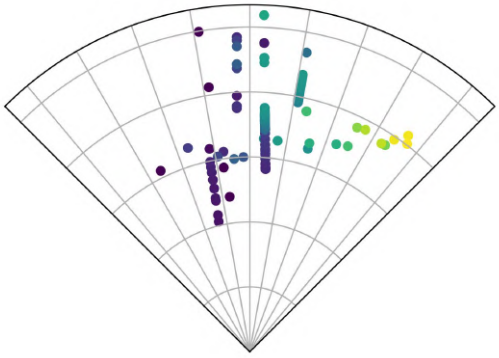
TÉCNICO LISBOA



Development of a Sense and Avoid System for Small Fixed-Wing UAV

Bruno Manuel Bento Pedro

Thesis to obtain the Master of Science Degree in

Aerospace Engineering

Supervisor: Prof. André Calado Marta

Examination Committee

Chairperson: Prof. Paulo Sérgio de Brito André
Supervisor: Prof. André Calado Marta
Member of the Committee: Prof. Alexandra Bento Moutinho

December 2024

Declaration

I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Acknowledgments

First of all, I would like to thank Prof. André Marta for being a steady source of support and insight, providing thoughtful guidance and feedback with patience and clarity throughout this work, since day one.

Further, I would like to thank my parents, my sister, and my friends, in particular, Dinis Pires and Rui Araújo, for the support over the last five years of academic journey at IST.

Lastly, a special note of gratitude to Andrew Brahim for his responsive technical assistance on the LiDAR integration with PX4.

Resumo

Dado o aumento do número de aplicações de Veículos Aéreos Não Tripulados (VANT) e consequente expansão desse mercado, sistemas de segurança de voo aprimorados precisam de ser desenvolvidos. O principal objetivo deste trabalho é desenvolver um sistema de *Sense and Avoid* (S&A) para pequenos VANTs de asa fixa. Para tal, primeiramente, elaborou-se uma revisão abrangente de literatura dos sensores e sistemas utilizados para detetar obstáculos, tanto de forma cooperativa como não cooperativa, seguida de uma revisão dos principais métodos locais e globais de planeamento de trajetória para evasão de colisões. Em seguida, propôs-se uma implementação em *hardware* composta por dois sensores ultrassónicos, dois telémetros laser e um LiDAR, integrados com um controlador de voo, um computador complementar, e outros componentes essenciais à operação de um VANT. Propôs-se, também, uma implementação em *software* completa, desde o estudo e adaptação do *software* de controlo de voo (PX4), com ênfase na manipulação e comunicação dos dados dos sensores, até ao desenvolvimento de um protótipo de software, baseado no método Histograma de Campo Vetorial, a ser executado no computador complementar, para receber os dados dos sensores, determinar as posições dos obstáculos e devolver pontos de uma trajetória de evasão de colisão. Os testes de validação mostraram que o sistema é capaz de, com base nos dados dos sensores, determinar posições dos obstáculos, transformá-las em formato de histograma polar e gerar trajetórias que representem manobras de evasão de pequenos desvios, com uma taxa de atualização de 10Hz e, portanto, capaz de atuar em tempo real.

Palavras-chave: deteção de obstáculos, evasão de colisão, Histograma de Campo Vetorial, controlador de voo, computador complementar, sensor ultrassónico, telémetro laser, LiDAR.

Abstract

Given the rising number of applications of Unmanned Aerial Vehicles (UAVs) and consequent expansion of that market, enhanced flight safety systems need to be developed. The main objective of this work is to develop a Sense and Avoid (S&A) system for small fixed-wing UAVs. To achieve this, firstly, a comprehensive literature review of the sensors and systems used to detect obstacles, cooperatively and non-cooperatively, was made, followed by a review of the main local and global path planning methods for collision avoidance. Then, a hardware implementation was proposed, consisting of two ultrasonic sensors, two laser rangefinders, and one LiDAR, integrated with a flight controller, a companion computer, and other components essential to the UAV operation. A complete software implementation was also proposed, ranging from the study and adaptation of the flight control software (PX4), with emphasis on the handle and communication of sensor data, to the development of a software prototype, based on the Vector Field Histogram (VFH) method, to be executed in the companion computer, to receive sensor data, obtain obstacle positions, and return setpoints of a collision avoidance trajectory. The validation tests have shown that the system is capable of, based on sensor data, compute obstacle positions, transform them to polar histogram format, and generate trajectory setpoints representing avoidance maneuvers of small deviations, with an update rate of 10Hz, thus real-time capable.

Keywords: obstacle detection, collision avoidance, Vector Field Histogram, flight controller, companion computer, ultrasonic sensor, laser rangefinder, LiDAR.

Contents

Acknowledgments . . . iii
Resumo . . . iv
Abstract . . . v
List of Tables . . . ix
List of Figures . . . x
Nomenclature . . . xii
Glossary . . . xvi

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Sense and Avoid Systems Overview . . . 3
1.3 Objectives and Deliverables . . . 5
1.4 Thesis Outline . . . 6

2 Obstacle Detection **7**
2.1 Cooperative Obstacle Detection . . . 8
2.1.1 Traffic Alert and Collision Avoidance System (TCAS) . . . 8
2.1.2 Automatic Dependent Surveillance-Broadcast (ADS-B) . . . 9
2.2 Non-cooperative Obstacle Detection . . . 10
2.2.1 Radio Detection and Ranging (RADAR) . . . 10
2.2.2 Laser Rangefinder . . . 11
2.2.3 Light Detection and Ranging (LIDAR) . . . 11
2.2.4 Ultrasonic Sensor . . . 12
2.2.5 Passive Sensors . . . 13
2.3 State Estimation of Obstacles . . . 15
2.3.1 Kalman Filter . . . 15
2.3.2 Markov Chain . . . 17

3 Collision Avoidance **19**
3.1 Global Path Planning . . . 20
3.1.1 A* Algorithm . . . 20
3.1.2 Rapidly-exploring Random Tree (RRT) . . . 21

3.2 Local Path Planning . . . 22
3.2.1 Geometric Methods . . . 22
3.2.2 Potential Field Methods . . . 23
3.2.3 Gap-based Methods . . . 25
3.3 Comparison of Collision Avoidance Methods . . . 29

4 Hardware Implementation **31**
4.1 Sensor Hardware . . . 31
4.1.1 Ultrasonic Sensors . . . 32
4.1.2 Laser Rangefinders . . . 33
4.1.3 LiDAR . . . 33
4.2 Flight Controller and Companion Computer . . . 34
4.2.1 Pixhawk 6X . . . 35
4.2.2 Companion Computer . . . 35
4.2.3 Pixhawk RPi CM4 Baseboard . . . 36
4.3 Electrical Layout . . . 38
4.4 Hardware Configuration . . . 40
4.4.1 Flight Controller Configuration . . . 40
4.4.2 Raspberry Pi CM4 Configuration . . . 40

5 Software Implementation **42**
5.1 Flight Control Software . . . 43
5.1.1 PX4 Stack and Architecture . . . 43
5.1.2 PX4 Software Configuration . . . 44
5.1.3 PX4 Internal Communication: uORB . . . 45
5.1.4 Drivers of Distance Sensors . . . 49
5.1.5 MAVLink Communication . . . 52
5.2 Ground Control Software . . . 58
5.3 Communication Between PX4 and Companion Computer . . . 59
5.3.1 Ethernet Connection Setup . . . 59
5.3.2 Comparison of MAVLink Interfaces for the Companion Computer . . . 61
5.3.3 MAVROS . . . 62
5.4 Obstacle Detection and Collision Avoidance Software . . . 64
5.4.1 Software Architecture . . . 65
5.4.2 Implementation of Obstacle Detection . . . 66
5.4.3 Implementation of Collision Avoidance . . . 68

6 Validation Tests **71**
6.1 Rover System Setup . . . 71
6.2 Static Vehicle and Static Obstacles . . . 72

6.3 Static Vehicle and Moving Obstacle 74
6.4 Moving Vehicle and Static Obstacles 76

7 Conclusions **78**
7.1 Achievements 78
7.2 Future Work 79

Bibliography **80**

A Pixhawk RPi CM4 Baseboard Scheme **91**

B Software Architecture of PX4 **92**

C Relevant File Directories **93**
C.1 ROS Catkin Workspace 93
C.2 PX4 v1.14.3 94

D Adaptation of MAVROS Files **95**
D.1 Adaptation of MAVROS_EXTRAS Distance Sensor Plugin 95
D.2 Adaptation of MAVROS Configuration File for PX4 96

E Custom ROS Message: RangeYaw **98**

List of Tables

3.1 Qualitative comparison of collision avoidance methods 30

4.1 Maxbotix I2CXL-MaxSonar-EZ Series ultrasonic sensor specifications [85]. 32
4.2 Lightware LW20/c laser rangefinder specifications [86]. 33
4.3 Lightware SF45/B LiDAR specifications [87]. 34
4.4 Dip Switch [90]. 37

5.1 uORB topic `distance_sensor` fields definition. 46
5.2 MAVLink message `DISTANCE_SENSOR` fields definition. 54
5.3 Definition of MAVLink enum message `MAV_DISTANCE_SENSOR` 55
5.4 Comparison of MAVLink Interfaces 61
5.5 Message flow between MAVLink and MAVROS. 64
5.6 MAVROS plugins that handle each MAVLink message. 64
5.7 Considered detection range values per sensor type. 67

6.1 Obstacle detection and collision avoidance software parameters. 72

List of Figures

1.1 Global UAV market size studies. . . . 2
1.2 TEKEVER AR4 [17]. . . . 3
1.3 Timeline of processes of S&A systems in UAVs [22]. . . . 4

2.1 Diagram of obstacle detection methods. . . . 7
2.2 Processes of TCAS [30]. . . . 8
2.3 Operation of ADS-B system [25]. . . . 9
2.4 Radar US-D1 (Source: Ainstein). . . . 10
2.5 Laser rangefinfer LW20/C (Source: LightWare). . . . 11
2.6 LIDAR SF45/B (Source: LightWare). . . . 12
2.7 3D LIDAR point cloud data representation shaded by height (Source: SureStar). . . . 12
2.8 Ultrasonic sensor MB1242 (Source: MaxBotix). . . . 13
2.9 Monocular camera Lumenera Lt-C1950 (Source: Lumenera). . . . 14
2.10 Stereo-vision camera Intel RealSense D455 (Source: Intel). . . . 14
2.11 Infrared sensor Sharp GP2Y0A710K0F (Source: Sharp). . . . 14

3.1 Diagram of collision avoidance methods. . . . 19
3.2 2D collision cone representation [69]. . . . 22
3.3 2D velocity obstacle representation [69]. . . . 22
3.4 2D APF representation with one waypoint (goal) and one obstacle [27]. . . . 24
3.5 Safety zones around an UAV [28]. . . . 24
3.6 Vector Field Histogram method [76]. . . . 26
3.7 2D primary polar histogram of 3DVFH+ [79]. . . . 27
3.8 VFH grid with reachable set of cells for fixed-wing UAV [81]. . . . 28
3.9 Nearness Diagram method [82]. . . . 28

4.1 Holybro Pixhawk 6X with Raspberry Pi CM4 baseboard [90]. . . . 36
4.2 Hardware electrical diagram. . . . 39
4.3 Holybro PM03D power module [96]. . . . 40

5.1 Diagram of S&A system software implementation. . . . 42
5.2 GUI of PX4 firmware configuration menu. . . . 44

5.3 Diagram of the high-level access to PX4 system through MAVLink console. 45
5.4 QGroundControl MAVLink Console printout example of `distance_sensor` uORB topic with 2 instances (one ultrasonic sensor and one laser rangefinder). 47
5.5 Over-the-wire format of a MAVLink 2 packet [115]. 53
5.6 Wireshark printouts of five `DISTANCE_SENSOR` MAVLink messages, each corresponding to one of the obstacle detection sensors. 56
5.7 Main interface of QGroundControl. 58
5.8 Vehicle setup tool of QGroundControl. 59
5.9 MAVLink Inspector of QGroundControl inspecting the `DISTANCE_SENSOR` message. 59
5.10 Software architecture of the obstacle detection and collision avoidance implementation. . 66
5.11 Flowchart of the algorithm for creation and continuous update of the polar histogram. . . . 69
5.12 Flowchart of the algorithm to generate avoidance setpoints. 70

6.1 Rover setup. 71
6.2 Static vehicle and static obstacles. 73
6.3 Polar histograms for $\gamma = 1$, $\gamma = 2$ and $\gamma = 3$. 74
6.4 Raw and filtered obstacle detection sensors data. 75
6.5 S&A system outputs as a function of time for static vehicle test. 76
6.6 S&A system outputs as a function of time for moving vehicle test. 77

A.1 Pixhawk RPi CM4 baseboard scheme [90]. 91

B.1 High-level software architecture of PX4 (Adapted from [105]). 92

C.1 Directory tree of relevant files of ROS Catkin Workspace. 93
C.2 Directory tree of relevant files of PX4 v1.14.3. 94

Nomenclature

Greek symbols

α	Width of polar histogram sectors
α_{PF}	Gain coefficient
β	Sensor orientation
γ	Polar histogram bin neighbours parameter
δ	Minimum distance to an obstacle
θ	Angle between desired direction of motion and obstacle
τ	Threshold
ν	Observation noise vector
Φ	State transition matrix
φ	Azimuth polar coordinate
ψ	Heading angle
ω	Process noise vector

Roman symbols

$\hat{\mathbf{A}}$	Geometric point A
a	Weight value
$\beta_{i,j}$	Direction of obstacle vector in a cell
$\mathbf{B}$	Control-input matrix
$\hat{\mathbf{B}}$	Geometric point B
C	Cartesian histogram grid
C^*	Active region of the Cartesian histogram grid
$c^*_{i,j}$	Certainty value of an active cell

$c_{i,j}$	Certainty value of a cell
CC	Collision cone
$\hat{d}$	Filtered sensor range measurement
d	Sensor range measurement
d_o	Distance between the UAV and the obstacle
$d_{i,j}$	Distance between vehicle and cell
$\mathbf{F}$	Force
$f(n)$	Cost function of node n
$g(n)$	Movement cost from initial node to current$_node$
$\mathbf{H}$	Measurement sensitivity matrix
H	Polar histogram
h'_k	Smoothed obstacle density in sector k of polar histogram
$h(n)$	Heuristic function reflecting distance from node to target
H^b	Binary polar histogram
H^m	Masked polar histogram
h_k	Obstacle density in sector k of polar histogram
$\hat{\mathbf{i}}, \hat{\mathbf{j}}, \hat{\mathbf{k}}$	Unit vectors
$\mathbf{K}$	Kalman gain matrix
L	List of obstacle points
l	Tuning parameter of smoothed polar histogram
$m_{i,j}$	Magnitude of obstacle vector in a cell
$\mathbf{N}$	Transition number
n	Number of sectors of a polar histogram
$\mathbf{P}$	Markov chain probability matrix
$\mathbf{P}$	State estimation covariance matrix
PND	Value of Nearness Diagram from central point
$\mathbf{Q}$	Accumulated uncertainty matrix
$\mathbf{R}$	Measurement error covariance matrix

r	Autocorrelation coefficient
r	Radial polar coordinate
R_a	Action radius
R_c	Collision radius
$\mathbf{S}$	Action vector
$\mathbf{s}_{dir}$	Swirl movement
t	Time instant
$\mathbf{u}$	Input vector
$\mathbf{v}$	Velocity vector is Cartesian coordinates
$\mathbf{WP}$	Waypoint
$\mathbf{X}$	Predict of next position matrix
$\hat{\mathbf{x}}$	State estimation vector
x, y	Cartesian components of 2D position
$\mathbf{z}$	Measurement vector

Subscripts

A	Geometric point A
att	Attractive
B	Geometric point B
$body$	Body frame
$close$	Closer waypoint in the path
$cutoff$	Cut-off
ENU	East-North-Up frame
$high$	High
k	Current discrete measurement
$k+1$	Next discrete measurement
$k-1$	Previous discrete measurement
low	Low
m	m-th previous detection

max Maximum

NED North-East-Down frame

$next$ Next waypoint in the path

obs Obstacle

rep Repulsive

$sens$ Sensor

$total$ Total

UAV Unmanned Aerial Vehicle

Superscripts

+ *A posteriori* estimate

- *A priori* estimate

-1 Inverse

m m-th order

T Transpose

Glossary

ACO	Ant Colony Optimization
ADS-B	Automatic Dependent Surveillance-Broadcast
APF	Artificial Potential Fields
API	Application Programming Interface
ATC	Air Traffic Control
Auto-GCAS	Automatic Ground Collision Avoidance System
BEC	Battery Eliminator Circuit
CAGR	Compounded Annual Growth Rate
CAN	Controller Available Network
CBF	Control Barrier Functions
CL-RTT	Closed-Loop Rapidly-exploring Random Tree
CM4	Computer Module 4
DAPF	Dynamic Artificial Potential Field
DHCP	Dynamic Host Configuration Protocol
EGPWS	Enhanced Ground Proximity Warning System
EKF	Extended Kalman Filter
ENU	East-North-Up
EO	Electro-Optical
ESC	Electronic Speed Controller
FAA	US Federal Aviation Administration
FC	Flight Controller
FGA	Fast Geometric Avoidance
FMU	Flight Management Unit
FOV	Field of View
GCS	Ground Control Station
GPIO	General-Purpose Input/Output
GPWS	Ground Proximity Warning System
GUI	Graphical User Interface
HUD	Head-Up Display
I2C	Inter-Integrated Circuit

IO	Input/Output
IR	Infrared
LIDAR	Light Detection and Ranging
MSD	Mass Storage Device
MSL	Mean Sea Level
MTOW	Maximum Take-Off Weight
MUX	Multiplexer
ND	Nearness Diagram
NED	North-East-Down
NSH	NuttShell
OS	Operating System
PCIe	Peripheral Component Interconnect Express
PHY	Physical layer
PWM	Pulse-width Modulation
RADAR	Radio Detection and Ranging
RCS	Radar Cross Section
ROS	Robotic Operating System
RPAS	Remotely Piloted Aerial System
RTOS	Real-Time Operating System
RTT	Rapidly exploring Random Tree
S&A	Sense and Avoid
SITL	Software-In-The-Loop
SNR	Signal-to-Noise Ratio
SoM	System on Module
SPI	Serial Peripheral Interface
TAWS	Terrain Avoidance and Warning System
TCAS	Traffic Collision Avoidance System
UART	Universal Asynchronous Receiver-Transmitter
UAS	Unmanned Aerial System
UAV	Unmanned Aerial Vehicle
UGV	Unmanned Ground Vehicle
USB	Universal Serial Bus
VFH	Vector Field Histogram
VTOL	Vertical Take-Off and Landing

Chapter 1

Introduction

This introductory chapter presents the motivation for this thesis, based on the applications of Unmanned Aerial Vehicles (UAV) and the current and foreseen size of the UAV market, as well as an overview of the aircraft Sense and Avoid (S&A) systems, with particular focus on the processes of obstacle sensing, detection and collision avoidance in UAVs. The objectives and deliverables of the thesis are also addressed, together with a summary of its structure.

1.1 Motivation

A UAV is an aircraft operated without a pilot on board and known as a drone, a Remotely Piloted Aerial System (RPAS), or an Unmanned Aerial System (UAS). Its flight is automatically controlled by on-board computers or a pilot on the ground through remote control [1].

Throughout the history, the development of UAVs was driven by military applications, with civilian applications following once the testing had been accomplished in the military arena [2]. Nowadays, UAVs are present in several civil and military applications, such as:

- **Surveillance:** Real-time location, identification, and monitoring of targets, whether for smart traffic monitoring, including detecting the speed of each vehicle and other traffic violations [3], for borders control, including tracking and recognizing illegal activities, unwanted infiltrations, and unauthorized trespassers [4], or for wildfire detection, applying image processing techniques to onboard visual and infrared sensors data [5];
- **Inspection:** Data gathering for inspection of difficult to access areas, such as bridges [6], wind turbine blades [7], penstocks [8], surfaces of high-rise buildings [9], or nuclear waste storage drums [10];
- **Agriculture:** Crops monitoring in Precision Agriculture, namely, weed mapping and management, vegetation growth monitoring and yield estimation, health monitoring and diseases detection, irrigation management, and corps spraying [11];

- **Transportation and Logistics:** Carrying small loads, UAVs can be incorporated with Unmanned Ground Vehicles (UGVs) for a fully automated inventory management system. They can, also, offer low-cost and faster delivery options [12];
- **Photography and Video Recording:** Low-cost aerial photo and video capture for media and entertainment through mounted or integrated cameras.

Most of these applications include flying at low altitudes, where various obstacles can be found, such as buildings, power lines, trees, and other UAVs. Equipping the UAVs with the ability to sense its surroundings, detect obstacles, and avoid collisions is an important step to reduce incidents and accidents that could result in damage, harm or injury, during the flight operations.

Furthermore, the size of the global UAV market has been increasing over the years and, until 2030, it is expected to maintain its expansion, according to recent studies from market research companies (Fig. 1.1). Some of the most conservative studies estimate a size value of US$17.31 billion in 2024, expected to reach US$ 32.95 billion in 2029, with a Compounded Annual Growth Rate (CAGR) of 13.74% [13], or a size of US$ 15.21 billion in 2023, expected to reach US$ 47.67 billion in 2031, with CAGR of 15.35 % [14]. More ambitious studies point to US$ 31.70 billion in 2023, to reach US$ 91.93 billion in 2030, with CAGR of 16.50% [15], or US$ 32.17 billion in 2022, to reach US$ 82.55 billion in 2030, with CAGR of 12.50% [16].

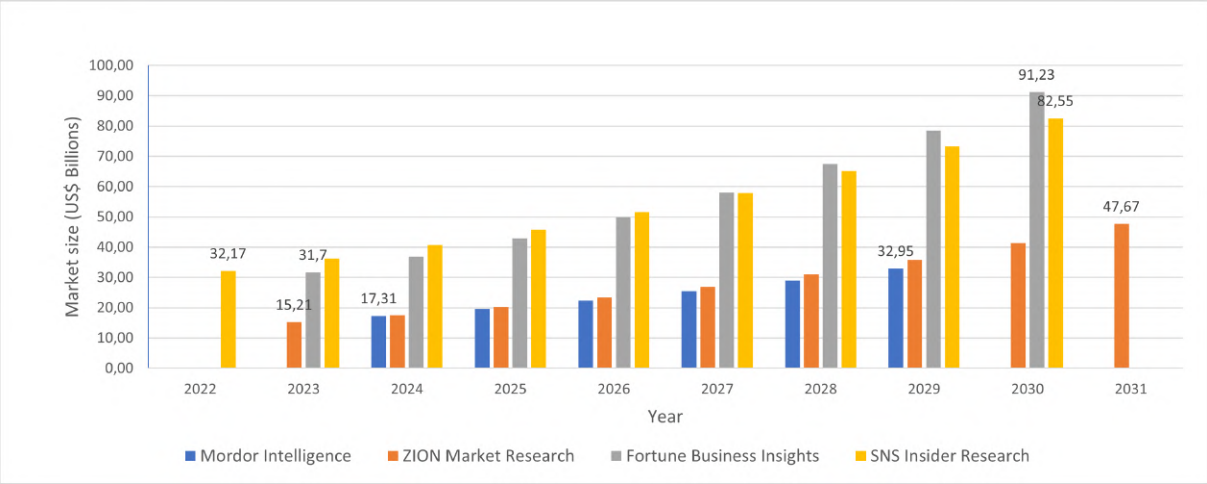


Figure 1.1: Global UAV market size (US$), in the forecasting period of 2022-2031, according to market studies from Mordor Intelligence [13], ZION Market Research [14], Fortune Business Insights [15], and SNS Insider Research [16].

The foreseen growth of the UAV market reflects an increase in the number of missions involving this type of vehicle, which leads to an urgency to develop safety systems that are cheap, in relation to the overall cost of the vehicle, allowing a massive implementation. In this sense, this thesis fits in the development and implementation of a safety enhancement system, addressing the problems of obstacle detection and collision avoidance, for small fixed-wing UAVs, characterized by a maximum takeoff weight (MTOW) under 25kg, a range under 10km, an endurance under 2h and flight altitude under 120m. An example of UAV with these specifications is the Tekever AR4 (Fig. 1.2), with MTOW of 4kg, wingspan of 2.1m, endurance of 2h and cruise speed of 15m/s [17].

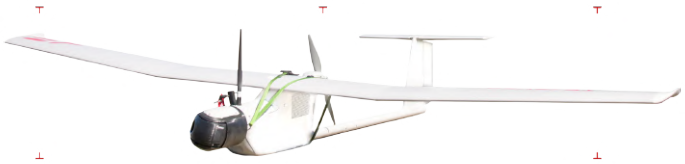


Figure 1.2: TEKEVER AR4 [17].

1.2 Sense and Avoid Systems Overview

From the moment people began to build and fly aerial vehicles, detection and avoidance of obstacles during flight has become a concern. In this regard, for manned aircraft, the US Federal Aviation Administration (FAA) introduces, in Title 14 CFR Part 91.113 (b) [18], the ”see and avoid” method that makes the pilot of the aircraft responsible to be vigilant, see the obstacles, make a judgment regarding the need for any trajectory change, and perform the necessary maneuvers to avoid collisions, when weather conditions permit. For unmanned aircraft, or when the weather conditions do not favor the pilot's vision, some efforts have been conducted to develop Sense and Avoid (S&A) systems, in the last decades, that make use of sensor data and on-board or off-board computers to automatically detect obstacles in real time and avoid collisions.

Some of the systems currently tested and implemented in civil and military manned aircraft, that contribute to the sense and detection of obstacles and collision avoidance, include:

- **Ground Proximity Warning System (GPWS):** Based on radio altimeter data, provides pilots with audio and visual signalling in case when the aircraft approaches dangerously near to the ground, when the descent speed is too high, the angle of descent is too large or in case of dangerous manoeuvres near the ground [19]. It was improved to the Enhanced Ground Proximity Warning System (EGPWS), also referred as Terrain Avoidance and Warning System (TAWS), which can predict the entrance into the dangerous area, by comparing data on the aircraft position, speed and vertical speed, radio and barometer altitude, and data obtained from navigation and inertia systems, with the data from a database about the terrain configuration [19];
- **Automatic Ground Collision Avoidance System (Auto-GCAS):** Developed for fast military aircraft, not only warns the pilot of ground proximity but also takes control and flies the aircraft out of danger before returning control to the pilot. Operates by projecting a predicted recovery trajectory over a virtual terrain map [20];
- **Traffic Collision Avoidance System (TCAS):** Generates alerts for the pilot for potential collision threats related to transponder-equipped aircraft. In addition to traffic advisories (TA), the TCAS II can provide resolution advisories (RA), supporting the pilot in the conflict resolution. The suggested collision avoidance maneuver is generated in a cooperative manner with the other aircraft [21].

For UAVs, the paradigm of S&A systems can be considered as a temporal sequence of the following processes: sense and detect obstacles, detect conflicts and make decisions, re-plan the path, and follow

the new path. Each of these processes needs a specific time interval to be executed (T_{det}, T_{dm}, T_{pp}, T_{pf}, respectively), as presented in Fig. 1.3 [22].

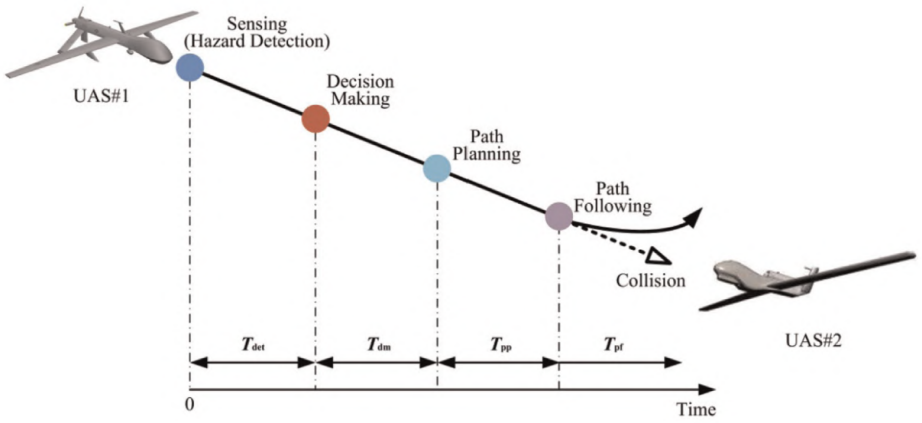


Figure 1.3: Timeline of processes of S&A systems in UAVs [22].

The sensing process is responsible for the continuous search of hazards in the surroundings of the UAV. It can be cooperative, when the hazard is another vehicle equipped with a similar S&A system with communication capabilities, or non-cooperative, when the hazard do not communicate and can only be detected by sensor hardware. The data containing the position of the potential obstacles is, then, sent to a computer, which can be on-board or off-board the UAV, where it is processed to gain confidence that the detection is valid and track the motion of the obstacles. Generally, estimation filters like Kalman filter or particle filters are used to estimate the trajectory of the obstacles, based on the information provided by the sensors [23]. This computer can combine information of the UAV motion with data of the obstacles and evaluate the existence of a conflict that could result in a collision. In case various obstacles are detected, resulting in more that one conflict, a decision should be made to prioritize the conflicts that configure a higher risk of collision, whether because the obstacles are closer to the UAV, have a larger size, or higher velocity.

Then, the system is required to make a decision regarding the avoidance maneuver, which should be feasible, considering the UAV dynamics, and minimize the deviation from the planned flight trajectory. There are several approaches to solve the problem of path re-planning for collision avoidance, including algorithms based on geometric relations between the UAV and the obstacle, potential field methods, optimized trajectory search methods, and others [24]. The new path plan that results from these algorithms can be a set of new waypoints that should be followed to execute the maneuver. Whether the computer that performs the previous tasks is on-board or off-board, the new waypoints have to be communicated to the flight controller, for it to compute inputs for the control surfaces of the UAV and allow the tracking of the new path. When the conflict is resolved, the UAV S&A system continues to sense potential obstacles and evaluate conflicts, repeating the previous processes.

In case of autonomous UAVs, there can, also, be a stage of pre-flight path planning, which consists of the definition, prior to the flight, of a set of waypoints to be followed during the flight operation. It makes it possible to take into account the presence of known obstacles, and avoid them from the outset, as

well as other important factors, such as path length, flight altitude, danger zones, energy consumption, threats, and flight time, resulting in an optimal path. Since this process is not done during flight, more computationally demanding algorithms can be used, such as the Mixed Integer Linear Programming or Genetic Algorithms [25].

1.3 Objectives and Deliverables

This work is part of a comprehensive obstacle detection and collision avoidance system for small UAVs, which represents a two-stage Sense and Avoid problem.

It is preceded by three theses, focused on the sensing, where virtual models of sensors have been developed and the optimal obstacle detection configurations have been found. In [25], the laser rangefinder, Radio Detection and Ranging (RADAR), and Light Detection and Ranging (LIDAR) sensors were studied and modeled, as well as their ability to detect obstacles using Kalman filters. It was also addressed the fusion of data from different redundant sensors, some collision avoidance algorithms, parametric studies on the impact of the range, field of view, and speed of the vehicle in the avoidance of obstacles, and optimization studies to determine the best configuration of sensors to be used. In [26], the previous work was extended with the model of an ultrasonic sensor, new parametric studies and experiments regarding the laser rangefinder and ultrasonic sensors, a proposed hardware and software implementation, and validation tests of the complete system using a rover robot. In [27], the modeling of all the previous sensors was revisited, optimization studies were conducted to evaluate new sensing configurations, a hardware and software implementation to test each sensor was proposed, the sensors were subject to bench tests, and the complete system was subject to rover tests.

In addition, one previous thesis [28] focused on the avoidance problem, having addressed and simulated pre-flight path planning algorithms, such as A* and Ant Colony Optimization, and the real-time path planning algorithm of Potential Fields, to avoid collisions in RPAS.

As sequence of the previous works, this thesis focuses on the problem of obstacle detection and collision avoidance for small fixed-wing UAVs. Its main goals are to:

1. Gather information, present in the literature, of the sensors and systems that can be used for obstacle detection, as well as the path planning algorithms used for collision avoidance, in small fixed-wing UAVs;

2. Present a solution for hardware implementation of an obstacle detection and collision avoidance system using the optimal multi-sensor configuration obtained in the previous works, a flight controller, a Ground Control Station (GCS), and a companion computer.

3. Present a solution for software implementation of an obstacle detection and collision avoidance system:

    3.1 Explore and adapt the flight control software, PX4, to receive and process data from the obstacle detection sensors;

3.2 Develop a software, to be executed on a companion computer, to communicate with the flight control software and receive data from the obstacle detection sensors, process them to obtain the position of the obstacles, run a collision avoidance algorithm, suitable for fixed-wing UAVs, determining a new set of trajectory setpoints, and send them back to the flight controller;

4. Bench test the sub-systems and validate the proposed implementation in a rover robot.

The fulfillment of the previous objectives aims to result in the deliver of a comprehensive report of the solutions present in the literature for obstacle detection and collision avoidance algorithms for UAVs, and a S&A system as a hardware/software solution, ready to be assembled in a small fixed-wing UAV and enhance its safety regarding active collision avoidance strategies.

1.4 Thesis Outline

This thesis is organized in chapters as follows:

- **Chapter 1** provides a motivation for the work, an overview of the topic, the objectives and deliverables, and the outline of the document.
- **Chapter 2** presents a review of the sensors and systems used in cooperative and non-cooperative detection of obstacles around an UAV, as well as methods to estimate their state;
- **Chapter 3** presents a review of some methods for collision avoidance in UAVs, regarding both global and local path planning approaches, ending with a qualitative comparison of the methods reviewed;
- **Chapter 4** presents a hardware implementation proposal for a complete obstacle detection and collision avoidance system for fixed-wing UAVs, including sensors for obstacle detection, a flight controller, a companion computer, and other auxiliary hardware;
- **Chapter 5** presents a software implementation proposal for the obstacle detection and collision avoidance system, regarding the flight control software (including an overview of its architecture, the necessary configurations, the messaging protocols for internal and external communication, and the drivers to handle the obstacle detection sensors), the ground control software, the communication between the flight controller and the companion computer (including a comparison of the tools available for this purpose and the adaptations applied to the chosen one), and, finally, the development of an obstacle detection and collision avoidance software based on the Vector Field Histogram (VFH) method;
- **Chapter 6** presents the critical validation tests of the system implemented, focusing both on the obstacle sensing capability and on a preliminary collision avoidance strategy.
- **Chapter 7** presents the conclusions taken from the results, the achievements of the work, and some notes regarding future work on this topic.

Chapter 2

Obstacle Detection

The sensing and detection of obstacles plays an essential role in a collision avoidance system for UAVs. Thus, in this chapter, some of the sensors and systems that can be used to detect obstacles around the UAV, during flight operation, are reviewed, distinguishing between cooperative and non-cooperative detection, as shown in the diagram of Fig. 2.1. Cooperative detection requires information exchange between the UAV and the obstacle (usually another aircraft), whilst non-cooperative detection is, generally, independent of the type of obstacle. Finally, it is addressed the state estimation of the obstacles using the data acquired from the sensors.

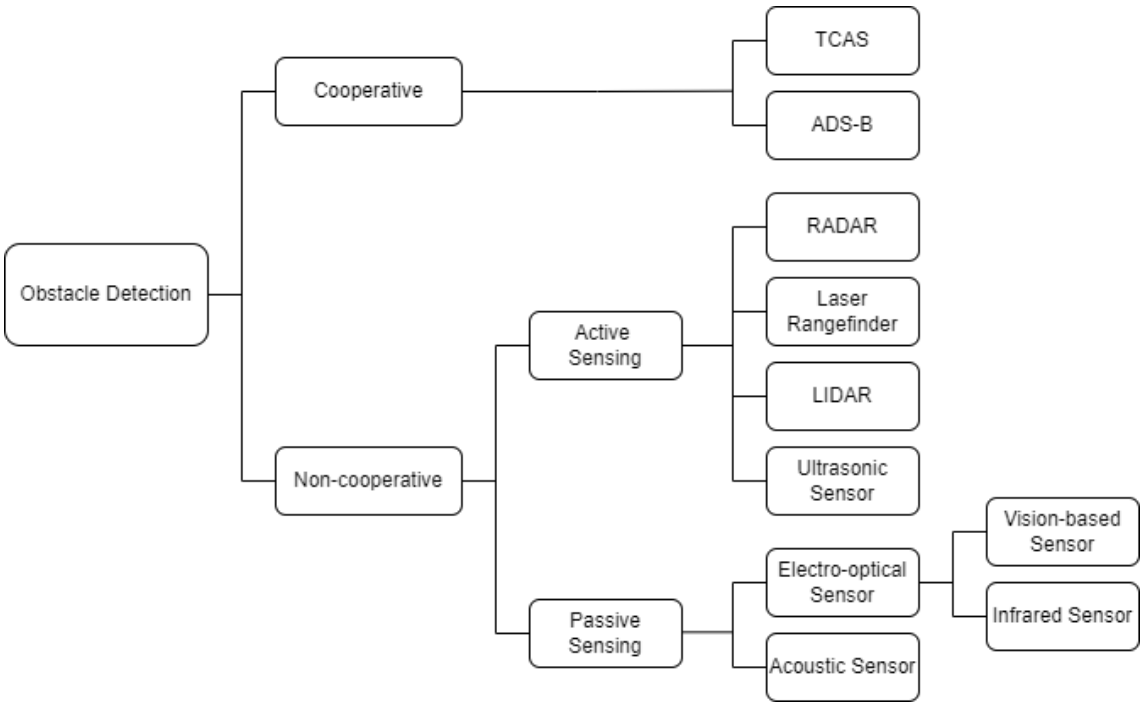


Figure 2.1: Diagram of obstacle detection methods.

