024/06348

Cost Factor	Cost [€]
Material costs (material + computer)	2483.68
License costs	0.00
Working hours	13671.00
Total gross costs	**16154.68**
Profit margin (6%) + Additional costs (13%)	3069.39
Total gross costs + Industrial profit margin + Additional costs	**19224.07**

Table 8.5: Total gross costs.

To obtain the final net budget needed for this project, the Value Added Tax must be counted, accounting for 21%. Table 8.6 contains this cost leading to the final total net budget.

Cost Factor	Cost [€]
Total gross costs	19224.07
Value Added Tax (VAT) of 21%	4037.05
Total net budget	**23261.12**

Table 8.6: Total net budget.

The final total net budget amounts to TWENTY-THREE THOUSAND TWO HUNDRED SIXTY-ONE EUROS AND TWELVE CENTS.

Bibliography

Alaimo, A., Artale, V., Milazzo, C., & Ricciardello, A. (2013). Comparison between euler and quaternion parametrization in uav dynamics. *AIP Conference Proceedings*, *1558*, 1228–1231. https://doi.org/10.1063/1.4825732 (cit. on p. 10).

Baumann, T., Meier, L., & Pollefeys, M. (2018). Obstacle avoidance for drones using a 3dvfh* algorithm, 67 (cit. on pp. 28, 41).

Bhat, G., Dudhedia, M., Panchal, R., Shirke, Y., Angane, N., Khonde, S., Khedkar, S., Pansare, J., Bere, S., Wahul, R., & Gawande, S. (2024). Autonomous drones and their influence on standardization of rules and regulations for operating–a brief overview. *Results in Control and Optimization*, *14*, 100401. https://doi.org/https://doi.org/10.1016/j.rico.2024.100401 (cit. on p. 3).

Borenstein, J., & Koren, Y. (1991). The vector field histogram-fast obstacle avoidance for mobile robots. *IEEE Transactions on Robotics and Automation*, *7*(3), 278–288. https://doi.org/10.1109/70.88137 (cit. on p. 27).

Bouabdallah, S., Becker, M., Perrot, V., & Siegwart, R. Y. (2007). Toward obstacle avoidance on quadrotors (cit. on pp. 23, 24, 27).

Brescianini, D., Hehn, M., & D'Andrea, R. (2013). *Nonlinear quadrocopter attitude control. technical report* (Report). Zürich, Eidgenössische Technische Hochschule Zürich, Departement Maschinenbau und Verfahrenstechnik. https://doi.org/10.3929/ethz-a-009970340 (cit. on p. 11).

Cordova, F., & Olivares, V. (2016). Design of drone fleet management model in a production system of customized products. *2016 6th International Conference on Computers Communications and Control (ICCCC)*, 165–172. https://doi.org/10.1109/ICCCC.2016.7496756 (cit. on p. 25).

Dardoize, T., Ciochetto, N., Hong, J.-H., & Shin, H.-S. (2019). Implementation of ground control system for autonomous multi-agents using qgroundcontrol. *2019 Workshop on Research,*

Education and Development of Unmanned Aerial Systems (RED UAS), 24–30. https://doi.org/10.1109/REDUAS47371.2019.8999717 (cit. on p. 7).

Emimi, M., Khaleel, M., & Alkrash, A. (2023). The current opportunities and challenges in drone technology. *1*, 74–89 (cit. on p. 4).

Feraru, V. A., Andersen, R. E., & Boukas, E. (2020). Towards an autonomous uav-based system to assist search and rescue operations in man overboard incidents. *2020 IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR)*, 57–64. https://doi.org/10.1109/SSRR50563.2020.9292632 (cit. on p. 3).

Findelair, A. (2021). Vision-based gesture-controlled drone. (Cit. on p. 26).

geaxgx. (2023). Depthai_hand_tracker [Accessed: 30.08.2024]. (Cit. on pp. 20, 36, 38).

