UAV Obstacle Collision Avoidance System

Subsystem integration for safer autonomous flights

By

Álvaro Melgosa Pascual

Department of Bioengineering and Aerospace Engineering
Universidad Carlos III de Madrid

September 2016

Abstract

The large growth that the civil Unmanned Aerial Vehicles (UAVs) market has experienced in the last decade is now triggering the urge of both professionals and enthusiasts to use this technology to perform tasks that would be more difficult to accomplish with their traditional procedures. However, many times these tasks require precision flight and do not allow the slightest physical contact with the UAV. Currently, very qualified pilots are needed since there have not been significant advancements on on-board obstacle detection technologies, and manual control is still a must.

The main goal of this thesis is to develop an affordable Obstacle Alert and Collision Avoidance System (OCAS) that can be easily deployed to a wide range of UAVs. The approach followed is to embark a series of ultrasonic rangefinders to continuously monitor the minimum distance of the vehicle with its surroundings. The data provided by the sensors is then processed on an onboard computer, and control commands are sent to the main controller board in the case that an obstacle is detected and a possible collision identified. The final result is an integrable payload subsystem that would improve the situational awareness capabilities of any UAV that integrates it, reducing the risk of collision with its surroundings.

Keywords: UAV, obstacle detection, collision avoidance, system integration, ultrasonic rangefinder, Ardupilot

Dedication and acknowledgements

Firstly, I would like to dedicate this thesis to my family, who have always supported me and are making a big effort to provide me with the best education.

Secondly, a big thank you to Xin Chen, who not only was the person which I could discuss technical issues with, but also motivated me every day through her endless optimism. Thank you also to Manuel Soler.

And last but not least, my appreciation for all my friends and classmates at UC3M, who accidentally excited me to keep working by showing their most sincere interest on the topic I was working on.

TABLE OF CONTENTS

List of Figures

List of Tables

CHAPTER 1

INTRODUCTION

This chapter will be used to acquaint the reader with the emerging UAV market, and the challenges it is facing on its way towards maturity. Also, the reasons for its rapid evolution will be exposed and finally, focusing on the contents of this thesis, the personal motivation and the methodology will be explained to further expand on the topics of interest in the following chapters.

1.1 Background information

The first remotely radio controlled models appeared in the early twentieth century as small prototypes for potential manned aircraft. Afterwards, and during most of the century, the investigation and development lines were directed towards the military scope, in which the main objective of UAVs, which is still applied today, was to substitute manned aircraft in three types of military operations, commonly known as “the three D’s" [4, 5]:

- Dirty: operations performed in a contaminated environment.
- Dangerous: operations entailing some risk for the pilot.
- Dull: long and monotone operations, such as monitoring operations.

In the 70’s and the 80’s, efforts were directed to improve the technical characteristics of these vehicles. But it was not until the late 80’s when a revolution in the industry took place with the introduction of the GPS navigation system, whose accuracy in geolocation opened a whole new spectrum of possibilities.

Regarding the civil sector, the potential applications of UAVs in the non-military field are much more diverse. Nowadays these vehicles are in the process of finding new niche positions in the civilian market, having been introduced up to now in different industry sectors such as agriculture, forest fire fighting, search and rescue, aerial photography, cartography, or security and surveillance, among others. Despite the latter, the use of UAVs for civil purposes is relatively recent in comparison with the military sector. This late implementation in the civilian field was caused mainly by two limitations which are of minor relevance in the fighting industry: legislation and economy. [6]

1.2 Socioeconomic environment

Apart from "the three D's" mentioned in Section 1.1, another reason for the embracement of UAVs within the industry shall be considered. The final goal of any company is to create profit to their shareholders, which can be done either by increasing the revenues or by decreasing the costs of their activities. UAVs enter in the latter category. The consistent usage of smaller tools as compared with the manned workpower usually means that the equipment costs can be lowered, as well as the man-hours needed to perform the task [7], not to mention that most of the time the number of workers needed can be reduced to as low as one or two, in charge of operating the UAS (Unmanned Aerial System[1]).

This phenomenon is already proving to be very effective for the companies taking advantage of it, but research also shows an even bigger potential that is still waiting to be exploited, claiming that UAVs could have replaced $127 billion worth of human labour in 2015 [1], distributed in the sectors shown in Figure 1.1.

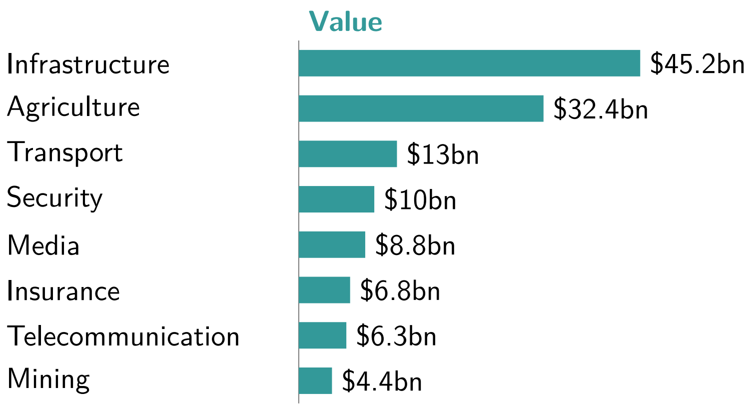


Figure 1.1: Distribution of potential UAV markets [1]

[1] UAS refers to the bigger system that incorporates one or more UAVs, as well as the Ground Control Station or other related subsystems

1.3 Legal framework

Due to the fast-evolving UAV industry, the aviation authorities have not yet been able to develop a reasonable set of regulations and standards to harmonize the legislation across borders. Aditionally, this regulatory framework should consider the idea that each system has unique capabilities and characteristics and also that development and innovation are very important concept in the field, and should not be dampened by restrictive rules [8]

However, there have already been some efforts from ICAO to outline some general rules to give a global sense of what is expected from the UAV sector [9]. In addition to that, some countries are creating their own legislation to enable the operation of Unmanned Aerial Vehicles within their territory.

For example, the Spanish government issued an urgent provisional regulation on October 2014 [10] that affects to Remotely Piloted Aircraft Systems (RPAS[2]) not exceeding 150 kg of Maximum Take Off Mass (MTOM). Heavier UAVs are subjected to European regulations [11] Focusing on the smaller segments, UAVs are separated according to their MTOM as follows:

MTOM < 2 kg: Flights Beyond Visual Line Of Sight (BVLOS) are allowed, but conditioned to the publication of a NOTAM (NOtice To AirMen). Apart from that, all the other rules in the 2 kg to 25 kg apply.

2 kg ≤ MTOM < 25 kg: Only operable in areas separated from groups of buildings in cities, or groups of people elsewhere. Flight shall always take place in uncontrolled airspace, within Visual Line Of Sight (VLOS) and at a maximum distance of 500 m from the position of the pilot, not exceeding 400 ft of heigt over the terrain.

25 kg < MTOM: Flight is only allowed for firefighting, search and rescue missions. They shall only operate in uncontrolled airspace and according to the limitations stablished in their Airworthiness Certificate, as emited by AESA.

Nevertheless, even if Spain or other countries have their own regulations to control the usage of UAVs in their territories, it is still important to have an international and stable legislation to allow the sector to grow to its full potential.

1.4 Motivation

Traditionally, the most important payload that could be carried in an aircraft was human beings, that would perform their mission while aloft. Nevertheless, the advancements on sensing technology and wireless communications have forced a change on traditional aviation. Apart from commercial aviation, where the final objective is to transport people form one place to

[2]RPAS are considered as a subset of the UAS group. Fully automatic vehicles do not belong to the RPAS category, since the existance of a remote pilot is required at any time

another, in almost any other mission the role of the human workforce is to pilot the aircraft and/or operate the payload systems. This secondary role of the human operators implies that, given the maturity of the involved technology, they could be substituted by intelligent computer systems or, at least, disembarked form the aircraft into a safer Ground Control Station (GCS). The process of “unmanning” the aircraft also brings the advantages of decreasing the weight of the aircraft and thus improving its endurance and manoeuvrability, avoids putting the pilot in a dangerous situation, and helps alleviate the errors associated with tedious and repetitive tasks, among others.

However, there are also some downsides. In the technical department, there are still some issues regarding the electromagnetic spectrum allocation for the data-link with the vehicle [12], as well as accommodating unmanned aircraft within the Airspace System [13]. In addition, the most accused issues for experienced pilots are those related with the loss of situational awareness that comes as a result of eliminating the physical cues (body inertia, vibrations. . .) and relying on instrumental readings only [14]. Hence, some enhanced systems need to be integrated into the vehicle to overcome this limitations, providing the pilots with additional information for the safe execution of the mission.

Finally, for this project, the goal is to provide a system that reduces the risk of the widest range of UAVs from crashing with nearby obstacles, so that regular operations are carried with a higher level of safety. Eventually, the authorities could consider the increase in overall safety as a standard, triggering the modification of existing regulations to a more permissive set, and allowing the industry to take advantage of all the benefits that the incorporation of UAVs could bring to their activities.

1.5 Project objectives

According to the motivation as stated in Section 1.4, the final goal of this project is to develop a working prototype for proof of concept of a system able to detect and avoid obstacles that threaten the integrity of the UAV. Towards that end, some more specific objectives can be defined as follows:

- Identify the requirements needed for the Obstacle Collision Avoidance System (OCAS) to correctly fulfill its purpose
- Define the functional architecture of the OCAS
- Define the interfaces (communication channels and protocols) to be used by the OCAS for its correct integration on the UAV.
- Define the interaction channels and procedures between the UAV equiped with OCAS and Ardupilot and the operator.

- Develop a first working prototype as proof of concept of the Obstacle Collision Avoidance System (both hardware and software) and integrate it on a real UAVtroduction.tex

Additionally, the architecture of the solution should be designed with modularity in mind, permitting easy adaptation of the algorithms for later research activities.

1.6 Methodology

As some may have noticed, the objectives defined in Section 1.5 remember of the first steps that are usually taken in the Systems Engineering approach for interdisciplinary design [15]. That approach will be adapted to the project, and some useful tools and concepts will be used [16], such as the requirements capture, the Functional Flow Block Diagram (FFBD), the Functional Architecture definition or the product integration via interfaces definition.

Finally the prototype created from the process will be tested in a series of common situations to prove that the product is capable of completing its task. Also, it will be demonstrated how the OCAS has been designed with flexibility and modularity in mind, explaining the possibilities to expand its features and proposing some ideas for future work.