2.1 Cooperative Obstacle Detection

The cooperative detection of obstacles is characterized by the requirement that the obstacle to be avoided be an aerial vehicle which is also equipped with a cooperative system, capable of exchanging information of its motion, that can be used to predict a collision, such as position, velocity and heading.

Among the current solutions of this type are the TCAS and ADS-B, widely used in manned aircraft. In the following sections, these systems are addressed, together with some of the approaches that have been studied to used them for UAVs.

2.1.1 Traffic Alert and Collision Avoidance System (TCAS)

TCAS is a family of airborne devices that are designed to reduce the risk of mid-air collisions between aircraft equipped with operating transponders [29]. Its mode of operations can be organised in several modules, as presented in Fig. 2.2. First, the surveillance of air traffic is based on air-to-air interrogations broadcast, once per second, from antennae on the TCAS aircraft to transponders on nearby intruder aircraft, which may reply to the interrogations with information that allows to compute its range, bearing and altitude. A linear extrapolation is performed to project trajectory of the intruder and its know information is displayed to the pilots. Then, an algorithm is used to decide if there is a threat and, if a collision is predicted to occur within the next 20 to 48 seconds, a traffic advisory (TA) is issued in the cockpit. From TCAS II, 15 to 35 seconds before the predicted collision, a resolution advisory (RA) is also issued to the pilots with a vertical command maneuver, coordinated with both aircraft [30].

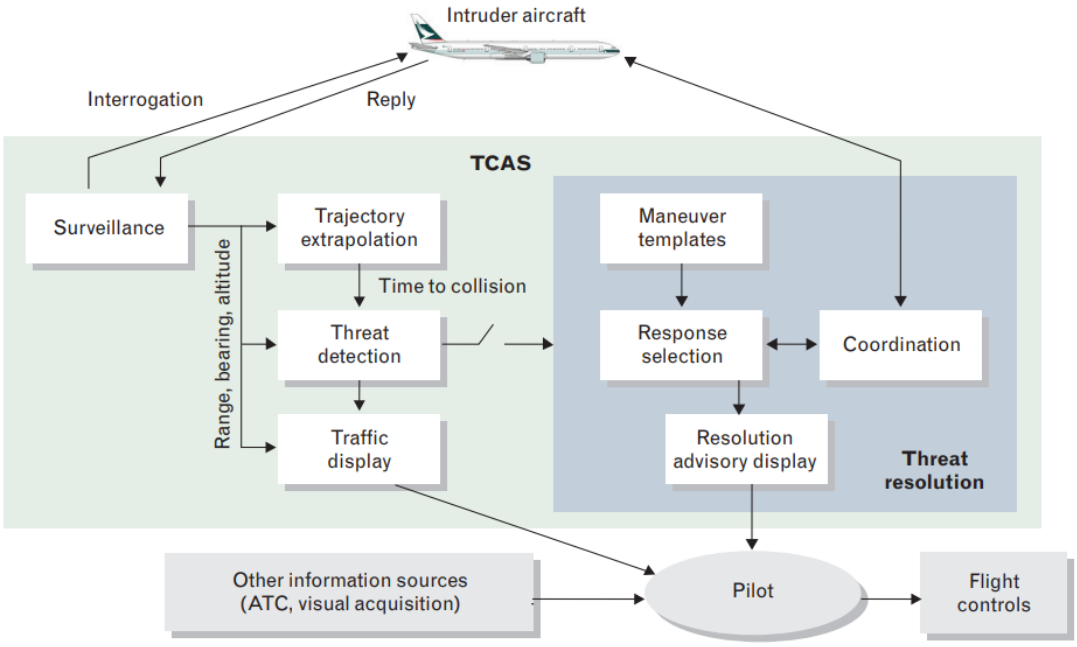


Figure 2.2: Processes of TCAS [30].

In the last two decades, the expansion of TCAS to UAVs have been studied. A safety evaluation process necessary to certify the integration of TCAS in UAVs, among other systems, is described in [31],

based on a statistically-valid estimate of collision avoidance performance, developed through a combination of airspace encounter modeling, fast time simulation, and system failure and event sensitivity analysis.

The implementation of TCAS II in UAVs is approached in [32], more specifically, its components, aural and visual annunciation, advisories, modes, functions, and interfaces, as well as the interface with a flight controller computer. It is concluded that the onboard directional antenna of TCAS II does not provide a precise directional information, and that the TCAS II is not assumed to be installed alone, but used as supplementary with other device which provides a precise direction. In [33], the implementation is further developed to a named Unmanned Aerial Collision Avoidance System (UCAS), that joins data from TCAS, Air Traffic Control (ATC), ground control, and, also, sensors of non-cooperative targets.

It is important to note that the TCAS alone cannot be implemented in autonomous UAVs, since it only provides advisories to the pilots and not path corrections that can be directly inputted to the flight controller.

2.1.2 Automatic Dependent Surveillance-Broadcast (ADS-B)

ADS-B is an air traffic surveillance system that enhances safety by making an aircraft visible, real-time, to Air Traffic Control (ATC), ground stations and other appropriately equipped ADS-B aircraft by broadcasting a Global Navigation Satellite System (GNSS) position, velocity and status periodically (Fig. 2.3). It is meant to replace the traditional Secondary Surveillance Radar systems [34].

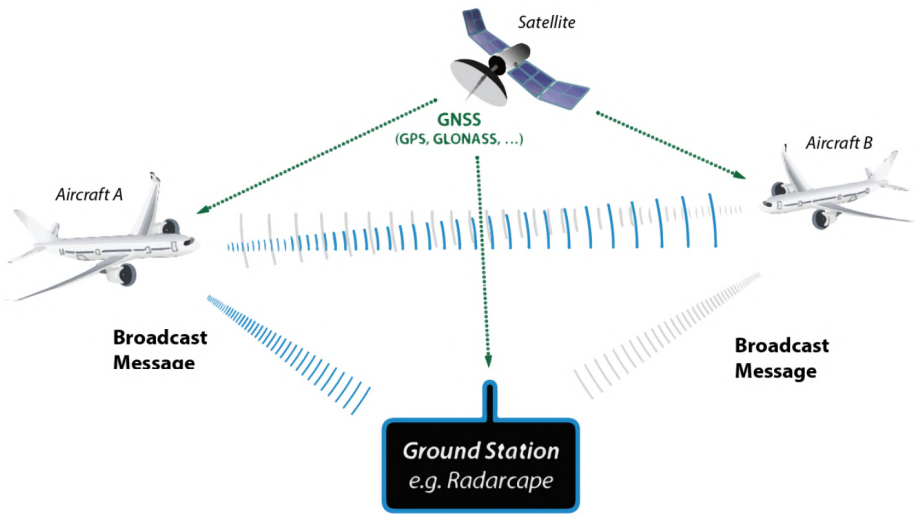


Figure 2.3: Operation of ADS-B system [25].

A cooperative S&A system for UAVs, that uses a commercial ADS-B receiver to receive data containing the latitude, longitude, altitude, heading, and horizontal and vertical velocities of an intruder aircraft, and performs a collision avoidance using a Closed-loop Rapidly-exploring Random Tree (CL-RTT) path planning algorithm, is developed and validated by Software-In-The-Loop (SITL) in [35]. In [34], the design of an airborne lightweight, small size and low power consumption device based on ADS-B, specific for small size UAVs, is approached, as part of a collision avoidance system. This device is equipped with

ADS-B Out and ADS-B In to broadcast and receive ADS-B data to and from other aircraft, respectively, together with signal processing units and a microcontroller unit, where the ADS-B data are processed and the collision avoidance algorithm is executed.

2.2 Non-cooperative Obstacle Detection

The non-cooperative detection of obstacles is not dependent on the type or action of the obstacles. This way, on-board sensors must be used, which can be active, when the data is retrieved from the reflection of a signal, such as the RADAR, LIDAR, Laser Rangefinder, and Ultrasonic Sensor, or passive, when it relies on external signals, such as electro-optical sensors and acoustic sensors. The following sections present a review of each of these sensors and their usage in UAV S&A systems.

2.2.1 Radio Detection and Ranging (RADAR)

A RADAR (Fig. 2.4) is an electromagnetic sensor used for detection of objects, whose operating principle is based on the production and emission of electromagnetic waves in the surrounding environment and reception of the echoed waves that resulted from their interaction with objects. The distance to the obstacle can be determined by measuring the time lapse between the emission of the signal and the reception of its reflection. The radial velocity of a moving obstacle can be determined by measuring the frequency difference between the emitted and received signal, i.e, the Doppler shift [36].

Figure 2.4: Radar US-D1 (Source: Ainstein).

RADAR sensors are considered accurate and precise in the detection of obstacles and can be used in adverse weather and low light conditions. Nevertheless, they have some drawbacks, such as the possible production of false readings in cluttered environments due to multiple path reflections, limited range resolution, and higher payload weight and higher power requirements when compared to other sensing systems [36].

In [37], a small-sized RADAR sensor is conceptually designed to be implemented in a collision avoidance system for UAVs, considering requirements of real-time measurement of relative range, range-rate, and bearing in azimuth or elevation, operational environment, payload constraints (the most critical requirement), and air safety regulations. A study of the probability of detection with range is performed, considering dependencies on the Signal-to-Noise Ratio (SNR) and targets RADAR Cross

Section (RCS), resulting in more than 90% of successful detection at the given required range. The performance of a collision avoidance maneuvering is also simulated based on various RADAR range and range-rate data in four different flight scenarios, such that the simulation results show that more than 85 % of probability of collision avoidance can be achieved within the requirements.

A low cost RADAR static solution, in X-band (10 GHz), based on the coherent multiple-input multiple-output (MIMO) principles for S&A in UAVs is presented in [38].

2.2.2 Laser Rangefinder

A laser rangefinder (Fig. 2.5) is a device that uses a light beam to detect the presence of an object and compute the distance to it, similarly to the RADAR, by measuring the time lapse between the emission of an electromagnetic wave, which, in this case, is a light pulse, and the reception of the wave that resulted from its reflection in the object.

Figure 2.5: Laser rangefinfer LW20/C (Source: LightWare).

Laser rangefinders are potentially more accurate than RADARs and ultrasonic sensors due to the usage of electromagnetic waves with shorter wavelength, allowing for higher spatial resolution. Also, they are lighter than LIDARs, and work in poor lighting conditions, unlike visual sensors [39]. On the other hand, working with a fixed light beam represents a limitation for scanning obstacles in an environment. One way to overcame this is with the inclusion of a servo motor to rotate the laser emitter, and get a wider angular scanning range, as in some LIDAR sensors.

Concerns regarding eye-safety and adverse weather propagation have been reducing the further development of solid state lasers and various semiconductor lasers, in favour of CO_2 lasers [40].

A system employing several body-fixed laser rangefinders as primary sensors for obstacle detection and collision avoidance in quadcopter UAVs is presented in [39], where a least squares linear regression was developed to filter the sensor measurements and identify the obstacles. The collision avoidance system was completed with a Batch Informed Trees algorithm for path planning, and its performance was evaluated through numerical simulations, whose results demonstrate the feasibility of the system.

2.2.3 Light Detection and Ranging (LIDAR)

The working principle of the LIDAR sensor (Fig. 2.6) is the same as the laser rangefinder, regarding the measurement of the time lapse between emission and reception of light pulses to determine the distance to objects, but with an expansion in the scanning capabilities provided by a mechanism to vary

the direction of the laser beam, such as a servo motor or an oscillating mirror, which allows the creation of a 2D or 3D point cloud data representation of the environment, as represented in Fig. 2.7.

Figure 2.6: LIDAR SF45/B (Source: LightWare).

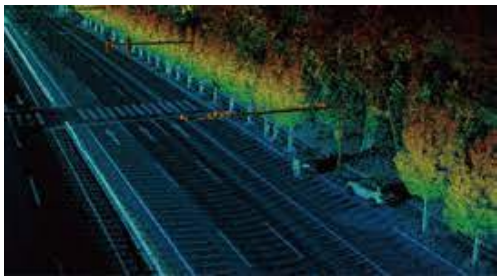


Figure 2.7: 3D LIDAR point cloud data representation shaded by height (Source: SureStar).

LIDAR sensors share most of the advantages of laser rangefinder, with emphasis on the high spatial resolution. Nevertheless, the beam-steering mechanism contribute to a higher payload, particularly in 3D LIDAR sensors, where the computational requirements are also higher. Consequently, most of the research is currently underway to use lightweight 2D LIDAR sensors in a computationally efficient way to meet the payload requirements of small UAVs [36].

In [40], a 3D LIDAR sensor is included as one of the main components of an obstacle warning and avoidance system for fixed-wing and multirotor human-controlled UAVs. In addition to hardware and software architecture, algorithms for obstacle avoidance trajectory generation, and design of a human-machine interface, some characteristics of the LIDAR behaviour in the detection of obstacles were specified, such as the fact that the used LIDAR beam scanned periodically (with frequency of 4 Hz) the area within a Field of View (FOV) of 40° in azimuth and 30° in elevation, and the azimuth orientation of the FOV could be changed by 20° left/right, during turning maneuvers. A similar application of a 3D LIDAR with 360° in azimuth and 30° in elevation is presented in [41], where simulations and ground tests with an octocopter provided good results for detection and avoidance of static obstacles.

2D LIDAR sensors are included in the obstacle detection phase of collision avoidance systems, for example, in [42], where the avoidance algorithm of Artificial Potential Fields (APF) was formulated as Control Barrier Functions (CBF) showing great results in simulation and experiments with a quadrotor, or in [43], where the 2D LIDAR is used for depth perception of planar obstacles, together with a monocular camera, presenting, also, good results in simulations and experiments with a quadrotor.

2.2.4 Ultrasonic Sensor

An ultrasonic sensor is a device used to detect obstacles and determine the distance to them, through the measurement of the time lapse between the emission of sound waves by a transducer and the reception of the respective sound waves that resulted from the reflection in the obstacle, considering the velocity of sound in the air. The frequency range of ultrasonic sensors typically falls between 20 kHz and 200 kHz [36].

Ultrasonic sensors are characterized to be cheaper and lighter than most of the other sensing systems. However, similarly to the laser, they are unidirectional, so their usage for environment scanning

Figure 2.8: Ultrasonic sensor MB1242 (Source: MaxBotix).

requires the presence of multiple sensors. Furthermore, they have a very limited detection range, typically up to a few meters, can be affected by external interference, including ambient noise and weather conditions [36], and cannot detect sound absorbing surfaces [44]. Given the limitation of detection range, most of the research of S&A with ultrasonic sensors is for multirotor UAVs in low-space environments, or in combination with other sensors with longer detection range.

A low-cost and low-computational solution for obstacle detection and distance controlled collision avoidance is shown in [44], with combination of one ultrasonic sensor, two infrared sensors and one pressure sensor for height control, and twelve ultrasonic sensors and sixteen infrared sensors more for 360° coverage in a quadrotor. The data fusion of the sensors is improved by inertial and optical flow sensors, used as a distance derivative for reference.

2.2.5 Passive Sensors

As stated before, passive sensors rely on capturing energy emitted by external sources. The most common passive sensors used in S&A systems for UAVs are electro-optical (EO) sensors, which can be vision-based or infrared (IR) sensors, and acoustic sensors. EO sensors consist of an array of small detectors, called pixels, capable of measuring the intensity of electromagnetic waves in the visible and infrared wavelengths range [24]. They detect targets by converting optical signals into electrical signals [45].

EO sensors that work in the visible wavelengths range can be designated as vision-based sensors and are one of the main components of the cameras that capture visual information in the form of images, which can be processed based on visual features, including edge, color, size, texture, shape, and optical flow, asynchronous events, and point clouds, to detect obstacles and estimate their position and motion [46]. They are characterized to be low cost and lightweight, but also sensitive to lightning conditions, occlusion, and motion blur [36]. Within vision-based cameras, it is worth highlighting the monocular (Fig. 2.9) and stereo-vision (Fig. 2.10) systems. The first one uses a single camera to capture the images and, consequently, lacks depth information, that must be estimated. Some of the methods for depth estimation with monocular cameras that were successfully applied in obstacle detection for UAVs are the Size Expansion Algorithm [47], that analyzes the size changes and expansion ratios of detected feature points, and the optical flow technique [48], that uses motion parallax to calculate the displacement of pixels between consecutive image frames and identify the motion vectors induced by the relative motion

between the camera and the obstacles. The stereo-vision system uses more than one camera, placed a fixed distance apart, to capture the same scene from different perspectives, providing depth perception and spatial information of the obstacles [36]. Its implementation in multirotor UAVs can be found in [49], where, after stereo matching based on epipolar line constraints and block matching, a two-stage image processing algorithm (thresholding of grayscale image and morphological processing), is applied to obtain a depth map, from which it was shown, in experimental tests, that can be extracted up to five obstacles in a distance of 15 meters.

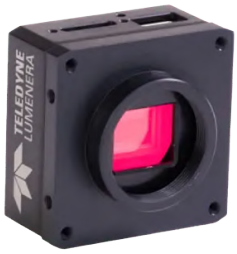


Figure 2.9: Monocular camera Lumenera Lt-C1950 (Source: Lumenera).

Figure 2.10: Stereo-vision camera Intel RealSense D455 (Source: Intel).

Passive infrared sensors can be considered the subtype of electro-optical sensors that work in the IR wavelengths range. Their main advantages are the possibility to work in low light conditions and detect objects based on their temperature, measuring the amount of heat they emit [45]. However, the output image of the IR sensors is blurry and distorted, with lower resolution when compared to visual cameras [46]. Thus, they are, usually, combined with other sensors to sense the environment, as in [44], where an implementation of IR sensors together with ultrasonic sensors for a quadrotor UAV S&A system is proposed.

Figure 2.11: Infrared sensor Sharp GP2Y0A710K0F (Source: Sharp).

Another type of passive sensors are the acoustic sensors, which are mainly composed of a microphone that can detect sound waves emitted by the intruding obstacles [24]. Consequently, these sensors are limited to the detection of noisy obstacles, such as aircraft intruders, and are vulnerable to noise interference. An application of this type of sensor in a S&A system for multirotor UAVs can be found in [50], where a laboratory prototype is developed and tested in real-world experiments. By recording the acoustic signature of the host UAV, prior to the experiments, and considering it in the data processing, it was possible to detect an intruder aircraft and determine its velocity and the range and time to the closest point of approach.

2.3 State Estimation of Obstacles

The sensors reviewed previously, in particular for non-cooperative systems, provide data of the environment which need to be processed in order to detect and track the obstacles. For any of the sensors, this process consists of the identification of obstacles in the environment and the estimation, at each step, of their states, including the position, velocity, size and shape, that allow for a prediction of their trajectory. The methods used in this process may vary depending on the nature of the data provided by the sensor, or even on the fusion of data from different sensors. This section addresses the application of the Kalman filter, Extended Kalman filter (EKF), and Markov chain in the estimation of obstacles' states.

2.3.1 Kalman Filter

A common way to estimate the unobservable or partially observable states of a system using available sensor measurements is through Kalman filtering. The Kalman filter is a real-time estimator that employs a predictor-corrector method that propagates the linear least mean squares estimate $\hat{\mathbf{x}}$ and its covariance of estimation uncertainty $\mathbf{P}$ forward in the time between measurements, predicting the estimate of the state variables along its covariance of estimation uncertainty before the next measurement is used. Then, the results of the measurements are used to correct the predicted values to reflect the influence of the information gained from the new measurements [25].

The Kalman filter [51] can be conceptualized in two distinct phases: predict and update. The predict phase uses the state estimate from the previous timestep to produce an estimate of the state at the current timestep (a priori estimate). The update phase multiplies the difference between the current a priori prediction and the current observation information by the optimal Kalman gain and combines with the previous estimate to refine the state estimate (a posteriori estimate).

In the predict phase, the predicted state estimate, $\hat{\mathbf{x}}_k^-$, and the predicted estimate covariance, $\mathbf{P}_k^-$, both at time instant t_k, are, respectively, given by

$$\hat{\mathbf{x}}_k^- = \phi_k \hat{\mathbf{x}}_{k-1}^+ + \mathbf{B}_k \mathbf{u}_k, \tag{2.1}$$

$$\mathbf{P}_k^- = \phi_k \mathbf{P}_{k-1}^+ \phi_k^T + \mathbf{Q}_k, \tag{2.2}$$

where, ϕ_k is the state transition matrix from t_{k-1} to t_k, $\hat{\mathbf{x}}_{k-1}^+$ is the a posteriori state estimate at t_{k-1}, $\mathbf{P}_{k-1}^+$ is the a posteriori state estimation covariance at t_{k-1}, $\mathbf{B}_k$ is the control-input matrix at t_k, $\mathbf{u}_k$ is the input vector at t_k, and $\mathbf{Q}_k$ is the covariance of the process noise.

In the update phase, the Kalman gain matrix, $\mathbf{K}_k$, the a posteriori state estimate, $\hat{\mathbf{x}}_k^+$, and the a posteriori estimation covariance, $\mathbf{P}_k^+$, all at time instant t_k, are, respectively, given by

$$\mathbf{K}_k = \mathbf{P}_k^- \mathbf{H}_k^T (\mathbf{R}_k + \mathbf{H}_k \mathbf{P}_k^- \mathbf{H}_k^T)^{-1}, \tag{2.3}$$

$$\hat{\mathbf{x}}_k^+ = \hat{\mathbf{x}}_k^- + \mathbf{K}_k (\mathbf{z}_k - \mathbf{H}_k \hat{\mathbf{x}}_k^-), \tag{2.4}$$

$$\mathbf{P}_k^+ = \mathbf{P}_k^- - \mathbf{K}_k \mathbf{H}_k \mathbf{P}_k^-, \tag{2.5}$$

where, $\mathbf{R}_k$ is covariance of the observation noise, $\mathbf{H}_k$ is the measurement sensitivity matrix, and $\mathbf{z}_k$ is the measurement vector, all at t_k.

In [52], a Kalman filter is applied in the detection and tracking of objects from range measurements of a 2D laser rangefinder scanner. The motion of detected objects is considered to be linear and constant between consecutive scans. The state vector, $\mathbf{x}_k$, and the state transition matrix, ϕ_k are defined as

$$\mathbf{x}_k = \begin{bmatrix} x_k \\ y_k \\ v_{x_k} \\ v_{y_k} \end{bmatrix} \text{ and } \phi_k = \begin{bmatrix} 1 & 0 & \Delta t_k & 0 \\ 0 & 1 & 0 & \Delta t_k \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{2.6}$$

where Δt_k is the sampling time between consecutive laser scans at iteration k, and (x_k, y_k) and (v_{x_k}, v_{y_k}) are, respectively, the position and velocity of the detected object. Since the laser only measures relative positions, the measurement vector, $\mathbf{z}_k$, and measurement sensitivity matrix, $\mathbf{H}_k$, are defined, respectively, as

$$\mathbf{z}_k = \begin{bmatrix} x_k \\ y_k \end{bmatrix} \text{ and } \mathbf{H}_k = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}. \tag{2.7}$$

The matrix $\mathbf{R}_k$ is calculated from the laser characteristics and $\mathbf{Q}_k$ depends on the errors imported by the approximation of the real motion by the equation of the model.

The Kalman filter is a very effective linear estimator, however, non-linear state estimation of obstacles can be handled with the Extended Kalman Filter (EKF) [51], which linearizes the system and measurement models at each time step. It uses first-order partial derivatives evaluated at the estimated value of the state vector. The state vector, $\mathbf{x}_k$, and state transition matrix, ϕ_k, are

$$\mathbf{x}_k = f_k(\mathbf{x}_{k-1}, \mathbf{u}_k) + \omega_k, \tag{2.8}$$

$$\phi_k = \left. \frac{\partial f_x}{\partial \mathbf{x}} \right|_{\mathbf{x}=\hat{\mathbf{x}}_{k-1}^+}, \tag{2.9}$$

where ω_k is the process noise, assumed to be a zero-mean multivariate Gaussian noise with covariance $\mathbf{Q}_x$. In the predict phase, Eq. (2.1) is replaced by

$$\hat{\mathbf{x}}_k^- = f_x(\hat{\mathbf{x}}_{k-1}^+, \mathbf{u}_k). \tag{2.10}$$

The measurement vector, $\mathbf{z}_k$ and measurement sensitivity matrix, $\mathbf{H}_k$ are, respectively, changed to

$$\mathbf{z}_k = h_k(\mathbf{x}_k^- + \nu_k), \tag{2.11}$$

$$\mathbf{H}_k = \left. \frac{\partial h_x}{\partial \mathbf{x}} \right|_{\mathbf{x}=\hat{\mathbf{x}}_k^-}, \tag{2.12}$$

where, ν_k is the observation noise, assumed to be a zero-mean multivariate Gaussian noise with covariance $\mathbf{R}_k$. Equation (2.4) must also be replaced by

$$\hat{\mathbf{x}}_k^+ = \hat{\mathbf{x}}_k^- + \mathbf{K}_k(\mathbf{z}_k - h_k(\mathbf{x}_k^-)). \tag{2.13}$$

The EFK is applied in [53] for state estimation (speed and heading in xy-plane, altitude and cartesian position) and trajectory prediction of moving obstacles around an UAV, but considering that the dynamic model of the obstacle is known. The quality of the predicted trajectory is evaluated by computing the variance of the prediction error (variance of the error between the modeled and predicted object trajectories). Two test-case simulations were performed, such that, in the first one, the EKF is used to estimate the states of an object in spite of process and measurement noises, and, in the second one, trajectories of the object presented in the first test case are predicted, in two different time instants, from the states estimated by the EKF. The first test-case results shown that the EKF succeeds in estimating adequate object setpoints, accurate model outputs and position estimates, filtering the noise of altitude measurement. The results of the second test-case are, also, satisfactory, with position predictions within the error margins.

2.3.2 Markov Chain

Another possibility for modeling the dynamics of obstacles over time and predict their states is to use Markov chains, which are stochastic models that describe a sequence of possible events, such that the probability of a future event depends only on the state of the present event and not past events (Markov property) [54].

This method is addressed in [55] to predict the position, speed and direction of obstacles flying around an UAV, in real-time, under dynamic unknown environments, considering that the relative position of the obstacles is measured by sensors at certain moments. In this case, a two-dimensional Cartesian rectangular grid was used to represent the environment and map the obstacles, such that the prediction of obstacle position in the grid is determined by the historical trajectory data of the obstacle, giving importance to recent data only. To simplify, it was considered that the obstacle can move between eight possible heading angles (0π, $\frac{\pi}{4}\pi$, $\frac{\pi}{2}\pi$, $\frac{3\pi}{4}\pi$, π, $\frac{5\pi}{4}\pi$, $\frac{3\pi}{2}\pi$, $\frac{7\pi}{4}\pi$), or stay in the same position. The m-th order transition probability matrix of the Markov chain, $\mathbf{P}^m$, was obtained from the first order transition probability matrix, $\mathbf{P}^1$, using the recurrence relation of the Chapman-Kolmogorov equation:

$$\mathbf{P}^m = \mathbf{P}^1\mathbf{P}^{(m-1)} = \mathbf{P}^{(m-1)}\mathbf{P}^1 = (\mathbf{P}^1)^m, \tag{2.14}$$

$$\mathbf{P}^1 = \begin{bmatrix} P_{11} & P_{12} & P_{13} & \dots & P_{19} \\ P_{21} & P_{22} & P_{23} & \dots & P_{29} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ P_{91} & P_{92} & P_{93} & \dots & P_{99} \end{bmatrix}, \tag{2.15}$$

$$P_{ij} = \frac{N_{ij}}{\sum_{j=1}^{9} N_{ij}}, \qquad i, j \in 1, 2, \ldots, 9, \tag{2.16}$$

where N_{ij} denotes the state transition number from state i to state j. The prediction for the next possible position of the obstacle is the matrix $\mathbf{X}(t)$ (1×9), in which each element is the probability of the corresponding flight action:

$$\mathbf{X}(t) = a_1 \mathbf{S}(t-1)\mathbf{P}^1 + \cdots + a_m + \mathbf{S}(t-m)\mathbf{P}^m + \cdots + a_k \mathbf{S}(t-k)\mathbf{P}^k, \tag{2.17}$$

where t is the current detection interval, k is the maximum order of prediction, $(t-m)$ is the m-th previous detection interval $(1 \leq m \leq k)$, $\mathbf{S}(t-m)$ is an 1×9 action vector of the UAV in the m-th previous detection interval, and $a_1, a_2, \ldots, a_m, \ldots, a_k$ are weight values that represent the influence degree of the prior actions, which are calculated from the autocorrelation coefficient determination method, to reduce the error caused by the experience. Having $A(t)$ as the current UAV action and $A(t-m)$ as the UAV action in the m-th previous detection interval, the autocorrelation coefficient of the m-th previous detection interval is given by

$$r_m = \frac{\sum_{m=1}^{k-m} A(t)A(t-m)}{\sum_{m=1}^{k} A^2(t)}, \tag{2.18}$$

and the weights are calculated by

$$a_m = \frac{|r_m|}{\sum_{m=1}^{k} |r_m|}. \tag{2.19}$$

Then, the largest element of $\mathbf{X}(t)$ is found and its position in the vector corresponds to the predicted next moving direction of the obstacle. Adding this direction the current position leads to the predicted next position of the obstacle.

Chapter 3

Collision Avoidance

The second major step of a S&A system for UAVs is to evaluate the existence of conflicts and potential collisions, re-plan the path in order to safely avoid a collision, and follow this path, based on the information of the detected obstacles obtained by the sensors. Thus, this chapter addresses some of the methods that can be used in UAVs to perform collision avoidance maneuvers, based on data from the position and motion of the obstacles. It is divided in global path planning, addressing the plan, done prior to the UAV flight, of a path to avoid obstacles with known location and characteristics, and local path planning, addressing the plan, done during the flight, in real-time, of the path to be followed to avoid specific obstacles that were not expected but were detected by on-board sensors. The collision avoidance methods addressed are presented in the diagram of Fig. 3.1. After a review of the main aspects of the most important methods and how they performed in previous simulations and experiments with UAVs present in the literature, a structured benchmark is presented to substantiate the decision of the algorithm to use in later software implementation.

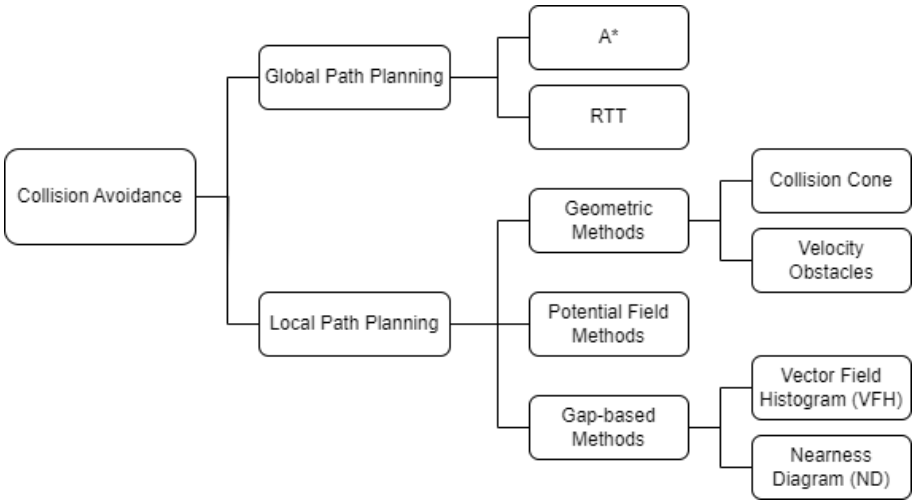


Figure 3.1: Diagram of collision avoidance methods.

3.1 Global Path Planning

The global path planning is particularly important for fully autonomous UAVs performing low-altitude missions, because it allows the establishment of a set of waypoints of the complete route, free from static obstacles. It requires the knowledge of an updated map of obstacles present in the environment, which can be computationally expensive [36]. It is also an opportunity to optimize the route to spend the least amount of energy. On the other hand, it is not robust to unpredictable obstacles, in particular, moving obstacles, as other aerial vehicles, such that it should be used in conjunction with a local path planner.

Among the possible methods of global path planning, the graph-based algorithms A*, Rapidly exploring Random Tree (RRT), and its variations are the most commonly found in UAV applications. The idea behind graph-based algorithms is to discretize the environment into nodes, with a cost associated, and connect them until the path to the goal that results in the lowest cost is found.

3.1.1 A* Algorithm

The A* is a graph search-based algorithm, introduced in [56] as an extension of the Dijkstra's algorithm [57], that uses a weighted graph, to get the path with the lowest cost between an initial node and a target node. It is based on the principle that one node, n, is considered to be part of a path (visited node) by evaluation of a cost, $f(n)$, given by

$$f(n) = g(n) + h(n), \tag{3.1}$$

where $g(n)$ is the movement cost from the initial node to that node, and $h(n)$ is an heuristic function that reflects the distance of that node to the target. The algorithm starts with the evaluation of the nodes adjacent to the initial one, to find the node with the least value of cost f and add it to a list of closed nodes. The nodes that already have a cost associated, but are not closed yet, remain on a list of open nodes. In the next iteration, the nodes adjacent to the node previously closed (parent node) are evaluated and, once again, the one with least f cost is found between the open nodes and added to the list of closed nodes. This process is repeated in a loop until the target node is reached, which is when the final path is found as the consecutive closed nodes with an overall lower f cost. The nodes whose position coincides with obstacles are not considered or have an infinity cost.

The A* algorithm guarantees to find the shortest path on graphs but does not guarantee to find the shortest path, smooth enough to be flyable, in a real continuous environment [58]. For that reason, some variations of the algorithm have been developed, such as D* (or Dynamic A*) and Theta*. Theta* was proposed in [59] and the main difference from A* is that a node does not need to be adjacent to its parent, rather, it can be any node, as long as there is a line-of-sight between them. There is an extension of this algorithm, called Lazy Theta*, proposed in [60], capable of generating similar paths in less time, by analysing only one line-of-sight per node. In [58], these two algorithms are compared with A* through a simulation in two 3D environment scenarios with different dimensions and static cuboid

obstacles. The idea was to simulate the pre-flight path planning of a UAV, addressing the computational time and path length. The results show that A* finds a path faster, but longer, than the Theta* and Lazy Theta* algorithms. In turn, Theta* and Lazy Theta* generate paths with the same length, but the latter is faster. It is also shown that the path of A* is less smooth, presenting several sudden heading variations, when compared to the other two.

3.1.2 Rapidly-exploring Random Tree (RRT)

RRT is a graph sampling-based algorithm for path planning, introduced in [61], based on the exploration of tree structures in high-dimensional spaces. It starts with an initial configuration of a graph with a single node and, at each iteration, a random configuration is sampled from the search space, the nearest node of the tree is identified, and the tree is extended toward the randomly sampled configuration with a new node, checking if the connection is feasible, that is, it does not cross any obstacle or other constraints. The distance between the new node and the tree is limited to a growth factor, avoiding the connection with nodes randomly placed too far from the tree. This results in a way to achieve Voronoi bias, meaning that the tree expands first towards areas with less nodes, creating a rapidly and aggressive search at the beginning, which eventually settles on uniform space coverage. One key advantage of RRT-based planning is that it can incorporate nonholonomic constraints [62].