Goyal, G., Di Pietro, F., Carissimi, N., Glover, A., & Bartolozzi, C. (2023). Moveenet: Online high-frequency human pose estimation with an event camera. *2023 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, 4024–4033. https://doi.org/10.1109/CVPRW59228.2023.00420 (cit. on p. 21).

Guide to mobile robot fleet management [Accessed: 03.07.2024]. (n.d.). (Cit. on p. 11).

Hamunen, K., Kurttila, M., Miina, J., Peltola, R., & Tikkanen, J. (2019). Sustainability of nordic non-timber forest product-related businesses – a case study on bilberry. *Forest Policy and Economics*, *109*, 102002. https://doi.org/https://doi.org/10.1016/j.forpol.2019.102002 (cit. on p. 3).

Idres, M., Mustapha, O., & Okasha, M. (2017). Quadrotor trajectory tracking using pid cascade control. *IOP Conference Series: Materials Science and Engineering*, *270*, 012010. https://doi.org/10.1088/1757-899X/270/1/012010 (cit. on p. 9).

Kiselov, N. (2021). Drone control via gestures using mediapipe hands. (Cit. on p. 26).

Lin, T., Goyal, P., Girshick, R. B., He, K., & Dollár, P. (2017). Focal loss for dense object detection. *CoRR*, *abs/1708.02002* (cit. on p. 22).

Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S., Fu, C.-Y., & Berg, A. C. (2016). Ssd: Single shot multibox detector. In *Lecture notes in computer science* (pp. 21–37). Springer International Publishing. https://doi.org/10.1007/978-3-319-46448-0_2 (cit. on pp. 18, 21).

Merz, M., Pedro, D., Skliros, V., Bergenhem, C., Himanka, M., Houge, T., Matos-Carvalho, J. P., Lundkvist, H., Cürüklü, B., Hamrén, R., Ameri, A. E., Ahlberg, C., & Johansen, G. (2022). Autonomous uas-based agriculture applications: General overview and relevant european case studies. *Drones*, *6*(5). https://doi.org/10.3390/drones6050128 (cit. on p. 3).

Mesuga, R., & Bayanay, B. (2022, May). *A deep transfer learning approach on identifying glitch wave-form in gravitational wave data*. https://doi.org/10.36227/techrxiv.19687590.v2 (cit. on p. 47).

Perez-Segui, R., Arias-Perez, P., Melero-Deza, J., Fernandez-Cortizas, M., Perez-Saura, D., & Campoy, P. (2023). Bridging the gap between simulation and real autonomous uav flights in industrial applications. *Aerospace*, *10*(9). https://doi.org/10.3390/aerospace10090814 (cit. on p. 3).

PX4-Development-Team. (2024). Px4 autopilot [Accessed: 01.07.2024]. (Cit. on pp. 9–11).

Ramirez-Atencia, C., & Camacho, D. (2018). Extending qgroundcontrol for automated mission planning of uavs. *Sensors*, *18*(7), 2339. https://doi.org/10.3390/s18072339 (cit. on p. 7).

Rinaldi, M., & Primatesta, S. (2024). Comprehensive task optimization architecture for urban uav-based intelligent transportation system. *Drones*, *8*(9). https://doi.org/10.3390/drones8090473 (cit. on p. 25).

Saghaei, H. (2016a). Design and implementation of a fleet management system using novel gps/glonass tracker and web-based software. (Cit. on pp. 25, 26).

Saghaei, H. (2016b). Design and implementation of a fleet management system using novel gps/glonass tracker and web-based software (cit. on p. 11).

Saha, S. (2018). A comprehensive guide to convolutional neural networks — the eli5 way [Accessed: 06.08.2024]. (Cit. on pp. 14, 15).

Sezer, V., & Gokasan, M. (2012). A novel obstacle avoidance algorithm: "follow the gap method". *Robotics and Autonomous Systems*, *60*(9), 1123–1134. https://doi.org/https://doi.org/10.1016/j.robot.2012.05.021 (cit. on p. 27).

Signal for help [Accessed: 30.07.2024]. (2024). (Cit. on p. 40).