1.7 Time planning

For any big project with defined deadlines, time management is of utmost importance. The elaboration of the thesis has been carried out during more than 10 months, and the different work phases have been monitored with a project management software tool. The resulting Gantt Diagram can be consulted on Figure 1.2.

It is worth mentioning that in the period from 01/11/2015 to 05/05/2016 I was doing my professional internships at Centum Solutions [17]. Thus, most of my research during that time was guided by the interests of the company. Nevertheless, that stage proved very useful for the summer period, when my work was exclusively focused towards my thesis.

1.8 Budget

This section describes all costs associated to the project and proposes an estimate of the budget needed to replicate it. The final cost results on **6391.86 €**, which is divided as follows.

1.8.1 Personnel expenses

The base annual engineering salary for an Engineering Degree holder in Spain is, according to the Spanish "XVI Convenio colectivo nacional de empresas de ingeniería y oficinas de estudios técnicos" [18], of at least 17,038.62 € per year, with a maximum of 1800 working hours. Thus, the

Figure 1.2: Gantt Diagram of the Project

Component	Unitary Price	Units	Total
F450 kit	399 €	1	399 €
Propeller blades	2 €	4	8 €
Radio transmitter / receiver	51 €	1	51 €
Telemetry radio	48 €	1	48 €
Primary battery	40.85 €	5	204.25 €
Secondary battery	14 €	1	14 €
Raspberry Pi	33.81 €	1	33.81 €
Wireless WiFi adapter	19.95 €	1	19.95 €
Ultrasonic range finder	4.50 €	8	36 €
Resistor	0.05 €	30	1.50 €
Connection wires	0.10 €	50	5 €
Camera	24 €	2	48 €
Optical flow sensor	36 €	1	36 €
TOTAL			**904.51 €**

Table 1.1: Prototype hardware costs

minimum salary can be calculated as 17038.62 €/1800 h = 9.47 €/h. With an estimated working time of approximately 550 hours, the calculated personnel expenses are 5216.50 €.

1.8.2 Software cost

Most of the work has been performed on open-source software systems, such as Arduino / Ardupilot and Raspberry Pi / Linux. However, the calibration and initial programming and testing, prior to the payload deployment, was done on a laptop machine running Windows 10 Pro, listed at 279 € per license on the Microsoft Store.

1.8.3 Hardware cost

For the hardware part of the project, the aforementioned laptop will be considered together with all the components needed to build the test platform.

The PC was bought for 500 €. Assuming a linear depreciation period of 4 years, and that the dedication to the project equals the amount of total labour hours, the resultant expense is estimated as 7.85 €.

Unexpectedly, the prototype is composed of numerous individual components acquired to different sources. In Table 1.1, an estimation of their cost is done according to current prices on relevant stores. Notice that the F450 kit already includes most of the components needed for the UAV to fly (motors, controller board, frame…)

Chapter 2

A brief introduction to Ardupilot

As it was mentioned in Section 1.4, it is intended to bring the technology developed within this project to the widest range of UAVs. However, there exist in the market several families of controller boards (which can be considered to be the brains of the vehicles, in charge of all the basic functions required for an stable flight) that can only be used with specific hardware and/or software, not being compatible with each other since they implement different communication protocols. Furthermore, some manufacturers work with proprietary software, of which little information on the low-level functioning is available to the public.

It is clearly impractical to try to target all the existing standards for this project, so a compromise needs to be made. The thesis will be elaborated for the Ardupilot family of controllers, for being the most widespread open-source[1] alternative. Some of the leading companies [19] in the sector actively support the Dronecode Project, of which Ardupilot is part, such as Intel, Qualcomm, Parrot, 3DR, Yuneec, AUAV, Walkera. . . [20]

It is important nonetheless to clarify some concepts and features of any Ardupilot-equipped UAV. More information can be found at `www.ardupilot.org`.

2.1 Basic features

The most basic but important feature of the controller is to give control to the pilot over the vehicle. There are several components that make this function possible.

Firstly, the pilot expresses the desired movements of the vehicle through a Radio Control (RC) transmitter, shown in Figure 2.1a. The signal at 2.4 GHz is received by the RC receiver located in the vehicle, depicted in Figure 2.1b. Then the receiver translates the electromagnetic wave into

[1] The software is being developed at GitHub: `https://github.com/ArduPilot/ardupilot`

several PWM (Pulse Width Modulation) signals, one for each input channel up to a maximum of 8 channels, which are inputted to the controller board. However, for the primary control of the vehicle, only 4 channels are needed: throttle, roll, pitch and yaw. The additional channels are used to control extra features such as the flight mode, the landing gear or the camera controls.

(a) RC transmitter (b) RC receiver

Figure 2.1: FlySky FS-i6 Remote Control (`www.flyskyrc.com`)

The second step is to translate the commands from the pilot into signals to the control elements of the vehicle. These can vary depending on the type of vehicle (for example the yaw command affects the rudder in the case of a fixed-wing aircraft, the tail rotor collective control for a conventional helicopter or the differential throttle in the diagonals for a multicopter) but the underlying processes are similar.

Every Ardupilot controller board must have at least and Inertial Measurement Unit (IMU) consisting of a 3-axis accelerometer plus a 3-axis gyroscope for the state determination of the vehicle. Additionally, a barometer, a GPS and other sensors can be integrated. Hence, reading the pilot's commands from the RC receiver and the state of the vehicle from the IMU, the output to the control elements can be computed by some regular PID control loops (more information on the topic can be found at [21]). To the output pins of the controller board are connected the control elements, be it some servo-motors for the control surfaces of a fixed-wing aircraft or brushless motors with propellers for the case of a multicopter. These elements are externally powered by the primary battery.

2.2 Ardupilot as part of a UAS

If Ardupilot wants to be used as a professional tool to enhance production or reduce costs, it can not rely on manual control only. For more advanced missions and proper calibration of vehicles with diverse configurations and physical properties, it is necessary to tweak the parameters that the control loops necessitate for their real-time computations. It is in those cases when a Ground Control Station can become useful. By connecting the vehicle to an external computer, the operator is no longer limited to the 8 input channels that the RC transmitter can provide.

Instead, the limit on the amount of information that the Ardupilot board can broadcast or absorb is only bound by the communication protocol that is implemented between the two.

For the Ardupilot ecosystem the protocol used is also open-source and receives the name of MAVlink[2] (MAV stands for Micro Aerial Vehicle). Its open nature allows developers to create a very diverse set of software and applications to communicate with the UAV, from the widespread Mission Planner and APM planner, to versions that run on Android devices for on-the-field operation or developer-oriented libraries that run under Python.

Another feature that is worth mentioning is the lightweight nature of the protocol, which not only permits the connection via USB cable, but also wirelessly through what is usually called a telemetry radio, which effectively is a serial transmission of data over a 433 MHz radio wave carrier.

An experienced operator can take advantage of all the mentioned features to receive real-time information on the state of the vehicle while it is on the air, and also to send high-level commands to the vehicle. Those options will be further discussed in Section 2.3.

Figure 2.2: Screenshot of Mission Planner GCS, implementing the MAVlink protocol

[2]More information on the protocol can be found on `qgroundcontrol.org/mavlink/start`. The message definitions and generator code can be found at its GitHub repository `github.com/mavlink/mavlink/`

2.3 Advanced features

For an Ardupilot UAV to be able to automate some missions and procedures there are some additional requirements. Firstly, the IMU is appropriate for the evaluation of the vehicle's state variables, but the knowledge of its environment can only be acquired through absolute positioning sensors. Those sensors are usually a GPS module for horizontal positioning and a barometer for altitude measurement. Secondly, a wireless data-link provides a much more flexible way of interacting with the UAV during the execution of the mission.

2.3.1 Flight modes

Ardupilot has separated the mentioned advanced features in different flight modes, which can be activated with the 5^{th} channel on the RC transmitter or from the GCS. At the time of writing, there are 15 different flight modes, but just the most relevant ones for the project will be described here. A summary of the most important features can be found in Table 2.1
[From this point onwards the concepts involving Ardupilot will be particularised for the multicopter variant, since it is the type of vehicle that will be used for the prototype. Similar information can be found for fixed-wing aircraft, helicopters and rovers at the support page.]

STABILIZE The default mode for manual control. Uses only the IMU data to control the flight. The pitch and roll channels define the Euler angles (instead of the rotation rate of Acrobatic mode) so that when the controls are released to neutral position, the vehicle will level off automatically. The yaw channel does control the yaw rate of the UAV instead, while the throttle channel Ardupilot will not compensate for wind or other disturbances.

ALTITUDE HOLD Very similar to the Stabilize mode. The only difference is on the throttle channel, which controls the ascension rate instead of raw power transmitted to the motors. When the throttle stick is centered, the vehicle will hold the current altitude using the information measured by the onboard barometer.

LOITER Incorporates the GPS data to the Altitude hold mode, making it possible for the UAV to compensate for wind and IMU drift. The pilot still has control on the vehicle similarly to Altitude hold but when the control sticks are released, the position will be kept within a 1 metre error (provided good quality GPS signal).

AUTO This mode allows to automate missions and procedures. With the help of a GCS application, such as the one shown in Figure 2.2, the operator can click on a map to define waypoints and actions (take of, land, point to a certain direction, etc. are within the options) and save a data file to the vehicle's internal storage. Later, when the mode is activated, the vehicle will follow the predefined route without the need of direct input from the pilot. The throttle, yaw, pitch and roll controls will be disregarded when Auto mode is active, but the pilot can change the active mode at any time from the RC transmitter.

RTL (RETURN TO LAUNCH) RTL mode is commonly used as a failsafe feature, when communication either with the pilot or the GCS is lost. It is a very specific version of the Auto mode that automatically starts the return to Home procedure, landing the vehicle exactly where it took of from. The

Mode	Sensors used	Throttle	Roll/Pitch	Features
Stabilize	IMU	Power	Euler angles	Fully manual
Altitude hold	IMU + Barometer	Ascension rate	Euler angles	Enhanced altitude control
Loiter	IMU + Barometer + GPS	Ascension rate	Euler angles	Disturbance rejection
Auto	IMU + Barometer + GPS	No control	No control	Mission automation
RTL	IMU + Barometer + GPS	No control	No control	Failsafe
Guided	IMU + Barometer + GPS	No control	No control	Real-time control

Table 2.1: Summary of the relevant flight modes

Home location is defined as the point were the motors were initially armed (the arming procedure resembles the engine startup of a conventional aircraft: the motors will not spin unless the vehicle is armed)

GUIDED The solely difference between the Auto and the Guided modes is that whereas in the Auto mode the mission needs to be completely defined and uploaded to the vehicle before it is executer, the Guided mode allows for on-the-fly control of the vehicle from the GCS, taking complete advantage of the MAVlink commands over the telemetry link. This characteristic makes it very flexible for real-time development of applications.