Despite being good and fast at finding a valid path from an initial node to a target, RRT does not guarantee to find the shortest nor the optimal path. In this sense, the variation RRT*, proposed in [63], modifies RRT to generate a path closer to optimal by improving the method of selecting parent nodes with the inclusion of a cost function in the node expansion process and rewiring the nodes on the existing tree after each iteration. Other variations have been developed to solve other problemas of RRT, for example, Bidirectional RRT, proposed in [64], increases the search speed of RRT by using two fast expanding random trees that grow from the initial state point to the target. A combination of the last two methods, called Bi-RRT*, is used in [65] for path planning for an UAV quadrotor, after detection of obstacles using a fusion of data from a RADAR and a monocular camera. The experimental tests performed show that the developed solution is able to plan a path to avoid obstacles within one second, making it capable for real-time usage.

In [66], the problem of global path planning for UAVs is addressed with the proposal of an improved RRT algorithm. This solution involves the inclusion of another algorithm, the Ant Colony Optimization (ACO) [67], in the process of extending the random tree, which can set pheromones on the path obtained by RRT and select the next extension point according to the pheromone concentration, leading to a convergence to optimal path. Simulations in four different maps with obstacles of different shapes, sizes and positions show that, in all cases, this algorithm is able to generate shorter paths than the RRT and ACO algorithms alone. A comparison of the computational time taken to generate paths with the same lengths also shows that the improved RRT algorithm is faster in most of the cases.

3.2 Local Path Planning

When a map of the environment is not available or there is a possibility of finding unexpected obstacles during a flight, it is crucial to have a local path planner to adapt the route of the UAV, in real-time, depending on the position and dynamics of the obstacles that are detected by the on-board sensors.

There are several methods to perform path planning in real-time. Here, it was chosen to review to most common in UAV applications, which can be divided in geometric methods, including collision cone and velocity obstacles approaches; potential field methods; gap-based methods, such as the Vector Field Histogram (VFH), its variants, and the Nearness Diagram (ND).

3.2.1 Geometric Methods

In general, geometric methods generate paths to avoid collisions with obstacles by taking advantage of their geometry, position, and relative velocity. In UAV applications, there are two main geometric approaches, namely, the collision cone and velocity obstacles.

The collision cone approach, firstly proposed in [68], is based on the establishment of a 2D or 3D cone area between the UAV (vertex of the cone) and a circular or spherical boundary around the obstacle, representing the potential collision zone. The imminence of a collision is detected when the relative velocity vector of the UAV with respect to the obstacle is, geometrically, in the area of this cone. Thus, the collision avoidance maneuver is such that this velocity vector is kept outside the collision cone area. Figure 3.2 presents the geometric representation of a two-dimensional collision cone, $CC_{A,B}$, between an UAV, that was reduced to the point $\hat{A}$, with velocity vector $\mathbf{v_A}$, and an obstacle at point $\hat{B}$, with a circular safety boundary and velocity vector $\mathbf{v_B}$. The relative velocity, $\mathbf{v_{A,B}}$ is also shown.

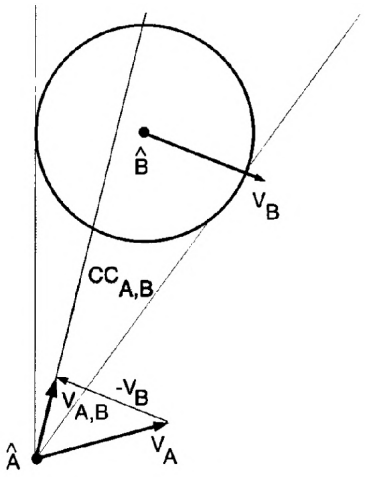


Figure 3.2: 2D collision cone representation [69].

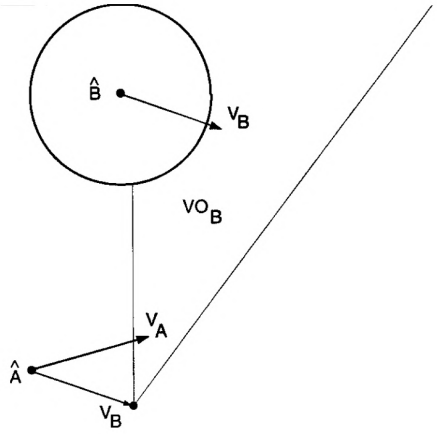


Figure 3.3: 2D velocity obstacle representation [69].

A solution of collision avoidance using the collision cone approach in a three-dimensional dynamic environment is proposed in [70] for an hexacopter. In this case, the state (position and velocity) of the obstacles, used to compute the spherical boundary and the collision cone, is estimated using a discrete-time Kalman filter with data obtained from a LIDAR sensor. It is shown that having the velocity

vector outside the collision cone is not enought to avoid a collision and a new strategy of avoidance is presented, based of the guidance to an aiming point, selected from the points on the intersection of the obstacle boundary and the collision cone, taking into account a prediction of the obstacle trajectory. Numerical simulations were performed to demonstrate that the hexacopter is able to avoid the collision with a moving obstacle, including when it is hovering, which raises the difficulty of having a velocity vector close to zero.

The velocity obstacles approach, introduced in [69], is similar to the previous one, but the potential collision zone is translated by the velocity vector of the obstacle, as seen in Fig. 3.3 using the same notation as the previous case. A collision occurs when the velocity of the UAV is within the velocity obstacle cone. This method is addressed in [71] to solve three-dimensional conflicts in UAVs, using a technique of avoidance-planes, and further developed in [72].

A different geometric method, named Fast Geometric Avoidance algorithm (FGA), is proposed in [73] for fixed-wing UAVs. It combines geometric avoidance of obstacles and selection of a critical avoidance start time based on kinematic considerations, collision likelihood, and navigation constraints. The method was simulated in MATLAB, in a SITL simulator, and implemented on a Navio2 board, having shown that it can deal efficiently with multiple obstacles, requires short computational time, and it is able to avoid the obstacles and get back to the original path with minimum deviation.

3.2.2 Potential Field Methods

The potential field methods are based on the physics-based potential fields, where objects are attracted or repelled by forces depending on their relative positions. The idea is to model the space around the UAV and the obstacles, creating a field (usually called Artificial Potential Field (APF)) of attractive and repulsive forces capable of guiding the UAV to avoid the obstacle. This forces can be obtained from the gradient of particular variables between the UAV and obstacles. The traditional APF states that the attractive forces are centered at the waypoints of the original path to be followed, whereas the repulsive forces are centered on the obstacles [74], as represented in Fig. 3.4. This approach presents some limitations, including the possibility of getting stuck in a local minimum, the presence of oscillations when passing by closely spaced obstacles and narrow passages, and the computational efficiency, which decreases when the number of obstacles increases [73]. The local minima problem is one of the limitations that raises more concerns and happens when the UAV, the target waypoint, and the obstacle are collinear and, at some point, the attractive force of the target equalizes the repulsive force of the obstacle.

A possible formulation for the APF is presented in the previous work [28], and states that the attractive force, $\mathbf{F}_{att}$, repulsive force, $\mathbf{F}_{rep}$, and total force, $\mathbf{F}_{total}$, are, respectively, given by

$$\mathbf{F}_{att} = \alpha_{PF} \frac{\mathbf{WP}_{close} - \mathbf{P}_{UAV}}{\|\mathbf{WP}_{close} - \mathbf{P}_{UAV}\|} + (1 - \alpha_{PF}) \frac{\mathbf{WP}_{next} - \mathbf{WP}_{close}}{\|\mathbf{WP}_{next} - \mathbf{WP}_{close}\|}, \tag{3.2}$$

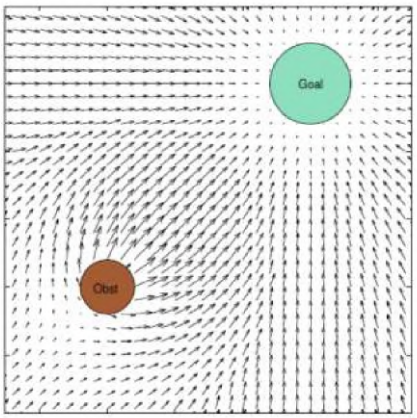


Figure 3.4: 2D APF representation with one waypoint (goal) and one obstacle [27].

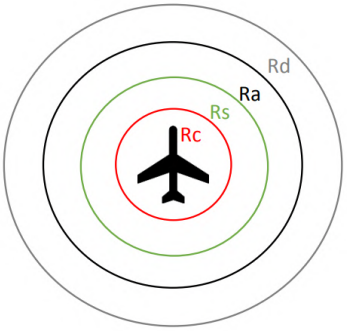


Figure 3.5: Safety zones around an UAV [28].

$$F_{rep} = \begin{cases} \infty, & d_o \leq R_c \\ \frac{\mathbf{P}_{UAV} - \mathbf{P}_{obs}}{d_o}, & R_a < d_o < R_c \\ 0, & d_o \geq R_a, \end{cases} \tag{3.3}$$

$$\mathbf{F}_{total} = \mathbf{F}_{att} + \sum_i \mathbf{F}_{rep}^i, \tag{3.4}$$

where $\mathbf{P}_{UAV}$ is the position of the UAV, $\mathbf{P}_{obs}$ is the position of the obstacle, $\mathbf{WP}_{close}$ is the position of the closer waypoint in the path followed by the UAV, $\mathbf{WP}_{next}$ is the next waypoint in the path, α_{PF} is a gain coefficient for the terms of the attractive force, $d_o = \|\mathbf{P}_{UAV} - \mathbf{P}_{obs}\|$ is the distance between the UAV and the obstacle, and R_a and R_c are the action and collision radius of the safety zones specified in Fig. 3.5. These forces acting in the acceleration vector of the UAV are, theoretically, enough to guarantee that it does not get too close to the obstacle. However, this formulation of the repulsive force leads to potential sudden variations of heading around the obstacle. Therefore, in the same work, and also in [25], it is improved with the inclusion of a swirling movement in the repulsive force, for a smooth circumvent of the obstacle, given by

$$\mathbf{s}_{dir} = S_{max} \frac{\hat{\mathbf{k}} \times (\mathbf{P}_{UAV} - \mathbf{P}_{obs})}{d_o}, \tag{3.5}$$

where $\hat{\mathbf{k}} = (0, 0, 1)$ and S_{max} is the intensity of the swirl. After the UAV passes the obstacle, the swirling effect of the repulsive force is canceled when an angle, θ, between the desired direction of motion and the direction of the obstacle, is higher than an arbitrary value, θ_{cutoff}. Resulting in a repulsive force, given by

$$F_{rep} = \begin{cases} \infty, & d_o \leq R_c \\ \mathbf{s}_{dir}, & R_c < d_o \leq R_s \\ \frac{R_a - d_o}{R_a - R_s}\mathbf{s}_{dir}, & R_s < d_o < R_a \\ 0, & d_o \geq R_a \vee \theta > \theta_{cutoff}. \end{cases} \tag{3.6}$$

In the literature, it is possible to find several solutions for UAV collision avoidance using potential field methods, with base principles similar to the ones used to formulate the forces of Eqs. (3.2), (3.3) and (3.4), but each focused on improving some specific features. In [75], the APF is optimized for multi-UAV environments, together with a solution, based on the variation of the UAV direction, for the problem of local minimization. In [74], the authors propose a Dynamic Artificial Potential Field (DAPF) for UAVs that improves the traditional APF method with the addition of a safety distance threshold that adjusts adaptively according to the UAV capacity and the motion of the obstacle, changes the repulsive force formula to include the speed steering force and a dependency on the threat level of the obstacle, splits the attractive force into position force and speed force, to make the UAV return to the original path stably after passing the obstacle, and introduces a damping force to the attractive force to prevent oscillations around the path.

3.2.3 Gap-based Methods

Unlike the previous methods that react to the presence of obstacles, gap-based methods can be categorized as the methods that react to the existence of space gaps between obstacles. Their working principle, based on an algorithm with predefined constraints that searches for a path to move the UAV through the most suitable gap in the environment, allows them to show good performance in cluttered environments with static and dynamic obstacles [36].

Vector Field Histogram (VFH)

The gap-based method most commonly used in UAV collision avoidance is the Vector Field Histogram (VFH), firstly proposed in [76] for mobile robots. It is characterized by the representation of the environment around the UAV in a polar histogram that is used to select the direction with less density of obstacles. In the first formulation of the method, data from onboard range sensors are used to discretize the area around the UAV in a two dimensional Cartesian histogram grid, C, where each cell, (i, j) holds a certainty value, $c_{i,j}$, of the probability of existence of an obstacle in that location. This grid is rapidly updated in real-time, such that only one cell is incremented at each range reading to lower the computational effort. Then, an active region of the histogram grid, C*, is defined as a square window of cells that accompanies the vehicle (Fig. 3.6a) and the content of each active cell is treated as an obstacle vector, with direction, $\beta_{i,j}$, from the cell to the center point of the vehicle and magnitude, $m_{i,j}$, dependent on the certainty value of the active cell, $c^*_{i,j}$, and the distance between the cell and the vehicle, $d_{i,j}$, such that $m_{i,j} = 0$ for the farthest active cell, increasing linearly for closer cells. Next, the active region, C*, is

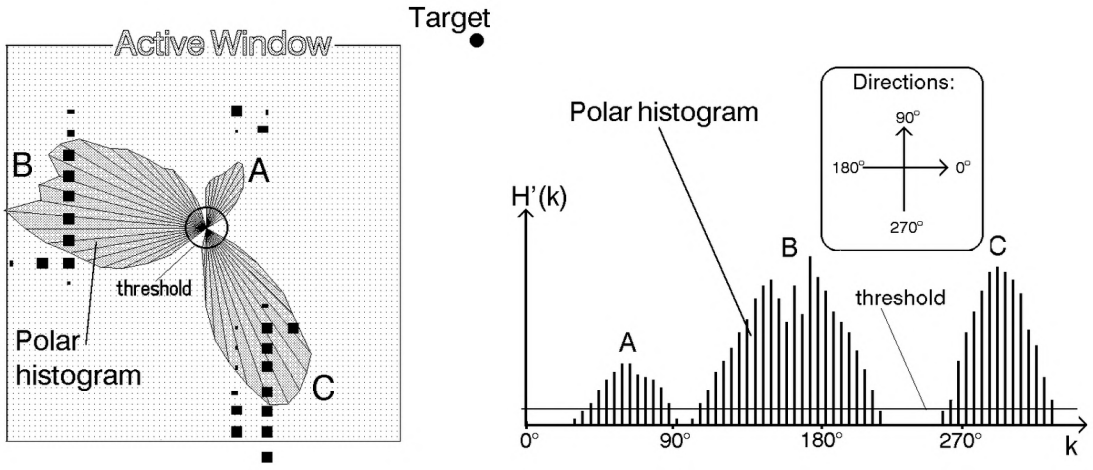
reduced to a one-dimensional polar histogram (Fig. 3.6b), H, with n angular sectors of width α around the vehicle, where each sector, k, holds a value, h_k, representing the polar obstacle density, given by

$$h_k = \sum_{i,j} m_{i,j}. \tag{3.7}$$

A smoothing function is, yet, applied to H, defining the smoothed polar obstacle density value, h'_k,

$$h'_k = \frac{h_{k-l} + 2h_{k-l+1} + ... + lh_k + ... + 2h_{k+l-1} + h_{k+l}}{2l+1}, \tag{3.8}$$

where l is an arbitrary value to be tuned (originally $l = 5$). The sectors of H with a polar obstacle density below a certain threshold value, τ, are considered candidate valleys and the algorithm chooses to follow the average direction of the valley closer to the target point.



(a) Active window.

(b) Polar histogram.

Figure 3.6: Vector Field Histogram method [76].

A variation of VHF, named VFH+, is proposed in [77] to improve reliability and the smoothness of the trajectories. It takes into account the width of the vehicle using an analytically determined low-pass filter, and the obstacle cells in the histogram grid are enlarged by its radius, allowing to update, at each time, all the histogram sectors that correspond to the enlarged cell, instead of one sector for each cell as in the traditional VFH. Moreover, the potential indecisive behaviour of VFH when an opening in the histogram alternates between an open and blocked state, causing the vehicle's heading to oscillate, is also solved by a hysteresis based on two thresholds, τ_{low} and τ_{high}, that are used to create a binary polar histogram, H^b, where, instead of polar density values, the sectors are open ($H^b_{k,n} = 0$ if $H_{k,n} < \tau_{low}$) or blocked ($H^b_{k,n} = 1$ if $H_{k,n} > \tau_{high}$) and, between thresholds, the value of the sector is kept unchanged. Another drawback of VFH that VFH+ tries to solve is the neglect of the dynamics and kinematics of the vehicle in the assumption that it is possible to instantly change direction of motion in every sampling time. For that, VFH+ assumes that the trajectory of the vehicle is based on circular arcs and straight lines, such that the maximum curvature is a function of the velocity. This way, the binary polar histogram is transformed in a masked polar histogram, H^m, where the value of each sector is 0 or 1 according to the evaluated possibility of the vehicle to go to that direction at the current speed and, in case all sectors are blocked,

a new evaluation is performed for a slower speed. In the selection of the direction to follow, VFH+ is, also, improved with the addition of a cost function that considers more than just the difference between the average direction of the candidate valleys and the target.

The same authors propose in [78] the VFH* method, as an improvement of the VFH+, to overcome problems related to the fact that VFH+ is a purely local obstacle avoidance algorithm and, in some situations, can make trajectory decisions that lead to blocked ways. It uses the A* search algorithm to project the position of the vehicle onto possible trajectories for each of the candidate directions that result from the VFH+, before making the final decision of a direction to follow. For every candidate direction, a cost function is calculated.

The previous VFH methods are for two-dimensional obstacle avoidance. An expansion for three-dimensional environments is done by the 3DVFH+ method, introduced in [79], which includes a prior stage for locating the position in space of the obstacles with an octomap framework. These positions are represented by active voxels (cubic volume elements) that lie within a bounding sphere inside a bounding box around the vehicle. The information of the active voxels is added into a 2D primary polar histogram, such that the x-axis and y-axis present, respectively, the azimuth and elevation angles of the voxel, as shown in Fig. 3.7. In the VFH+, only the azimuth angle was included in the polar histogram. Then, similarly to VFH+, the algorithm considers the physical characteristics of the vehicle with the enlargement of the voxels, creates a 2D binary polar histogram based on the previous one using threshold values as a function of the elevation, and selects a path to follow.

A method based on the 3DVFH+ is implemented in the flight control software PX4 for UAV multi-copters in [80].

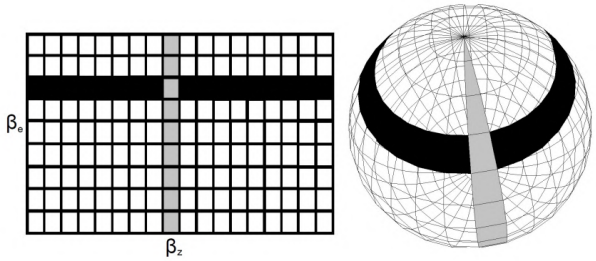


Figure 3.7: 2D primary polar histogram of 3DVFH+ [79].

In [81], the basic VFH method is improved for fixed-wing UAV cooperative collision avoidance. The environment is discretized in a reachable set of cells (Fig. 3.8) based on the nose pointing direction and the maximum turning rate of the UAV, which are then reduced to a polar histogram using Eqs. (3.7) and (3.8) with $l = 4$. As in basic VFH, the path of avoidance is selected as the one with average direction of the candidate valley (obstacle density bellow threshold) closer to the target waypoint. Since, in this case, the grid only covers space within the turning rate of the UAV, any direction can be directly considered to perform the avoidance maneuver, without the addition of circular arcs to take into account the dynamics of the UAV, as it happens in VFH+. This cooperative method was successfully validated in real flight tests with seven similar UAVs, using onboard host computers running Ubuntu 14 and Pixhawk flight controllers running a feed-forward PID control algorithm.

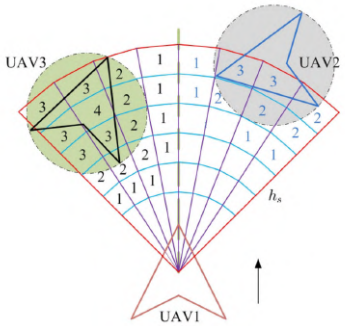
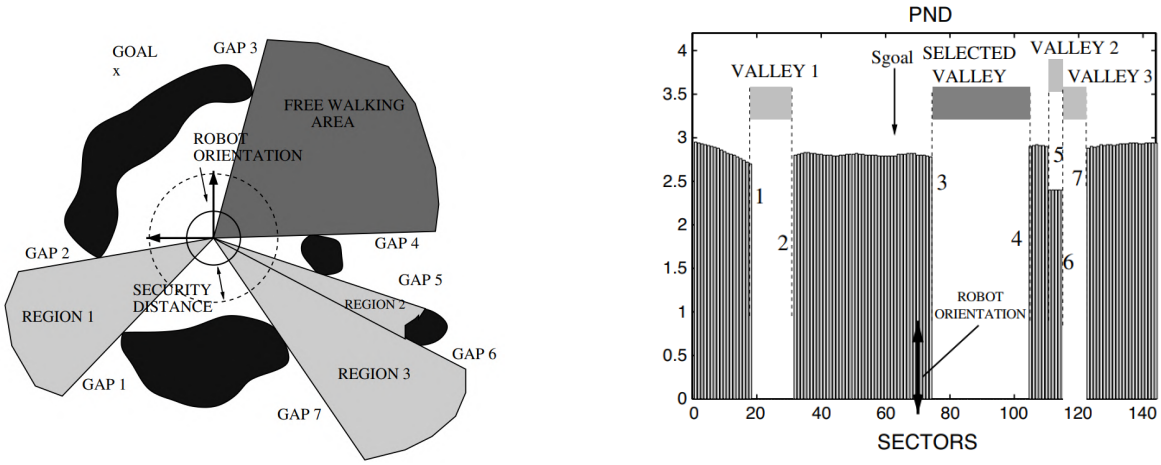


Figure 3.8: VFH grid with reachable set of cells for fixed-wing UAV [81].

Nearness Diagram (ND)

Another method using a gap-based logic, similar to VFH, is the Nearness Diagram (ND), introduced in [82] for mobile robots, which uses a "divide and conquer" strategy to simplify the navigation in densely cluttered and troublesome scenarios. It assumes that sensory information of the obstacles is available, at least, as depth point maps. The space around the robot is divided in n sectors, the obstacle points perceived are saved in a list L and, for each sector i, the minimum distance to an obstacle, $\delta_i(L)$, is computed. If there is no obstacle, then $\delta_i(L) = 0$. This information is represented in a nearness diagram from the central point (PND), such that $PND_i = d_{max} + 2R - \delta_i(L)$, where $d_{max} = max(\delta_i(L))$ is the maximum range of the sensor and R is the radius of the robot. Figure 3.9a presents an example of a two-dimensional space around a robot, with the gap regions highlighted, which were obtained from the PND with $n = 144$ shown in Fig. 3.9b, namely, from the sections where $\delta_i(L) = 0$. The region closer to the goal point is selected and checked for navigability. One of the limitations of this method is the availability of a clear sensor scan, since it is not very robust to sensor noise. Moreover, despite being a possible candidate for UAV collision avoidance, it is not possible to find an implementation for this type of vehicles in the literature, yet.



(a) Obstacles and gap regions.

(b) PND.

Figure 3.9: Nearness Diagram method [82].

3.3 Comparison of Collision Avoidance Methods

Having reviewed some of the methods of collision avoidance and their application in UAV, whether in pre-flight path planning and real-time path correction, it is important to compare them to substantiate the decision of the algorithm to use in later software implementation. In this sense, this section presents a literature-based comparison of the collision avoidance methods presented previously.

First, it is necessary to clearly define the performance metrics that are going to be considered in the benchmark of the methods, namely, the globality/locality, computational power required, complexity, robustness, and path optimization capacity. The first one is meant to evaluate if the method is more suitable for global or local path planning, regarding its characteristics and the predominance type of application that can be found in the literature. The computational power required metric quantifies the computational resources, such as processing time and memory usage, needed to execute the collision avoidance algorithm, which, in turn, can be used to assess its efficiency and scalability. Directly related to the previous one is the complexity metric, which evaluates, on the one hand, the simplicity and ease of understanding of the idea behind the method and, on the other hand, the feasibility to implement it in a host computer that communicates with a flight controller. The robustness metric assesses the ability of the methods to maintain a safe navigation in static and dynamic environments and the response to uncertainties and adversities. Finally, the path optimization capacity evaluates the final path followed by the UAV after the avoidance maneuvers regarding its length and smoothness.

Starting with the methods addressed in the global path planning section, the A* and its variants, D* and Theta*, are mainly found in global applications, because, as graph-search algorithms, they require more computational power, resulting in processing times that are not feasible in real-time application using a computer onboard the UAV. The A* can, eventually, be used for real-time path re-planning, as in [83], but it might present a major drawback related to processing time, such as the need of the UAV to stop during the mission while waiting for a new path to be planned, which is unfeasible for fixed-wing UAVs. However, these methods are simple to implement when there is a map of the environment available. In terms of path optimization, A* is reasonable, presenting paths with short lengths but lacking smoothness to be flyable, aspect where D* and Theta* get better results. Similarly to A*, RRT and its variants are found, more frequently, in global path planning, due to high computational power requirements, despite their relatively low complexity, resulting in a better performance in static environments. RRT cannot find the path with shortest length, which leads to a lack in path optimization, but RRT* was developed to improve this characteristic.

Regarding the methods addressed in the local path planning section, the geometric approaches of collision cone, velocity obstacles, and FGA share some characteristics, such as being fast and computationally efficient, that make them good for real-time path re-planning. Despite begin based on simple principles, the complexity of their implementation is highly dependent on the estimation of the position and velocity vectors of the obstacles. The simulations and real flight tests using these methods show that, usually, they are not guaranteed to find optimal trajectories, especially with multiple intruders. Also, they do not account for noisy sensor information or potential delay in sensor readings [24].

The Artificial Potential Field (APF) methods are also suitable for local path planning, requiring low computational power. The formulation of the attractive and repulsive forces that are used in these methods is also not very complex and can be easily implemented in a onboard computer. The main drawbacks of APF are the lack of robustness caused by the possibility of the UAV getting trapped in a local minimum and the oscillations around the path that make it far from optimal. There are some methods to get around these problems, but with increased complexity.

Finally, the gap-based methods, namely, the Vector Field Histogram (VFH), its variations, VFH+, VFH* and 3DVFH+, and the Nearness Diagram (ND) are characterized by low computational power requirements, that make them suitable for real-time path re-planning. The VFH and its variations show a particularly good performance in cluttered environments with static and dynamic obstacles. The paths produced by the VFH are not optimal and might not be feasible to fly due to sudden heading changes requirements, which is why the VFH+ was developed, that, despite increasing the complexity, is able to smooth the path generated by the algorithm. the VFH* increases even more the robustness of the VFH+, avoiding paths that lead to blocked ways. The ND is simpler to implement, but was not tested, yet, in UAV applications, such that it is not possible to infer its real performance.

The qualitative comparison of the collision avoidance methods described previously is summarized in Tab. 3.1, regarding the globality/locality, computational power required, complexity, robustness, and path optimization metrics.

Table 3.1: Qualitative comparison of collision avoidance methods

Refs.	Method	Global/Local	Computational Power Required	Complexity	Robustness	Path Optimization
[36] [58] [83]	A*	global	high	low	medium	medium
[58]	D*	global	high	low	medium	high
[58]	Theta*	global	high	low	medium	high
[61][62] [64] [65] [66]	RRT	global	high	low	medium	medium
[63]	RRT*	global	high	low	medium	high
[68]	Collision cone	local	low	medium	low	low
[69] [70] [71] [72]	Velocity obstacles	local	low	medium	low	low
[73]	FGA	local	low	medium	medium	high
[74] [75]	APF	local	low	low	low	medium
[36] [76]	VFH	local	low	low	medium	low
[36] [77]	VFH+	local	low	medium	medium	high
[36] [78]	VFH*	local	low	medium	high	high
[82]	ND	local	low	low	-	-

Chapter 4

Hardware Implementation

This chapter covers the physical components chosen to make up the obstacle detection and collision avoidance system and how they are configured and connected to complete an hardware implementation of the system.

First, the sensor hardware is addressed, regarding the characteristics and connections of the distance measuring onboard sensors chosen to sense the obstacles. Then, the flight controller hardware is explored, which is the main component of the system, where all the other hardware is connected, since it is responsible to, among other aspects, manage the flight dynamics and navigation of the UAV. Another important hardware component explored, also connected to the flight controller, is the companion computer, where the data measurements from the obstacle detection sensors are processed and the collision avoidance method is executed. Finally, an overall layout of the hardware is presented, including all the components and electrical connections necessary for an implementation in the UAV.

4.1 Sensor Hardware

The hardware used for obstacle sensing is composed of three different types of non-cooperative active sensors. From the obstacle detection sensors previously considered in Sec. 2.2, it was chosen to use two ultrasonic sensors, two laser rangefinders, and one LiDAR, given their availability.

The communication interface of the sensors is serial, meaning that data is transmitted in binary format, using a series of electrical pulses, one bit at a time. As it is going to be seen later, both the ultrasonic sensors and laser rangefinders use a specific type of synchronous serial communication protocol, which is the Inter-Integrated Circuit (I2C). This protocol allows multiple devices to be connected to the same bus using a I2C splitter, with only two bidirectional data signals: serial data line (SDA) and serial clock line (SCL) [84]. The devices on a I2C bus can be masters or slaves, such that the master is the one that initiates and controls the communications, generating the clock signal, and the slave responds to requests from the master. In this case, the master device is the flight controller and the slaves are the sensors. For this communication with multiple sensors on the same I2C bus to be possible, every slave device must have a unique address that allows the master to identify it. The standard I2C address

format, that is used in this case, is a 7-bit number, allowing for 128 possible addresses (in hexadecimal, range from 0x00 to 0x7F), followed by a read/write (1/0) bit, that indicates whether the master is reading from or writing to the slave.

4.1.1 Ultrasonic Sensors

Two different ultrasonic sensors from the Maxbotix I2CXL-Maxsonar-EZ line are used, namely the MB1202 and MB1242 (Fig. 2.8) models. Their main specifications, retrieved from the datasheet of the sensors [85], are summarized in Tab. 4.1.

Table 4.1: Maxbotix I2CXL-Maxsonar-EZ Series ultrasonic sensor specifications [85].

	MB1202	MB1242
Range (cm)	**25-765**	**20-765**
Resolution (cm)	1	1
Recommended update rate (Hz)	10	10
Beam pattern type	**Wide**	**Narrow**
Noise sensitivity	**High**	**Low**
Power supply voltage (V)	3.3-5	3.3-5
Power supply current (mA)	2.7(50 peak)-4.4(100 peak)	2.7(50 peak)-4.4(100 peak)
Operational temperature (° C)	0-65	0-65
Ultrasonic signal frequency (kHz)	42	42
Communication interface	I2C	I2C
7-bit I2C decimal address (hexadecimal)	**104 (0x68)**	**112 (0x70) - default**
Dimensions (mm)	22x19x15	22x19x15
Weight (g)	5.9	5.9

Both sensors detect obstacles up to 765cm, with 1cm resolution, but only provide range information to obstacles from 25cm, for MB1202, and 20cm, for MB1242. Regarding the update rate, it is recommended 10Hz, because the duration of a ranging cycle is proportional to the distance between the sensor and the target, i.e, a target at the minimum distance results in the fastest ranging cycle, around 25ms, whereas, at the maximum distance, or if there is no target, the ranging cycle takes around 100ms. Furthermore, setting a rate that is too high, and, consequently, sending I2C requests while the sensor is actively ranging, can introduce additional noise into the sensor and negatively impact performance [85].

Other notes pointed in [85] are that the I2CXL-MaxSonar-EZ sensors do not apply target size compensation to detected objects, resulting in larger size targets reported as being at a closer distance, or smaller size targets reported as being at a further distance, than the actual distance. The sensor calibrates itself each time it takes a range reading, and does not apply any compensation for the variation of speed of sound with the temperature.

The major difference between MB1202 and MB1242 ultrasonic sensors is the type of beam pattern. Wide beam patterns, as for MB1202, are associated with picking up more noise clutter, and detecting smaller obstacles, whereas, narrow beam patterns, as for MB1242, are associated with a an higher rejection of noise clutter and more difficult to detect smaller obstacles.

Both ultrasonic sensors use I2C serial communication, with a default 7-bit I2C address of 112 (decimal) or 0x70 (hexadecimal). However, to operate two sensors simultaneously on the same I2C bus and differentiate their range measurements, it is necessary to change the address of, at least, one of the sensors. In [85], there is a procedure outline for this purpose, as well as a C++ function to implement in Arduino, which was followed to change the I2C address of MB1202 to 104 (decimal) or 0x68 (hexadecimal) using an Arduino UNO board. The I2C address of MB1242 was kept default.

4.1.2 Laser Rangefinders

Two identical laser rangefinders, Lightware LW20/c (Fig. 2.5), are considered. Its main specifications, retrieved from the datasheet of the sensor [86], are summarized in Tab. 4.2.

Table 4.2: Lightware LW20/c laser rangefinder specifications [86].

	LW20/c	LW20/c
Range (cm)	20-10000	20-10000
Resolution (cm)	1	1
Update rate (Hz)	40-388 (default is 48)	40-388 (default is 48)
Accuracy (cm)	± 10	± 10
Beam divergence ($^\circ$)	< 0.5	< 0.5
Power supply voltage (V)	4.5-5.5	4.5-5.5
Power supply current (mA)	100	100
Laser class	1M	1M
Operational temperature ($^\circ$C)	-10 to +50	-10 to +50
Communication interface	Serial or I2C	Serial or I2C
7-bit I2C decimal address (hexadecimal)	**102 (0x66) - default**	**103 (0x67)**
Dimensions (mm)	30x20x43	30x20x43
Weight (g)	20	20

It is noteworthy that the detection range of the sensor goes up to 100m, much higher than the ultrasonic sensor's. Relying on the speed of light, instead of the speed of sound, allows the update rate to be higher, being configurable to a value from 40 to 388 measurements per second. Moreover, it is tolerant to changes in background lighting conditions, wind and noise, and the accuracy is not generally affected by the color or texture of the target surface, nor the angle of incidence of the beam [86].

Since it was chosen to use the I2C communication interface, once again, the I2C address of one of the two sensors had to be changed. This was done connecting the sensor to a computer with a TTL to USB adapter and using the Lightware Studio software provided by the manufacturer. One of the sensors kept the original I2C address of 102 (decimal) or 0x66 (hexadecimal) and the other was changed to 103 (decimal) or 0x67 (hexadecimal).

4.1.3 LiDAR

The previous sensors were only able to provide distance to obstacles at fixed directions. To scan the whole area ahead the UAV, the Lightware SF45/B LiDAR sensor (Fig. 2.6) is used. Its specifications are

present in Tab. 4.3.

Table 4.3: Lightware SF45/B LiDAR specifications [87].

	SF45/B
Range (cm)	20-5000
Linear resolution (cm)	1
Angular resolution (°)	< 0.2
Update rate (Hz)	50-5000
Accuracy (cm)	± 10
Scanning angle (°)	20-320
Beam divergence (°)	< 0.5
Power supply voltage (V)	4.5-5.5
Power supply current (mA)	300
Laser class	1M
Operational temperature (° C)	-10 to +50
Communication interface	Serial or I2C
Dimensions (mm)	51x48x44
Weight (g)	59

With a detection range up to 50m, the major features of this LiDAR are the scanning angle, which can go from 20° to 320° , and the update rate, configurable from 50Hz to 5000Hz. The speed of rotation is dependent on the scan angle and can go up to 5 sweeps per second. Just like the laser rangefinder, it is also tolerant to changes in background lighting conditions, wind and noise [87].

The scanning angle was configured to range from -45° to 45° , given the turning rate limitations of a fixed-wing UAV that make it unnecessary to detect obstacles at wider angles.

Regarding the communication interface, it was chosen not to use I2C, but serial communication through one of the telemetry (TELEM) ports of the flight controller, that is going to be addressed next.

4.2 Flight Controller and Companion Computer

The flight controller is the most important hardware component for the operation of an UAV and, also, for the S&A system. By processing data from measurements of various sensors, including gyroscopes, accelerometers, barometers, magnetometers and GPS, and generating outputs for the motors and servos of the UAV, it is responsible for performing several tasks during flight, such as stabilization and control of attitude, altitude and position, as well as manage navigation through waypoints. Moreover, it supports communication protocols with other devices, which can be used in telemetry for transmitting and receiving data to and from a Ground Control Station (GCS).