Simpson, A., & Sabo, C. (2016). Mav obstacle avoidance using biomimetic algorithms. https://doi.org/10.2514/6.2016-0403 (cit. on p. 27).

Tan, W., Liu, J., Chen, T., & Marquez, H. J. (2005). Robust analysis and pid tuning of cascade control systems. *Chemical Engineering Communications*, *192*(9), 1204–1220 (cit. on p. 9).

TensorFlow. (2022). Movenet pose detection [Accessed: 05.08.2024]. (Cit. on p. 21).

Ulrich, I., & Borenstein, J. (1998). Vfh+: Reliable obstacle avoidance for fast mobile robots. *Proceedings. 1998 IEEE International Conference on Robotics and Automation (Cat. No.98CH36146)*, *2*, 1572–1577 vol.2. https://doi.org/10.1109/ROBOT.1998.677362 (cit. on pp. 27, 28).

Vanneste, S., Bellekens, B., & Weyn, M. (2014). 3dvfh+: Real-time three-dimensional obstacle avoidance using an octomap. *CEUR Workshop Proceedings*, *1319* (cit. on p. 28).

Yam-Viramontes, B., & Mercado-Ravell, D. A. (2020). Implementation of a natural user interface to command a drone. *2020 International Conference on Unmanned Aircraft Systems (ICUAS)*, 1139–1144 (cit. on p. 26).

Zhang, F., Bazarevsky, V., Vakunov, A., Tkachenka, A., Sung, G., Chang, C.-L., & Grundmann, M. (2020). Mediapipe hands: On-device real-time hand tracking. (Cit. on pp. 22, 23).

Appendix A

Rubrica TFM

Imprimir formulario Limpia formulario

ESCUELA TÉCNICA SUPERIOR INGENIERÍA INDUSTRIAL VALENCIA

NORMAS DE ESTILO PARA LA PRESENTACIÓN DEL TFG Y DECLARACION DE HONESTIDAD ACADEMICA

La tipografía (tipo de letra/tamaño) y maquetación (sangrías, párrafos, listas o viñetas…) tienen un formato libre pero debe permitir una fácil lectura del documento y el alumno debe respetar ese formato en TODO el documento.

Marque las casillas que cumple y adjunte esta hoja a la entrega del documento (debe cumplir todos los puntos para ser presentado):

- ☑ Conforme al código de honestidad académica, certifico que el trabajo presentado es original y desarrollado íntegramente por mí, citando adecuadamente las fuentes externas que haya utilizado en la realización de este trabajo.
- ☑ Las páginas están numeradas en la parte inferior derecha (numeración correlativa de todos los capítulos).
- ☑ Contiene índice, asociando los temas a la numeración de página.
- ☑ Si se utilizan tablas, figuras o ecuaciones, siguen una numeración correlativa. Una serie para tabla, otra para figuras y otra para ecuaciones. (El alumno puede elegir si el título de estos elementos está encima o debajo del elemento, pero mantiene el mismo criterio en todos).
- ☑ Se cita correctamente las fuentes en el texto formato Autor (Año). Ejemplos: http://blog.apastyle.org/apastyle/2011/01/writing-in-text-citations-in-apa-style.html

  Cita directa: los sistemas..... (Pérez y Martínez, 2007; Alba, 2010)
  Cita indirecta: como afirman Pérez y Martínez (2007) los sistemas.....
  Cita con más de dos autores: (Gutiérrez y otros, 2003)

- ☑ Si se citan fuentes externas, existe sección de Bibliografía con las referencias formateadas adecuadamente (se sugiere formato APA: http://www.apastyle.org/). Ejemplos recomendados en:

  http://www.upv.es/laboluz/master/metodologia/textos/citar.pdf
  https://biblioguias.uam.es/citar/estilo_apa
  http://www.ub.edu/biblio/citae-e.htm

- ☑ Si se citan fuentes externas, existe concordancia entre las citas en texto y la lista de referenciadas.
- ☑ Sin faltas de ortografía, ni errores tipográficos.
- ☑ Figuras e imágenes de buena calidad (si no existen figuras o imágenes, marca la casilla como cumplida).
- ☑ Si hay gráficas, estas son claras y están bien etiquetadas (si no existen gráficas, marca la casilla como cumplida).
- ☑ Si hay ecuaciones, estas son claras y están bien escritas (si no existen ecuaciones, marca la casilla como cumplida).
- ☑ Uso correcto de símbolos, anagramas, denominaciones, etc. (si no existen estos elementos, marca la casilla como cumplida).