CHAPTER 3

STATE OF THE ART

The relatively recent history of "small" (considered as weighting less than 25 kg) commercial UAVs implies that the technologies involved have not reached a high Technology Readiness Level (TRL) [22] until very recently, or are still under development (TRL 6-7). For that reason, most of the work considered in the present chapter is not yet ready for the market and has a great capacity to improve. Universities are actively exploring the autonomous aptitudes of UAVs, but the economical exploitation potential is still small due to the low reliability issues that these systems are encountering in non-controlled environments.

3.1 Environment sensing

Knowing the environment is the initial step towards any interaction of a system with its surroundings. That knowledge could in principle be acquired a-priori and then copied into the system, but when either time-changing or completely unknown scenarios are considered, having a means of sensing the environment is essential.

3.1.1 Radar

Radar stands for RAdio Detection And Ranging. Its usage dates back to 1904, when Hülsmeyer registered the patent for his Telemobiloscope [23], using a primitive variant of the radar for detecting metallic ships.

Nowadays, the radar works by emitting one powerful radio signal impulse against the object that needs to be located. If the signal reaches a solid object on its path, the electromagnetic wave will be distorted and part of it will be reflected back to the emission point, where a second listening antenna can detect it.

The distance of the detected object is calculated by measuring the flight time of the signal since it was emitted until it is detected, applying the signal propagation speed ($3 \cdot 10^8 \, m/s$ for electromagnetic waves) as correction factor. The azimuth position of the object can be estimated if the initial signal was created by a highly directional antenna, since the orientation of the transmitting antenna can be measured and the sensed object must be contained in the electromagnetic radiation field.

The fact that radar uses electromagnetic waves implies that the response time can be very high, although some technical issues may arise for the same fact at the time of processing the returned signal. For that reason, radar-based sensor systems focus on temporal processing techniques to improve the results of the measurements [24].

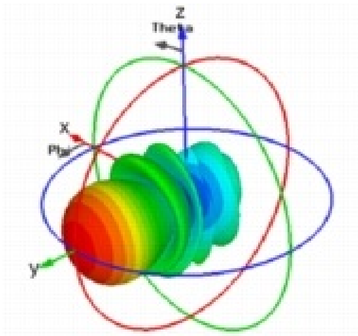


Figure 3.1: Highly directional radiation pattern. Source: `cisco.com`

3.1.2 Sonar

The ultrasonic rangefinder (commonly known as sonar, for SOund Navigation And Ranging) relies, like the radar, on the measurement of the flight time of a signal rebounding against the target. However, instead of being electromagnetic waves, the carriers are sound waves, with a frequency usually beyond the human hearing range upper threshold (hence ultrasonic).

It is important to mention that the calculation of distance from the flight time of the rebounded signal is to some degree dependent on the environment: The speed of propagation of sound depends on the temperature of the medium through which it propagates as $a = \sqrt{\gamma R_g T}$. Thus, the temperature of the air should be monitored to compensate for variations during the flight. Fortunately, the error associated with temperature changes around room temperature is of the same order than the accuracy of the sensor itself, so it is safe to consider ambient temperature at the initial calibration stage only and assume it stays constant thereafter, since it can be proven that for the measurement of distance to an object at approximately 1 metre, the error due to a temperature change of 10 K is of the order of 1 milimetre.

3.1.3 Lidar

LIght Detection and Ranging also works by measuring the time of flight of a signal. In this case, it is a visible light pulse, usually produced by a laser for its high coherence and low dispersion.

This system is convenient for ranging objects that are small or at a long distance from the sensor, but the small measuring point of the laser beam has some limitations when the mapping of a large area is required. In these situations, multiple measurements are usually performed sequentially with a rotation of the laser emitter between pulses, but the procedure significantly increases the latency of the system for obstacle detection, and also the complexity of the sensor system itself.

In summary, lidar is a good alternative for ranging, but not so much for detection.

3.1.4 Computer vision

The usage of regular cameras for physical environment sensing is certainly different than the previously considered, since it is not the flight time of a pulsated signal what encodes the information. Instead, one or more cameras provide a two-dimensional array of data each that can be processed to extract information of distance and location, among others, of any object that is within frame.

It is in that processing where both the advantages and draw backs of computer vision lie. On one hand, many different transformations can be applied to the stream of data from the cameras, and information can be extracted on very diverse aspects such as colour, luminosity, movement, position, texture... [2] Additionally, advanced processing techniques and even artificial intelligence can be applied to the image, which is a very flexible source of information. On the other hand though, the aforementioned processing tasks should be computed on a relatively high rate video stream, which implies several transformations on various video streams (for stereoscopic vision) with a few million pixels per image, at several frames per second. Furthermore, if obstacle detection and ranging needs to be performed, more than one viewpoint (and hence more than one camera) are required, and the matching problem of the objects seen by the two and the geometric transformations are also some quite demanding computational problems. Such a workload can only be dealt with by higher-end computer graphics cards nowadays at a reasonable rate for the effective control of a moving vehicle.

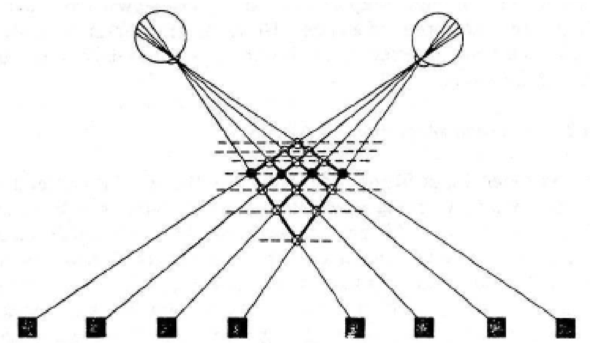


Figure 3.2: The stereo matching problem Source: [2]

3.2 Collision avoidance

The conventional sense and control problem has two very defined parts: the data acquisition stage that has been studied in Section 3.1 and the actuation of the control variables to modify the state of the system. In this section, the application alternatives of the latter in TRL 9 [22] products will be studied.

3.2.1 TCAS on conventional aircraft

The Traffic alert and Collision Avoidance System is the standard system mandated for use by commercial aircraft. It creates a virtual safety volume around the aircraft, which is based on the time to the Closest Point Approach (CPA) [25].

For the system to work, the host aircraft and the threatening aircraft must both equip an ATC transponder. If the external aircraft has only a Mode A transponder, only Traffic Advisories (TA) can be issued by the TCAS; when it is Mode C or S, also Resolution Advisores (RA) are issued; if the external aircraft is also equipped with a TCAS II, vertical coordination between the two aircraft is additionally provided by means of an ask-answer procedure.

The system relies on two antennas that are usually placed at the top and bottom of the fuselage to provide antenna diversity. The signal carriers are 1030 MHz radio waves for the asking signal and 1090 MHz for the replying ones.

Regarding the actuation component of the system, it is fully manual. When a TA or RA are issued, the pilots are responsible to take the control of the aircraft and perform the recommended manoeuvre. Thus, the system itself cannot be considered to be the one avoiding the collision, but rather helping the higher-level aircraft + crew system accomplish it.

3.2.2 DJI Phantom 4

An Obstacle Collision Avoidance System was not available for commercial UAVs until as recent as March 2016. Chinese manufacturer DJI unveiled their Phantom 4 emphasising its “aerial camera” role for video professionals.

Their reason for including such a system is to facilitate the creative process of filming while the UAV manages the flying aspect of the mission as autonomously as possible. To that end, apart from the primary recording camera, the Phantom 4 also incorporates two smaller front-facing cameras that use stereoscopic vision algorithms to detect potential obstacles in the planned trajectory and modify it on-the-fly. In addition, it also incorporates a lower-level failsafe: when the vehicle is not able to compute a reasonable alternative trajectory before the obstacle enters within a predefined distance to the vehicle as measured by the stereoscopic cameras, the flight controller will command a full stop to stationary flight until the obstacle is manually cleared by the pilot.

Figure 3.3: DJI Phantom 4 with stereo camera OCAS

Chapter 4

Problem statement

The objectives of the project have already been stated in Section 1.5. Following those ideas, the problem that is to be answered in this thesis can be stated with the following question:

> *Is it possible to improve the operational safety of a wide range of UAVs by developing an intermediate functional layer that prevents physical collisions between the UAV and its surroundings?*

The above statement tries to condense the main idea of the thesis in a compact and precise manner. Nonetheless, some concepts within it might need some clarification:

Operational safety: A reliable collision avoidance system reduces the workload of the pilot so that higher-level tasks directly related to the mission can be performed more efficiently and safely.

Wide range of UAVs: As stated in Chapter 2 the project will focus on the widely-spread Ardupilot firmware, which is currently the leading alternative of open-source UAV controller software available.

Intermediate functional layer: The proposed solution shall be easily integrable within existing UAVs; offered as an enhancement to the toolbox of functions of the system. Thus, the solution shall incorporate additional features to the UAS, while the functions provided by Ardupilot should not be modified.

The problems is believed to be worthwhile answering since, as it was proven in the State of the Art chapter (Chap. 3), the technology is not mature and has not been implemented except on very specific products like the DJI Phantom 4.

Furthermore, a fully operable and reliable OCAS would allow for more autonomous operation of the UAV, avoiding the pilot from needing to be focused on the immediate surroundings of the vehicle, thus permitting for an improved situational awareness and leading to a better overall execution of the mission.

Ultimately, a higher safety level of general UAV operations could lead the authorities to reconsider the possibility of them flying for civil purposes with less restrictions, thus enabling companies to save vast amounts of money and manpower on the execution of activities that are currently being accomplished in a less effective way by human workers.

Chapter 5

System design

As mentioned in Section 1.6, the design of the Obstacle Collision Avoidance System will follow the Systems Engineering approach. The main reason is that Systems Engineering provides some methods that prevent the errors with the highest consequences when the system to be designed is complex. As explained by Rolls-Royce Global Chief of Systems Engineering [26]:

> *Systems Engineering collects and organises all the information needed to understand the whole problem, explores it from all angles, and then finds the most appropriate system solution.*

Furthermore, A key study published through INCOSE [3] looked at the phase of detection of errors, and the consequent cost of fixing them. Cost modelling was validated against a cross-industry range of defence and aerospace projects. Figure 5.1 shows the results of the study.