It is to the flight controller that the obstacle detection sensors are connected, since it is, also, capable of collecting their measurements and processing them in a first instance. However, due to its limited computation power, the processing necessary to the application of a collision avoidance method is left to a more powerful companion computer that is able to directly communicate with the flight controller through a serial port or Ethernet.

The flight controller chosen for this application is the Pixhawk 6X from Holybro [88], which, together with the Raspberry Pi Computer Module 4 (CM4) [89] as companion computer, are integrated in the Holybro Pixhawk RPi CM4 baseboard [90].

4.2.1 Pixhawk 6X

Inside the Pixhawk 6X Flight Controller (FC), presented in Fig. 4.1b, there is a Flight Management Unit (FMU) microcontroller (STM32H753), from STMicroelectronics, containing a 32-bit Arm Cortex-M7 core running at up to 480 MHz, with 2MB flash memory and 1MB RAM, as well as a IO microcontroller (STM32F100) containing a 32-bit Arm Cortex-M3 core running at up to 24MHz, with 8KB SRAM. It is paired with onboard sensors from Bosch and InvenSense, namely, 3x accelerometers/gyroscopes (ICM-20649 or BMI088, ICM-42688-P, and ICM-42670-P), 1x magnetometer (BMM150), and 2x barometers (BMP388). There is, also, a vibration isolation system to filter out high frequency vibration and reduce noise from sensor readings, and an Ethernet interface for high-speed integration with a companion computer [88].

The FMU hardware standard followed in the design of Pixhawk 6X is the DS-012 Pixhawk Autopilot FMUv6X Standard, that can be found in [91].

4.2.2 Companion Computer

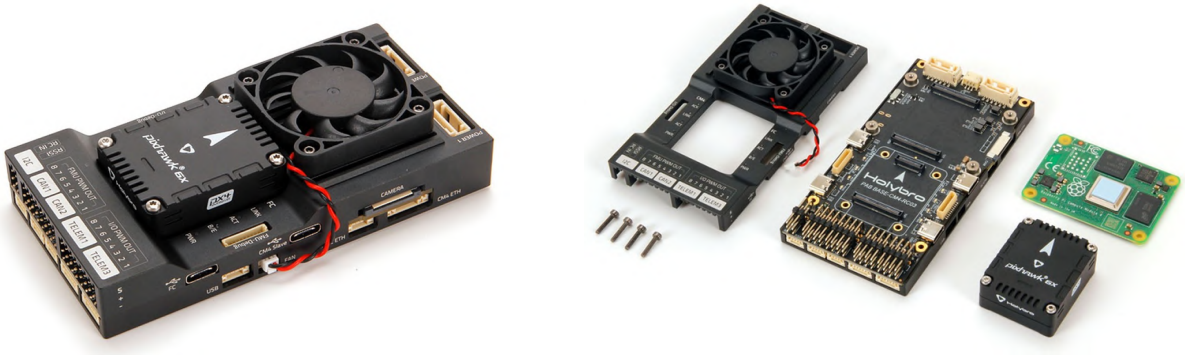
The version of the Raspberry Pi Compute Module 4 (CM4) that was chosen to be used as on-board companion computer, shown in Fig. 4.1b, is a System on Module (SoM) containing a Broadcom BCM2711 64-bit quad-core ARM Cortex-A72 processor running at up to 1.5GHz, 4GB of LPDDR4-3200 SDRAM, 16GB of eMMC flash storage with peak bandwidth of 100MBps, no wireless module, a Broadcom BCM54210PE on-board Gigabit Ethernet physical layer (PHY), a Peripheral Component Interconnect Express (PCIe) Gen2 x1 interface, a USB 2.0 interface, 28 GPIO pins, dual HDMI 2.0 interface, and MIPI Display serial Interface (DSI) and Camera Serial Interface (CSI) for display and camera connectivity, respectively [89].

The Raspberry Pi CM4 is designed to be mounted on an external board over PCIe, as the Pixhawk RPi CM4 Baseboard addressed in Sec. 4.2.3. The idea is to use only the latter on-board the UAV. However, the former can be considered an easier solution for configuring the CM4 and developing and implementing software on it, in a first instance, given its available interfaces and connectors: 2x full-size HDMI 2.0 connectors, Gigabit Ethernet RJ45 with PoE support, 2x USB 2.0 sockets with header for two more sockets, Micro USB socket, microSD card socket, standard fan connector, external power connector, 2x MIPI DSI display FPC connectors, 2x MIPI CSI-2 camera FPC connectors, Raspberry Pi HAT connector, and Real time clock (RTC) with battery socket [92]. A 12V 3.5A power supply unit was considered to power this board.

4.2.3 Pixhawk RPi CM4 Baseboard

Depending on the desired application, Pixhawk 6X can be integrated in different baseboards, which must follow the interface standards: DS-010 Pixhawk Autopilot Bus Standard [93] and DS-009 Pixhawk Connector Standard [94].

To simplify the inclusion of the companion computer within the hardware, the Holybro Pixhawk RPi CM4 baseboard (Fig. 4.1) was selected. It combines the Pixhawk 6X FC module with the Raspberry Pi CM4, offering two distinct solutions for communication between them: serial and Ethernet.



(a) Pixhawk RPi CM4 Baseboard assembled.

(b) Parts (from left to right): case with fan, baseboard, Raspberry PI CM4, and Pixhawk 6X.

Figure 4.1: Holybro Pixhawk 6X with Raspberry Pi CM4 baseboard [90].

A detailed scheme of this baseboard is present in App. A and the name and functionality of each of its ports is described below:

- **POWER1** and **POWER2** ports are used to power the Pixhawk from a Power Module with a 6-wire cable;
- **Serial** ports, which are used to communicate with peripheral hardware in short distance and can be configured for different functions:
  - **TELEM1** (6 pins) is configured, by default, as a MAVLink serial port suitable for connection to a GCS via a telemetry module;
  - **TELEM2** (6 pins) internally connects the FC module to the Raspberry Pi CM4:
    CM4 GPIO14 - FMU TXD TELEM2,
    CM4 GPIO15 - FMU RXD TELEM2,
    CM4 GPIO16 - FMU CTS TELEM2,
    CM4 GPIO17 - FMU RTS TELEM2;
  - **TELEM3** (6 pins) is configured, by default, as a MAVLink serial port suitable for connection with another device;
  - **GPS1** (10 pins) is configured to connect the primary GPS with integrated compass, LED, buzzer and safety switch and **GPS2** (6 pins) may be used to connect a secondary GPS;

- **I2C** port (4 pins) is prepared to communicate with devices using the I2C synchronous serial communication bus, introduced in Sec. 4.1;
- **UART4 & I2C** (7 pins) is a general purpose serial port with I2C and a Universal Asynchronous Receiver-Transmitter (UART) interface, which is an asynchronous serial communication bus with separated lines for transmitting (TX) and receiving (RX). The port includes, yet, an additional GPIO line for external NFC reader;
- **SPI** port (11 pins) is prepared to communicate with devices using the Serial Peripheral Interface (SPI), which is synchronous serial communication bus with four logic signals: serial clock (SCK), chip select (CS) from master to enable communication with a specific slave device, master out-slave in (MOSI), for serial data from the master, and master in-slave out (MISO), for serial data from slave;
- **CAN1** and **CAN2** ports (4 pins) are prepared to communicate with devices using the Controller Available Network (CAN) bus protocol;
- **AD & I/O** port (8 pins) is an Analog-to-Digital conversion (ADC) input and a GPIO;
- **USB** port (4 pins), as well as **FC USB-C**, can be used to directly connect the Pixhawk with a GCS using the Universal Serial Bus (USB) protocol;
- **CM4 Slave USB-C** is used to power and flash the Raspberry Pi CM4;
- **CM4 Host1 & Host2 USB-C** are used to connect external devices to the Raspberry Pi CM4.

- **Dip Switch** can change the functions of **CM4 Slave** and **CM4 Host1 & Host2** ports according to Tab. 4.4;

Table 4.4: Dip Switch [90].

	CM4 Slave	CM4 Host1 & Host2
RPi	Data Connected Power IN and Data	Data Not Connected Power out only
EMMC	Data Not Connected Power IN only	Data Connected Power out and Data

- **Ethernet** ports:
  - **FC ETH** port (4 pins) is prepared to establish communication between the FC and external devices through Ethernet;
  - **CM4 ETH** port (8 pins) is prepared to establish communication between the Raspberry Pi CM4 and external devices through Ethernet. **FC ETH** and **CM4 ETH** ports can be used to directly connect the FC and CM4 through Ethernet.
- **Radio Control (RC)** ports:
  - **DSM RC** port (3 pins) port is used to connect to a Spektrum/DSM radio receiver;

  - **RC IN** port (3 pins) is used to connect to a Serial BUS (SBUS) or Pulse Position Modulation (PPM) radio receiver;
  - **RSSI** port (3 pins) is used to input the Received Signal Strength Indication (RSSI) from the radio receiver.

- **Debug** ports:
  - **I/O Debug** port (10 pins);
  - **FMU Debug** port (10 pins) run the PX4 System Console and the Serial Wire Debug (SWD) interface.

- **PWM output** ports (signals sent to servos and motors):
  - Motors and servos are connected to **I/O PWM OUT** (3 pins) (MAIN) or to **FMU PWM OUT** (3 pins) (AUX).

- **Micro HDMI** is used for Raspberry Pi CM4 video output;
- **FAN** port (2 pins) is used to power the fan of the baseboard;
- **CAMERA**;
- **Micro SD** slot for log recording in a Micro SD card.

The complete list of pin assignments for each port is present in [95] and [90].

4.3 Electrical Layout

Having selected the main hardware components of the system, an overall electrical layout of the connections between them can be designed. Figure 4.2 presents a simplified diagram of such electrical layout.

It can be noticed that some auxiliary hardware components, essential for the flight operation of a fixed-wing UAV, were included, such as a power module, a battery, a DC motor, an Electronic Speed Controller (ESC), a GPS module, a radio receiver, and a telemetry module.

The chosen power module is the Holybro PM03D Power Module, shown in Fig. 4.3. It has both XT-30 and XT-60 connectors preinstalled for motor ESCs and battery (supports 2S to 6S batteries), as well as two 5V Battery Eliminator Circuit (BEC) and a selectable 8V/12V output for different kinds of peripheral devices [96]. It must be connected to the baseboard with a 6-pin CLIK-mate cable through POWER1 or POWER2 ports to power the FC and with a 4-pin to USB-C cable through CM4 USB-C Slave port to, separately, power the Raspberry Pi CM4.

The ESC that controls the DC motor is powered by the power module and connected to the FMU PWM OUT port of the baseboard to receive Pulse-width Modulation (PWM) signals from the FC. The four servos, one for each control surface - left aileron, right aileron, elevator, and rudder - are connected to the I/O PWM OUT port. The FMU and I/O PWM OUT ports must be powered separately, so the

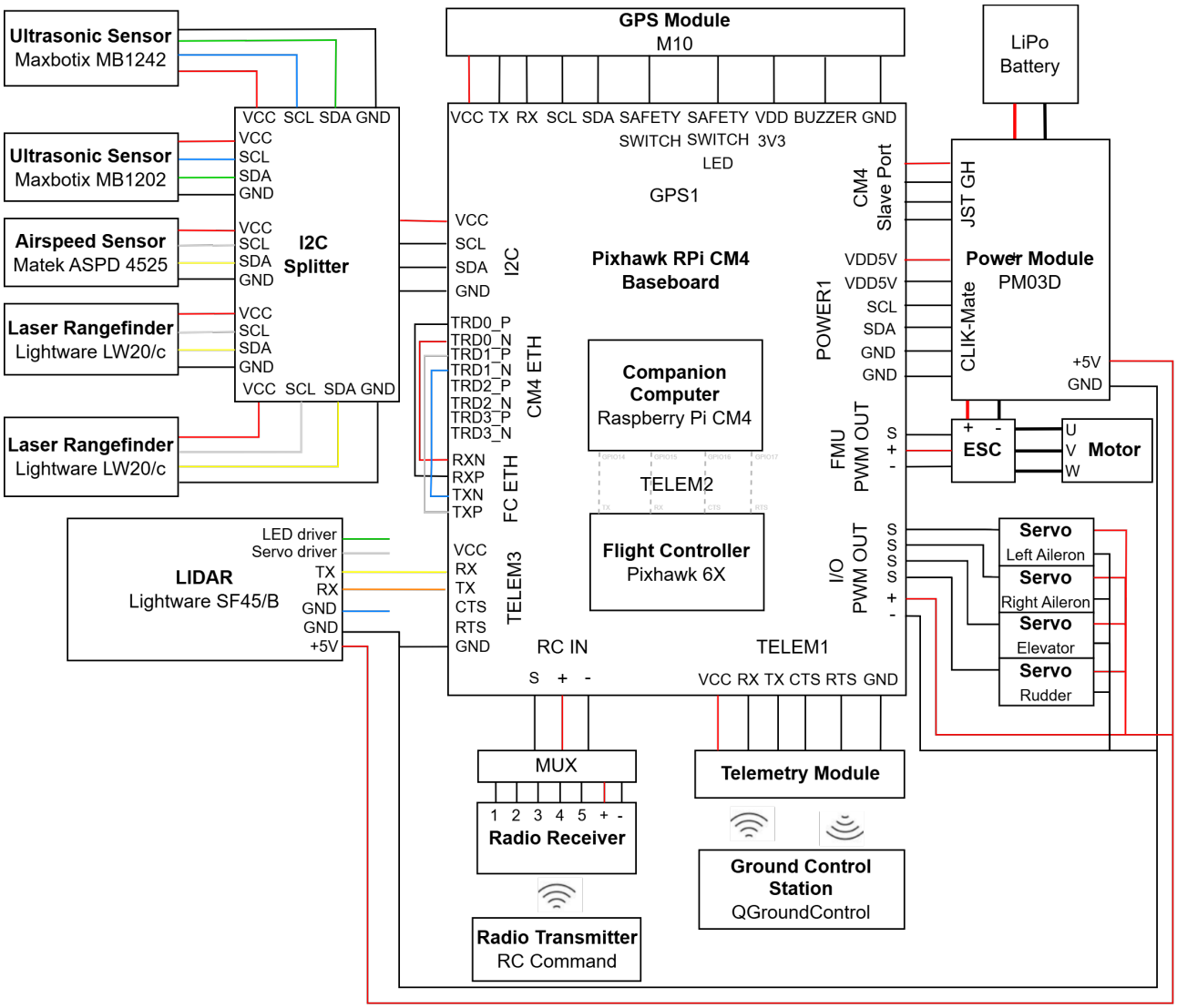


Figure 4.2: Hardware electrical diagram.

former is powered through the connection with the ESC and the latter is powered from the 5V BEC of the power module.

The radio receiver communicates wirelessly with the radio transmitter, receiving PWM signals. These signals are converted into a single Pulse-Position Modulation (PPM) signal through a multiplexer (MUX), which is connected to the RC IN port of the flight controller. The telemetry module is connected to TELEM1 port of the baseboard and communicates wirelessly with another telemetry module that is connected to the GCS.

Regarding the connection of the sensors for obstacle detection. The Ultrasonic sensors and Laser Rangefinders are all connected to the same I2C port using a I2C Splitter device. The LiDAR sensor is connected to the RX and TX pins of the TELEM3 port, but it is externally powered using the 5V BEC of the power module. The airspeed sensor is also connected to the the I2C bus through the I2C Splitter.

Even though the companion computer and the flight controller are internally connected in the baseboard through the serial TELEM2 port, a connection over Ethernet is used instead, due to higher bandwidth, from the CM4 ETH and FC ETH ports.

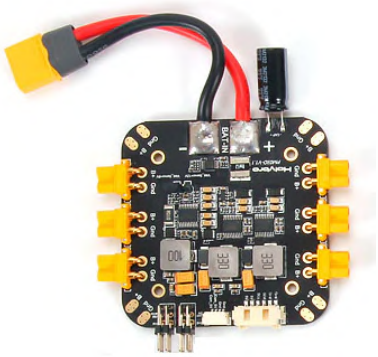


Figure 4.3: Holybro PM03D power module [96].

4.4 Hardware Configuration

This section addresses some of the aspects to take into account in the setup and configuration of the previously described hardware.

4.4.1 Flight Controller Configuration

Just like any flight controller hardware of Pixhawk series, Pixhawk 6X (Sec. 4.2.1) needs to be loaded with a firmware based on a flight control software to work as autopilot onboard a UAV. The flight control software to be used in this S&A system is the open-source PX4-Autopilot, developed by the Dronecode organization, and addressed with detail in Sec. 5.1.

The first step to load the desired firmware into the flight controller, is to establish a physical connection between the flight controller and a computer over USB (USB-C through the FC Port, in the case of the Pixhawk RPi CM4 Baseboard). Then, it is as simple as having QGroundControl installed on that computer, the Ground Control Software addressed in detail in Sec. 5.2, and following the steps presented in [97], which include the selection of the desired firmware to load in the Vehicle Setup tab. QGroundControl offers the possibility to load the most recent official releases of PX4, but also custom firmware versions.

Pixhawk hardware usually comes with an appropriate bootloader version pre-installed, such that it should not be a problem during configuration.

4.4.2 Raspberry Pi CM4 Configuration

Before use, the Raspberry Pi CM4 requires a prior configuration to prepare the eMMC storage to flash an Operating System (OS) image. This setup involves using another computer, to work as host device, an IO board for the CM4, such as one of those mentioned previously, and a USB cable to connect them. The detailed steps can be found in [98], using the CM4 IO Board, and, yet, in [90], using the Pixhawk RPi CM4 Baseboard.

The first step is to disable boot from eMMC at the hardware level, by fitting a jumper J2, in the case of

the CM4 IO board, or by switching the Dip-Switch (Tab. 4.4) to RPi position, in the case of the Pixhawk RPi CM4 Baseboard. Next, on the host computer, the software *usbboot* must be downloaded from the official Github repository [99], along with some auxiliary software, such as *libusb* [100] and *pkg-config* [101], if necessary. After connecting the host computer and the CM4 via USB (MicroUSB for CM4 IO board or USB-C for Pixhawk RPi CM4 Baseboard), the *usbboot* program must be compiled and the file *rpiboot* executed to boot the CM4 with a firmware that makes the eMMC storage emulate a USB Mass Storage Device (MSD), allowing the host OS to access it through the file system. In a first configuration, an empty USB MSD is expected to be found. From this step, the CM4 is ready to be flashed with an OS image and the easiest way to do so is by using the *rpi-imager* software [102] on the host computer, which offers a Graphical User Interface (GUI) to select the MSD storage and the OS to flash, from a list of supported possibilities. Finally, to boot the CM4 from the eMMC, the jumper J2 must be removed, in the case of CM4 IO board, or the Dip-switch changed to EMMC position, in the case of the Pixhawk RPi CM4 Baseboard.

Aiming to flash a Linux distribution in the CM4, in the first attempts to carry out the previous configuration, the host OS persistently failed to recognize the eMMC storage of the CM4 as a USB MSD after execution of *rpiboot*, either using the CM4 IO Board or the Pixhawk RPi CM4 Baseboard. The procedure followed to diagnose this problem and seek a solution included trying different OS for the host computer, trying different releases of *usbboot*, and following the troubleshooting guide present in [99] for this purpose, which includes some steps, such as hardware checks, running an alternative *mass-storage-gadget* version of *rpiboot* that uses Linux instead of a custom VPU firmware to implement the MSD, and running a *recovery* version of *rpiboot* that updates the SPI EEPROM bootloader. However, none of the these options solved the issue, leading to a possibility of that particular CM4 in use being defective. This latter possibility ended up being verified, since repeating the configuration procedure with a different CM4 did not result in any error.

Finally, The working Raspberry Pi CM4 was flashed with an OS image of Ubuntu Server 20.04 LTS, followed by the installation of a desktop GUI, resulting in the overall installation of the Ubuntu Desktop 20.04 LTS. The choice of this OS for the companion computer was made based on compatibility constraints with the Robotic Operating System (ROS) version Noetic, that is used by the collision avoidance system (see Sec. 5.3).

Chapter 5

Software Implementation

The hardware implementation described in the previous chapter requires the presence of adequate software to complete the obstacle detection and collision avoidance system, which can be rather extensive and complex when considered as a whole. Thus, the software implementation addressed in this work can be seen as an application with additional developments of existing open-source solutions.

Firstly, the flight control software that is used as base software is addressed and dissected in the most important aspects for this application, including the middleware protocol for internal communication between modules, the drivers for the sensors used in the obstacle detection, and the messaging protocol to establish external communication with other devices. Secondly, the ground control software selected to serve as central interface for configuring, monitoring and managing the flight control software and the UAV is presented. Then, the communication interface between the flight control software and the companion computer is discussed. Finally, a software implementation in the companion computer of a selected collision avoidance method is proposed, with emphasis on the data processing for detection of obstacles and potential conflicts, and translation of that information in the real-time generation of new trajectories for the UAV.

Figure 5.1 presents a diagram of the main components of the S&A system software implementation and high-level interaction between them.

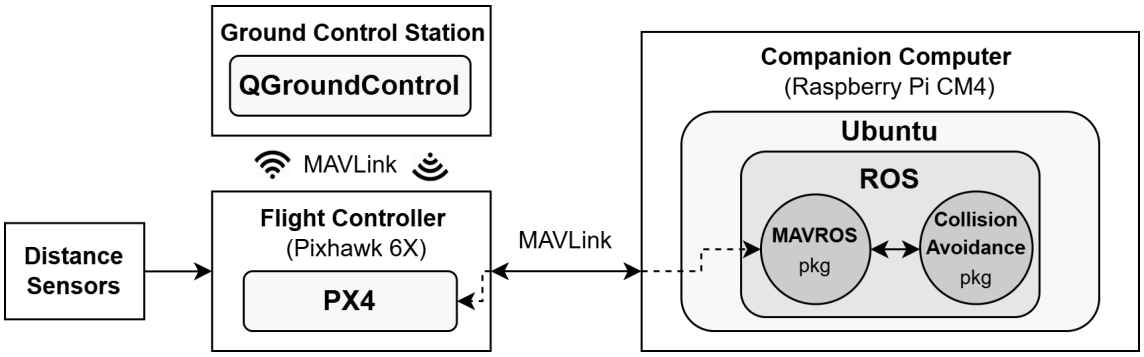


Figure 5.1: Diagram of S&A system software implementation.

5.1 Flight Control Software

The development of a software for obstacle detection and collision avoidance in UAVs requires a flight control software, that is responsible for functions important for a flight, such as autopilot, navigation, mission management, and communication with a ground control station. Of the most popular open-source flight control software projects that support autonomous flight of fixed-wing UAVs, it is worth highlighting the Ardupilot [103] and the PX4 [104] projects. The software adopted in this work is the latter, due to its reliability, modular architecture, allowing for extension of functionalities, good documentation, and increasing presence in the industry, with a growing community of users and developers.

In the following sections, key components and aspects of the PX4 software critical for the implementation of the S&A system are addressed, including a description of the software stack and architecture of PX4, the steps to configure and use it, key aspects of its communication protocol between internal modules, the drivers for the obstacle detection sensors, and the MAVLink protocol used for external communication between PX4 and other devices.

5.1.1 PX4 Stack and Architecture

PX4 is an open-source, BSD-licensed, flight control software, that supports different types of vehicles, such as multicopters, fixed-wing UAVs (traditional and VTOL), airships, autogyros, balloons, helicopters, rovers, and submarines. It is part of the non-profit organization Dronecode, administered by the Linux Foundation.

The architecture of PX4 [105], represented in a simplified version in App. B, consists of two main layers: the middleware and the flight stack. On the one hand, the middleware is, among other things, responsible for:

- Hardware integration, including the drivers of sensors, such as cameras, gimbals, GPS, IMU, airspeed sensors, the obstacle detection sensors referred in Sec. 4.1, etc.;
- External communication with other devices (GCS, companion computers, etc.) using the MAVLink (Sec. 5.1.5) or uXRCE-DDS protocols;
- Internal communication between modules using the uORB protocol (Sec. 5.1.3);
- Storage and log of parameters and other information in the EEPROM/SD Card;
- Simulation layer to run on a desktop operating system.

On the other hand, the flight stack is responsible for the flight control system. Including guidance, navigation and control algorithms for drones with fixed-wing, multirotor or VTOL airframes, as well as estimators for attitude and position.

PX4 runs on various operating systems that provide a POSIX-API (Linux, macOS, NuttX or QuRT). However, NuttX is the primary Real-time Operating System (RTOS) option for PX4, as well as the one used in this work, due to the fact that it is open source, light-weight, efficient and very stable [105].

5.1.2 PX4 Software Configuration

The setup and configuration steps needed, including additional developments in the PX4 software are described next.

The source code of PX4 can be found in the official Github repository PX4/PX4-Autopilot [106]. This work will be based on the stable version 1.14.3, released on May 24th 2024, whose source files can be obtained through the Git repository cloning command:

```
git clone https://github.com/PX4/PX4-Autopilot.git --branch v1.14.3
```

Given its modular architecture, it is possible to add or remove modules, drivers, and other features before building the software for a specific board. This process can be done in the PX4 Firmware Configuration menu (Fig. 5.2) or writing directly in the adequate `.px4board` file. In this case, for the Pixhawk 6X board, the `default.px4board` file (see App. C.2) is written using the Firmware Configuration menu, accessed through the following terminal command, at the PX4-Autopilot root folder:

```
make px4_fmu-v6x_default boardconfig
```

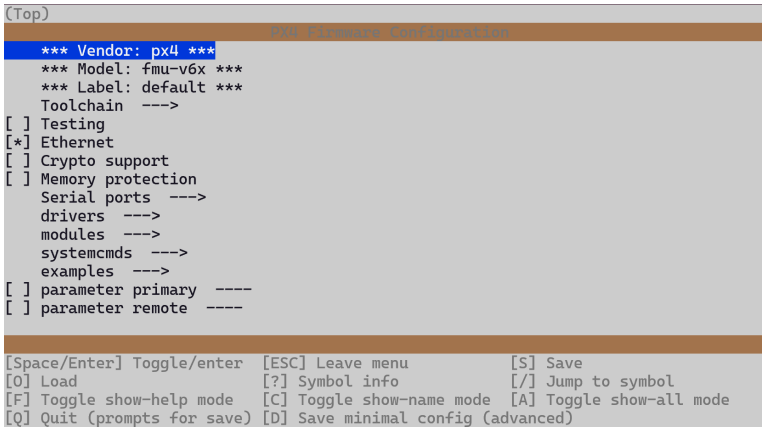


Figure 5.2: GUI of PX4 firmware configuration menu.

Finally, PX4 can be built using the Make tool, which performs an automated sequence of actions, previously defined in one or more Makefiles, to convert the source code in a standalone software artifact. The build process depends on the board the software is supposed to run in and the previous `.px4board` configuration file. Thus, in this case, for Pixhawk 6X, the build command is:

```
make px4_fmu-v6x_default
```

After building, a resulting file with `.px4` extension (see App. C.2) can be loaded to the corresponding flight controller, following the procedure stated in Sec. 4.4.1.

Terminal Access to PX4 System:

PX4 enables terminal access to the system through the System Console or the NuttShell (NSH).

The System Console runs on a specific UART port of the flight controller, for example the FMU Debug port of the Pixhawk RPi CM4 Baseboard (Sec. 4.2.3), and provides low-level access to the

system outputs. This allows, for example, for low-level debugging of the system boot process, PX4 modules, etc.

The NSH [107] is a NuttX integrated shell that provides high-level access to the system, which can be used to run commands and receive outputs of PX4 modules. Since the high-level interaction with PX4 modules is essential to the integration of obstacle detection sensors with PX4 and development of the S&A system, this shell is frequently used in this work. The easiest way to access it is through the connection, over telemetry or USB, of the flight controller to a Ground Control Station (GCS) that has a console that emulates the NSH, and using MAVLink communication (Sec. 5.1.5) between them. In this case, the MAVLink Console of the QGroundControl, the Ground Control Software addressed with detail in Sec. 5.2, is used to send commands to PX4 and receive outputs, since it emulates interacting directly with the NSH. This high-level interaction with PX4 from GCS, over MAVLink, is represented in the diagram of Fig. 5.3. As shown in the following sections, this method is used to send commands and get PX4 system outputs related, for example, to the internal communication between modules of PX4 (Sec. 5.1.3), the drivers of sensors (Sec. 5.1.4), and MAVLink streaming (Sec. 5.1.5).

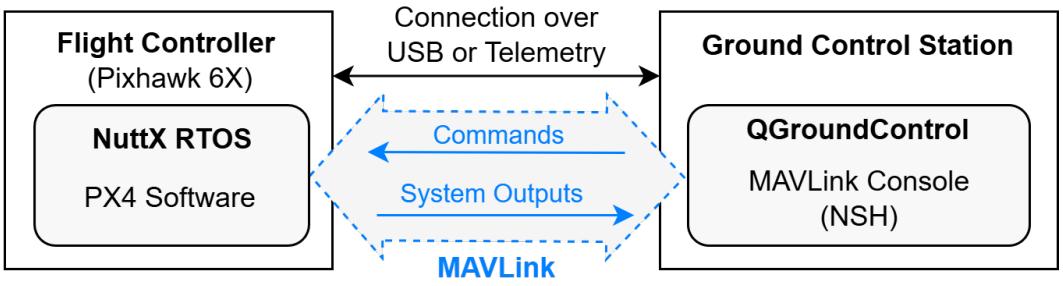


Figure 5.3: Diagram of the high-level access to PX4 system through MAVLink console.

5.1.3 PX4 Internal Communication: uORB

The communication between internal modules of PX4 is done using the publish/subscribe message bus named uORB, standing for micro Object Request Broker. It consists of an asynchronous messaging API used for inter-thread/inter-process communication in PX4. It works such that, after a uORB topic is advertised at least once, data can be published to the topic from anywhere in the system, then subscribed, resulting in a reactive communication system where the subscriber of a topic receives instant updates when new data is published to that topic. The asynchronism means that a publisher does not wait for a subscriber and vice-versa [108]. A graph with all built-in uORB topics can be found in [109].

The uORB communication provides, yet, a mechanism to publish multiple independent instances of the same topic, giving subscribers the freedom to choose the instances they want to subscribe to.

A uORB topic must have a prior definition of the fields that make up its message context. This definition is done in .msg files that can be found in `PX4-Autopilot/msg` directory (see App. C.2) and requires the inclusion of the type and name for each field.

Distance Sensor uORB Topic

In the case of the S&A system, the uORB communication will be responsible for handling the data of the obstacle detection sensors. Since the sensors addressed in Sec. 4.1 simply measure, at each instance, the distance to an obstacle at a given direction, their data can be published in the built-in `distance_sensor` topic by the driver of the sensor. The way data is published in the uORB topic is going to be covered with more details in Sec. 5.1.4. Every message of this topic consists of the fields presented in Tab. 5.1, described in the `DistanceSensor.msg` file (see App. C.2).

Table 5.1: uORB topic `distance_sensor` fields definition.

Name	Type	Units	Description
`timestamp`	`uint64`	ms	Time since system start .
`device_id`	`uint32`	-	Unique device ID for the sensor.
`min_distance`	`float32`	m	Minimum distance the sensor can measure.
`max_distance`	`float32`	cm	Maximum distance the sensor can measure.
`current_distance`	`float32`	cm	Current distance reading.
`current_yaw`	**`float32`**	**deg**	**Direction of sensor in the horizontal plane.**
`variance`	`float32`	m^2	Measurement variance.
`signal_quality`	`int8`	%	Signal quality of the sensor.
`type`	`uint8`	-	Type of distance sensor.
`h_fov`	`float32`	rad	Sensor horizontal Field of View.
`v_fov`	`float32`	rad	Sensor vertical Field of View.
`q`	`float[4]`	-	Quaternion of the sensor orientation.
`orientation`	`uint8`	-	Direction the sensor faces.

The first field of a `distance_sensor` message is the `timestamp`, in microseconds, since the system starts, useful for logging purposes. Next, the `device_id` is a unique numeric ID of the sensor sending the data, which does not change between power cycles. It is determined by the sensor driver using different parameters, such as characteristics of the serial or I2C connection to the pixhawk. The `min_distance` and `max_distance` are, respectively, the minimum and maximum distance, in meters, defined in the driver, the sensor can measure. The `current_distance` is the current distance, between the minimum and maximum, in meters, the sensor is measuring at each timestamp. Generally, it is set to -1 for invalid measurements. The `current_yaw` is the only non-standard field, added to include the direction, in degrees (from $-45°$ to $45°$), in the horizontal plane (yaw) of the sensors, in particular for the case of the LiDAR, which changes its direction at each timestamp to values that do not fit the `orientation` field. The `variance` is the variance, in square meters, for unknown/invalid readings. The `signal_quality` is the quality of the signal from the sensor, in percentage, defined in the driver, where 0 is invalid signal, 100 is a perfect signal, and -1 is unknown signal quality. The `type` is a standard field of integers to define the type of distance sensor: 0 is laser rangefinder, 1 is ultrasonic sensor, 2 is infrared sensor, and 3 is RADAR. The `h_fov` and `v_fov` are, respectively, the horizontal and vertical fov of the sensor, in degrees. The `q` is an optional quaternion definition of the sensor orientation with respect to the vehicle body frame. Finally, `orientation` is a standard field of integers to specify a three-dimensional direction

of the sensor, among a predefined list of possibilities, for example: 0 is forward facing (yaw of 0°), 1 is yaw of 45° , 2 is right facing (yaw of 90°), 3 is yaw of 135° , 4 is backward facing (yaw of 180°), 24 is upward facing (pitch of 90°), 25 is downward facing (pitch of 270°), etc. The `q` field is supposed to be used when the `orientation` is 100, which stands for custom value, however, given then the fact that, in the case of this S&A system for fixed-wing UAVs, all the sensors are placed in the horizontal plane, for the sake of simplicity, both `q` and `orientation` field are going to be discarded, and the direction of each sensor, at each timestamp, is defined in the custom `current_yaw` field.

The idea is to have a single `distance_sensor` uORB topic with messages being published in different instances, one for each sensor. But, due to limitations of the sensor drivers (Sec. 5.1.4), the `distance_sensor` topic will have 4 instances in total: one for both ultrasonic sensors, one for each laser rangefinder, and one for the LiDAR sensor.

The messages that are being published in uORB topics while PX4 is running in a Pixhawk flight controller can be accessed in real-time through the MAVLink Console in QGroundControl (Sec. 5.1.2). For example, to view the content of `distance_sensor` topic for 2 messages the following NSH command can be used:

```
nsh> listener distance_sensor 2
```

Figure 5.4 shows the printout of such a command with one ultrasonic sensor and one laser rangefinder connected. It is, also, useful to have access to the publishing frequency of each topic in real-time with the command:

```
nsh> uorb top
```

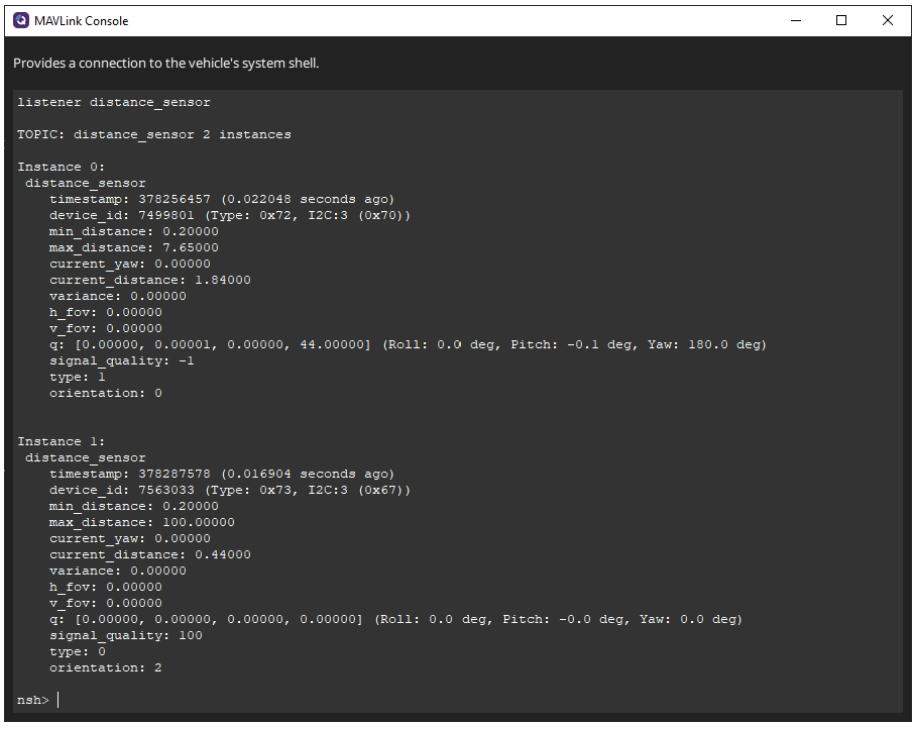


Figure 5.4: QGroundControl MAVLink Console printout example of `distance_sensor` uORB topic with 2 instances (one ultrasonic sensor and one laser rangefinder).