Fecha: 17.09.2024

Firma: P. Haas

Plantilla para volcar puntuaciones de las rúbricas

ESCUELA TÉCNICA SUPERIOR INGENIERÍA INDUSTRIAL VALENCIA

Título TFM: Developing enhanced Drone Operations: Enabling Gesture Recognition with Integration of a Custom Fleet Management System based on ROOSTER

Autor TFM: Patrick Haas

Evaluador: EL TUTOR

Tutor TFM: Blanes Noguera, Juan Francisco

Use las descripciones detalladas de la rúbrica para seleccionar puntuación.
Marque con un tick la casilla correspondiente.

ELABORACION TFM	Exc	Alto	Med	Insuf	Defic
P-01 Viabilidad del proyecto y/o de su trabajo de campo	■				
P-02 El alumno sabe recoger la información necesaria		■			
P-03 Comprensión de la tarea	■				
P-04 Motivación, iniciativa e independencia	■				
P-05 Organización y planificación de las tareas		■			
P-06 Innovación, creatividad y emprendimiento	■				

INFORME TFM depositado	Exc	Alto	Med	Insuf	Defic
I-01 Comunicación escrita		■			
I-02 Introducción-Objetivos	■				
I-03 Antecedentes		■			
I-04 Metodología/ Desarrollo	■				
I-05 Resultados		■			
I-06 Conclusiones		■			
I-07 Documentaci-ción	■				
I-08 Honestidad académica	■				

COMENTARIOS:

COMENTARIOS:

Firma:

JUAN FRANCISCO| BLANES| NOGUERA

Firmado digitalmente por JUAN FRANCISCO| BLANES|NOGUERA
Fecha: 2024.09.17 14:28:19 +02'00'

P. Haas

Plantilla para volcar puntuaciones de las rúbricas

ESCUELA TÉCNICA SUPERIOR INGENIERÍA INDUSTRIAL VALENCIA

Título TFM: Developing enhanced Drone Operations: Enabling Gesture Recognition with Integration of a Custom Fleet Management System based on ROOSTER

Autor TFM: Patrick Haas

Evaluador: El TUTOR

Tutor TFM: Blanes Noguera, Juan Francisco

Use las descripciones detalladas de la rúbrica para seleccionar puntuación.
Marque con un tick la casilla correspondiente.

ELABORACION TFM	Exc	Alto	Med	Insuf	Defic
P-01 Viabilidad del proyecto y/o de su trabajo de campo	X				
P-02 El alumno sabe recoger la información necesaria	X				
P-03 Comprensión de la tarea	X				
P-04 Motivación, iniciativa e independencia	X				
P-05 Organización y planificación de las tareas	X				
P-06 Innovación, creatividad y emprendimiento	X				

COMENTARIOS:

INFORME TFM depositado	Exc	Alto	Med	Insuf	Defic
I-01 Comunicación escrita		X			
I-02 Introducción-Objetivos		X			
I-03 Antecedentes		X			
I-04 Metodología/ Desarrollo		X			
I-05 Resultados	X				
I-06 Conclusiones		X			
I-07 Documentaci-ción		X			
I-08 Honestidad académica	X				

COMENTARIOS:

Firma:

JUAN FRANCISCO| BLANES| NOGUERA

Firmado digitalmente por JUAN FRANCISCO| BLANES|NOGUERA
Fecha: 2024.09.17 14:28:03 +02'00'