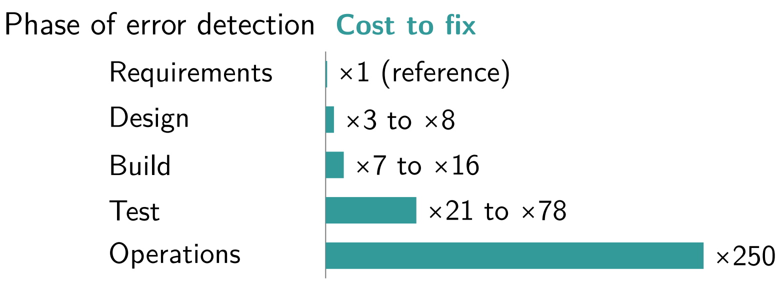


Figure 5.1: Cost to fix a design error. Source: [3]

Hence, in the present chapter, some of the most relevant Systems Engineering tools from the NASA Systems Engineering Handbook [16] will be applied

5.1 Requirements capture

The design process for a system is requirement driven, since the requirements are what will define the cost, design, schedule... A requirement is a statement about or a characteristic of something that is needed.

Requirements can be derived from a variety of sources, like customer needs, stakeholders, regulations, procedures, constrains, etc. However, for this project, customers and stakeholders will be disregarded (since none exist) and the motivation as stated in Section 1.4 will be used instead.

In the present section some requirements will be posed, but only those that directly apply to the OCAS subsystem or its interfaces, since the platform is considered to be completely functional prior to the introduction of the solution (following the modularity concept).

Req. ID	Requirement	Traceability (sourced from)	Traceability (allocated to)
Certification			
1.1	The UAV shall meet European regulations	EC No 218/2008	All
1.2	The UAV shall meet Spanish regulations	Ley 18/2014	All
Architecture			
2.1	The OCAS shall work independently of the UAV	Motivation	Power, communication
2.2	The OCAS shall be self-contained within the UAV	Integration	Power
Functionality			
3.1	The OCAS shall detect obstacles surrounding the UAV	Motivation	Sensors
3.2	The OCAS shall avoid collisions with the detected obstacles	Motivation	Processing, actuation
3.3	The OCAS shall take control of the UAV in case of danger	Req. 3.2	Actuation
3.4	The OCAS shall not interfere with existing Ardupilot functions	Motivation	Communication
3.5	The UAV shall maintain a communications data-link with the GCS at all time	Safety / FFBD	Communication
Performance			
4.1	The OCAS shall detect obstacles closer than 4 m to the UAV	Technical constraint	Sensors
4.2	The OCAS shall detect obstacles of at least 0.5 m across	Technical constraint	Sensors
4.3	The OCAS shall be powered along the full mission	Safety / FFBD	Power
Interfaces			
5.1	The OCAS shall know the state of the UAV	FFBD	Communication
5.2	The OCAS shall send commands to the UAV	FFBD	Communication
5.3	The OCAS shall be accessible from the GCS	Human factors	Communication
5.4	The OCAS shall be activated and deactivated by the pilot	Safety / Human factors	Communication
Safety			
6.1	The OCAS shall improve the operational safety of the UAV	Motivation	Processing, actuation
6.2	The operation of the OCAS shall not be disrupting to the workflow of the pilot	Motivation	Communication

Reliability			
7.1	The OCAS shall avoid any physical collision	Motivation	Actuation
7.2	The OCAS shall be operative regardless of the state of the controller board	Safety	Power, processing
Ergonomics and human factors			
8.1	The OCAS shall be operable after a short training by any pilot	Motivation	Communication
8.2	The OCAS should be engaged and disengaged at discretion of the pilot	Safety / FFBD	Communication
Loads			
9.1	The OCAS shall stand the same loads as the UAV	Integration	Structure
Weight			
10.1	The UAV + OCAS shall not weight more than the limit of the UAV segment	Regulations	Hardware
Environment			
11.1	The OCAS shall withstand the effect of open-air flight	Integration	Hardware

Table 5.1: OCAS System-level Requirements

Notice that Table 5.1 is not static, and should be updated during the design process, since some of the tools of Systems Engineering are designed to expose missing requirements. Thus, some requirements have been written at later design stages, as the "Traceability (sourced from)" column shows. Also, the fourth column is to be completed in the subsystem design stage, when the system requirements will be allocated to one or more specific subsystems or components.

5.2 Logical Decomposition

The Logical Decomposition is an intermediate step between the Requirements Capture and the Design phases. Its purpose is to understand the manner in which the requirements affect the way that the system functions, for the requirements loop; and to identify a feasible solution that functions in a way that meets the requirements, for the design loop, as shown in Figure 5.2

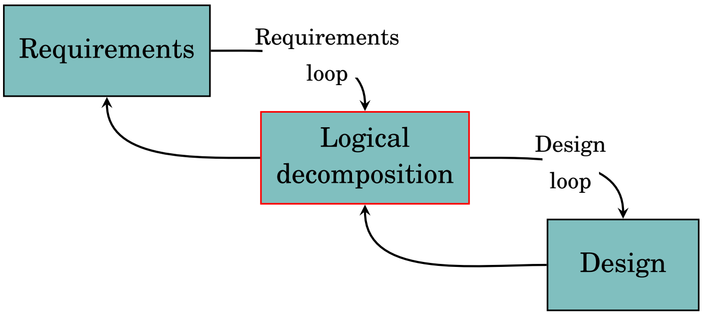


Figure 5.2: The Logical Decomposition phase

5.2.1 Functional Architecture

The logical decomposition performed during the functional analysis decomposes the top level requirements and allocates them down to the lowest desired levels. The main outcome of the process is the Functional Architecture (Figure 5.3), which helps establish relationships between requirements, and ultimately build a System Architecture.

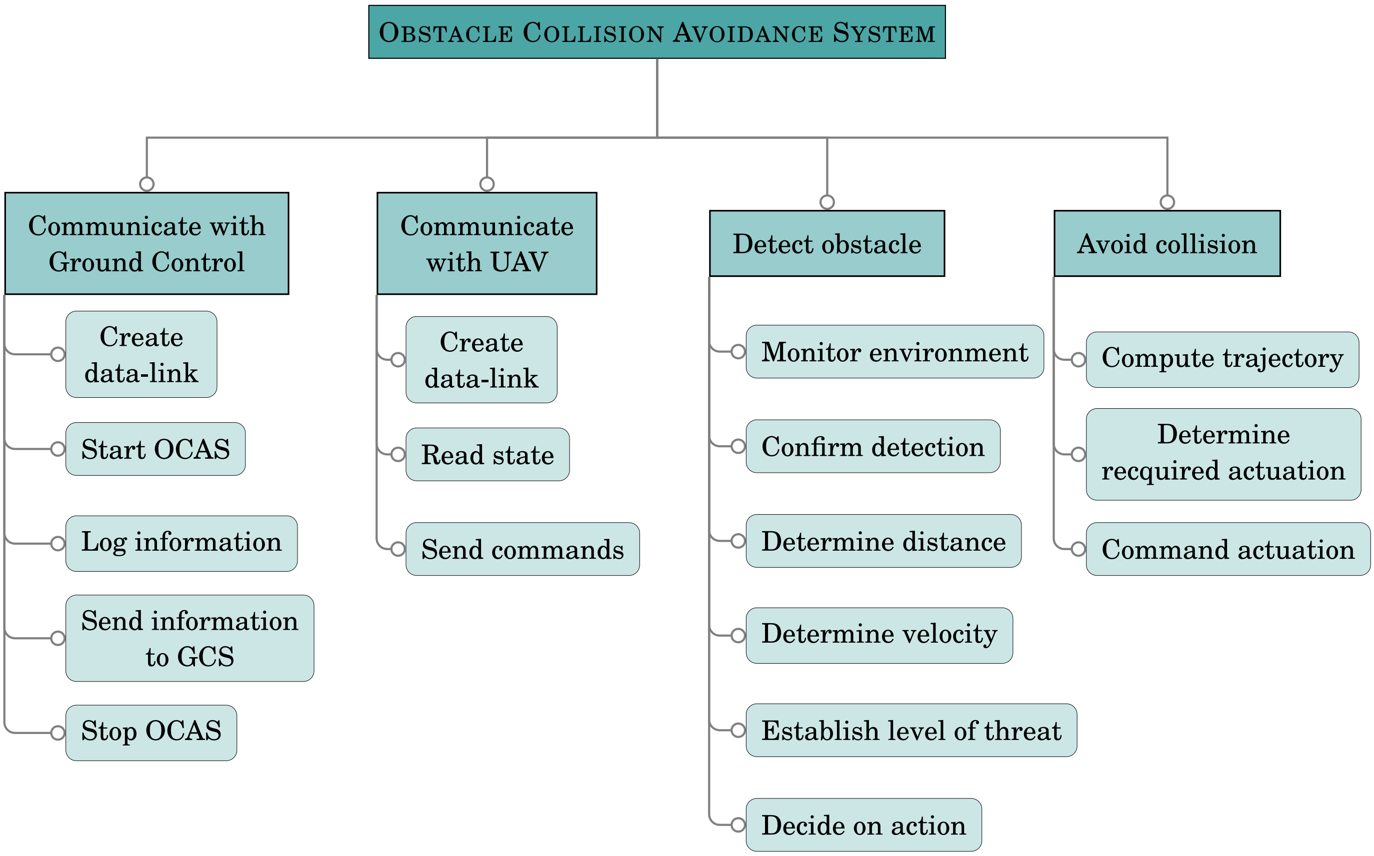


Figure 5.3: OCAS Functional Architecture

The main purpose is to create an association between the requirements and the functions that the system needs to be able to perform to meet them. In the process, any discrepancy or missing items can (and should) be identified and corrected in an iterative manner.

5.2.2 Functional Flow Block Diagram (FFBD)

Once the functions of the system are defined, it is useful to dispose them so that the sequential use of each of them during the mission is shown. To that end, the Functional Flow Block Diagram is used. In the FFBD each function is represented by a block, and it is described in terms of inputs, outputs and interfaces. In the case that a function is composed of several sub-functions, those will be represented hierarchicallyfrom the top level down to the most specific sub-function, maintaining the general flow.

The FFBD shows *what* must happen, and provides an end-to-end path considering all the functionality of the system and the predefined use-case scenarios. Parallel or alternate paths might be considered.