The sensor data that is internally organized in the `distance_sensor` uORB topic can, then, be streamed over the MAVLink protocol (Sec. 5.1.5), both to the Ground Control Station and the companion computer. In the latter, different approaches of interfaces and API can be followed to receive and interpret the data, as discussed in Sec. 5.3.

Other Relevant uORB Topics

In addition to internally communicate data of the obstacle detection sensors, the uORB message bus is also responsible to communicate data from other sensors of the flight controller that are going to be useful to the S&A system, in particular to the real-time generation of a trajectory to avoid the obstacles.

The `vehicle_local_position` uORB topic, defined in the homonym `.msg` file (see App. C.2), is used to communicate the UAV local position, velocity and acceleration estimates, produced by the EKF module of PX4, in a NED (North-East-Down) earth-fixed frame, whose origin is defined as the position at the moment the EKF is initialized. Besides that, the messages of this topic have fields regarding the validity of those estimates, estimator reset counters, the heading of the UAV (yaw) relative to the NED frame, the global position of the local NED frame origin in WGS84 coordinates, the distance to the ground (when available), the standard deviation of horizontal/vertical position and velocity errors, a flag to indicate if the estimation was obtained through dead-reckoning, and, finally, the vehicle limits of horizontal/vertical speed and height level above ground, specified in the estimator. Similarly, the `vehicle_global_position` topic is used by the position estimator of PX4 to publish the global position of the UAV in WGS84, based on the fusion of the GPS data with other sources.

Regarding the process of trajectory generation by a collision avoidance system, PX4 includes some uORB topics to communicate trajectories, allowing different approaches. As it is going to be discussed in Sec. 5.4, in PX4, it is possible to describe the path and trajectory of an autonomous operation of the UAV using whether setpoints or waypoints, depending on the desired flight mode.

On the one hand, setpoints are a real-time stream of position, velocity, attitude or thrust commands, which can be generated outside the flight controller and sent to feed the correspondent control loops of PX4. The `trajectory_setpoint` uORB topic is used to internally communicate position, velocity and acceleration setpoints in the local earth-fixed NED frame, as well as the desired yaw and yaw speed. The setpoints published to this topic are used as inputs of the PID position controller of PX4. The messages of the topic `vehicle_local_position_setpoint` can be used to monitor the setpoints used by the PID position controller.

On the other hand, waypoints are predefined points in space, in a global or local frame, that make up a plan of the path to be followed during an autonomous mission. The `vehicle_trajectory_waypoint` uORB topic is used to internally communicate up to five waypoints to be inputted in the position controller. Each of these waypoints is defined in a message of the `trajectory_waypoint` uORB topic with position, velocity and acceleration in the local NED frame, together with yaw and yaw speed, and the type of waypoint. There is, yet, the `vehicle_trajectory_waypoint_desired` uORB topic, which is a variation of the `vehicle_trajectory_waypoint` to be used to monitor the current waypoints sent to the position controller or, eventually, to an external device, like a companion computer.

5.1.4 Drivers of Distance Sensors

The interface between the sensor hardware and the PX4 requires intermediary software components, called sensor drivers. These components are responsible for the initialization of the sensors, including the setup of the communication protocol and initial configurations, acquisition of data measurements from the sensors, primary data processing, and communication with the uORB messaging bus.

Not all drivers are enabled by default in PX4. Usually, parameters are used to enable and disable drivers. The PX4 parameters, whose complete list is found in [110], can be manually changed by searching their definition in the source code or, more conveniently, through the Parameters screen in the Vehicle Setup tab of QGroundControl (Sec. 5.2).

In general, drivers can be manually controlled (start, stop, change specifications, check status, etc.) through dedicated NSH commands in MAVLink Console of QGroundControl. The full list of commands to control drivers of distance sensors is present in [111]. Moreover, since the PX4 system startup is controlled by shell script files [112], some of these commands, in particular the ones that are used to startup the drivers, are included, conditioned by the respective parameters, in the PX4 shell script file `rc.sensors`, that can be found in the `PX4-Autopilot/ROMFS/px4fmu_common/init.d` directory (see App. C.2), for the case of NuttX RTOS, to manage the automatic startup of drivers in PX4.

The source code of the drivers of distance sensors can be found in `PX4-Autopilot/src/drivers/distance_sensors` directory (see App. C.2). The following sections present some important details of the drivers used in the obstacle detection part of the S&A system, as well as any changes made to them.

Ultrasonic Sensor Driver

The ultrasonic sensors MB1202 and MB1242 from Maxbotix (Sec. 4.1.1) are controlled by the built-in `mb12xx` PX4 driver. This driver can be enabled (1) or disabled (0) from the firmware using the `SENS_EN_MB12XX` parameter and manually controlled using the NSH commands defined in [111] for this driver:

```
mb12xx <command> [arguments...]
 Commands:
   start
     [-I]         Internal I2C bus(es)
     [-X]         External I2C bus(es)
     [-b <val>]   board-specific bus (default=all) (external SPI: n-th bus (default=1))
     [-f <val>]   bus frequency in kHz
     [-q]         quiet startup (no message if no device found)
   set_address
     [-a <val>]   I2C address (default=112)
   stop
   status         print status info
```

By default, when the parameter is enabled, it is commanded, in the PX4 startup shell script file `rc.sensors`, to startup the driver in the external I2C bus:

```
# mb12xx sonar sensor
```

```
if param greater -s SENS_EN_MB12XX 0
then
  mb12xx start -X
fi
```

From the analysis of the driver source code, in the file `mb12xx.cpp` (see App. C.2), it can be found that it was build in such a way that a single instance of the driver is enough to control multiple ultrasonic sensors connected to the same I2C bus, as long as they have different I2C addresses. This is accomplished through a descending scanning through I2C addresses, from 112 (decimal) or 0x70 (hexadecimal) to 90 (decimal) or 0x5A (hexadecimal), in the `init()` function, in order to find and count all the ultrasonic sensors that are connected in the I2C bus. However, it was found that scanning through a large amount of I2C addresses could produce unwanted effects when sensors of different types are, also, connected to that I2C bus, in particular the laser rangefinders. In this sense, to avoid conflicts with sensors of different types, the scanning of I2C addresses was restricted to the addresses that were previously assigned to the MB1202 and MB1242 sensors in Sec. 4.1.1, namely, 104 (decimal) or 0x68 (hexadecimal) and 112 (decimal) or 0x70 (hexadecimal), respectively.

The time interval between sensor reads, or update rate, is also defined by the driver. This should be in accordance with the hardware limitation referred in Sec. 4.1.1, that is, take into account the time of a ranging cycle when the sensor is at the maximum distance of a target (765cm), or if there is no target, which is around 100ms. Since, in this case, the same driver is responsible to control multiple sensors, it is possible to define a time interval between reads of consecutive sensors. In this case, this latter value was set to 50ms, since only two sensors are used and an interval of 100ms between reads of each sensor is required.

Since a single instance of the driver is controlling both ultrasonic sensors, sensor data is published (by instruction in the `collect()` function of the driver source code) in a single instance of the `distance_sensor` uORB topic every 50ms for each sensor. Although this is not ideal, it is possible to distinguish the data from each sensor because a different `device_id` field of `distance_sensor` is assigned to each.

Laser Rangefinder Driver

The laser rangefinder sensors Lightware LW20/c (Sec. 4.1.2) are controlled by the built-in `lightware_laser_i2c` PX4 driver. Since this driver is used to control seven different models of the Lightware SF1XX family, it can be enabled (1 to 7) or disabled (0) from the firmware using the `SENS_EN_SF1XX` parameter, by matching an integer to the correspondent model, as documented in [110]. In the case of the Lw20/c model, SENS_EN_SF1XX must be 6. The driver can, also, be manually controlled using the NSH commands defined in [111]:

```
lightware_laser_i2c <command> [arguments...]
 Commands:
   start
     [-I]        Internal I2C bus(es)
     [-X]        External I2C bus(es)
```

```
    [-b <val>]  board-specific bus (default=all) (external SPI: n-th bus (default=1))
    [-f <val>]  bus frequency in kHz
    [-q]        quiet startup (no message if no device found)
    [-a <val>]  I2C address (default: 102)
    [-R <val>]  Sensor rotation - downward facing by default (default: 25)
  stop
  status        print status info
```

Contrary to what happens with the ultrasonic sensors, the driver of the laser rangefinders is unable to control, in a single instance, multiple sensors with different I2C addresses in the same I2C bus. For this reason, the solution found to have two lasers connected at the same time is by starting two independent instances of the driver in the startup shell script `rc.sensors` (see App. C.2), using the previous start driver command with the external I2C bus argument, as well as the correspondent i2C address of each sensor, which, as defined in Sec. 4.1.2, are 102 (decimal) or 0x66 (hexadecimal), for one, and 103 (decimal) or 0x67 (hexadecimal), for the other:

```
# Lightware i2c lidar sensor
if param greater -s SENS_EN_SF1XX 0
then
  lightware_laser_i2c start -X -a 102
  lightware_laser_i2c start -X -a 103
fi
```

Analysing the source code of this driver, present in the file `lightware_laser_i2c.cpp` (see App. C.2), it can be seen that the way data is published to the `distance_sensor` uORB topic also differs from the ultrasonic sensor driver. Instead of a direct instruction to publish to the topic, it uses an instance of the auxiliary class `PX4Rangefinder`, defined in `PX4Rangefinder.hpp` (see App. C.2). Moreover, since two independent instances of the driver are started, it, also, publishes data in two independent instances of the `distance_sensor` uORB topic, making them easily distinguishable, besides the differences in the `device_id` field.

LiDAR Sensor Driver

The LiDAR sensor (Sec. 4.1.3) is controlled by the built-in `lightware_sf45_serial` PX4 driver. Unlike the previous drivers, it is not included in the firmware by default, so it needs to be manually enabled in the PX4 Firmware Configuration (see Sec. 5.1.2), or by adding the following line to the configuration file `default.px4board` (see App. C.2):

```
CONFIG_DRIVERS_DISTANCE_SENSOR_LIGHTWARE_SF45_SERIAL=y
```

This driver has four configuration parameters [110]: `SENS_EN_SF45_CFG` to disable or enable the driver, by choosing on which serial port to run the sensor, `SF45_ORIENT_CFG` to select if the sensor is mounted facing upward or downward in the UAV frame, `SF45_UPDATE_CFG` to set the update rate, in Hz, from 12 possible values between 50Hz and 5000Hz, and `SF45_YAW_CFG` to select the horizontal orientation of the sensor from backward, forward, right and left. The first and third referred parameters are the

most important to configure the driver. SENS_EN_SF45_CFG was set to TELEM3 serial port (103) and SF45_UPDATE_CFG is set to 50Hz.

The driver can be manually controlled using the following NSH commands [111]:

```
lightware_sf45_serial <command> [arguments...]
    Commands:
        start           Start driver
            -d <val>     Serial device
            -R <val>     Sensor rotation - downward facing by default
        stop           Stop driver
        status         print status info
```

Listing 5.1: NSH commands for Lightware SF45 serial driver

Similarly to the laser rangefinder driver, the publishing of sensor data to the distance_sensor uORB topic is done resorting to the auxiliary class PX4Rangefinder, and results in an independent instance of the topic. In this case, it is also expected that the use multiple LiDAR sensors would require the initialization of multiple instances of the driver in different serial ports.

Even though the LiDAR sensor measures the scanning angle at each instant, the standard version of its driver does not publish these measurements, which are necessary to fulfill the custom current_yaw field added to the distance_sensor topic (Sec. 5.1.3). Since it is crucial to have access to data of the angle at which the sensor is scanning, at each time instant, to be able to determine the direction of a potential obstacle, the function s_update(), developed in [27], was included in the file PX4Rangefinder.cpp (see App. C.2). This function is responsible for publishing the scan angle of the LiDAR, ranging from $-45°$ to $45°$, to the distance_sensor uORB topic.

The version of the lightware_sf45_serial driver present in release 1.14.3 of PX4 presented some relevant performance issues regarding the publishing of sensor measurements data in the distance_sensor uORB topic. However, those issues were later resolved by the maintainer and developer of the driver. Thus, a newer version of the lightware_sf45_serial.cpp and lightware_sf45_serial.hpp files was used instead, which can be found in [113].

5.1.5 MAVLink Communication

In addition to the internal PX4 communication ensured by uORB message bus (Sec. 5.1.3), there is a need to establish external communication between the flight controller and other devices, such as the Ground Control Station and the companion computer. MAVLink, standing for Micro Air Vehicle Link, is a lightweight messaging protocol, designed for efficiently sending messages over unreliable low-bandwidth radio links, capable of providing this type of communication [114]. So far, MAVLink has been deployed in two versions, being MAVLink 2.0 the current version used with PX4 and the one referred in this work.

The MAVLink messages are encapsulated in data packets. The over-the-wire format of the MAVLink 2.0 packets is present in Fig. 5.5, which includes, from left to right, a packet start marker, payload length, incompatibility flags, compatibility flags, packet sequence number, system ID (sender), component ID

(sender), message ID, payload message, checksum, and signature (optional). The payload message can have up to 255 bytes, and information about its structure is not included, so the sender and receiver must share a common understanding of its meaning, order and size [115].

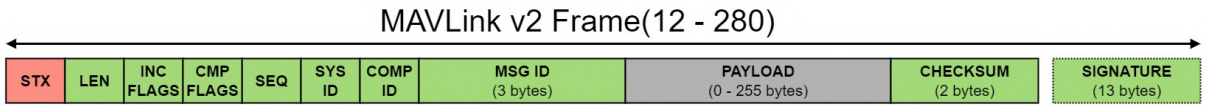


Figure 5.5: Over-the-wire format of a MAVLink 2 packet [115].

MAVLink messages are characterized by a name, an id, and fields containing the data to be transmitted. They are defined in XML files that are used by code generators to create programming-language-specific libraries from these definitions for sending and receiving the messages. Most of the messages that are used are standardized by MAVLink and their definitions are included in the following definition XML files: `development.xml` (includes definitions that are proposed to be part of the standard), `common.xml` (includes the definitions that meet the most common UAV use cases), `standard.xml` (includes definitions that are actually standard), `minimal.xml` (includes definitions required by a minimal MAVLink implementation), and other dialect XML files that are protocol-specific and vendor-specific [114]. Some of the messages defined in the XML files are enumerations (enums), which are used to define named values that may be used as options in messages - for example to represent types, errors, states, or modes [116].

MAVLink also includes high-level protocols, called microservices, used to communicate information, such as commands, parameters, missions, trajectories, images, and other data, that may not be able to be sent in a single message or require acknowledgment or error reporting [114].

PX4 includes MAVLink as a module, and builds `common.xml` MAVLink definitions by default, which can be found in the `PX4-Autopilot/src/modules/mavlink/mavlink/mavlink_definitions/v1.0` directory (see App C.2), together with other XML definition files.

Generally, in PX4, MAVLink messages stream data of an already existing uORB message (Sec. 5.1.3) with similar fields. Every MAVLink message requires the presence of a streaming class that is defined in `.hpp` header files present in `PX4-Autopilot/src/modules/mavlink/streams` directory (see App. C.2) and included in the file `mavlink_messages.cpp` present in `PX4-Autopilot/src/modules/mavlink` directory (see App. C.2). It is, also, possible to choose which messages are streamed by default in the file `mavlink_main.cpp` (see App. C.2) and organize them by MAVLink modes, which separate different applications of the MAVLink communication, for example, `MAVLINK_MODE_NORMAL` refers to streaming to a GCS and `MAVLINK_MODE_ONBOARD` to a companion computer.

Furthermore, MAVLink can have different instances to communicate with different peripheral devices simultaneously. Each instance `X` has its own configuration parameters: `MAV_X_CONFIG` to set the serial port, `MAV_X_MODE` to set the MAVLink mode, `MAV_X_RATE` to set the maximum data rate in bytes/second, and `MAV_X_FORWARD` to enable forwarding received packets onto other interfaces.

The following sections address the streaming of some of the most important MAVLink messages for the S&A system.

MAVLink Stream of Distance Sensors Data

The MAVLink message DISTANCE_SENSOR (ID=132) is a standard message, defined in the common.xml file, that can be used to communicate data from the obstacle detection sensors between the PX4 and external devices, such as a companion computer.

In order to suit the S&A system, the standard DISTANCE_SENSOR message was slightly modified. The fields of the message version considered in this work are presented in Tab. 5.2, together with their name, type, unit, whether they are standard or custom made, and a brief description of their content.

Table 5.2: MAVLink message DISTANCE_SENSOR fields definition.

Name	Type	Units	Standard/Custom	Description
time_boot_ms	uint32_t	ms	Standard	Timestamp.
min_distance	uint16_t	cm	Standard	Minimum range distance.
max_distance	uint16_t	cm	Standard	Maximum range distance.
current_distance	uint16_t	cm	Standard	Current distance reading.
type	uint8_t	-	Standard	Type of distance sensor.
id	uint8_t	-	Standard	Onboard index of the sensor (not used).
orientation	uint8_t	-	Standard	Direction of the sensor (not used).
covariance	uint8_t	cm^2	Standard	Measurement variance (not used).
horizontal_fov	float	rad	Standard	Horizontal Field of View (not used).
vertical_fov	float	rad	Standard	Vertical Field of View (not used).
quaternion	float[4]	-	Standard	Quaternion of sensor orientation (not used).
signal_quality	uint8_t	%	Standard	Signal quality of the sensor (not used).
current_yaw	**float**	**deg**	Custom	**Current horizontal direction.**
device_id	**uint32_t**	**-**	Custom	**Unique sensor ID.**

Most of the fields are similar to the ones of distance_sensor uORB topic, addressed in Sec. 5.1.3. The streaming of data from this uORB topic to the MAVLink message DISTANCE_SENSOR is done when the MavlinkStreamDistanceSensor class, derived from the MavlinkStream class, is defined in the file DISTANCE_SENSOR.hpp (see App. C.2). In the private section of the class, the multiple instances of the uORB topic are subscribed and the desired fields are copied to a structure that contains the fields of the MAVLink message.

In this sense, by default, the DISTANCE_SENSOR fields: time_boot_ms, min_distance, max_distance, current_distance, type, orientation, and covariance are copied from the correspondent fields of the uORB topic. Once again, the type and orientation fields are selected within predefined option values, however, in this case, they are directly provided by the MAVLink enumeration messages MAV_DISTANCE_SENSOR (Tab. 5.3) and MAV_SENSOR_ORIENTATION, respectively.

The remianing fields: horizontal_fov, vertical_fov, quaternion, and signal_quality are discarded, since their correspondent fields in the uORB topic are left unfilled.

The standard id field is meant to represent an ID of each onboard sensor. However, it is defined, by default, as an index of the instance of the distance_sensor uORB topic that is subscribed. Despite being a reasonable solution to identify the two laser rangefinders and LiDAR, since each of them has an independent instance in the uORB topic, it results in an ambiguous value for the case of the two

Table 5.3: Definition of MAVLink enum message MAV_DISTANCE_SENSOR

Name	Value
MAV_DISTANCE_SENSOR_LASER	0
MAV_DISTANCE_SENSOR_ULTRASOUND	1
MAV_DISTANCE_SENSOR_INFRARED	2
MAV_DISTANCE_SENSOR_RADAR	3
MAV_DISTANCE_SENSOR_UNKNOWN	4

ultrasonic sensors, because they share the same instance. Thus, this field will not be considered to distinguish sensors.

The standard DISTANCE_SENSOR message lack some important fields that are previously defined in the namesake uORB topic and are important for this implementation, such as the current_yaw, representing the direction, in degrees, in the horizontal plane (yaw) of the sensors, and the device_id, which includes an unique sensor ID defined in its drivers. Since MAVLink 2 allows the addition of extension fields to a standard message without breaking the message serialization, as long as the maximum payload of 255 bytes is respected, the following steps were taken to include the previous fields in the message:

1. Add the field current_yaw with the float type and the device_id with uint32_t type to the common.xml file (see App. C.2), in the message with ID=132, after the extensions/> tag;
2. In the DISTANCE_SENSOR.hpp file (see App. C.2), copy the current_yaw and device_id fields that come from the uORB topic (in the msg structure) to the MAVLink message (dist_sensor structure);
3. Add the previous fields to the mavlink_receiver.cpp file (see App. C.2);
4. Build PX4 again (see Sec. 5.1.2).

To test the addition of the previous custom fields and analyze the MAVLink traffic of the DISTANCE_SENSOR message, an external tool, Wireshark, was used. This tool consists of an open-source network protocol analyzer that can be used to inspect MAVLink traffic following the instructions in [117]. That process requires the generation of the Wireshark plugin Lua from the new common.xml definitions file, using the generator tool of MAVLink libraries, mavgen, included in MAVLink. In order to inspect, for example, the MAVLink traffic flowing to the GCS (Sec. 5.2), the MAVLInk network port (14550 by default) needs to be specified in the plugin file generated, mavlink_2_common.lua, which, in turn, needs to be added in the Wireshark installation directory. The following filter may be applied in Wireshark to visualize only the DISTANCE_SENSOR MAVLInk message and hide the Internet Control Message Protocol (ICMP) traffic:

```
mavlink_proto.msgid==132 && not icmp
```

Figure 5.6 presents the Wireshark printouts of a sequence of five DISTANCE_SENSOR MAVLink messages received in QGroundControl, with Pixhawk 6X connected via USB, each corresponding to one of the five obstacle detection sensors used in this system (two ultrasonic sensors, two laser rangefinders and one LiDAR sensor).

LiDAR Sensor

```
MAVLink Protocol (58)
> Header
v Payload: DISTANCE_SENSOR (132)
    time_boot_ms (uint32_t) [ms]: 193805
    min_distance (uint16_t) [cm]: 20
    max_distance (uint16_t) [cm]: 5000
    current_distance (uint16_t) [cm]: 143
    type (MAV_DISTANCE_SENSOR): MAV_DISTANCE_SENSOR_LASER (0)
    id (uint8_t): 3
    orientation (MAV_SENSOR_ORIENTATION): MAV_SENSOR_ROTATION_NONE (0)
    covariance (uint8_t) [cm^2]: 0
    horizontal_fov (float) [rad]: 0 (0 deg)
    vertical_fov (float) [rad]: 0 (0 deg)
    quaternion[0] (float): 0
    quaternion[1] (float): 0
    quaternion[2] (float): 0
    quaternion[3] (float): 0
    signal_quality (uint8_t) [%]: 0
    current_yaw (float) [deg]: -54
    device_id (uint32_t): 7536653
  Message CRC: 0x7ac1
```

Ultrasonic Sensor (0x70)

```
MAVLink Protocol (58)
> Header
v Payload: DISTANCE_SENSOR (132)
    time_boot_ms (uint32_t) [ms]: 193851
    min_distance (uint16_t) [cm]: 20
    max_distance (uint16_t) [cm]: 765
    current_distance (uint16_t) [cm]: 233
    type (MAV_DISTANCE_SENSOR): MAV_DISTANCE_SENSOR_ULTRASOUND (1)
    id (uint8_t): 0
    orientation (MAV_SENSOR_ORIENTATION): MAV_SENSOR_ROTATION_NONE (0)
    covariance (uint8_t) [cm^2]: 0
    horizontal_fov (float) [rad]: 0 (0 deg)
    vertical_fov (float) [rad]: 0 (0 deg)
    quaternion[0] (float): 0
    quaternion[1] (float): 0
    quaternion[2] (float): 0
    quaternion[3] (float): 0
    signal_quality (uint8_t) [%]: 0
    current_yaw (float) [deg]: 0
    device_id (uint32_t): 7499801
  Message CRC: 0x47d1
```

Laser Rangefinder (0x67)

```
MAVLink Protocol (58)
> Header
v Payload: DISTANCE_SENSOR (132)
    time_boot_ms (uint32_t) [ms]: 193837
    min_distance (uint16_t) [cm]: 20
    max_distance (uint16_t) [cm]: 10000
    current_distance (uint16_t) [cm]: 209
    type (MAV_DISTANCE_SENSOR): MAV_DISTANCE_SENSOR_LASER (0)
    id (uint8_t): 2
    orientation (MAV_SENSOR_ORIENTATION): MAV_SENSOR_ROTATION_YAW_90 (2)
    covariance (uint8_t) [cm^2]: 0
    horizontal_fov (float) [rad]: 0 (0 deg)
    vertical_fov (float) [rad]: 0 (0 deg)
    quaternion[0] (float): 0
    quaternion[1] (float): 0
    quaternion[2] (float): 0
    quaternion[3] (float): 0
    signal_quality (uint8_t) [%]: 0
    current_yaw (float) [deg]: 0
    device_id (uint32_t): 7563033
  Message CRC: 0x9ebf
```

Ultrasonic Sensor (0x68)

```
MAVLink Protocol (58)
> Header
v Payload: DISTANCE_SENSOR (132)
    time_boot_ms (uint32_t) [ms]: 193901
    min_distance (uint16_t) [cm]: 20
    max_distance (uint16_t) [cm]: 765
    current_distance (uint16_t) [cm]: 59
    type (MAV_DISTANCE_SENSOR): MAV_DISTANCE_SENSOR_ULTRASOUND (1)
    id (uint8_t): 0
    orientation (MAV_SENSOR_ORIENTATION): MAV_SENSOR_ROTATION_YAW_45 (1)
    covariance (uint8_t) [cm^2]: 0
    horizontal_fov (float) [rad]: 0 (0 deg)
    vertical_fov (float) [rad]: 0 (0 deg)
    quaternion[0] (float): 0
    quaternion[1] (float): 0
    quaternion[2] (float): 0
    quaternion[3] (float): 0
    signal_quality (uint8_t) [%]: 0
    current_yaw (float) [deg]: 0
    device_id (uint32_t): 7497753
  Message CRC: 0x1926
```

Laser Rangefinder (0x66)

```
MAVLink Protocol (58)
> Header
v Payload: DISTANCE_SENSOR (132)
    time_boot_ms (uint32_t) [ms]: 193846
    min_distance (uint16_t) [cm]: 20
    max_distance (uint16_t) [cm]: 10000
    current_distance (uint16_t) [cm]: 264
    type (MAV_DISTANCE_SENSOR): MAV_DISTANCE_SENSOR_LASER (0)
    id (uint8_t): 1
    orientation (MAV_SENSOR_ORIENTATION): MAV_SENSOR_ROTATION_NONE (0)
    covariance (uint8_t) [cm^2]: 0
    horizontal_fov (float) [rad]: 0 (0 deg)
    vertical_fov (float) [rad]: 0 (0 deg)
    quaternion[0] (float): 0
    quaternion[1] (float): 0
    quaternion[2] (float): 0
    quaternion[3] (float): 0
    signal_quality (uint8_t) [%]: 0
    current_yaw (float) [deg]: 0
    device_id (uint32_t): 7562777
  Message CRC: 0x65aa
```

Figure 5.6: Wireshark printouts of five `DISTANCE_SENSOR` MAVLink messages, each corresponding to one of the obstacle detection sensors.

Analyzing the behaviour of the `id` field, it can be observed that, as expected, both ultrasonic sensors have `id=0`, one laser rangefinder has `id=1`, the other has `id=2`, and the LiDAR has `id=3`, corresponding to the instance's index of the `distance_sensor` uORB topic subscribed at the moment of definition of the MAVLink stream. Meaning that this field is not enough to distinguish the sensors, in particular the ultrasonic sensors. The `device_id`, on the other hand, presents an unique integer for each sensor, allowing its unequivocal identification.

MAVLink Stream of Local Position Data

The MAVLink message `LOCAL_POSITION_NED` (`ID=32`) is a standard message, defined in the `common.xml` file, that can be used to communicate data of the local position of the UAV between PX4 and external devices.

It is defined as a stream of data from the vehicle_local_position uORB topic, addressed in Sec. 5.1.3. However, from all the fields of this uORB topic, only the fields with the three-dimensional position and velocity in the local earth-fixed NED frame are retrieved in the file LOCAL_POSITION_NED.hpp (see App. C.2) to the MAVLink message.

MAVLink Stream of Setpoints and Waypoints

In PX4, it is possible to describe the path and trajectory of an autonomous operation of the UAV using whether setpoints or waypoints, therefore, it is important to address how their communication can be done between PX4 and a companion computer.

Starting with the case of setpoints, it is worth remembering that the setpoint data must be published in the trajectory_setpoints uORB topic in order to be inputted in the position controller. To do so, a stream of position, velocity or acceleration setpoints defined in a companion computer can be sent over MAVLink to PX4 using the SET_POSITION_TARGET_LOCAL_NED MAVLink message (ID=84). This message requires the completion of the coordinate_frame field with one of the MAV_FRAME MAVLink enumerations to indicate the frame of the setpoint (in this case, should be MAV_FRAME_LOCAL_NED), and the type_mask field with one of the POSITION_TARGET_TYPEMASK enumerations to indicate dimensions of the setpoint to be ignored. Moreover, the MAVLink message POSITION_TARGET_LOCAL_NED (ID=85) retrieves data from the vehicle_local_position_setpoint uORB topic to monitor the setpoints that are actually being sent to the position controller of PX4.

For the case of waypoints, MAVLink has a specific microservice called Path Planning Protocol (Trajectory Interface) [118] that can be used. This protocol allows PX4 to request dynamic path planning from a companion computer. It works when PX4 has a pre-planned mission of waypoints and, with the parameter COM_OBS_AVOID=1, it sends the desired path to a companion computer through the MAVLink message TRAJECTORY_REPRESENTATION_WAYPOINTS (ID=332) at a frequency of 5Hz, and the companion computer sends back a similar MAVLink message but with a new generated waypoint at a minimum frequency of 2Hz. The TRAJECTORY_REPRESENTATION_WAYPOINTS message allows to send information of position, velocity and acceleration of 5 different waypoints in the local NED frame. The desired path sent by PX4 is made up of 3 waypoints (current waypoint adapted by FlightTaskAutoMapper, current waypoint unmodified and next waypoint unmodified), whilst the companion computer sends back only 1 next waypoint [119].

In addition, the standard MAVLink message VFR_HUD streaming head-up display (HUD) information, such as airspeed, groundspeed, heading, throttle, altitude MSL, and climb rate will also be relevant for the S&A system.

5.2 Ground Control Software

A Ground Control Station (GCS) is a ground based system that allows a human operator to monitor, control and manage the systems of an UAV in real-time. There are different hardware and software possibilities to built a GCS. In this work, QGroundControl was used as GCS software in a computer running Windows 10 OS.

QGroundControl (Fig. 5.7) is an open-source, cross-platform GCS software solution from Dronecode, prepared to communicate with flight controllers running PX4, Ardupilot, and other MAVLink enabled systems. Among its features, it is important to highlight the full setup/configuration and flight support of PX4 equipped vehicles, mission planning for autonomous flight, and real-time flight map display of the vehicle position, flight track, waypoints and vehicle instruments [120]. The communication between QGroundControl and PX4 can be done in a bidirectional data link through either USB or telemetry radio.

The Vehicle Setup tool of QGroundControl (Fig. 5.8) allows to upload Firmware to the flight controller, select the airframe, calibrate the sensors of the flight controller (accelerometers, gyroscopes, compass, etc.), configure Radio Communcation (RC) channels, flight modes, actuators, safety and camera, and read and change parameters of PX4.

In the Analyze Tools tab of QGroundControl (Fig. 5.9), it is possible to access the log of flight data, the MAVLink Console to interact with the NSH Shell of PX4, as addressed in Sec. 5.1.2, and the MAVLink Inspector tool that allows to inspect MAVLink messages communicated with the flight controller in real-time.

The MAVLink communication between the flight controller and the GCS is configured in PX4 through the following parameters: `MAV_0_CONFIG` is set to TELEM1, since it is the port used to connect the telemetry module, `MAV_0_MODE` is set to Normal, `MAV_0_RATE` is set to 1200B/s, and `MAV_0_FORWARD` is disabled.

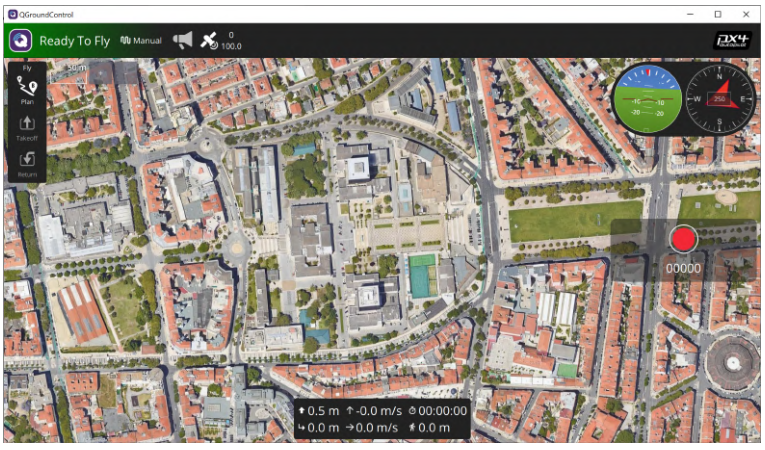


Figure 5.7: Main interface of QGroundControl.

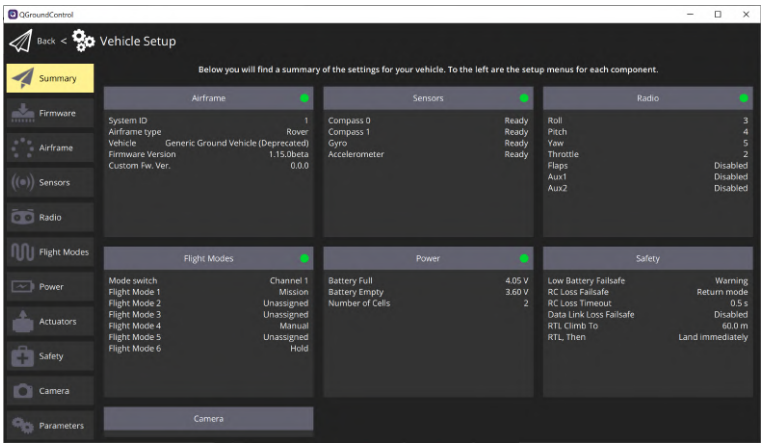


Figure 5.8: Vehicle setup tool of QGroundControl.

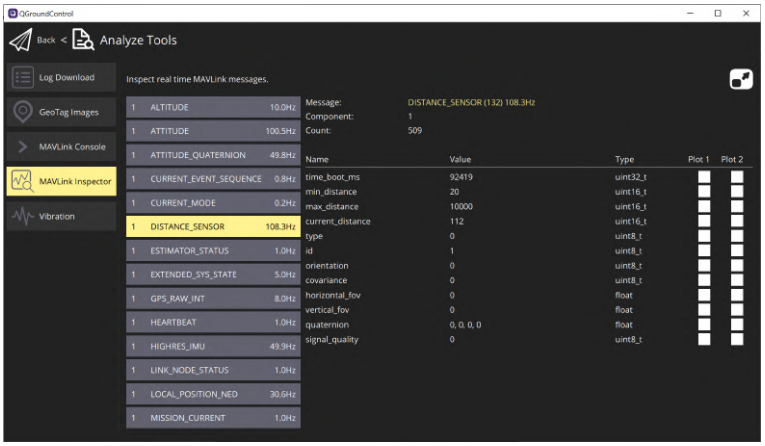


Figure 5.9: MAVLink Inspector of QGroundControl inspecting the `DISTANCE_SENSOR` message.

5.3 Communication Between PX4 and Companion Computer

The data of the obstacle detection sensors that is obtained and processed by the flight control software, PX4, needs to be transmitted, in real-time, to the companion computer, where the next processes of the S&A system are executed, as illustrated in Fig. 5.1. Similarly, the avoidance trajectory needs to be transmitted from the companion computer back to the PX4. This communication is based on the MAVLink protocol, addressed in Sec. 5.1.5.

This section covers the Ethernet connection setup between these devices and a discussion of the most suitable MAVLink interface to use in the companion computer (among MAVSDK, pymavlink and MAVROS options), how it manages data from PX4 and allows the integration of a S&A system.