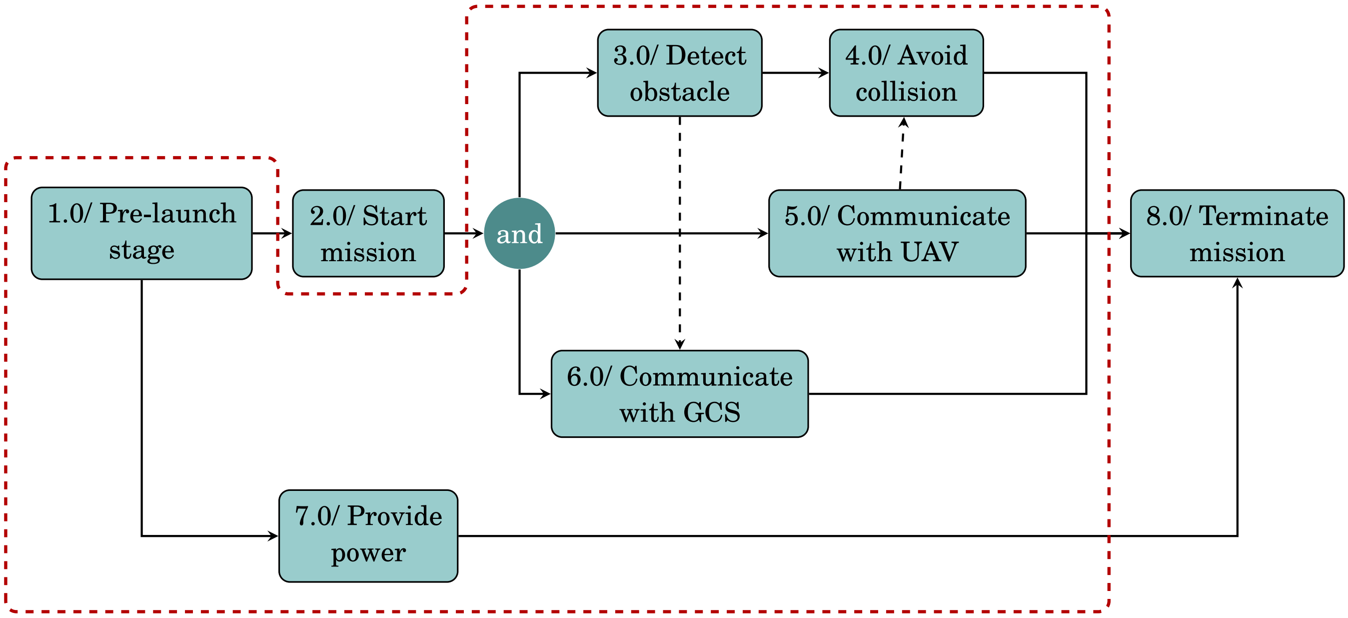


Figure 5.4: OCAS Functional Flow Block Diagram. TOP LEVEL

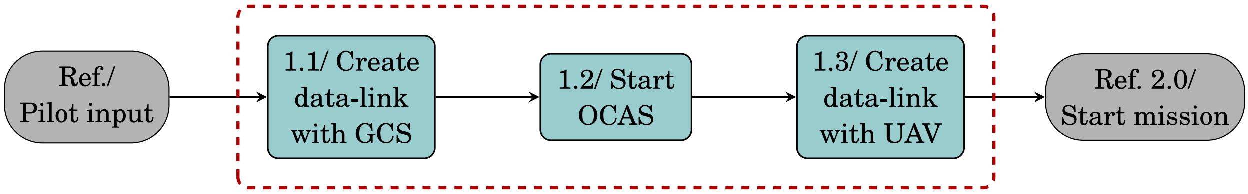


Figure 5.5: OCAS Functional Flow Block Diagram. 1st STAGE

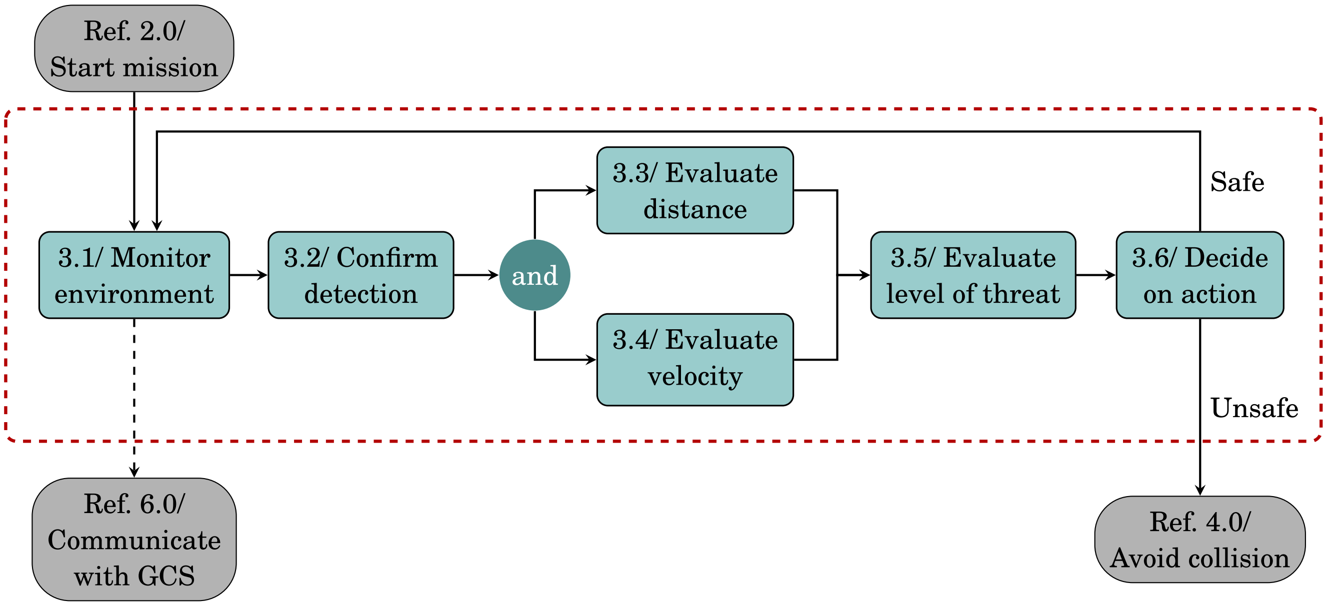


Figure 5.6: OCAS Functional Flow Block Diagram. 3rd STAGE

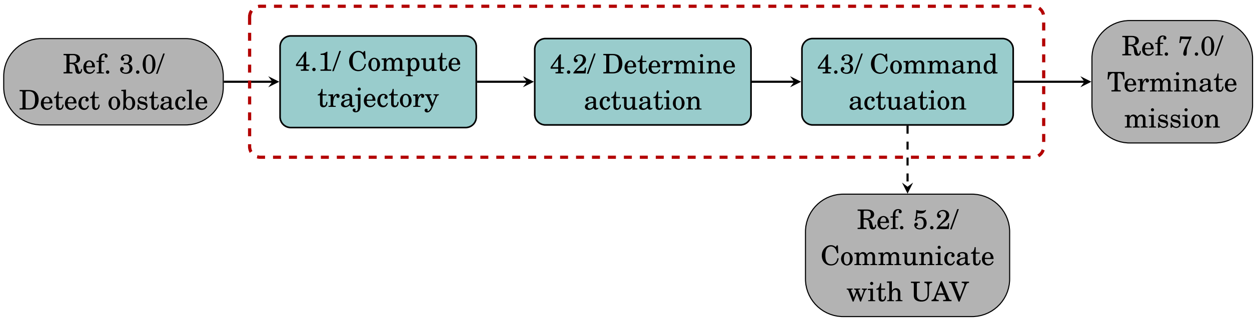


Figure 5.7: OCAS Functional Flow Block Diagram. 4th STAGE

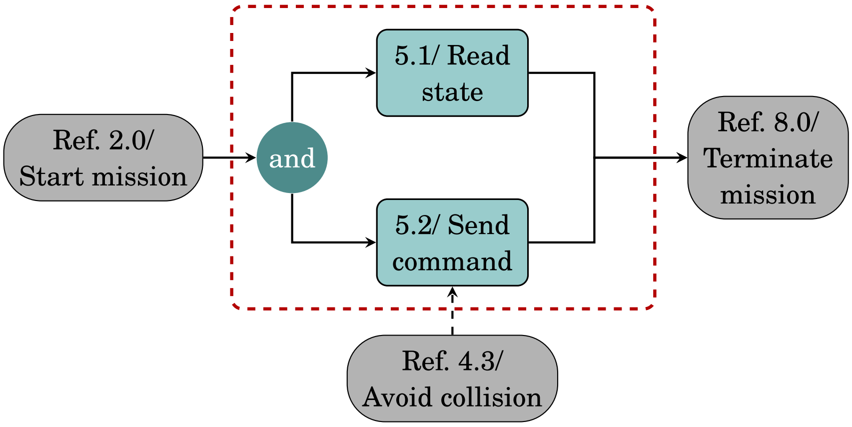


Figure 5.8: OCAS Functional Flow Block Diagram. 5th STAGE

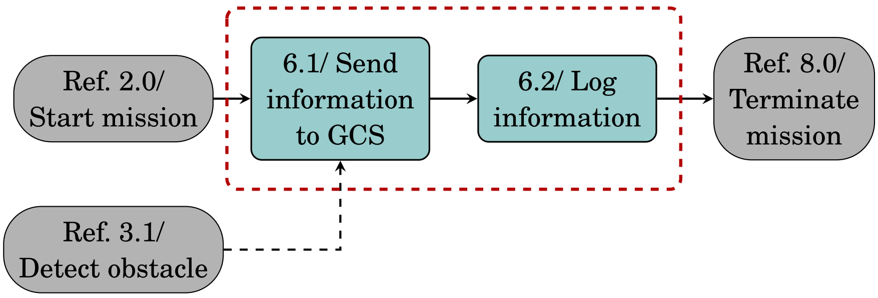


Figure 5.9: OCAS Functional Flow Block Diagram. 6th STAGE

For the block diagrams depicted in Figures 5.4 to 5.9, the symbology explained below is used:

- represents an individual function or subfunction as defined in the Functional Architecture from Figure 5.3.
- represents a logical *and* or *or* gate for defining parallel or alternative paths, respectively.
- represents a reference block that specifies the origin or destination of a path from an external function of the system.
- represents the boundaries of the functional description, be it the whole system or a subfunction of it.
- → indicates the secuential order that is to be followed from one function to another.
- - -> indicates an information flow between two functional blocks.

5.2.3 Product Breakdown Structure (PBS)

Once the functions are properly defined, they need to be allocated to the subsystems that will be in charge of accomplishing them. To that end, the system is decomposed in its forming subsystems ensuring that all the functions can be achieved by the system. The decomposition process is visually show via the Product Breakdown Structure, which is represented in Figure 5.10

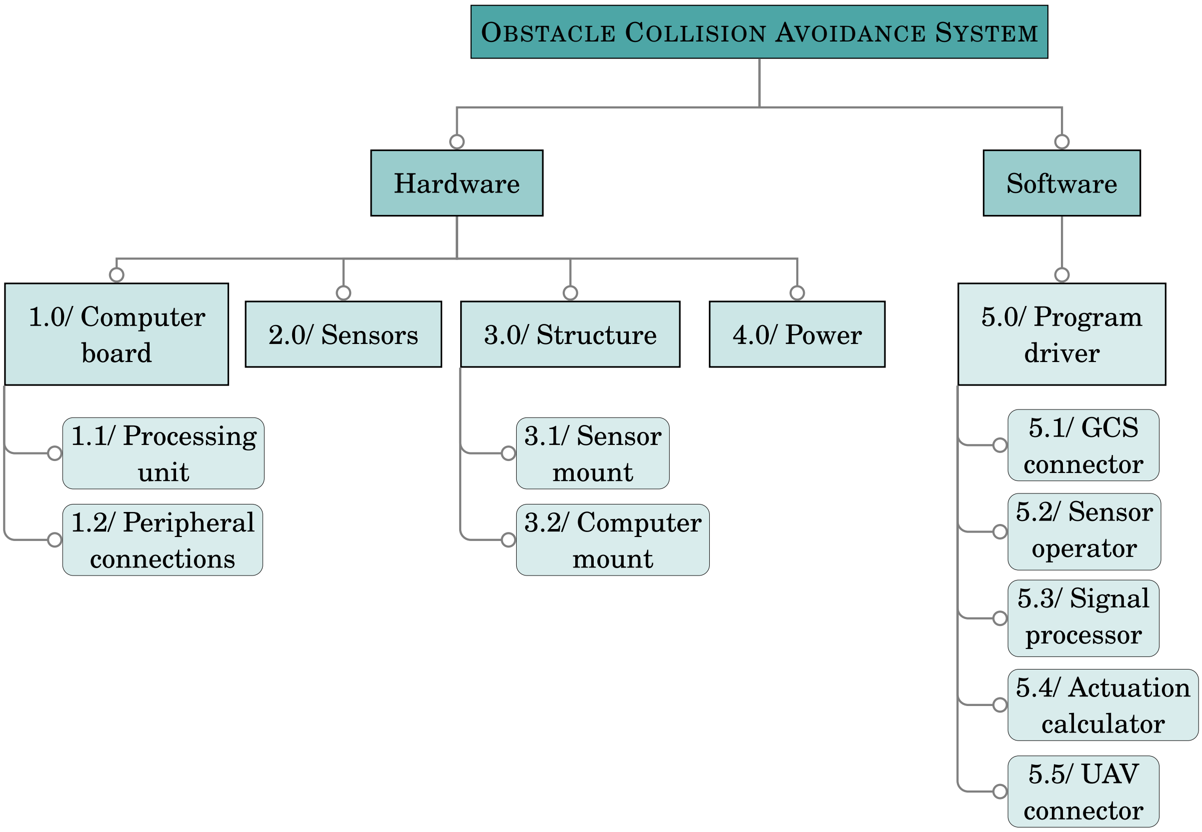


Figure 5.10: OCAS Product Breakdown Structure

5.2.4 Functional-Physical matrix

Finally, to couple the Requirements (Table 5.1) with the Functional Architecture (Figure 5.3) and with the Physical product (Figure 5.10), a functional-physical matrix can be built. This tool is very relevant since it exposes possible mismatches between the three steps, which would lead to requirements not being met or a product that cannot perform its intended functions. Thus, by filling the matrix, the designer can go back to previous steps and adjust anything that is needed in order to avoid the exponential increase in cost that was mentioned at the beginning of the present Chapter.

For the matrix represented in Table 5.2, the requirements, functions and subsystems have been represented by their ID as defined in Table 5.1, Figures 5.4 to 5.9 and Figure 5.10, respectively.

		Subsytem ID										
		Hardware						Software				
Req. ID	Funct. ID	1.1	1.2	2.0	3.1	3.2	4.0	5.1	5.2	5.3	5.4	5.5
1.1	All	*	*	*	*	*	*	*	*	*	*	*
1.2	All	*	*	*	*	*	*	*	*	*	*	*
2.1	All						*					
2.2	All				*	*	*					
3.1	3.0	*	*	*	*		*		*			
3.2	4.0	*	*				*			*	*	*
3.3	4.3	*	*				*		*	*	*	*
3.4	5.2	*	*								*	*
3.5	6.0	*	*				*	*				
4.1	3.1-3.4			*	*		*		*	*		
4.2	3.1-3.4			*	*		*		*	*		
4.3	7.0						*					
5.1	5.1	*	*				*					*
5.2	5.2	*	*				*					*
5.3	6.1	*	*				*	*				*
5.4	1.2	*	*				*	*				
6.1	4.0	*	*	*	*	*	*	*	*	*	*	*
6.2	6.0		*		*	*	*	*			*	
7.1	4.0	*	*								*	*
7.2	6.0	*	*				*	*	*			
8.1	1.0,6.0	*	*					*				
8.2	1.2		*					*				
9.1	Perf.	*	*	*	*	*	*					
10.1	Perf.	*	*	*	*	*	*					
11.1	Perf.	*	*	*	*	*	*					

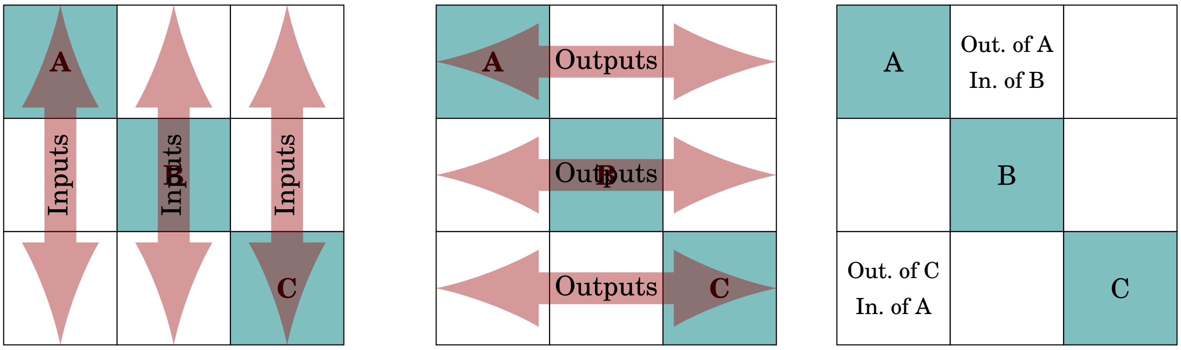
Table 5.2: OCAS Functional-Physical matrix

As it can be seen for requirements 9.1, 10.1 and 11.1, they are requirements affecting the entire system, and thus cannot be associated to any function: they are requirements that impose hardware constrains only.

5.2.5 Interfaces definition (N^2 diagram)

For the correct integration of the system the definition of its interfaces is of utmost importance. The N^2 diagram is commonly used for the development of those interfaces.

In the N^2 diagram, an N×N matrix is built. In the main diagonal all the systems or subsystems are placed, while the upper and lower triangles are reserved for the interfaces, which are clasified into inputs and outputs. An input to any of the modules is represented through its vertical cells, while the output is place on the horizontal rows. An example N^2 diagram can be found in Figure 5.11



(a) N^2 inputs (b) N^2 outputs (c) N^2 diagram

Figure 5.11: Example of a N^2 diagram

For the Obstacle Collision Avoidance System N^2 diagram in Figure 5.12, the subsystems defined in the Product Breakdown Structure (Figure 5.10) will be placed in the main diagonal.

In this Figure, the grey blocks represent external systems to the OCAS, while the blue ones are subsystems of the OCAS itself. In a regular UAV, with no OCAS implemented, the pilot would only be executing the outermost green loop: only in direct control with the UAV. Nonetheless, the OCAS provides an additional layer of safety that is functionally placed between the pilot and the UAV; but also adds another interface the pilot has to deal with: the GCS connector is the gate to the OCAS, which in turn connects via the Central Processing Unit to the rest of the system. Hence, the "green" interfaces designate the human links with both the OCAS and the UAV.

Further, the "yellow" interfaces have been highlighted since they represent the traditional sensing and control problem. Notice how the state of the UAV is transmitted from the UAV connector (that information is relayed by the UAV) and the sensors to the Processing Unit. From the hardware-software interface downwards, the information is processed, the actuation calculated and ultimately the command is sent to the UAV.

The "blue" interface has been created for information logging in response to Function 6.2 as defined in the FFBD. It could also serve as a debugging channel during the implementation phase.

Finally, the "black" interfaces on the hardware side shows how the system has been designed meeting requirements 2.1 and 2.2, which stated that the OCAS shall be independent of the UAV and self-contained. In this case, it has been decided that the OCAS would carry its own power source to provide energy to its components. However, the power is not transmitted down to the UAV, which could be more weight efficient but would significantly modify the architecture of the original UAV, disregarding one of the main requirements from the motivation of the project.