5.3.1 Ethernet Connection Setup

As referred in Sec. 4.3, the Raspberry Pi CM4 that is used as companion computer can be electrically connected to the Pixhawk RPi CM4 Baseboard both through serial (on the internal TELEM2 port) and Ethernet connections. Of these options, the Ethernet connection was selected for this work, because

it can reach higher data transmission speed (the Raspberry Pi CM4 supports Ethernet speeds up to 1Gbps), being also a reliable and flexible solution.

The steps for the Ethernet connection setup can be found in [121, 122]. The most important aspect to take into account is that both PX4 and companion computer must be configured to run on the same IP network. The approach followed to achieve that was through the manual allocation of static IP addresses in the range 192.168.0.XXX (chosen arbitrarily) in both systems.

The static Ethernet IP address of PX4 was configured to 192.168.0.4 in the `net.cfg` of the Pixhawk 6X microSD card, through the following MAVLink Console commands:

```
nsh> echo DEVICE=eth0 > /fs/microsd/net.cfg
nsh> echo BOOTPROTO=static >> /fs/microsd/net.cfg
nsh> echo IPADDR=192.168.0.4 >> /fs/microsd/net.cfg
nsh> echo NETMASK=255.255.255.0 >>/fs/microsd/net.cfg
nsh> echo ROUTER=192.168.0.254 >>/fs/microsd/net.cfg
nsh> echo DNS=192.168.0.254 >>/fs/microsd/net.cfg
nsh> netman update -i eth0
```

The static Ethernet IP address of the Raspberry Pi CM4, which is running Ubuntu Desktop 20.04 LTS, as referred in Sec. 4.4.2, was set to 192.168.0.3 creating the `01-network-manager-all.yaml` file in the `/etc/netplan` directory of Ubuntu and adding the following configuration:

```
network:
version: 2
renderer: NetworkManager
ethernets:
    eth0:
        addresses:
            - 192.168.0.3/24
        nameservers:
            addresses: [192.168.0.3]
        routes:
            - to: 192.168.0.3
              via: 192.168.0.3
```

The automatic allocation of IP addresses by a DHCP server should be disabled in the Wired Connection IPV4 settings window of Ubuntu. Changes are applied with the following terminal command:

```
sudo netplan apply
```

To test the Ethernet connection established, one can do `ping 192.168.0.3` in the MAVLink Console, expecting the reception of packets from the companion computer, and do `ping 192.168.0.4` in the Ubuntu terminal, expecting the reception of packets from PX4.

The MAVLink communication between the flight controller and the companion computer is configured in PX4 through the following parameters: `MAV_2_CONFIG` is set to Ethernet, `MAV_2_MODE` is set to Onboard, `MAV_2_RATE` is set to 10000B/s, and `MAV_2_FORWARD` is disabled.

5.3.2 Comparison of MAVLink Interfaces for the Companion Computer

After the Ethernet connection was established between PX4 and the Raspberry Pi CM4, it was necessary to choose a MAVLink interface to use in the companion computer to exchange data with PX4 in the form of MAVLink messages. Since there are a few different open-source options available to accomplish that, this section is going to present a discussion and comparison of three main possibilities: **MAVSDK**, **pymavlink** and **MAVROS**, regarding their functionalities, design, usage and underlying purpose, in order to select the most appropriate to implement in the S&A system. A summary of that comparison is presented in Tab. 5.4.

Table 5.4: Comparison of MAVLink Interfaces

Aspect	MAVSDK [123]	pymavlink [124]	MAVROS [125]
Programming Interface	High-level API	Low-level library	ROS middleware
Programming Language	C++, Python, Java, Swift, Rust, JavaScript, C#, and Go	Python	C++, Python
OS Supported	Linux, macOS, Windows, iOS, and Android	Linux, macOS, Windows	Linux (Ubuntu only) with ROS1 or ROS2
Level of Message Abstraction	High	Low	High (ROS topics)

Starting with **MAVSDK** [123], it is a software development kit (SDK) with a collection of libraries to interface with MAVLink enabled systems in multiple languages, with an high-level object-oriented API. Its core library is written in C++, but it implements a gRPC server, a open-source Remote Procedure Call framework used to build APIs, and resorts to Protocol Buffers, a language-neutral open-source mechanism for serializing structured data, to support the development in different languages, and across various platforms. It has a modular plugin-based design to allow different functionalities to be loaded as needed. Currently, its APIs can be used in MAVLink communication with UAV systems to send commands, change parameters, manage missions, receive telemetry, control cameras, and others.

The main advantages of using MAVSDK are the ease of use provided by the abstraction of the MAVLink protocol complexity, and the support for multiple languages and platforms. However, the high abstraction of MAVSDK limits the low-level access and control over the MAVLink messages, which can represent a problem when dealing with complex messages. In the case of the S&A system, given that the `DISTANCE_SENSOR` MAVLink message was modified to allow the communication of data from five different sensors, it is important to have lower-level access to the message fields, in order to properly distinguish the data from each sensor at the moment of reception in the companion computer. Therefore, MAVSDK does not represent a good solution for interfacing with MAVLink in the companion computer.

In turn, **pymavlink** [124] is a Python library to handle MAVLink messages, that provides low-level access to the MAVLink protocol, without significant abstraction layers, from a set of modules: `dialects` with source XML message definition files for MAVLink v1 and v2, `mavutil` with utility functions for setting up communication links, receiving and decoding messages, running periodic tasks, and others, `mavwp` to load/save waypoints, `mavparm` to load/save sets of MAVLink parameters, `mavextra` with useful functions for converting values and messages, and `mavexpression` with functions for expression evaluation. It is used by developers in applications that require a fine-grained control of the MAVLink messages, such

as in the creation of custom applications that include the construction, parsing and handling of custom messages or the advanced usage of data fields included in the standard messages.

The customization, flexibility and lightweight of a low-level interaction with MAVLink are the main advantages of pymavlink. Nonetheless, these characteristics come naturally associated with a steeper learning curve in the development process, due to the need of a deeper understanding of the MAVLink protocol and increase of the manual work. Given this, it was considered that the S&A system does not require enough customization of the MAVLink communication to justify the development of a pymavlink interface in the companion computer.

Finally, **MAVROS** [125] is a Robot Operating System (ROS) package that acts as a bridge between ROS and MAVLink enabled systems, by translating the MAVLink messages into ROS messages, organized in ROS topics, and vice-versa, allowing the integration of the MAVLink communication in ROS-based systems. It is organized into plugins, where each plugin handles a specific set of messages. Moreover, it includes tools to support serial and network communication (UDP/TCP) with PX4 or Ardupilot, conversion of coordinates, and support to Offboard flight mode. Another package called MAVROS_EXTRAS complements the functionalities of MAVROS with additional plugins, including the one to handle the `DISTANCE_SENSOR` message. This way, MAVROS includes all the functionalities of MAVSDK, such as send commands, change parameters, manage missions, receive telemetry, and more, within the ROS environment.

The advantages of the integration of MAVLink with ROS are the possibility of high-level interaction with ROS topics, instead of directly with MAVLink messages, the extensibility of the MAVROS plugins to include new functionalities, and the possibility to include the collision avoidance software as a ROS package, better integrated with the remaining system. The main drawbacks are performance concerns, since the ROS middleware can be resource-intensive, requiring more computational power from the companion computer than the previous solutions, and may also introduce some latency in the MAVLink communication. Despite that, given the previous analysis to the available solutions, MAVROS was the solution selected to interface with MAVLink in the companion computer.

The following section presents more details of the MAVROS interface with MAVLink, with emphasis on the reception of data from the obstacle detection sensors.

5.3.3 MAVROS

There are similar versions of MAVROS for both ROS1 and ROS2. Their differences lie mainly in the differences of the ROS versions, which can be found in more detail in [126]. In summary, ROS1 is the oldest version of ROS and it is stable, well-tested and well-documented, but presents limitations regarding real-time and multi-robot systems, which are the aspects that ROS2, the latest version of ROS, aims to improve.

For the obstacle detection and collision avoidance system, MAVROS version 1.19.0 for ROS1 Noetic was the solution selected to interface with MAVLink in the Raspberry PI CM4 with Ubuntu 20.04. The version of MAVROS for ROS2 was not selected because the system will interact with a single robot

(UAV), and the documentation available for ROS1 is more extensive and complete.

Having Ubuntu 20.04 already installed in the companion computer, first, it is necessary to install ROS Noetic following the official documented instruction in [127]. Then, the `mavros`, `mavros_extras` and `mavros_msgs` packages are installed, together with `GeographicLib` datasets following the instructions in [128]. Here, a catkin workspace is created, which is a folder to modify and manage packages. Any modification of packages has to be followed by a `catkin build` command to take effect.

The UDP connection between PX4 and MAVROS is configured, accordingly to the previous Ethernet setup (Sec. 5.3.1) in the launch file, that, in this case, is px4.launch (see App. C.1), by adding the link `udp://@192.168.0.4:14550` to the "`fcu_url`" argument. The following command line is used to launch MAVROS:

```
roslaunch mavros px4.launch fcu_url:=udp://@192.168.0.4:14550
```

Adaptation of MAVROS Distance Sensor Plugin

Now, it is important to address the MAVROS reception of the MAVLink message `DISTANCE_SENSOR` (Sec. 5.1.5) containing the data of the five obstacle detection sensors. That bridge is done by the MAVROS_EXTRAS plugin `distance_sensor.cpp` (see App. C.1), which needs to, first of all, be removed from the blacklist of `px4_pluginlists.yaml`.

Analyzing the plugin, it is found that it is prepared to receive data of different sensors from MAVLink, by searching in a previously configured sensor map which data corresponds to which sensor, using its unique ID. By default, the standard `id` field of the `DISTANCE_SENSOR` message is considered. However, given that, as referred in Sec. 5.1.5, this field was discarded and replaced by the new `device_id` field, a unique integer defined by the sensor driver in PX4, that modification needs to be reflected in the sensor mapping. Then, in the `px4_config.yaml` file (see App. C.1), where a few parameters of each sensor are defined, the `id` parameter is manually replaced by the `device_id` of each sensor (see App. D.2). The plugin is also modified to map the sensors using the new `device_id` parameter of `px4_config.yaml`.

Moreover, it can be noticed that the plugin translates the `DISTANCE_SENSOR` MAVLink message into the `Range` ROS message from the standard `sensor_msgs` ROS package. That message does not include all the fields of the MAVLink message, but only some of the most relevant, such as `timestamp` (in the form of the ROS message `Header`), `type`, `field_of_view`, `min_range`, `max_range`, and `range`. As expected, the `device_id` and `current_yaw`, added in Sec. 5.1.5, are not included. To solve that, instead of modifying a standard ROS message, a new ROS package called `custom_msgs` was created to include a new ROS message called `RangeYaw`, which includes all the fields of `Range`, plus the new ones (see App. E). Once again, the plugin needs to be modified, not only to replace the `Range` message by `RangeYaw`, but also to retrieve the `device_id` and `current_yaw` data into the new fields of the `RangeYaw` ROS message. All modifications made to the plugin code can be found in App. D.1.

The plugin is responsible for publishing data of the sensors in ROS topics named after the `frame_id` parameter defined in `px4_config.yaml` for each sensor.

The MAVLink message definitions `common.xml` file in the catkin workspace (see App. C.1) also need

to be updated accordingly to the previous modifications of the DISTANCE_SENSOR message.

Other MAVROS Topics and Plugins

The ROS topics where the data from the most relevant MAVLink messages are organized in ROS environment are presented in Tab. 5.5, together with the corresponding ROS messages used to retrieve that data.

Table 5.5: Message flow between MAVLink and MAVROS.

MAVLink Message	ROS Topic	ROS Message
DISTANCE_SENSOR	mavros/distance_sensor/sonar1	custom_msgs/RangeYaw
DISTANCE_SENSOR	mavros/distance_sensor/sonar2	custom_msgs/RangeYaw
DISTANCE_SENSOR	mavros/distance_sensor/laser1	custom_msgs/RangeYaw
DISTANCE_SENSOR	mavros/distance_sensor/laser2	custom_msgs/RangeYaw
DISTANCE_SENSOR	mavros/distance_sensor/lidar	custom_msgs/RangeYaw
LOCAL_POSITION_NED	mavros/local_position/pose	geometry_msgs/PoseStamped
LOCAL_POSITION_NED	mavros/local_position/velocity	geometry_msgs/TwistStamped
SET_POSITION_TARGET_LOCAL_NED	mavros/setpoint_position/local	geometry_msgs/PoseStamped
SET_POSITION_TARGET_LOCAL_NED	mavros/setpoint_velocity/cmd_vel	geometry_msgs/TwistStamped
TRAJECTORY_REPRESENTATION_WAYPOINT	mavros/trajectory/desired	mavros_msgs/Trajectory
TRAJECTORY_REPRESENTATION_WAYPOINT	mavros/trajectory/generated	mavros_msgs/Trajectory
VFR_HUD	/mavros/global_position/compass_hdg	std_msgs/UInt16

Table 5.6 presents the MAVROS or MAVROS_EXTRAS plugins responsible for handling the previous MAVLink messages.

Table 5.6: MAVROS plugins that handle each MAVLink message.

MAVLink Message	MAVROS Plugin File (App. C.1)
DISTANCE_SENSOR	distance_sensor.cpp
LOCAL_POSITION_NED	local_position.cpp
SET_POSITION_TARGET_LOCAL_NED	setpoint_position.cpp
SET_POSITION_TARGET_LOCAL_NED	setpoint_velocity.cpp
TRAJECTORY_REPRESENTATION_WAYPOINT	trajectory.cpp
VFR_HUD	global_position.cpp

5.4 Obstacle Detection and Collision Avoidance Software

ROS provides a flexible framework for writing robotic software. Therefore, given that the MAVROS package is able to provide an interface with MAVLink using plugins to translate data from PX4 (Sec. 5.1) to ROS topics, a new software can be developed in the ROS framework to complete the remaining steps of the S&A system.

The approach of building such a system in ROS is not a novelty, since there is already an open-source ROS package called PX4-Avoidance [80], developed by the PX4 community, to enable obstacle avoidance in autonomous flights of multicopters. This package includes a Local Planner that provides

reactive path adjustments to avoid obstacles during a flight operation, using the three-dimensional Vector Field Histogram avoidance method (3DVFH*) detailed in [79], a Global Planner that plans an optimal path from the start location to the destination, taking into account the entire environment map, using a graph-based method detailed in [129], and, finally, a Safe Landing Planner, that identifies potential hazards on the ground and finds a safe location to land. However, besides not having been tested in fixed-wing UAV, this package tackles the detection of obstacles based on data from a stereo-vision camera hardware and computer vision algorithms, which is not the approach followed in this work for obstacle detection (Sec. 4.1).

Having said that, a new ROS package was created to include the development of a new software prototype using the hardware of Chapter 4. Regarding the choice of programming language, both C++ and Python were considered. C++ is a compiled language known for its better performance (faster execution speed), fine-grained control over memory allocation, and real-time capabilities, but with a more complex syntax that can lead to a slower development. Python is an interpreted language known for having a simpler syntax, which allows a faster development, and a vast ecosystem of libraries, but with worse performance (lower execution speed). Taking this into account, despite the importance of execution speed in this application, Python was chosen in this first phase of development for rapid prototyping and testing of different solutions.

The Python library that provides an interface with ROS is `rospy` [130]. It allows the creation of nodes, subscription and publish of topics, and interaction with ROS services and parameters.

The software prototype was divided in two main parts: **obstacle detection** and **collision avoidance**. The former is responsible for processing the data from the distance sensors and transform it into two-dimensional positions in space for the obstacles. Based on the positions of the obstacles, the latter is responsible for generating, in real-time, a new trajectory for the UAV that avoids the collision with obstacles. The approach followed here was based on the **Vector Field Histogram (VFH)** method, addressed in Sec. 3.2.3.

A multithreading approach was considered, with the `threading` Python module, to allow multiple tasks to run concurrently within a single process, such as different loops to wait for data to be published in ROS topics, process that data, and publish to other topics. This approach was chosen over multiprocessing because the tasks of the system are expected to be I/O-bounded and not CPU-bounded, i.e., the speed of execution is limited by the reception of data from the ROS topics and not by the speed of the processor. As such, threading locks need to be carefully applied where exclusive access to a variable is needed.

The following sections present the general architecture of the software prototype, and a detailed description of the obstacle detection and collision avoidance implementations.

5.4.1 Software Architecture

The architecture of the software prototype is illustrated in Fig. 5.10, including the files, classes, methods (functions) of each class, and the data flow between methods.

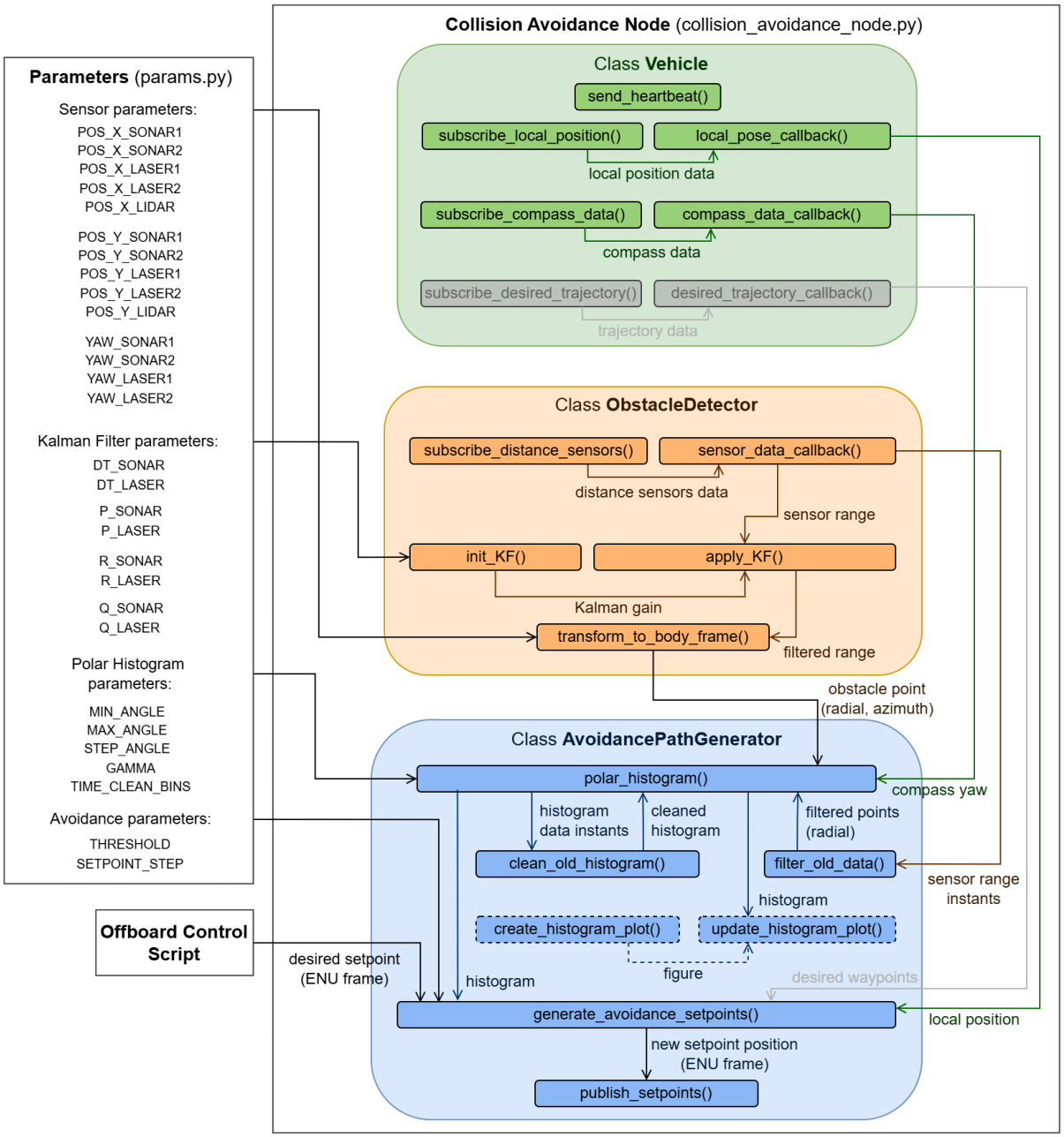


Figure 5.10: Software architecture of the obstacle detection and collision avoidance implementation.

The software is organized in two main files: Parameters (`params.py`), where the main parameters of the system, related to the distance sensors, Kalman filter, polar histogram, and avoidance process are configured/tuned; and the Collision Avoidance Node (`collision_avoidance_node.py`), where all the code developments are included. The latter is organized in three classes: `Vehicle`, `ObstacleDetector`, and `AvoidancePathGenerator`, which exchange data among them. The following sections provide details of the rationale behind each method of these classes.

5.4.2 Implementation of Obstacle Detection

The obstacle detection part of the software is implemented within the class **ObstacleDetector**. It starts with the subscription of the five ROS topics: `mavros/distance_sensor/` (Tab. 5.5), where data

from the distance sensors are being published by MAVROS, using instances of the `rospy.Subscriber()` function from the `rospy` library. As arguments of these functions, not only the ROS topic and message (`RangeYaw`) are specified, but also the callback functions, named `<sensor>_data_callback()`, that are called everytime new data is received on the corresponding topic.

Each callback function is responsible for processing the incoming ROS messages of a topic. First, the current time instant of data reception is saved using the Python standard `Time` utility module for later use in the the `filter_old_data()` method of the class AvoidancePathGenerator. If the sensor range received is within a reasonable range of values for proper detection (Tab. 5.7), a Kalman filter is applied to get the filtered range.

Table 5.7: Considered detection range values per sensor type.

Sensor	Detection Range (m)
Ultrasonic sensor	1-7
Laser rangefinder	1-50
LiDAR	1-50

It was decided to include a one-dimensional Kalman filter in the range measurements of the ultrasonic sensors and laser rangefinders to smooth noisy sensor data and provide a better estimate of the true distance to the obstacles. The theory behind a Kalman filter was previously addressed in Sec. 2.3.1. In this case, there is only one state variable: the distance measured at a known direction. Some tunable parameters of the Kalman filter were included, for each sensor type, in the `params.py` file: the time step, dt, the initial covariance of estimation uncertainty, P, the covariance of measurement noise, R, and the covariance of process noise, Q. The LiDAR was excluded from this kind of Kalman filtering because, as it scans around the environment, each range measurement corresponds to a different direction, meaning that different variables are being measured (distances at different directions).

The implementation of the Kalman filters was made using the `filterpy` Python package: the method `init_KF()` was created to initialize the Kalman filter's variables of this package and the method `apply_KF()` was created to apply the prediction and update steps and return the resulting estimation. For each of the ultrasonic sensors and laser rangefinders, an instance of `init_KF()` was used with the respective parameters. Then, `apply_KF()` is applied to each range measurement of these sensors to return a corresponding filtered range.

Having the filtered range measurements of the sensors, $\hat{d}$, it is necessary to convert them all to the same reference frame, such that they can be seen as 2D positions of obstacles. The `transform_to_body_frame()` method is used to convert the range of each sensor to a 2D position in the body frame of the UAV, in polar coordinates. For this, the position in Cartesian coordinates (x_{sens}, y_{sens}) and orientation, β_{sens}, of each sensor, in the body frame, are specified as parameters. Then, the radial and azimuthal components of the polar coordinates, (r_{obs}, φ_{obs}), of the obstacle detected, in the body frame, are, respectively, obtained from

$$r_{obs} = \sqrt{(\hat{d}\cos(\beta_{sens}) + x_{sens})^2 + (\hat{d}\sin(\beta_{sens}) + y_{sens})^2} \tag{5.1}$$

and

$$\varphi_{obs} = \arctan\left(\frac{\hat{d}\sin(\beta_{sens}) + y_{sens}}{\hat{d}\cos(\beta_{sens}) + x_{sens}}\right). \tag{5.2}$$

The position of the obstacles was chosen to be represented in polar coordinates in the body frame for convenience of the VFH method, since the next step involves representing the obstacle density in space in the form of a polar histogram.

It is worth noting that, in this implementation, the measurements of each sensor are processed independently, i.e., there is no fusion of sensor data.

5.4.3 Implementation of Collision Avoidance

The collision avoidance part of the software is implemented mainly within the class **AvoidancePathGenerator**, using some methods of the class **Vehicle**.

Having access to the position, in body frame, of the obstacle detected by a certain sensor, in polar coordinates, from the class ObstacleDetector, and aiming to apply the VFH method, the first thing to worry about is the generation of a polar histogram representing the obstacle density in space. This is done by the `polar_histogram()` method, which has the following parameters as arguments: the minimum (`MIN_ANGLE`) and maximum (`MAX_ANGLE`) angles of the polar histogram, the size of each histogram bin (`STEP_ANGLE`), and the parameter γ (`GAMMA`), created to adjust the number of neighbors of a main bin.

Figure 5.11 presents the algorithm flowchart of the `polar_histogram()` method to create and continuously update the histogram (with a frequency of 20Hz), using always the most recent obstacle position data available from each sensor. Before that data is used to update the histogram, the `filter_old_data()` method is run to check for sensor data that was updated more than 0.2 seconds ago, and, if so, erase it to avoid using outdated data. Furthermore, the `clean_old_histogram()` method is also run during the histogram update to check through all the data that is currently on the histogram and clean the data that is older than a certain time value defined in the parameter `TIME_CLEAN_BINS`, to avoid, once again, using outdated data.

For the most recent obstacle detected by each sensor, with position (r_{obs}, φ_{obs}) in body frame, the equivalent obstacle position in East-North-Up (ENU) frame $(r_{obs}^{ENU}, \varphi_{obs}^{ENU})$, is computed using the compass heading of the UAV, $\theta_{compass}$. Then, the bin of the polar histogram, in ENU frame, in which φ_{obs}^{ENU} is inserted is found and the corresponding obstacle density, h_k, of that bin is computed using an arbitrary function that, in this case, is given by $h_k = \frac{50 - r_{obs}}{50}$.. The time instant is saved for `clean_old_histogram()`. For safety reasons, the obstacle density of a bin is spread to its neighbor bins, using a function controlled by the parameter γ, as $h_{k\pm a} = h_k$ $(a = 1, ..., \gamma)$.

Optionally, the methods `create_histogram_plot()` and `update_histogram_plot()` can be integrated in the previous algorithm to plot the histogram in a figure updated in real-time. However, it increases the latency of the algorithm, and should not be done in practice, order than for debugging or demonstration purposes.

Next, concurrently to the update of the polar histogram, the most recent histogram data is used by the `generate_avoidance_setpoints()` method to generate, in a loop with frequency of approximately

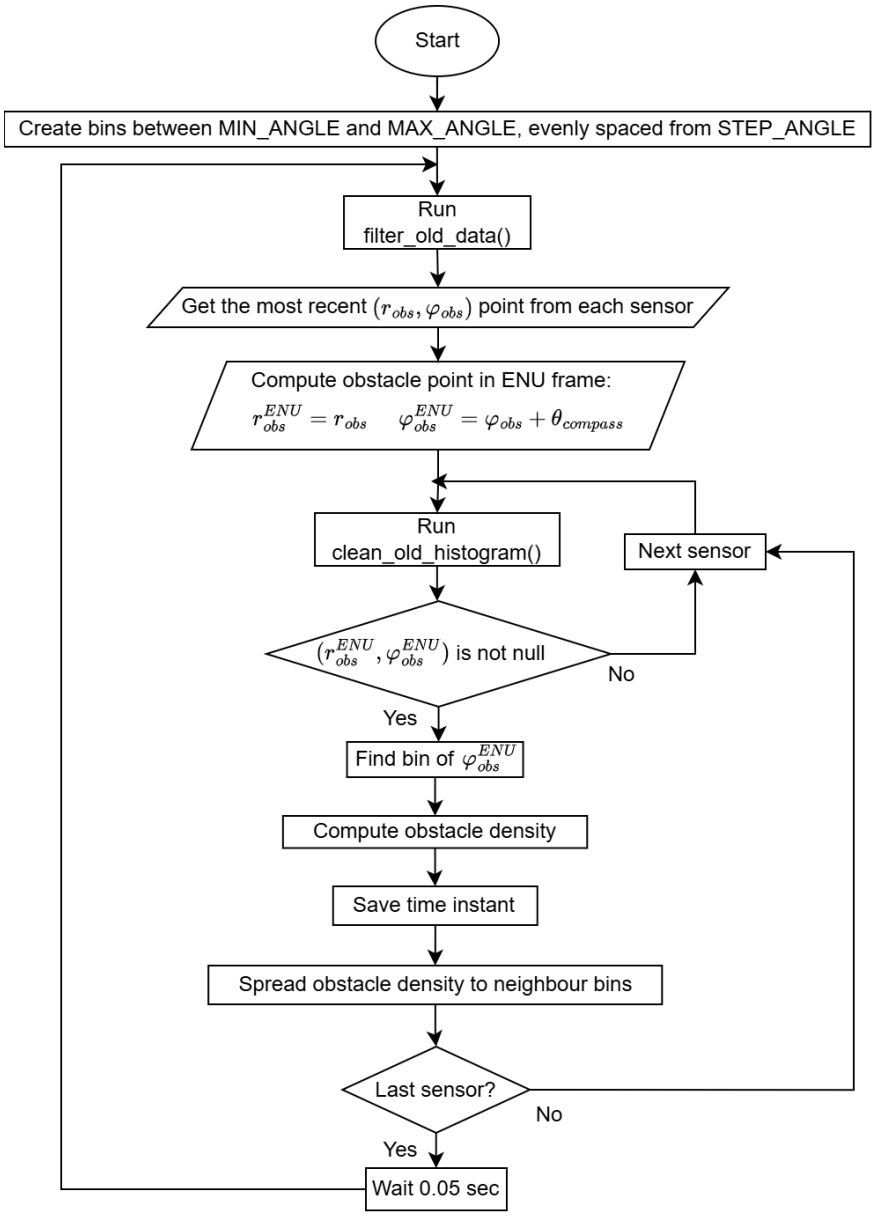


Figure 5.11: Flowchart of the algorithm for creation and continuous update of the polar histogram.

10Hz, new setpoints for the UAV to follow, in order to build a new real-time trajectory for the UAV. This method is fed, as arguments, with desired setpoint positions that make up a desired trajectory for the UAV to follow. The process of sending desired setpoints is usually called offboard navigation control and can be made, for example, by an external Python script and using the Offboard Mode of PX4. The flowchart of the algorithm of `generate_avoidance_setpoints()` is presented in Fig. 5.12.

It is important to note that, since ROS works with a ENU coordinates system, this system is the convention used by the positions published in ROS topics, even though PX4 works with a North-East-Down (NED) coordinates system.

Having the desired setpoint position in ENU frame, the desired direction can be computed. Then, the bins of the polar histogram with an obstacle density below the `THRESHOLD` parameter (available bins) are selected and, from them, the one corresponding to a direction closer to the average between the desired

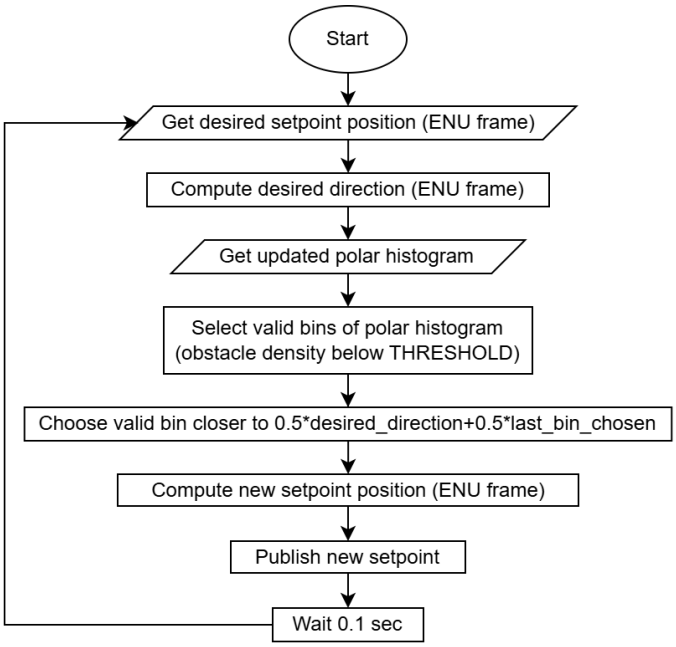


Figure 5.12: Flowchart of the algorithm to generate avoidance setpoints.

direction and the direction followed in the last iteration is chosen. From that direction, a new setpoint velocity in Cartesian coordinates is generated using the `SETPOINT_STEP` parameter, as well as a new setpoint position in the ENU frame using the local position obtained from the `local_position_callback()` method of the class Vehicle.

Finally, it can be chosen to publish the new setpoint position to `mavros/setpoint_position/pose` or publish the new setpoint velocity to `mavros/setpoint_velocity/cmd_vel` using the `publish_setpoint()` and `publish_setpoint_vel()` methods, respectively. The continuous publish of setpoint positions or velocities, at a frequency of 10Hz, is able to generate an avoidance trajectory for the UAV to follow.

In this software prototype, different threads were created and started to execute concurrently the methods responsible for getting the vehicle local position and compass data, getting sensor data, updating the polar histogram and generating avoidance setpoints.

There is an alternative way of generating an avoidance maneuver by using waypoints in the Mission mode of PX4, instead of setpoints in Offboard mode. In the final parts of Secs. 5.1.3 and 5.1.5, the uORB and MAVLink messages that can be used to communicate a new path in waypoints with PX4 are addressed. In this sense, the `subscribe_desired_trajectory()` and `desired_trajectory_callback()` methods were implemented in the class Vehicle to receive the desired waypoints in ENU frame and use them to compute new waypoints in a similar way that is done with setpoints in `generate_avoidance_setpoints()`. However, due to constrains of PX4 related to failsafe configurations, it was not possible to validate this approach, so it was discarded from the software prototype.

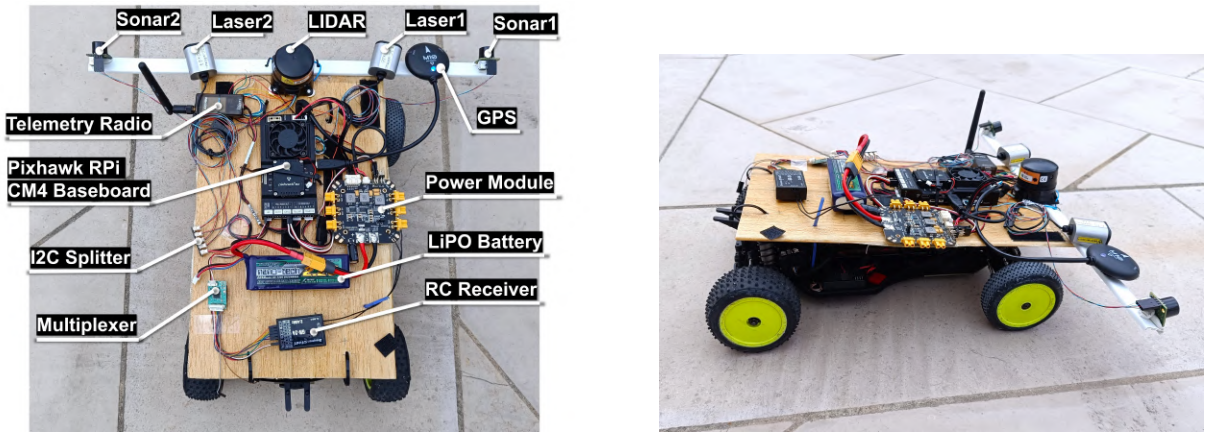
Chapter 6

Validation Tests

To validate the previous hardware and software implementations of the system, a few real-world bench tests were performed. However, given the risks associated with testing new developments in flight, a small Unmanned Ground Vehicle (UGV), hereinafter referred to as rover, was used instead of a fixed-wing UAV. The following sections present the hardware and software setup of the system in the rover and the bench tests performed with static vehicle and static obstacles, static vehicle and a moving obstacle and, finally, moving vehicle and static obstacles.

6.1 Rover System Setup

Regarding the hardware setup for testing in a rover, all the components referred in Chap. 4 were used and connected following the electrical layout of Sec. 4.3, with the exception of the servos connected to the I/O PWM OUT port of the flight controller, which were replaced by the steering servo of the rover. Moreover, a second LiPO battery was included to power exclusively the motor and steering servo of the rover. The complete hardware setup of the rover is presented in Fig. 6.1, with labeled components.



(a) Top view (labeled hardware components).

(b) Right-side view.

Figure 6.1: Rover setup.

As for the software, an important modification had to be made in the PX4 configuration step, ad-

dressed in Sec. 5.1.2. To activate the PX4 support for rovers, the `rover_pos_control` module had to be included in the `default.px4board` file (see App. C.2) through the line:

```
CONFIG_MODULE_ROVER_POS_CONTROL=y
```

This module has been considered deprecated in more recent versions of PX4, being replaced by the `rover_ackermann` and `rover_differencial` modules. However, it was found that the latter do not support Offboard mode, which is important for testing the collision avoidance implementation described in Sec. 5.4.3. To monitor, control and manage the rover in the GCS, the airframe had to be set to "Generic Ground Vehicle (deprecated)" in the Vehicle setup tab of QGroundControl.

6.2 Static Vehicle and Static Obstacles

The first test was conducted with the rover statically positioned in front of three obstacles, labeled A, B and C, also static, arranged at different distances and angles from the rover, as shown in Fig. 6.2a. Obstacles A and B had a cross-section of 0.55m^2, and C had a cross-section of 0.35m^2. The objective of this test was to validate the capabilities of the system to detect obstacles, estimate its relative positions and translate them to the polar histogram of the VFH method.

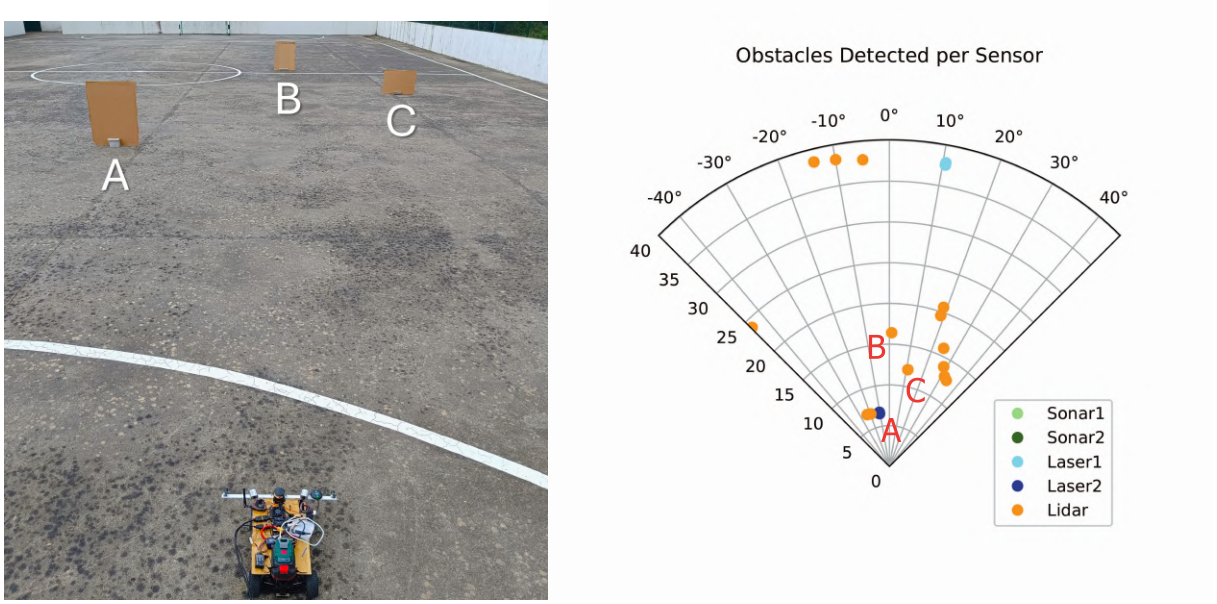
Table 6.1: Obstacle detection and collision avoidance software parameters.

Parameter	Value	Parameter	Value	Parameter	Value
POS_X_SONAR1 (m)	0.2	POS_Y_LASER2 (m)	-0.1	R_LASER	1
POS_Y_SONAR1 (m)	0.2	YAW_LASER2 (°)	-10	Q_LASER	$\begin{bmatrix} 10^{-1} & 0 \\ 0 & 10^{-1} \end{bmatrix}$
YAW_SONAR1 (°)	0	POS_X_LIDAR (m)	0.2	MIN_ANGLE (°)	0
POS_X_SONAR2 (m)	0.2	POS_Y_LIDAR (m)	0	MAX_ANGLE (°)	360
POS_Y_SONAR2 (m)	-0.2	DT_SONAR	0.1	STEP_ANGLE (°)	10
YAW_SONAR1 (°)	0	P_SONAR	7	GAMMA	2
POS_X_LASER1 (m)	0.2	R_SONAR	1	TIME_CLEAN_BINS (s)	0.1
POS_Y_LASER1 (m)	0.1	Q_SONAR	$\begin{bmatrix} 10^{-1} & 0 \\ 0 & 10^{-1} \end{bmatrix}$	THRESHOLD	0.9
YAW_LASER1 (°)	10	DT_LASER	0.05	SETPOINT_STEP	3
POS_X_LASER2 (m)	0.2	P_LASER	50		

The flight controller and companion computer were turned on and the system was run for around 10 seconds, maintaining the rover and obstacles in static positions. The parameters of sensors, Kalman filter, polar histogram, and avoidance chosen for the obstacle detection and collision avoidance software prototype are presented in Tab. 6.1.

The data of the obstacles positions in polar coordinates of the rover body frame, (r_{obs}, φ_{obs}), that were computed by the system during its execution, were saved and plotted in Fig. 6.2b, separated by the sensors that originated them. It can be observed that none of the ultrasonic sensors detected obstacles, which was expect because obstacle A, the only one below the maximum detection distance considered for these sensors (see Tab. 5.7), was not in the direction of any ultrasonic sensor. The laser

rangefinders detected obstacles in precise positions along the directions they were pointed. And the LiDAR presented the most detected obstacles, in different directions.



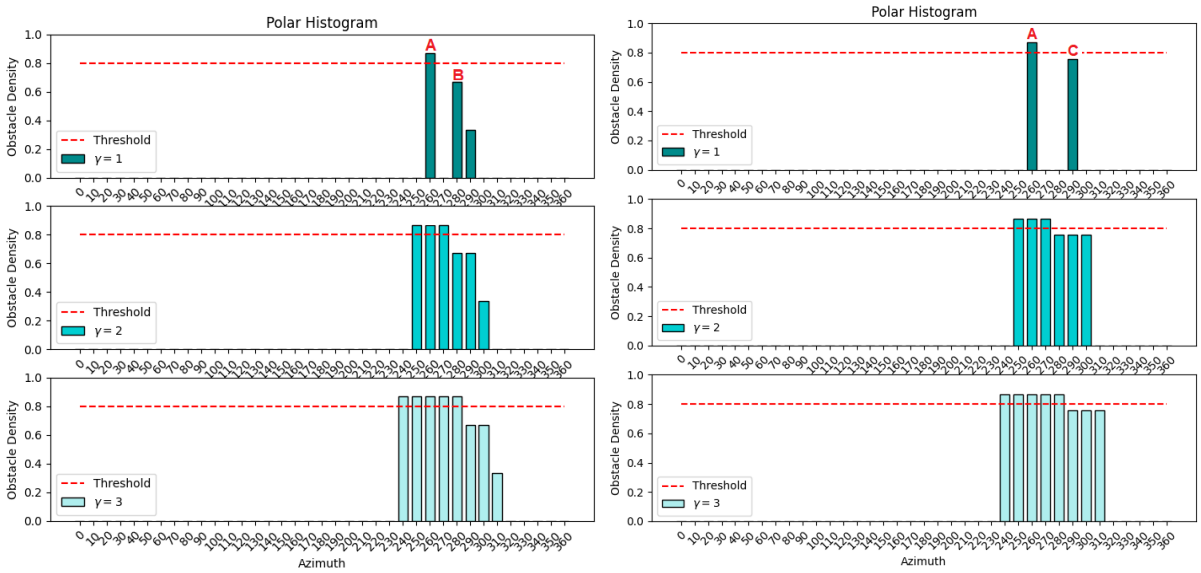
(a) Arrangement of static obstacles A, B and C.

(b) Obstacles detected by sensors during the test.

Figure 6.2: Static vehicle and static obstacles.

Among the points detected, it is possible to identify their correspondence in the real-world scenario and infer about their accuracy. Obstacle A was successfully detected by the Laser2 and the LiDAR, whilst obstacles B and C were only detected by the LiDAR and, consequently, never both at the same time. Moreover, the cluster of LiDAR points in the right-side, as well as the single point in the left-side, and the points in the top of the plot, including the one from Laser1, correspond to the detection of the field walls. This way, it can be concluded that the system was able to detect the target obstacles successfully.

To assess the translation of the obstacles positions to the format of polar histogram, two histograms, built with obstacle density bins ranging from $0°$ to $360°$ and a step angle of $10°$, were plotted from two time instants, t_1 and t_2, when the pairs of obstacles A,B and A,C were detected, respectively. In the first plot of Fig. 6.3a, it can be observed that there are three main bins, one for $260°$ representing the Laser2 detection of obstacle A, another for $280°$ representing the LiDAR detection of obstacle B and, yet, a smaller bin for $290°$ representing the Laser1 detection of the wall. In the first plot of Fig. 6.3b, there is, also, one main bin for $260°$ representing the Laser2 detection of obstacle A and another for $290°$ representing the LiDAR detection of obstacle C. The second and third plots of the previous figures, present the effect of spreading the obstacle density to neighbor bins, which are controlled by γ (`GAMMA` parameter in Tab. 6.1). Having $\gamma = 1$ leads to 0 neighbors, $\gamma = 2$ leads to 2 neighbors for each main bin, and $\gamma = 3$ leads to 3 neighbors for each main bin. An example of a threshold line of 0.8 is also presented - the bins with obstacles densities above 0.8 are considered unavailable and the others are considered available.



(a) $t_1 \approx 1.8s$. (b) $t_2 \approx 5.4s$

Figure 6.3: Polar histograms for $\gamma = 1$, $\gamma = 2$ and $\gamma = 3$.

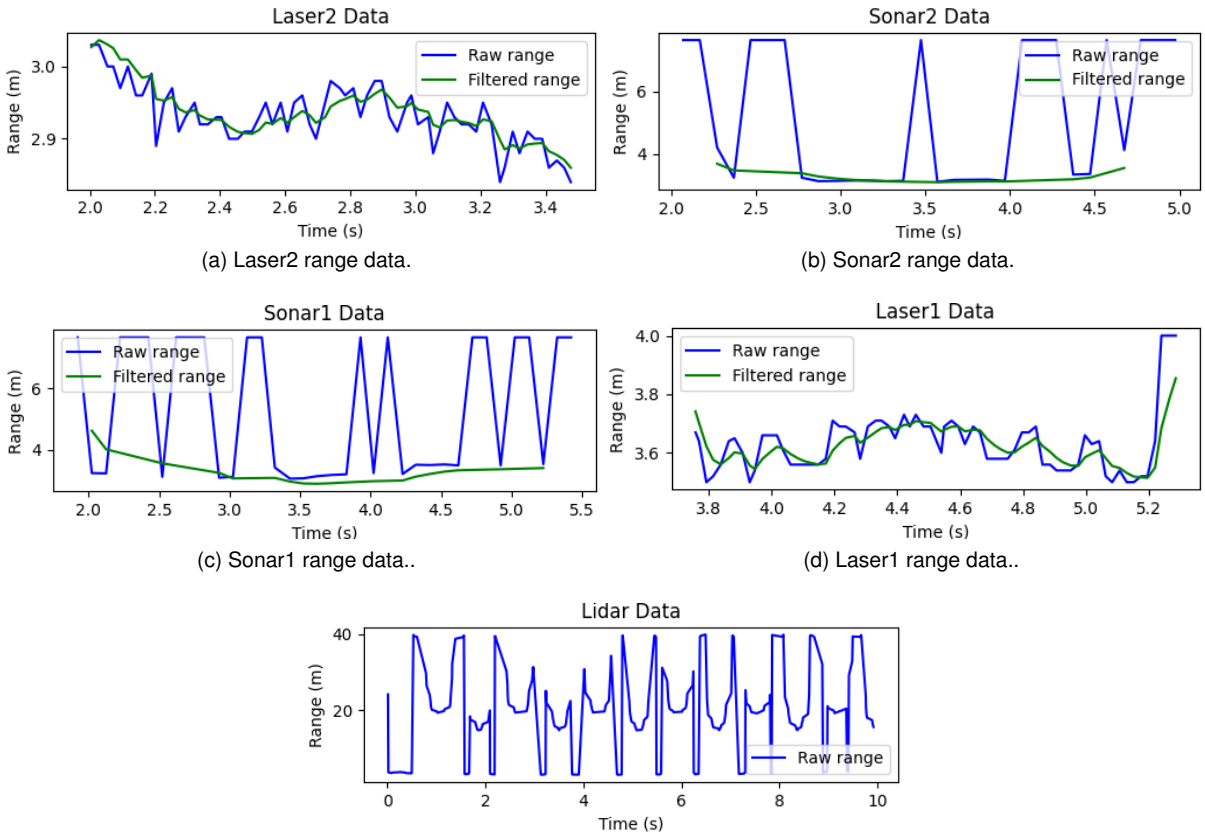
6.3 Static Vehicle and Moving Obstacle

The next test was conducted with the rover statically positioned while a single obstacle with $0.55m^2$ cross-section was moved in front of it, from the left to right, at a speed of approximately $1m/s$ and a distance of around 3m. The objective of this test was to validate the capabilities of the system to detect dynamic obstacles and reflect it in dynamic variations of the polar histogram, from which new directions to follow are computed in the form of chosen bins. The software was tuned with the same parameters of Tab. 6.1, with exception to `THRESHOLD`, which was changed to 0.90, to make the system react only for obstacles below 5m of radial distance.

To assess the characteristics of the sensor data that feed the system, the raw and filtered range measurements of the sensors were saved, truncated to the time intervals where the obstacle is detected, and plotted in Figs. 6.4a, 6.4b, 6.4c, 6.4d, and 6.4e, respectively from the first to the last to detect the obstacle, i.e., Laser2, Sonar2, Sonar1, Laser1 and LiDAR. Given the positions of the sensors in the vehicle, the sequence of detection by the lasers and ultrasonic sensors is as expected for an obstacle moving from left to right.

It can be observed that, for all cases, the obstacle is detected through range measurements between 2.8m and 4m, which is within the tolerated distances. Moreover, the Kalman filter performed reasonably for the lasers and ultrasonic sensors, reducing the noise and dampening the effect of outlier measurements that could lead the system to unnecessary reactions. Measurements of the ultrasonic sensors above 7m were not considered for filtering, since these sensors measure the maximum range (7.65m) when no obstacles are detected. Finally, the LiDAR data was not subject to any filtering process, but presented good results by detecting the obstacle at each scan.

The transformation of the data from all sensors to obstacle positions in polar coordinates, in the vehicle body frame, over the execution of the test, resulted in Fig. 6.5a. The points are labeled by the



(a) Laser2 range data.

(b) Sonar2 range data.

(c) Sonar1 range data..

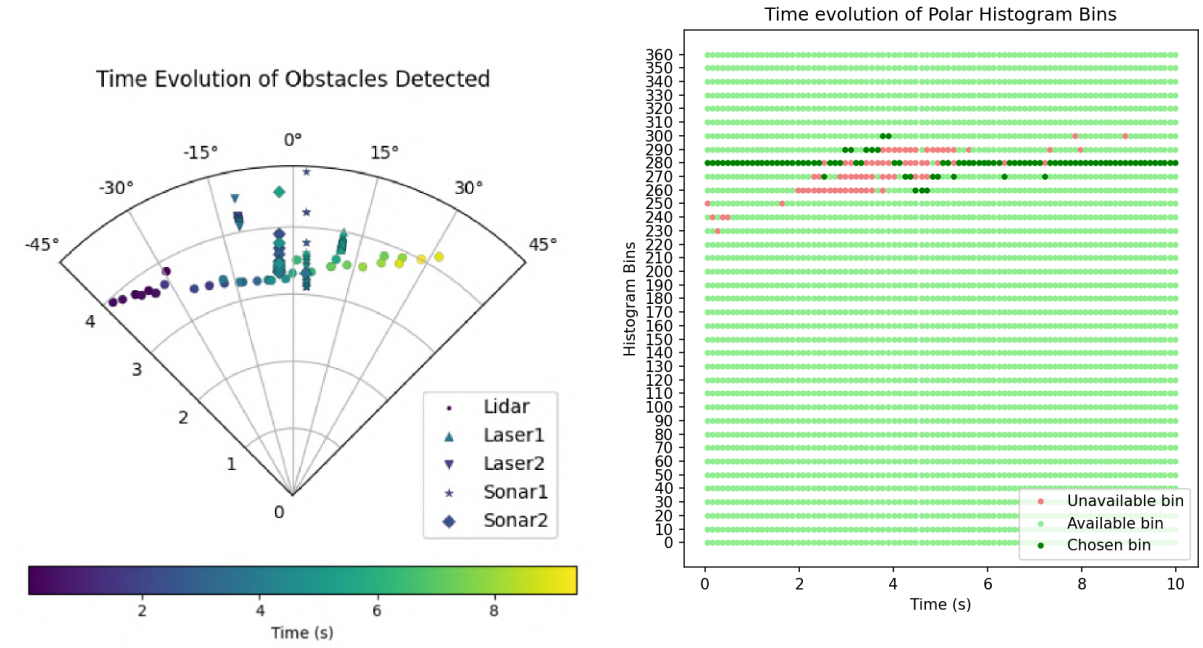
(d) Laser1 range data..

(e) LiDAR range data.

Figure 6.4: Raw and filtered obstacle detection sensors data.

sensors that originated them, such that the cluster of points distributed approximately along the $-11°$ azimuth came from Laser2, the points along the $-3°$ azimuth came from Sonar2, the points along the $3°$ azimuth came from Sonar1, the points along the $11°$ azimuth came from Laser1 and the other scattered points came from LiDAR. Once again, the evolution of the point positions in time is in accordance with the trajectory of the obstacle from left to right.

Given that, at each time instant, the system translated the previous obstacle positions into polar histograms, labeling the bins with obstacle density above the `THRESHOLD` as unavailable, and the others as available, that classification was saved and plotted over time in Fig. 6.5b. Moreover, the bin chosen by the system at each instant is also presented. In this case, the desired setpoints arbitrarily imputed to the system were such that the desired direction was of $280°$. It can be observed that, as soon as the obstacle covered that direction, the corresponding bin became unavailable and the system was forced to chose another bin direction. Throughout the time, as the obstacle moved to the right-side, the bins affected were dynamically blocked and released, while the system was dynamically choosing the available bin closer to the average between the desired direction and the bin chosen in the previous instant. This way, it can be concluded that the system successfully detects a dynamic obstacle and presents an intended solution to avoid a collision.



(a) Position of obstacles detected in polar coordinates body frame, over time.

(b) Available (light green), unavailable (red), and chosen bins (dark green) of the polar histogram, at each time instant.

Figure 6.5: S&A system outputs as a function of time for static vehicle test.

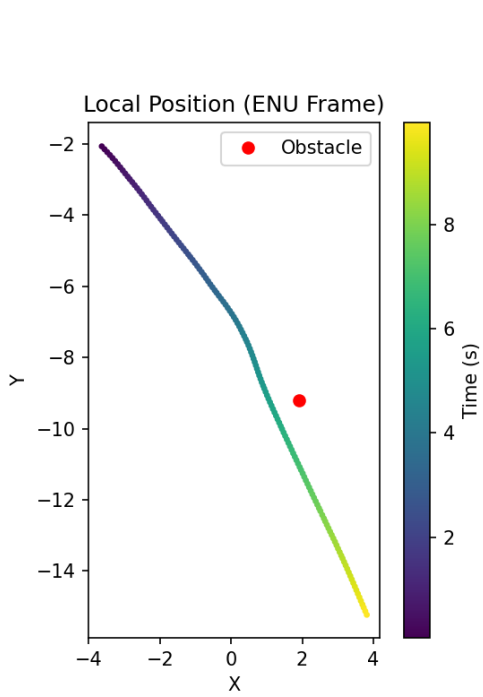
6.4 Moving Vehicle and Static Obstacles

Finally, a test was performed with the rover moving towards one static obstacle with 0.5m^2 of cross-section. The objective of this test was to validate the capabilities of the system to, based on a dynamic detection of obstacles, perform a real-world collision avoidance using all the features of the implemented version of the VFH method. The software was tuned with the parameters of Tab. 6.1, changing to $\gamma = 4$ and `TIME_CLEAN_BINS=1`, to enhance safety. Regarding the avoidance part of the software, addressed in detail in Sec. 5.4.3, it was decided to use only the function to publish setpoint positions.

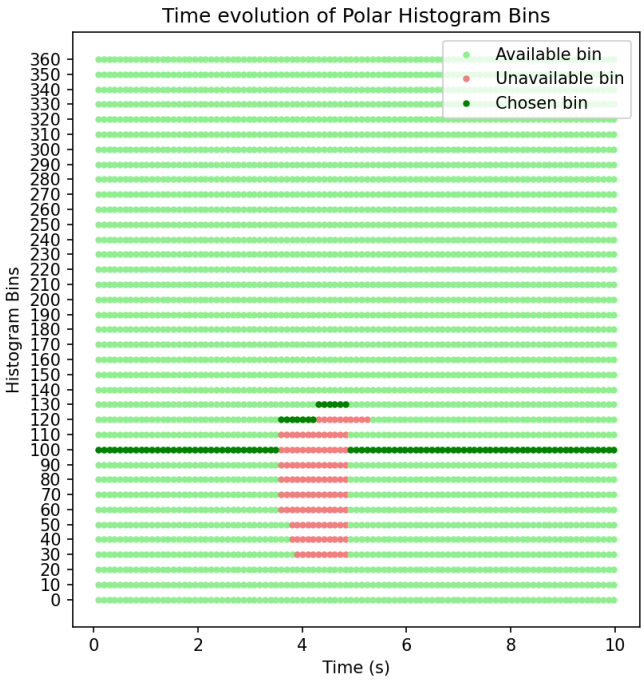
After running the system, and selecting Offboard mode in the GCS, the rover performed the test at an average speed of $2m/s$. To present the results, the local position of the rover, in earth-fixed ENU frame, was plotted, over time, in the Fig. 6.6a, together with the approximated position of the obstacle. This plot shows that the rover was following a linear trajectory towards the obstacle and, around 3m before colliding to it, a small deviation to the right on the trajectory was made, allowing the successful avoidance of a collision with the obstacle.

Similarly to the previous test, the background computation of the classification of available, unavailable and chosen polar histogram bins, at each time instant, were plotted in Fig. 6.6b. It can be observed that, initially, the rover was physically aligned to the desired direction of 100° and, at the time instant $t = 3.6$s, the detection of the obstacle led to the blocking of bins from 60° to 110°, forcing the system to choose the direction of 120° and start the avoidance trajectory. In the following seconds, the blocked bins eventually evolved to the range from 30° to 120°, forcing the vehicle to follow the direction of 130°,

until a moment when the polar histogram data is cleaned and, since no further obstacles were detected, the system returned to the desired direction of 100° , finishing the avoidance trajectory.



(a) Local position of Vehicle in earth-fixed ENU frame over time.



(b) Available (light green), unavailable (red), and chosen bins (dark green) of the polar histogram, at each time instant.

Figure 6.6: S&A system outputs as a function of time for moving vehicle test.

It is important to note that, as shown in Fig. 6.6a, the desired direction followed before starting the avoidance maneuver and the desired direction followed after passing the obstacle was not the same, even though the system published setpoint positions in direction of 100° in both cases. This can be due to a faulty calibration of the compass, which led to inaccurate heading measures during the test.

This way, it can be concluded that the system was able to perform obstacle detection and collision avoidance in the presence of a single obstacle, at a speed around $2m/s$, by generating a trajectory of avoidance setpoint position at a frequency of 10Hz.

Chapter 7

Conclusions

7.1 Achievements

This work was carried out as part of a comprehensive study to enhance the flight safety of small fixed-wing UAV through the search and application of Sense and Avoid technologies. This way, it focused on the development of a system able to perform non-cooperative obstacle detection and collision avoidance.

A hardware implementation was proposed for the obstacle detection and collision avoidance system using two ultrasonic sensors, two laser rangefinders, and one LiDAR as detection sensors, a Pixhawk 6X as flight controller, a Raspberry Pi CM4 as companion computer, and many other components to complete the system, such as GPS module, LiPO battery, power module, ESC, motor, control surface servos, telemetry module, and radio receiver and transmitter, all properly connected in a proposed electrical layout.

A software implementation for the system was proposed next. First, the flight control software, PX4, was adapted and dissected into the most important parts for this system, in particular, the procedure to properly configure it, the uORB messaging bus for internal communication, with emphasis on the `distance_sensor` message, the drivers that handle the operation and data of the sensors, and the MAVLink protocol used for external communication with the GCS and companion computer, once again with emphasis on handling the `DISTANCE_SENSOR` message. Then, the communication between PX4 and the companion computer with MAVROS was addressed. This way, a MAVROS plugin was adapted to receive the data from the sensors, properly distinguishing them. Finally, the software prototype to implement the VFH method and perform the obstacle detection and collision avoidance was developed, in ROS environment. It implemented obstacle detection by converting the range measurements from the sensors in obstacle positions in polar coordinates, and collision avoidance by generating a polar histogram from those positions, based on which the avoidance trajectory is determined as a generation of setpoints at the chosen directions.

Finally, the system was validated through real-world tests in a rover. The first test, with the vehicle statically in front of three static obstacles, showed that the system was able to determine obstacle positions and translate them to a polar histogram. The second, with static vehicle and one moving obstacle,

showed the capabilities of the system to detect dynamic obstacles and the process of choosing, in real-time, the direction to follow, based on polar histogram bins. The last test, with the rover moving towards one static obstacle, showed the overall S&A system capabilities on performing obstacle detection and collision avoidance through a small deviation maneuver from trajectory setpoint positions published at a rate of 10Hz.

7.2 Future Work

The present work leaves an open path to some future improvements and further developments, regarding both obstacle detection and collision avoidance for fixed-wing UAVs.

Based on the work that was done to handle multiple sensors in PX4 and receive their data separately in the companion computer, it would be possible to work on the implementation of sensor fusion techniques, resorting, for example, to more advanced applications of the Kalman filter and its variants, which could leverage features such as tracking of obstacles through estimation of their position, velocity and acceleration. In this regard, there is also the future possibility to change the detection sensors to vision-based sensors, as stereo cameras, and obtain the position of the obstacles through the application of computer vision algorithms, opening the possibility for a three-dimensional detection.

Regarding collision avoidance, the VFH method implemented here served has a preliminary avoidance solution, presenting some lack of trajectory optimization. Thus, more advanced methods, such as the variants of VFH (e.g VFH+ and VFH*), could be implemented to provide a more reliable and optimized solution. Then, real flight tests with a fixed-wing UAV could be done to properly validate the system.

Bibliography

[1] I. Heiets, Y.-W. Kuo, J. La, R. C. K. Yeun, and W. Verhagen. Future Trends in UAV Applications in the Australian Market. *Aerospace*, 10(6):555, 2023. doi: 10.3390/aerospace10060555.

[2] P. G. Fahlstrom, T. J. Gleason, M. H. Sadraey, P. Belobaba, J. Cooper, and A. Seabridge, editors. *Introduction to UAV Systems*. Aerospace Series. Wiley, 5th edition, 2022. ISBN: 9781119802617.

[3] N. A. Khan, N. Z. Jhanjhi, S. N. Brohi, R. S. A. Usmani, and A. Nayyar. Smart traffic monitoring system using Unmanned Aerial Vehicles (UAVs). *Computer Communications*, 157:434–443, 2020. doi: 10.1016/j.comcom.2020.04.049.

[4] S. Berrahal, J.-H. Kim, S. Rekhis, N. Boudriga, D. Wilkins, and J. Acevedo. Border surveillance monitoring using Quadcopter UAV-Aided Wireless Sensor Networks. *Journal of Communications Software and Systems*, 12:67–82, 2016. doi: 10.24138/jcomss.v12i1.92.

[5] J. V. R. Sousa and P. J. V. Gamboa. Aerial Forest Fire Detection and Monitoring Using a Small UAV. *KnE Engineering*, 5(6):242–256, 2020. doi: 10.18502/keg.v5i6.7038.

[6] A. Khaloo, D. Lattanzi, K. Cunningham, R. Dell'Andrea, and M. Riley. Unmanned aerial vehicle inspection of the Placer River Trail Bridge through image-based 3D modelling. *Structure and Infrastructure Engineering*, 14:124–136, 2018. doi: 10.1080/15732479.2017.1330891.

[7] A. Kulsinskas, P. Durdevic, and D. Ortiz-Arroyo. Internal Wind Turbine Blade Inspections Using UAVs: Analysis and Design Issues. *Energies*, 14:294, 2021. doi: 10.3390/en14020294.

[8] T. Özaslan, S. Shen, Y. Mulgaonkar, N. Michael, and V. Kumar. *Inspection of Penstocks and Featureless Tunnel-like Environments Using Micro UAVs*, pages 123–136. Springer International Publishing, Cham, 2015. ISBN 978-3-319-07488-7. doi: 10.1007/978-3-319-07488-7_9.

[9] Y. Tan, S. Li, H. Liu, P. Chen, and Z. Zhou. Automatic inspection data collection of building surface based on BIM and UAV. *Automation in Construction*, 131:103881, 2021. doi: 10.1016/j.autcon.2021.103881.

[10] R. A. Clark, G. Punzo, C. N. MacLeod, G. Dobie, R. Summan, and G. Bolton. Autonomous and scalable control for remote inspection with multiple aerial vehicles. *Robotics and Autonomous Systems*, 87:258–268, 2017. doi: 10.1016/j.robot.2016.10.012.

[11] D. C. Tsouros, S. Bibi, and P. G. Sarigiannidis. A Review on UAV-Based Applications for Precision Agriculture. *Information*, 10(11):349, 2019. doi: 10.3390/info10110349.

[12] A. Gupta, T. Afrin, E. Scullt, and N. Yodo. Advances of UAVs toward Future Transportation: The State-of-the-Art, Challenges, and Opportunities. *Future Transportation*, 1(2):326–350, 2021. doi: 10.3390/futuretransp1020019.

[13] UAV Market Size & Share Analysis - Growth Trends & Forecasts (2024 - 2029). Technical report, Mordor Intelligence, 2024. URL https://www.mordorintelligence.com/industry-reports/uav-market. (accessed on 18 Feb 2024).

[14] Unmanned Aerial Vehicle (UAV) Market. Technical report, ZION Market Research, 2023. URL https://www.zionmarketresearch.com/report/unmanned-aerial-vehicle-uav-market. (accessed on 18 Feb 2024).

[15] Unmanned Aerial Vehicle (UAV) Market. Technical report, Fortune Business Intelligence, 2023. URL https://www.fortunebusinessinsights.com/industry-reports/unmanned-aerial-vehicle-uav-market-101603. (accessed on 18 Feb 2024).

[16] Unmanned Aerial Vehicle (UAV) Market Growth Dynamic Report 2023-2030. Technical report, SNS Insider Research, 2023. URL https://www.researchgate.net/publication/372439486_Unmanned_Aerial_Vehicle_UAV_Market_Growth_Dynamic_Report_2023-2030. (accessed on 18 Feb 2024).

[17] TEKEVER. AR4, 2014. URL https://www.tekever.com/models/ar4/. (accessed on 4 Dez 2024).

[18] Code of Federal Regulations (CFR) Title 14 Part 91.113, 2024. URL https://www.ecfr.gov/current/title-14/chapter-I/subchapter-F/part-91/subpart-B/subject-group-ECFRe4c59b5f5506932/section-91.113. (accessed on 18 Feb 2024).

[19] D. Franjkovic, T. Bucak, and N. Hoti. Ground Proximity Warning System - GPWS. *Promet - Traffic&Transportation*, 11(5):293–301, 1999. URL https://traffic.fpz.hr/index.php/PROMTT/article/view/1143.

[20] D. E. Swihart, A. F. Bartfield, E. M. Griffin, R. C. Lehmann, S. C. Whitcomb, B. Flynn, M. A. Skoog, and K. E. Processor. Automatic Ground Collision Avoidance System design, integration, & flight test. *IEEE Aerospace and Electronic Systems Magazine*, 26(5):4–11, 2011. doi: 10.1109/MAES.2011.5871385.

[21] M. Orefice, V. D. Vito, and G. Torrano. *Sense and Avoid: Systems and Methods*. 12 2015. ISBN 9780470754405. doi: 10.1002/9780470686652.eae1149.

[22] X. Yu and Y. Zhang. Sense and avoid technologies with applications to unmanned aircraft systems: Review and prospects. *Progress in Aerospace Sciences*, 74:152–166, 2015. doi: 10.1016/j.paerosci.2015.01.001.

[23] B. N. Chand, P. Mahalakshmi, and V. P. S. Naidu. Sense and avoid technology in unmanned aerial vehicles: A review. In *2017 International Conference on Electrical, Electronics, Communication, Computer, and Optimization Techniques (ICEECCOT)*, pages 512–517, 2017. doi: 10.1109/ICEECCOT.2017.8284558.

[24] M. Skowron, W. Chmielowiec, K. Glowacka, M. Krupa, and A. Srebro. Sense and avoid for small unmanned aircraft systems: Research on methods and best practices. *Proceedings of the Institution of Mechanical Engineers, Part G: Journal of Aerospace Engineering*, 233(16):6044–6062, 2019. doi: 10.1177/0954410019867802.

[25] N. Alturas. Modeling and Optimization of an Obstacle Detection System for Small UAVs. MSc Thesis in Aerospace Engineering, Instituto Superior Técnico, Lisboa, Portugal, January 2021.

[26] P. Serrano. Optimization of Obstacle Detection for Small UAVs. MSc Thesis in Aerospace Engineering, Instituto Superior Técnico, Lisboa, Portugal, June 2022.

[27] M. Portugal. Optimal Multi-Sensor Collision Avoidance System for Small Fixed-Wing UAV. MSc Thesis in Aerospace Engineering, Instituto Superior Técnico, Lisboa, Portugal, November 2023.

[28] J. Alves. Path Planning and Collision Avoidance Algorithms for Small RPAS. MSc Thesis in Aerospace Engineering, Instituto Superior Técnico, Lisboa, Portugal, June 2017.

[29] C. Munoz, A. Narkawicz, and J. Chamberlain. A TCAS-II Resolution Advisory Detection Algorithm. In *AIAA Guidance, Navigation, and Control (GNC) Conference*, Boston, Massachusetts, USA, August 2013. doi: 10.2514/6.2013-4622.

[30] J. K. Kuchar and A. C. Drumm. The traffic alert and collision avoidance system. *Lincoln Laboratory Journal*, 16(2):277, 2007.

[31] J. K. Kuchar. Safety Analysis Methodology for Unmanned Aerial Vehicle (UAV) Collision Avoidance Systems. In *6th USA/Europe Air Traffic Management R&D Seminar*, Baltimore, MD, USA, June 2005.

[32] H.-C. Lee. Implementation of collision avoidance system using TCAS II to UAVs. In *24th Digital Avionics Systems Conference*, volume 2, page 9, Washington, D.C, USA, November 2005. doi: 10.1109/DASC.2005.1563410.

[33] J. Asmat, B. Rhodes, J. Umansky, C. Villavicencio, A. Yunas, G. Donohue, and A. Lacher. UAS Safety: Unmanned Aerial Collision Avoidance System (UCAS). In *2006 IEEE Systems and Information Engineering Design Symposium*, pages 43–49, Charlottesville, VA, USA, April 2006. doi: 10.1109/SIEDS.2006.278711.

[34] L. Lin, Y. Cheng, L. Zhiyong, L. Yinchuan, and L. Nisi. A UAV Collision Avoidance System Based on ADS-B. In *Proceedings of the 5th China Aeronautical Science and Technology Conference*, pages 159–167, Wuzhen, Zhejiang, China, 2022. Springer Singapore. ISBN 978-981-16-7423-5. doi: 10.1007/978-981-16-7423-5_16.

[35] Y. Lin and S. Saripalli. Sense and avoid for Unmanned Aerial Vehicles using ADS-B. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 6402–6407, Seattle, WA, USA, May 2015. doi: 10.1109/ICRA.2015.7140098.

[36] M. Z. Butt, N. Nasir, and R. B. A. Rashid. A review of perception sensors, techniques, and hardware architectures for autonomous low-altitude UAVs in non-cooperative local obstacle avoidance. *Robotics and Autonomous Systems*, 173:104629, 2024. doi: 10.1016/j.robot.2024.104629.