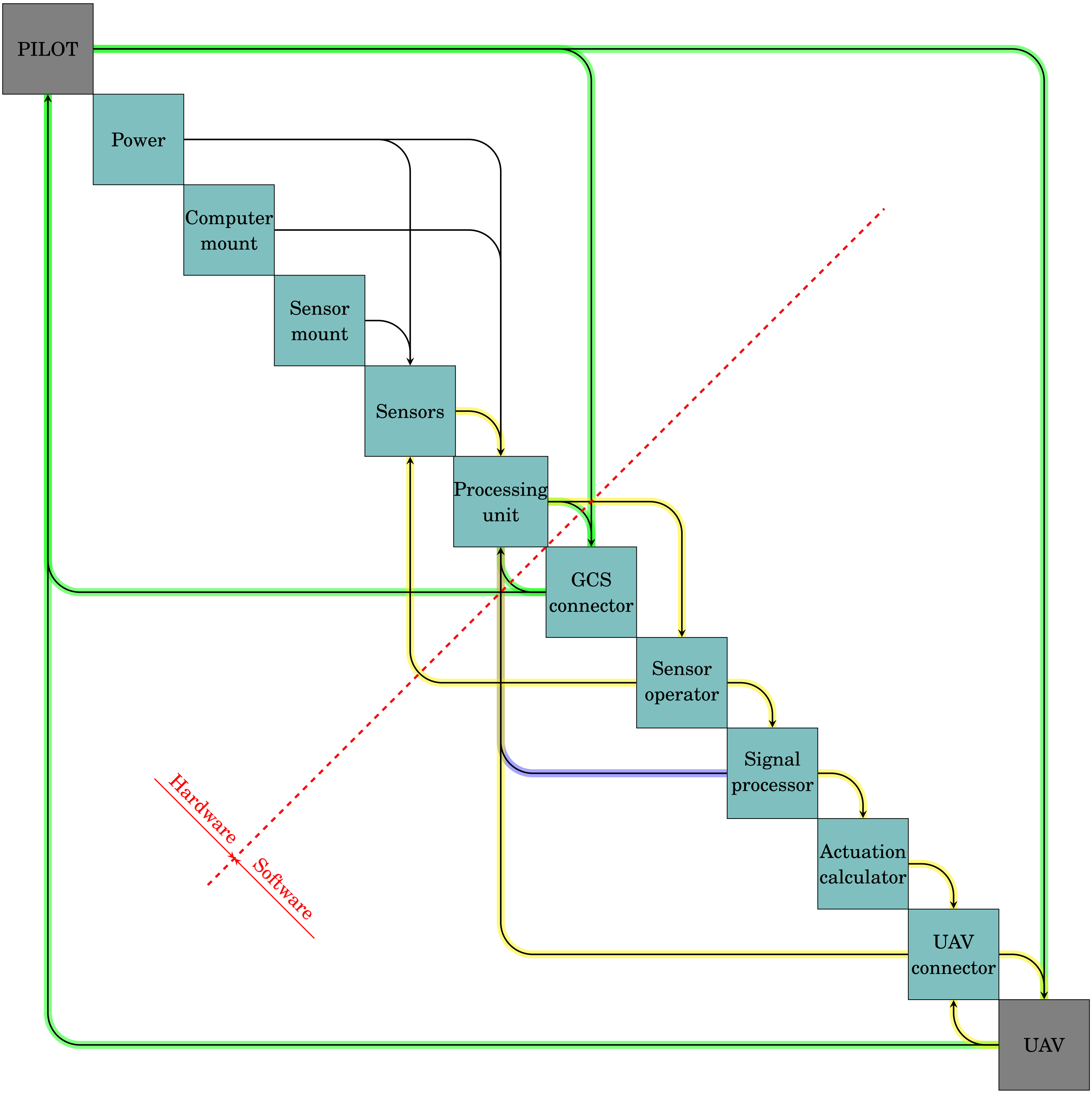


Figure 5.12: OCAS N^2 diagram for interfaces definition

5.3 Component choice

Only after properly defining the requirements, functions and subsystems of the Obstacle Collision Avoidance System can the components be chosen to ensure that the system is correctly designed to meet specifications.

5.3.1 Sensors

The most relevant sensing alternatives were already exposed in Section 3.1. In this Section, the most appropriate one for the project will be chosen according to the specifications.

Certainly, all the options considered have their advantages and disadvantages. The purpose of the selection process is to evaluate all of those in a trade-off study which ensures that the final selection provides the best alternative to the system. The steps to a successful trade-off study are:

1. Definition of the problem
2. Definition of the constraints
3. Generation of alternative solutions
4. Definition of the evaluation criteria
5. Definition of the weight factors
6. Fulfillment of the trade study
7. Ranking of the solutions

For the three first steps, those have already been done in Chapter 4, Section 5.1 and Section 3.1, respectively.

For the rest of the steps, a tabular format will be used.

The evaluation criteria can be defined as the parameters considered important for the sensors to fulfill towards the correct achievement of their function, and will be presented on the first column of the table.

A weight factor will be given to each of the evaluation criteria according to their importance in the sensing and integration problems. Those will be normalised ensuring that the sum of all of them add up to unity, and will be presented in the second column of the table.

The trade study will then be fulfilled by rating each of the alternatives on the evaluation criteria as previously defined. The mark given, ranging from 0 to 1, represents the capability of the alternative on the corresponding evaluation criteria.

Finally, the ranking of the solutions is performed by combining (multiplying) the criteria's weight factors with the ratings given to the alternatives on each of those criteria, to ultimately sum all of those and obtain a single figure for every alternative considered. The objectively most appropriate alternative is the one with highest rating after the trade-off study.

As it can be seen in Table 5.3, the most appropriate sensor to be used in the OCAS is the ultrasonic rangefinder. The reasons for that are mainly the high score obtained in the ease of operation, integration and processing, as will be seen during the implementation process, as well as its low cost; which were all considered to be important properties for the chosen sensor to meet for this project.

		Computer vision		Sonar		Lidar		Radar	
Criteria	Weight factor	Rating	Combined	R.	C.	R.	C.	R.	C.
Accuracy	0.1	0.4	0.04	0.8	0.08	1	0.1	0.6	0.06
Range	0.12	1	0.12	0.4	0.048	0.8	0.096	0.6	0.072
Ease of operation	0.12	0.6	0.072	1	0.12	0.6	0.072	0.4	0.048
Ease of integration	0.15	0.6	0.09	0.8	0.12	0.6	0.09	0.2	0.03
Ease of processing	0.12	0.2	0.024	1	0.12	0.8	0.096	0.6	0.072
Availability	0.12	0.8	0.096	0.8	0.096	0.6	0.072	0.6	0.072
Cost	0.1	0.6	0.06	1	0.1	0.2	0.02	0.8	0.08
Flexibility	0.05	1	0.05	0.4	0.02	0.4	0.02	0.6	0.03
Weight	0.12	1	0.12	0.8	0.096	0.4	0.048	0.8	0.096
TOTAL			0.672		**0.8**		0.614		0.56

Table 5.3: Sensor alternatives trade-off study

Figure 5.13: Chosen ultrasonic rangefinder: HC-SR04 Source: `arduinolearning.com`

5.3.2 Computer board

The choice of processing unit for the OCAS is not nearly as complex as the sensor case. The main available alternatives are either a microcontroller board or a Single Board Computer (SBC), which differ in the type of CPU architecture.

A microcontroller board can be as simple as a single Atmel AVR microchip, although they generally incorporate additional features for easier programming and connection with other peripherals. The best example of microcontroller boards is the Arduino family. These board usually incorporate an 8-bit processing unit, which can be considered computationally underpowered according to present standards, and thus the programs are frequently coded in C/C++ languages due to their high resource efficiency. Also, these boards do not have any software feature out-of-the-box, which implies that every required function needs to be programmed from scratch on the chip.

On the other hand, an SBC can be thought of as a full Personal Computer, except in a reduced form-factor. These computers do not generally exceed the footprint of a credit card, albeit embodying all the necessary components such as RAM and non-volatile memory, USB ports and even the convenient General Purpose Input / Output (GPIO) pins for low-level hardware integration. Moreover, SBCs are driven by complete Operating Systems (OS), featuring convenient general kernel and communications tools. Additionally, these computers are able to run virtually any computer software available, which also means that applications can be programmed in a wide range of languages.

This brief analysis should be enough to prove that an SBC is more capable on almost any aspect than a microcontroller board and significantly more flexible. Thus, for this project, a Raspberry Pi 2 Model B SBC was selected for the reasons mentioned above and additionally because it is widely available and runs a full Debian Operating System. The only disadvantage is its higher power consumption as compared with the Arduino family of microcontrollers, which can nevertheless be neutralised by powering the OCAS with an off-the-shelf portable USB battery pack which outputs a continuous current of 5V, 2A: just enough to provide energy to the Raspberry Pi and all the peripherals.

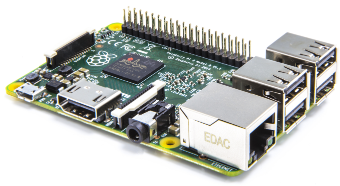


Figure 5.14: Raspberry Pi 2 Model B. Source: `raspberrypi.org`

5.3.3 Other components

Clearly, the most important components of the OCAS are its sensors and processing unit. The other elements are less critical and the selection process was less exhaustive. The component list considered for the prototype will only be listed here for completeness and to aid any interested researcher on the reproduction of the project.

POWER SOURCE As mentioned in the previous section, any portable USB battery pack with at least 5V, 2A continuous current will sufice to power the computer board and its peripherals. The one used for the project is the Amazon Basics 5600 mAh battery, which can potentially last more than two hours powering the OCAS.

NETWORK ADAPTER For the connection with the Ground Control Station, a wireless WiFi network will be used. Unfortunately, the Raspberry Pi does not include one, so the external TP-Link TL-WN822N has to be mounted on the testing platform, although any other model would be just as suitable.

TESTING PLATFORM The testing part of this project is based upon the Bachelor Thesis by M. Arteta [27], which provided an already calibrated and flight-capable F450 quadrotor UAV.

Figure 5.15: Testing platform, with OCAS already integrated

Chapter 6

System implementation

The design phase of the Systems Engineering approach will be presented in the current chapter instead of the previous one since the focus of this phase was put in the software part, for the hardware one (mainly structural mounts) being too dependant on the configuration of the existing UAV. Hence, all the design, implementation and testing of the software branch were conducted in a parallel manner, as will be exposed in this part of the thesis. Nevertheless, the final hardware assembly of the system in the working prototype will also be discussed at the end of the present chapter.

Chapter 6 will describe the complete implementation of the Obstacle Collision Avoidance System within the Unmanned Aerial System starting from the uppermost level and deepening through the execution of the interfaces and the software layers down to the custom-built control script.

6.1 The OCAS within the UAS

This section describes the architecture of the UAS prior to the implementation of the OCAS. Then, the uppermost integration level is explained, emphasising the compliance with Requirement 3.4 (lack of interference with the Ardupilot functions)

6.1.1 Overview of the existing UAS

The regular Ardupilot-based Unmanned Aerial System with Ground Control Station capabilities is composed of three main subsystems:

Firstly, the UAV, which is considered to be fully operable. That is, the UAV concept encloses the airframe, propulsion, power source and all the other components as described in [27]; but most importantly, the controller board with the Ardupilot software, considered as the “brain” of the UAV.