[37] Y. K. Kwag and C. H. Chung. UAV based collision avoidance radar sensor. In *2007 IEEE International Geoscience and Remote Sensing Symposium*, pages 639–642, Barcelona, Spain, July 2007. doi: 10.1109/IGARSS.2007.4422877.

[38] S. Kemkemian, M. Nouvel-Fiani, P. Cornic, and P. Garrec. A MIMO radar for Sense and Avoid function: A fully static solution for UAV. In *11th International Radar Symposium*, pages 1–4, Vilnius, Lithuania, June 2010. ISBN 978-1-4244-5614-7.

[39] S. Shougaev and M. Idan. Laser Range-Finder Based Obstacle Avoidance for Quadcopters. In *AIAA Scitech 2019 Forum*, San Diego, CA, USA, January 2019. doi: 10.2514/6.2019-1408.

[40] S. Ramasamy, R. Sabatini, A. Gardi, and J. Liu. LIDAR obstacle warning and avoidance system for unmanned aerial vehicle sense-and-avoid. *Aerospace Science and Technology*, 55:344–358, 2016. doi: 10.1016/j.ast.2016.05.020.

[41] A. Moffatt, E. Platt, B. Mondragon, A. Kwok, D. Uryeu, and S. Bhandari. Obstacle detection and avoidance system for small uavs using a lidar. In *2020 International Conference on Unmanned Aircraft Systems (ICUAS)*, pages 633–640, Athens, Greece, September 2020. doi: 10.1109/ICUAS48674.2020.9213897.

[42] A. Singletary, K. Klingebiel, J. Bourne, A. Browning, P. Tokumaru, and A. Ames. Fast Obstacle Avoidance Motion in Small Quadcopter operation in a Cluttered Environment. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, page 8129–8136, Prague, Czech Republic, September 2021. doi: 10.1109/IROS51168.2021.9636670.

[43] C. S. Gadde, M. S. Gadde, N. Mohanty, and S. Sundaram. Fast Obstacle Avoidance Motion in Small Quadcopter operation in a Cluttered Environment. In *2021 IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT)*, pages 1–6, Bangalore, India, July 2021. doi: 10.1109/CONECCT52877.2021.9622631.

[44] N. Gageik, P. Benz, and S. Montenegro. Obstacle Detection and Collision Avoidance for a UAV With Complementary Low-Cost Sensors. *IEEE Access*, 3:599–609, 2015. doi: 10.1109/ACCESS.2015.2432455.

[45] J. Tang, S. Lao, and Y. Wan. Systematic Review of Collision-Avoidance Approaches for Unmanned Aerial Vehicles. *IEEE Systems Journal*, 16(3):4356–4367, 2022. doi: 10.1109/JSYST.2021.3101283.

[46] L. Lu, G. Fasano, A. Carrio, M. Lei, H. Bavle, and P. Campoy. A comprehensive survey on non-cooperative collision avoidance for micro aerial vehicles: Sensing and obstacle detection. *Journal of Field Robotics*, 40(6):1697–1720, 2023. doi: 10.1002/rob.22189.

[47] A. Al-Kaff, F. García, D. Martín, A. De La Escalera, and J. M. Armingol. Obstacle Detection and Avoidance System Based on Monocular Camera and Size Expansion Algorithm for UAVs. *Sensors*, 17(5), 2017. doi: 10.3390/s17051061.

[48] H.-Y. Lin and X.-Z. Peng. Autonomous Quadrotor Navigation With Vision Based Obstacle Avoidance and Path Planning. *IEEE Access*, 9:102450–102459, 2021. doi: 10.1109/ACCESS.2021.3097945.

[49] Y. Xiao, X. Lei, and S. Liao. Research on UAV Multi-Obstacle Detection Algorithm based on Stereo Vision. In *2019 IEEE 3rd Information Technology, Networking, Electronic and Automation Control Conference (ITNEC)*, pages 1241–1245, Chengdu, China, March 2019. doi: 10.1109/ITNEC.2019.8729183.

[50] A. Finn and S. Franklin. Acoustic sense & avoid for UAV's. In *2011 Seventh International Conference on Intelligent Sensors, Sensor Networks and Information Processing*, pages 586–589, Adelaide, Australia, December 2011. doi: 10.1109/ISSNIP.2011.6146555.

[51] M. S. Grewal and A. P. Andrews. *Kalman Filtering: Theory and Practice Using MATLAB*. John Wiley & Sons, Inc, New York, USA, 2nd edition, 2001. ISBN: 0-471-26638-8.

[52] F. Fayad and V. Cherfaoui. Tracking objects using a laser scanner in driving situation based on modeling target shape. In *2007 IEEE Intelligent Vehicles Symposium*, pages 44–49, 2007. doi: 10.1109/IVS.2007.4290089.

[53] C. G. Prevost, A. Desbiens, and E. Gagnon. Extended Kalman Filter for State Estimation and Trajectory Prediction of a Moving Object Detected by an Unmanned Aerial Vehicle. In *2007 American Control Conference*, pages 1805–1810, 2007. doi: 10.1109/ACC.2007.4282823.

[54] W. Li and C. Zhang. Markov Chain Analysis. In A. Kobayashi, editor, *International Encyclopedia of Human Geography (Second Edition)*, pages 407–412. Elsevier, Oxford, second edition edition, 2009. ISBN 978-0-08-102296-2. doi: 10.1016/B978-0-08-102295-5.10403-2.

[55] J. Feng, J. Zhang, G. Zhang, S. Xie, Y. Ding, and Z. Liu. UAV Dynamic Path Planning Based on Obstacle Position Prediction in an Unknown Environment. *IEEE Access*, 9:154679–154691, 2021. doi: 10.1109/ACCESS.2021.3128295.

[56] P. E. Hart, N. J. Nilsson, and B. Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968. doi: 10.1109/TSSC.1968.300136.

[57] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1): 269–271, 1959. doi: 10.1007/bf01386390.

[58] D. Mandloi, R. Arya, and A. K. Verma. Unmanned aerial vehicle path planning based on A* algorithm and its variants in 3d environment. *International Journal of System Assurance Engineering and Management*, 12:990–1000, 2021. doi: 10.1007/s13198-021-01186-9.

[59] K. Daniel, A. Nash, S. Koenig, and A. Felner. Theta*: Any-Angle Path Planning on Grids. *Journal of Artificial Intelligence Research*, 39:533–579, 2010. doi: 10.1613/jair.2994.

[60] A. Nash, S. Koenig, and C. Tovey. Lazy Theta*: Any-Angle Path Planning and Path Length Analysis in 3D. *Proceedings of the AAAI Conference on Artificial Intelligence*, 24(1):147–154, 2010. doi: 10.1609/aaai.v24i1.7566.

[61] S. M. LaValle. Rapidly-exploring random trees: a new tool for path planning. *The annual research report*, 1998. URL `https://api.semanticscholar.org/CorpusID:14744621`.

[62] Y. Dong, E. Camci, and E. Kayacan. Faster RRT-based Nonholonomic Path Planning in 2D Building Environments Using Skeleton-constrained Path Biasing. *Journal of Intelligent & Robotic Systems*, 89(3):387–401, Mar 2018. doi: 10.1007/s10846-017-0567-9.

[63] S. Karaman, M. R. Walter, A. Perez, E. Frazzoli, and S. Teller. Anytime Motion Planning using the RRT*. In *2011 IEEE International Conference on Robotics and Automation*, pages 1478–1483, Shanghai, China, May 2011. doi: 10.1109/ICRA.2011.5980479.

[64] J. Kuffner and S. LaValle. RRT-connect: An efficient approach to single-query path planning. In *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No.00CH37065)*, pages 995–1001 vol.2, San Fransisco, CA, USA, April 2000. doi: 10.1109/ROBOT.2000.844730.

[65] H. Yu, F. Zhang, P. Huang, C. Wang, and L. Yuanhao. Autonomous Obstacle Avoidance for UAV based on Fusion of Radar and Monocular Camera. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 5954–5961, Las Vegas, NV, USA, October 2020. doi: 10.1109/IROS45743.2020.9341432.

[66] F. Yang, X. Fang, F. Gao, X. Zhou, H. Li, H. Jin, and Y. Song. Obstacle Avoidance Path Planning for UAV Based on Improved RRT Algorithm. *Discrete Dynamics in Nature and Society*, 2022: 4544499, 2022. doi: 10.1155/2022/4544499.

[67] M. Dorigo and T. Stutzle. *Ant Colony Optimization*. MIT Press, 2004. ISBN: 9780262256032.

[68] A. Chakravarthy and D. Ghose. Obstacle avoidance in a dynamic environment: a collision cone approach. *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans*, 28(5):562–574, 1998. doi: 10.1109/3468.709600.

[69] P. Fiorini and Z. Shiller. Motion Planning in Dynamic Environments Using Velocity Obstacles. *The International Journal of Robotics Research*, 17(7):760–772, 1998. doi: 10.1177/027836499801700706.

[70] J. Park and N. Cho. Collision Avoidance of Hexacopter UAV Based on LiDAR Data in Dynamic Environment. *Remote Sensing*, 12(6), 2020. doi: 10.3390/rs12060975.

[71] Y. I. Jenie, E.-J. V. Kampen, C. C. de Visser, J. Ellerbroek, and J. M. Hoekstra. *Three-Dimensional Velocity Obstacle Method for UAV Deconflicting Maneuvers*. doi: 10.2514/6.2015-0592.

[72] C. Y. Tan, S. Huang, K. K. Tan, and R. S. H. Teo. Three Dimensional Collision Avoidance for Multi Unmanned Aerial Vehicles Using Velocity Obstacle. *Journal of Intelligent & Robotic Systems*, 97 (1):227–248, Jan 2020. doi: 10.1007/s10846-019-01055-5.

[73] Z. Lin, L. Castano, E. Mortimer, and H. Xu. Fast 3D Collision Avoidance Algorithm for Fixed Wing UAS. *Journal of Intelligent & Robotic Systems*, 97(3):577–604, Mar 2020. doi: 10.1007/s10846-019-01037-7.

[74] Y. Du, X. Zhang, and Z. Nie. A Real-Time Collision Avoidance Strategy in Dynamic Airspace Based on Dynamic Artificial Potential Field Algorithm. *IEEE Access*, 7:169469–169479, 2019. doi: 10.1109/ACCESS.2019.2953946.

[75] Y. Yan, Z. Lv, J. Yuan, and S. Zhang. Obstacle Avoidance for Multi-UAV system with Optimized Artifical Potencial Field Algorithm. *International Journal of Robotics and Automation*, 36, 01 2021. doi: 10.2316/J.2021.206-0610.

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

[77] I. Ulrich and J. Borenstein. VFH+: reliable obstacle avoidance for fast mobile robots. In *Proceedings. 1998 IEEE International Conference on Robotics and Automation (Cat. No.98CH36146)*, volume 2, pages 1572–1577 vol.2, Leuven, Belgium, May 1998. doi: 10.1109/ROBOT.1998.677362.

[78] I. Ulrich and J. Borenstein. VFH/sup */: local obstacle avoidance with look-ahead verification. In *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No.00CH37065)*, volume 3, pages 2505–2511, San Fransisco, CA, USA, April 2000. doi: 10.1109/ROBOT.2000.846405.

[79] S. Vanneste, B. Bellekens, and M. Weyn. 3DVFH+: Real-Time Three-Dimensional Obstacle Avoidance Using an Octomap. In *MORSE 2014 Model-Driven Robot Software Engineering: proceedings of the 1st International Workshop on Model-Driven Robot Software Engineering co-located with International Conference on Software Technologies: Applications and Foundations (STAF 2014)*, number 1319, pages 91–102, York, England, July 2014.

[80] PX4. PX4-Avoidance, 2022. URL `https://github.com/PX4/PX4-Avoidance`. (accessed on 30 Mar 2024).

[81] S. Zhao, X. Wang, H. Chen, and Y. Wang. Cooperative Path Following Control of Fixed-wing Unmanned Aerial Vehicles with Collision Avoidance. *Journal of Intelligent & Robotic Systems*, 100(3):1569–1581, Dec 2020. doi: 10.1007/s10846-020-01210-3.

[82] J. Minguez and L. Montano. Nearness diagram (ND) navigation: collision avoidance in troublesome scenarios. *IEEE Transactions on Robotics and Automation*, 20(1):45–59, 2004. doi: 10.1109/TRA.2003.820849.

[83] A. Alexopoulos, A. Kandil, P. Orzechowski, and E. Badreddin. A Comparative Study of Collision Avoidance Techniques for Unmanned Aerial Vehicles. In *2013 IEEE International Conference on Systems, Man, and Cybernetics*, pages 1969–1974, 2013. doi: 10.1109/SMC.2013.338.

[84] J. Mankar, C. Darode, K. Trivedi, M. Kanoje, and P. Shahare. Review of I2C protocol. *International Journal of Research in Advent Technology*, 2(1), 2014. URL https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=314537daa1f601f83044b25b68e2af6c8f331f3f.

[85] Maxbotix. I2CXL-MaxSonar - EZ Series, 2023. URL https://maxbotix.com/pages/i2cxl-maxsonar-ez-datasheet. (accessed on 25 May 2024).

[86] Lightware. LW20/c Manual, 2020. URL https://www.documents.lightware.co.za/LW20%20-%20LiDAR%20Manual%20-%20Rev%2012.pdf. (accessed on 25 May 2024).

[87] Lightware. SF45/B Guide, 2021. URL https://support.lightware.co.za/sf45b/. (accessed on 02 Jun 2024).

[88] Holybro Docs. Pixhawk 6X, 2023. URL https://docs.holybro.com/autopilot/pixhawk-6x. (accessed on 8 Apr 2024).

[89] Raspberry Pi. Raspberry Pi Computer Module 4 Datasheet, 2023. URL https://datasheets.raspberrypi.com/cm4/cm4-datasheet.pdf. (accessed on 05 Jun 2024).

[90] Holybro Docs. Pixhawk RPi CM4 Baseboard, 2023. URL https://docs.holybro.com/autopilot/pixhawk-baseboards/pixhawk-rpi-cm4-baseboard. (accessed on 8 Apr 2024).

[91] L. Meier, D. Sidrane, and R. Roche. DS-012 Pixhawk Autopilot FMUv6X Standard. Technical report, Pixhawk, 2022. URL https://github.com/pixhawk/Pixhawk-Standards/blob/master/DS-012%20Pixhawk%20Autopilot%20v6X%20Standard.pdf. (accessed on 8 Apr 2024).

[92] Raspberry Pi. Computer Module 4 IO Board Datasheet, 2023. URL https://datasheets.raspberrypi.com/cm4io/cm4io-datasheet.pdf. (accessed on 1 Jul 2024).

[93] L. Meier, D. Sidrane, D. Yoon, R. Roche, and A. Smith. DS-010 Pixhawk Autopilot Bus Standard. Technical report, Pixhawk, 2022. URL https://github.com/pixhawk/Pixhawk-Standards/blob/master/DS-010%20Pixhawk%20Autopilot%20Bus%20Standard.pdf. (accessed on 8 Apr 2024).

[94] L. Meier and A. Willee. DS-009 Pixhawk Connector Standard. Technical report, Pixhawk, 2022. URL https://github.com/pixhawk/Pixhawk-Standards/blob/master/DS-009%20Pixhawk%20Connector%20Standard.pdf. (accessed on 8 Apr 2024).

[95] Holybro Docs. Pixhawk Baseboard Ports, 2023. URL https://docs.holybro.com/autopilot/pixhawk-baseboards/pixhawk-baseboard-ports. (accessed on 8 Apr 2024).

[96] Holybro Docs. PM03D - RPi CM4 Base Wiring Guide, 2023. URL https://docs.holybro.com/autopilot/pixhawk-baseboards/pixhawk-rpi-cm4-baseboard/pm03d-rpi-cm4-base-wiring-guide. (accessed on 8 Jun 2024).

[97] PX4 Guide (v1.14). Loading Firmware, 2023. URL https://docs.px4.io/v1.14/en/config/firmware.html. (accessed on 3 Jul 2024).

[98] Raspberry Pi. Flash an image to a Compute Module, 2023. URL https://www.raspberrypi.com/documentation/computers/compute-module.html#flash-an-image-to-a-compute-module. (accessed on 1 Jul 2024).

[99] Raspberry Pi. usbboot, 2024. URL https://github.com/raspberrypi/usbboot. (accessed on 1 Jul 2024).

[100] libsub. libusb, 2024. URL https://github.com/libusb/libusb. (accessed on 1 Jul 2024).

[101] freedesktop Wiki. pks-config, 2021. URL https://www.freedesktop.org/wiki/Software/pkg-config/. (accessed on 1 Jul 2024).

[102] Raspberry Pi. rpi-imager, 2024. URL https://github.com/raspberrypi/rpi-imager. (accessed on 1 Jul 2024).

[103] Ardupilot. ArduPilot Documentation, 2024. URL https://ardupilot.org/ardupilot/. (accessed on 3 Jul 2024).

[104] PX4. Software Overview, 2024. URL https://px4.io/software/software-overview/. (accessed on 3 Jul 2024).

[105] PX4 Guide (v1.14). PX4 Architectural Overview, 2023. URL https://docs.px4.io/v1.14/en/concept/architecture.html. (accessed on 25 May 2024).

[106] PX4. PX4-Autopilot, 2024. URL https://github.com/PX4/PX4-Autopilot. (accessed on 1 Jun 2024).

[107] Brennan Ashton. NuttX Documentation: NuttShell (NSH), 2019. URL https://cwiki.apache.org/confluence/pages/viewpage.action?pageId=139629410. (accessed on 3 Jul 2024).

[108] PX4 Guide (v1.14). uORB Messaging, 2023. URL https://docs.px4.io/v1.14/en/middleware/uorb.html. (accessed on 3 Jul 2024).

[109] PX4 Guide (v1.14). uORB Graph, 2022. URL https://docs.px4.io/v1.14/en/middleware/uorb_graph.html. (accessed on 3 Jul 2024).

[110] PX4 Guide (v1.14). Parameter Reference, 2023. URL https://docs.px4.io/v1.14/en/advanced_config/parameter_reference.html. (accessed on 3 Jul 2024).

[111] PX4 Guide (v1.14). Modules Reference: Distance Sensor (Driver), 2023. URL https://docs.px4.io/v1.14/en/modules/modules_driver_distance_sensor.html. (accessed on 3 Jul 2024).

[112] PX4 Guide (v1.14). System Startup, 2023. URL https://docs.px4.io/v1.14/en/concept/system_startup.html. (accessed on 3 Jul 2024).

[113] Andrew Brahim. SF45 fixes to restart the state machine if sensor does not init correctly, 2024. URL https://github.com/PX4/PX4-Autopilot/pull/23565. (accessed on 28 Ago 2024).

[114] PX4 Guide (v1.14). MAVLink Messaging, 2023. URL https://docs.px4.io/v1.14/en/middleware/mavlink.html. (accessed on 18 Jul 2024).

[115] MAVLink Guide. Packet Serialization, 2024. URL https://mavlink.io/en/guide/serialization.html. (accessed on 18 Jul 2024).

[116] MAVLink Guide. How to Define MAVLink Messages & Enums, 2024. URL https://mavlink.io/en/guide/define_xml_element.html. (accessed on 18 Jul 2024).

[117] MAVLink Guide. Parsing MAVLink in Wireshark, 2024. URL https://mavlink.io/en/guide/wireshark.html. (accessed on 20 Jul 2024).

[118] MAVLInk Guide. Path Planning Protocol (Trajectory Interface), 2024. URL https://mavlink.io/en/services/trajectory.html. (accessed on 30 Sep 2024).

[119] PX4 Guide (v1.14). Path Planning Protocol, 2023. URL https://docs.px4.io/v1.14/en/computer_vision/path_planning_interface.html. (accessed on 30 Sep 2024).

[120] QGroundControl User Guide. Overview, 2024. URL https://docs.qgroundcontrol.com/master/en/qgc-user-guide/. (accessed on 15 Oct 2024).

[121] PX4 Guide (v1.14). PX4 Ethernet Setup, 2023. URL https://docs.px4.io/v1.14/en/advanced_config/ethernet_setup.html. (accessed on 29 Jul 2024).

[122] Ferhang Naderi. Get the Pixhawk Raspberry Pi CM4 Baseboard by Holybro talking with PX4, 2023. URL https://px4.io/get-the-pixhawk-raspberry-pi-cm4-baseboard-by-holybro-talking-with-px4/. (accessed on 29 Jul 2024).

[123] MAVSDK Guide. Introduction, 2024. URL https://mavsdk.mavlink.io/main/en/index.html. (accessed on 11 Sep 2024).

[124] MAVLink Developer Guide. Using Pymavlink Libraries (mavgen), 2024. URL https://mavlink.io/en/mavgen_python/#using-pymavlink-libraries-mavgen. (accessed on 11 Sep 2024).

[125] ROS Wiki. MAVROS, 2024. URL https://wiki.ros.org/mavros. (accessed on 11 Sep 2024).

[126] J. C. Castilho. ROS 2.0–Study and Evaluation of ROS 2 in comparison with ROS 1. MSc Thesis in Electrical and Computer Engineering, Faculdade de Ciências e Tecnologias, Universidade de Coimbra, 2022.

[127] ROS Documentation. Ubuntu install of ROS Noetic, 2023. URL http://wiki.ros.org/noetic/Installation/Ubuntu#Installation. (accessed on 5 Oct 2024).

[128] PX4 Guide (v1.14). ROS with MAVROS Installation Guide, 2023. URL https://docs.px4.io/v1.14/en/ros/mavros_installation.html. (accessed on 5 Oct 2024).

[129] V. Vilhjalmsson. Risk-based Pathfinding for Drones. MSc Thesis in Computer Science, Swiss Federal Institute of Technology Zurich, 2016.

[130] ROS Documentation. rospy, 2017. URL http://wiki.ros.org/rospy. (accessed on 5 Oct 2024).

Appendix A

Pixhawk RPi CM4 Baseboard Scheme

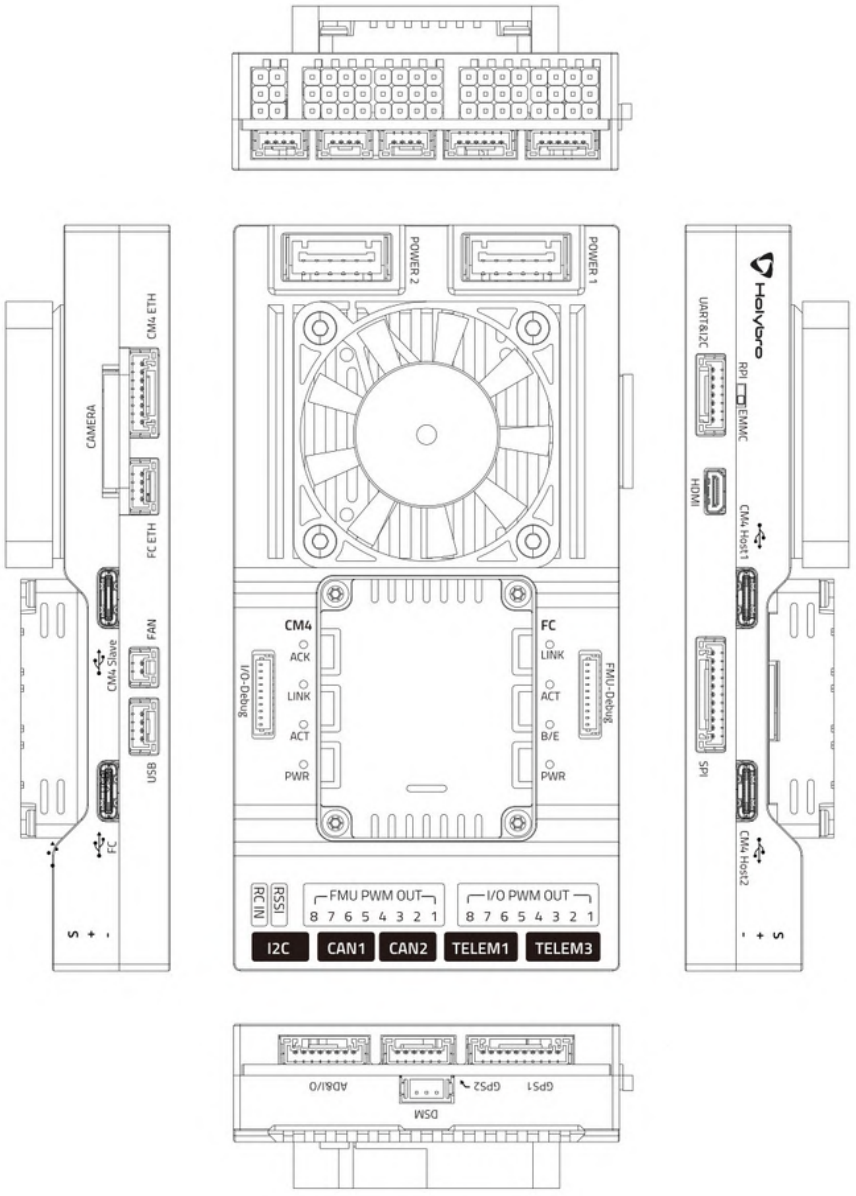


Figure A.1: Pixhawk RPi CM4 baseboard scheme [90].

Appendix B

Software Architecture of PX4

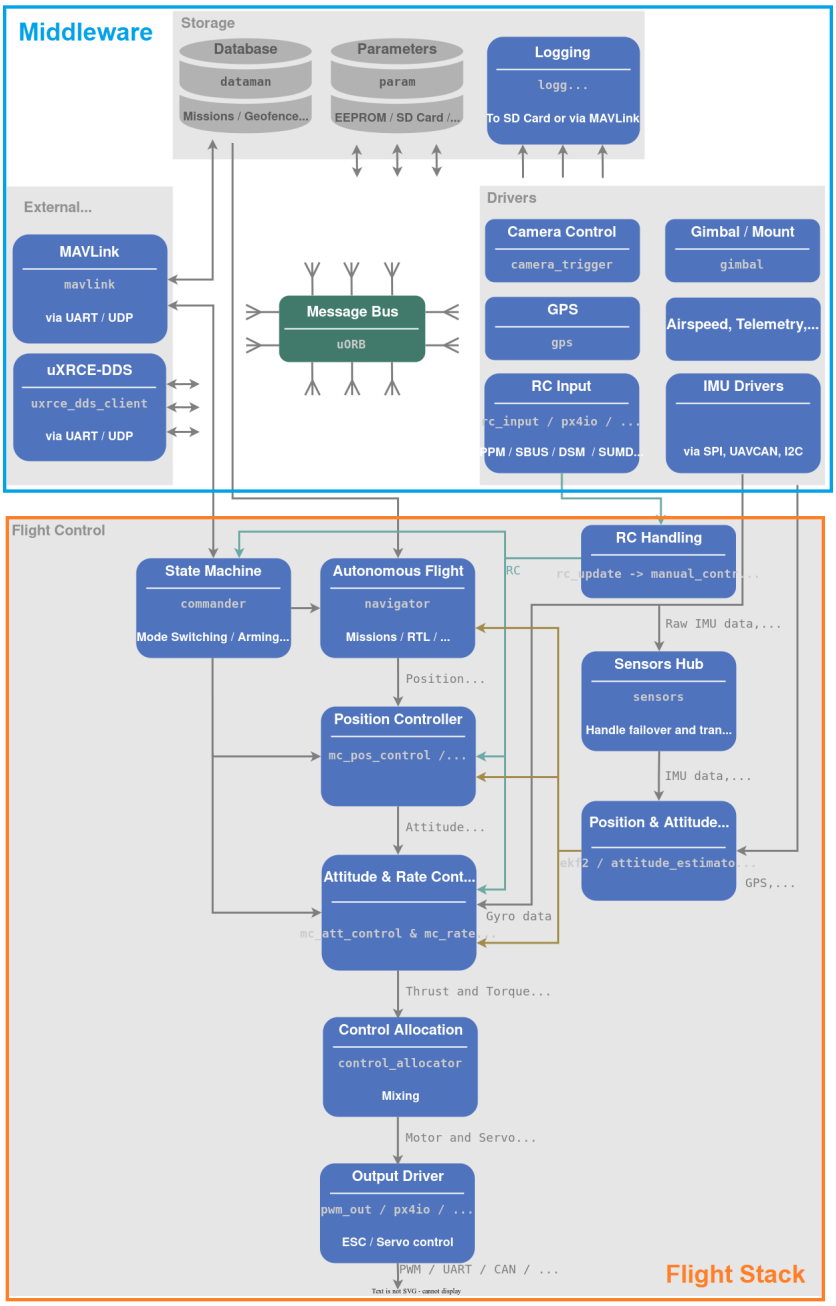


Figure B.1: High-level software architecture of PX4 (Adapted from [105]).

Appendix C

Relevant File Directories

C.1 ROS Catkin Workspace

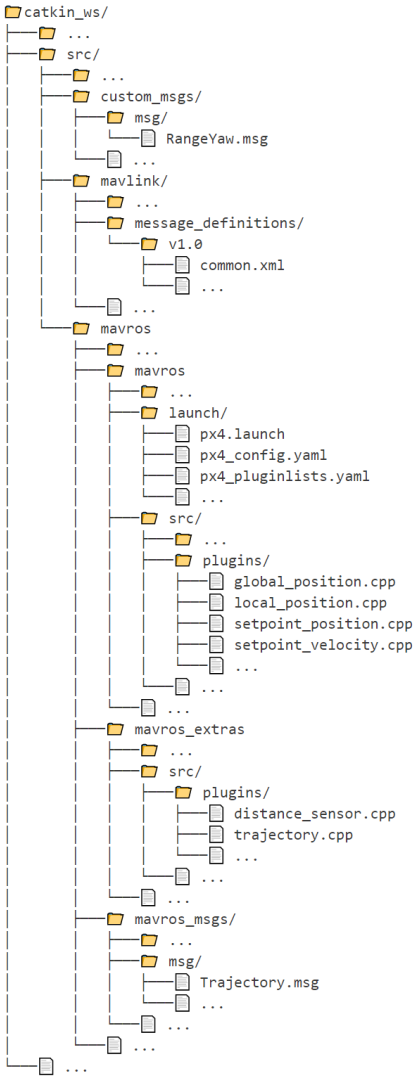


Figure C.1: Directory tree of relevant files of ROS Catkin Workspace.

C.2 PX4 v1.14.3

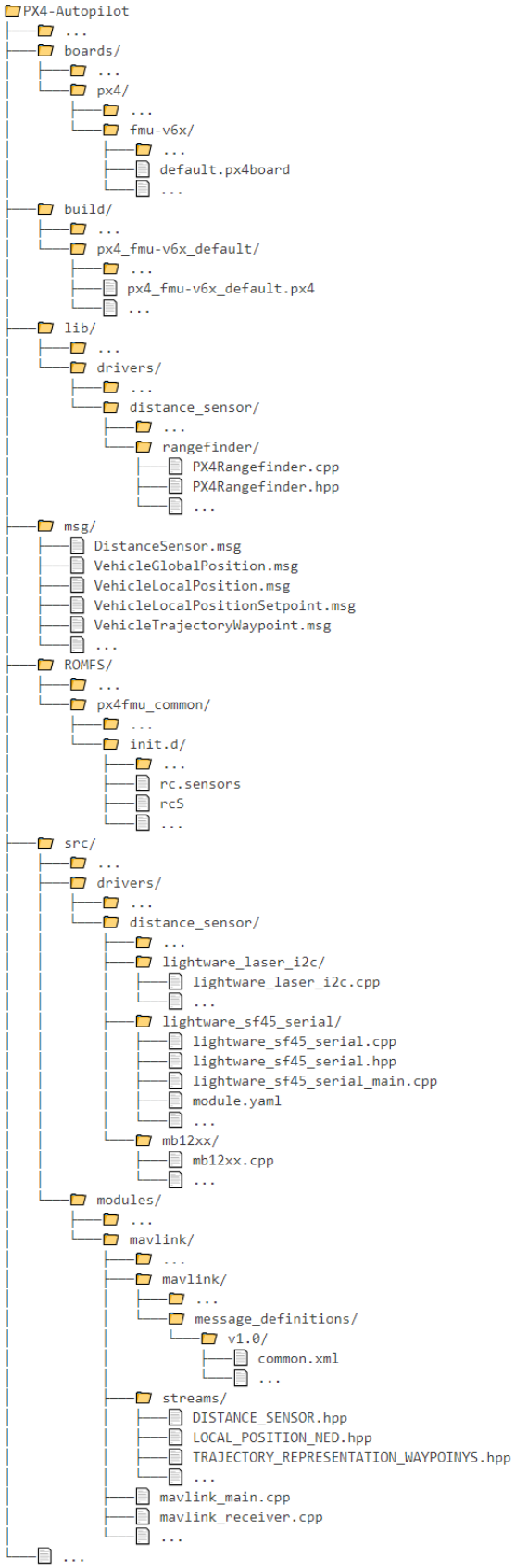


Figure C.2: Directory tree of relevant files of PX4 v1.14.3.

Appendix D

Adaptation of MAVROS Files

D.1 Adaptation of MAVROS_EXTRAS Distance Sensor Plugin

distance_sensor.cpp

```
(...)
#include <custom_msgs/RangeYaw.h>
(...)
class DistanceSensorItem {
public:
    (...)
    DistanceSensorItem() :
        (...)
        device_id(0)
    // params
    (...)
    uint32_t device_id;  //!< Unique sensor ID defined by its driver
    (...)
    void range_cb(const custom_msgs::RangeYaw::ConstPtr &msg);
class DistanceSensorPlugin : public plugin::PluginBase {
public:
    (...)
    void initialize(UAS &uas_) override{
        (...)
        for (auto &pair : map_dict) {
            (...)
            if (it)
                sensor_map[it->device_id] = it;
            (...)}}
private:
    (...)
    void distance_sensor((...), uint32_t device_id){
        (...)
        ds.current_yaw = current_yaw;
        ds.device_id = device_id;
```

```
        (...)}
    void handle_distance_sensor(...){
        (...)
        auto it = sensor_map.find(dist_sen.device_id);
        (...)
        auto range = boost::make_shared<custom_msgs::RangeYaw>();
        (...)
        range->range = dist_sen.current_distance* 1E-2; // in meters
        range->current_yaw = dist_sen.current_yaw;      // in degrees
        range->device_id = dist_sen.device_id;
        (...)}
void DistanceSensorItem::range_cb(const custom_msgs::RangeYaw::ConstPtr &msg){
    (...)
    if (msg->radiation_type == custom_msgs::RangeYaw::LASER)
        type = enum_value(MAV_DISTANCE_SENSOR::LASER);
    else if (msg->radiation_type == custom_msgs::RangeYaw::ULTRASOUND)
        type = enum_value(MAV_DISTANCE_SENSOR::ULTRASOUND);
    (...)
    owner->distance_sensor((...), msg->current_yaw, msg->device_id);
}
DistanceSensorItem::Ptr DistanceSensorItem::create_item(DistanceSensorPlugin *owner, std
    ::string topic_name){
    (...)
    int temp_device_id; // temporary variable to store device_id as an int
    (...)
    p->device_id = static_cast<uint32_t>(temp_device_id); // cast device_id to uint32_t
    (...)
    // create topic handles
    if (!p->is_subscriber)
        p->pub = owner->dist_nh.advertise<custom_msgs::RangeYaw>(topic_name, 10);
    (...)}
(...)
```

D.2 Adaptation of MAVROS Configuration File for PX4

px4_config.yaml

```
(...)
distance_sensor:
    sonar1_pub:
        device_id: 7497753
        frame_id: "sonar1"
        orientation: YAW_45
        field_of_view: 0.0
    sonar2_pub:
        device_id: 7499801
        frame_id: "sonar2"
        orientation: 0
```

```
        field_of_view: 0.0
    laser1_pub:
        device_id: 7562777
        frame_id: "laser1"
        orientation: 0
        field_of_view: 0.0
    laser2_pub:
        device_id: 7563033
        frame_id: "laser2"
        orientation: YAW_90
        field_of_view: 0.0
    lidar_pub:
        device_id: 7536653
        frame_id: "lidar"
        orientation: 0
        field_of_view: 0.0
(...)
```

Appendix E

Custom ROS Message: RangeYaw

RangeYaw.msg

```
Header header
uint8 LASER=0
uint8 ULTRASOUND=1
uint8 INFRARED=2
uint8 RADAR=3
uint8 UNKNOWN=4
uint8 radiation_type
float32 field_of_view
float32 min_range
float32 max_range
float32 range
float32 current_yaw     # NEW FIELD: current sensor yaw data [deg]
uint32 device_id        # NEW FIELD: unique sensor ID defined by its driver
```