Secondly, the pilot with the Radio Control transmitter (see Figure 2.1a) will also be considered a subsystem of the UAS. He/she has direct control of the UAV when flying in manual mode, plus is responsible of the operation of the GCS when the UAV is in Automatic mode (see Chapter 2).

Thirdly, the computer running the GCS software and having a real-time wireless connection with the UAV while in the air.

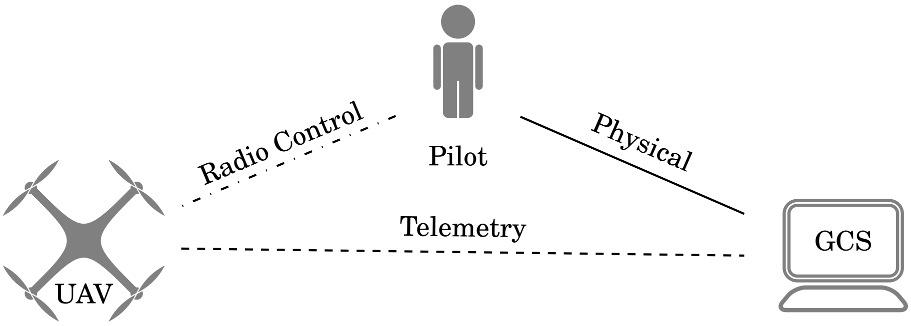


Figure 6.1: Regular Ardupilot UAS architecture

In addition, the interfaces between these subsystems are depicted in Figure 6.1 and work as follows:

The Radio Control (RC) link is established between the RC transmitter held by the pilot and the RC receiver that is directly connected to the controller board. A 2.4 GHz signal transmits information on the position of the control sticks as a PWM directly to the Ardupilot software, as explained in Section 2.1.

Likewise, the telemetry link consists of a 433 MHz duplex radio wave that carries MAVlink messages from the UAV to the GCS and viceversa, allowing for configuration, calibration and operation of the autonomous flight modes while the vehicle is aloft.

6.1.2 Integration of the OCAS

The introduction of the OCAS into the UAS shall preserve the basic Ardupilot functions. Thus, the architecture should not be significantly modifyied. The final decision on the UAS architecture after the integration of the OCAS is shown in Figure 6.2

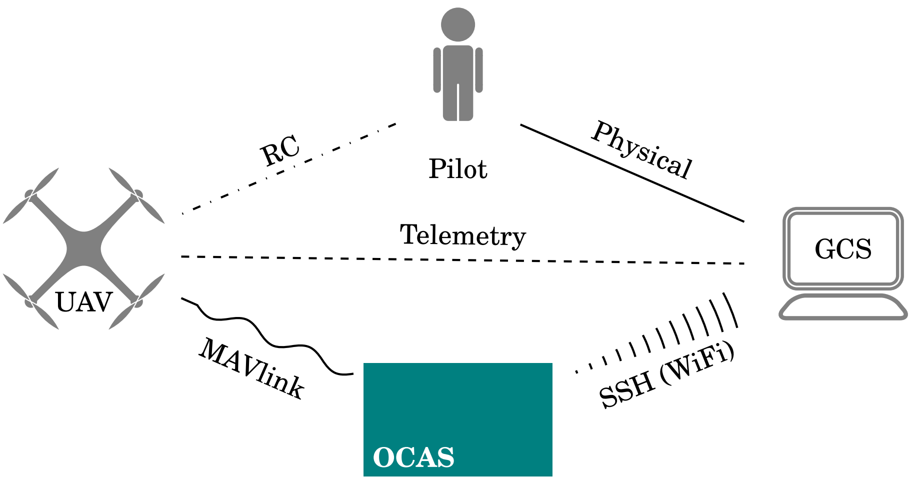


Figure 6.2: OCAS-equiped UAS architecture

With this setup, the original connections and functions are maintained while the OCAS, which is mounted onboard the UAV, communicates with it through a USB cable via the MAVlink protocol (the same one used for the telemetry link). Additionally the GCS has a second wireless link to the OCAS via WiFi, making use of the SSH (plus optional X Window System forwarding) protocol. More information on these interfaces is provided in Section 6.2.

6.2 OCAS peripheral connections (hardware interfaces)

As already stated in Section 5.2.5, the main component of the OCAS is the computer board, which can be considered as a hub on which the rest of the components of the OCAS are brought together. Thus, the first step is to define the information pathways of the Raspberry Pi with the oter hardware components.

The physical layout of the OCAS is shown in Figure 6.3. Notice that only Raspberry Pi peripherals are being considered. They are handled in the following manner:

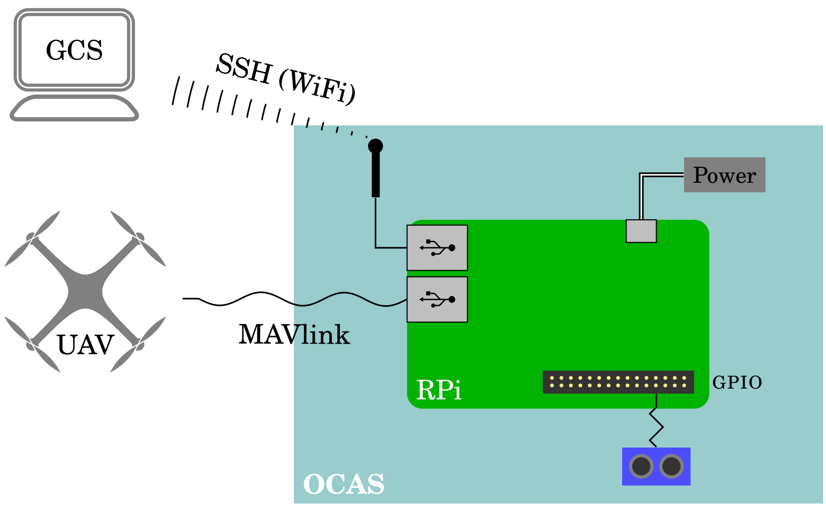


Figure 6.3: OCAS hardware layout

6.2.1 Power connection

For the Raspberry Pi to boot up, the only requirement is to provide a continuous current of 5V and enough current to power any other peripheral as well as the board, which in any case will not be higher than 2A. Thus, the battery pack, providing a continuous source of energy during during the whole duration of the mission is enough to meet the requirement. It is connected to the Raspberry Pi SBC via a conventional USB type A, at the battery end, to micro-USB type B, at the computer end; no additional action being required.

6.2.2 MAVlink connection

The connection with the UAV (i.e. with the Ardupilot controller board) is done also via a regular USB cable, making use of the serial communications protocol. The serial protocol is a simple manner of tranfering information which consists on transmitting the data one bit at a time, avoiding the synchronisation problem. Hence, the only issue is that both ends must agree in advance on the transmission rate. This is done by setting a common “baud rate”, where a baud is the unit for symbol change (signal event) rate, commonly measured in bits per second. In the particular case of communicating with the UAV, the messages transmitted through the serial link are defined according to the MAVlink protocol.

6.2.3 GCS connection

The link with the Ground Control Station is composed of two intermediate steps:

On one hand, the network adapter is connected to a USB port on the Raspberry Pi to provide the SBC with wireless networking capabilities. This connection is entirely handled by the kernel, the adapter’s drivers and the operating system, and needs no further action from the engineer.

The second step is decidedly more complex. Firstly, the Raspberry Pi needs to be set up to wirelessly connect to the same network as the GCS computer. There are several ways to achieve this goal, but an uncomplicated one is to create an ad-hoc network from the GCS computer (running Windows) to which the Raspberry Pi is directly connected. The specific details are explained in Appendix A. This approach has been mainly chosen for its simplicity and portability, but notice that there exist more advanced network architectures that could provide significantly better performance. Secondly, the SSH connection needs to be stablished over the network. The process involves searching for the Raspberry Pi’s address on the network, connecting to the SSH port and, optionally, setting up an X server for an easier Graphical User Interface (GUI) with the OCAS. More details on the steps to be taken are developed in Appendix B.

6.2.4 GPIO connection

The General Purpose Input / Output pins on the Raspberry Pi operate on a notably lower level than the previous hardware connections. As their name implies, the GPIO pins are the most general type of connection the Raspberry Pi can handle. The reason is that these pins have to be manually operated; that is, each of the pins can be set via software to either a HIGH or LOW state, meaning 3.3V or 0V with respect to the Ground (GND) potential, respectively.

Hence, in this project, the GPIO pins will be used to both trigger the ultrasonic rangefinders and read the returning signal that encodes the information on the distance from the sensor to the detected obstacle.

Besides, the sonar is equipped with its own microcontroller, which handles the lowest-level signals. For its operation, it counts with 4 different pins (see Figure 5.13)

1. GND, or Ground, specifies the reference voltage of the device.
2. VCC, which stands for Voltage Continuous Current, powers the sensor at 5V.
3. Trigger is an input signal pin. A HIGH value on this pin triggers (hence the name) a series of short bursts of sound from the piezoelectric speaker, which will rebound on any close obstacle.
4. Echo is the output signal pin. The sensor’s microcontroller processes the sound captured by the microphone and sends a short pulse through the echo pin exactly after the rebounded sound signal is received. Knowing the speed of propagation of sound (given by $a = \sqrt{\gamma R_g T}$) and the time taken for the wave to travel to the obstacle and back, the distance can be calculated with $d = v \cdot t/2$

The ultrasonic rangefinder’s technical documentation can be found in Appendix C.

On the Raspberry Pi side, the VCC pin shall be connected to any 5V pin, the GND pin to a Ground pin, and the Trigger and Echo pins to any numbered GPIO pins, depicted in Figure 6.4.

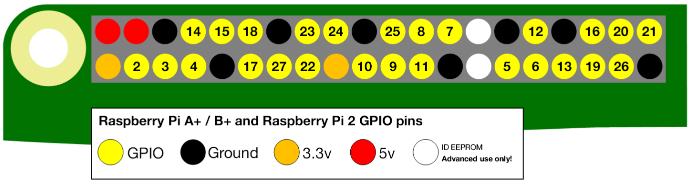


Figure 6.4: GPIO pins on the Raspberry Pi 2 model B Source: `raspberrypi.org`

There is one important issue that needs to be noticed, though. The rangefinders work on 5V only and, while the Raspberry Pi can provide 5V to power the sensors, the GPIO pins can be damaged if operated at more than 3.3V. Thus, the signal pins must be reduced from 5V to 3.3V before being connected to the SBC. The solution to the problem is to use a “voltage divider”, which is a passive circuit that outputs a fraction of the input voltage by means of a pair of resistors, which are connected as shown in Figure 6.5.

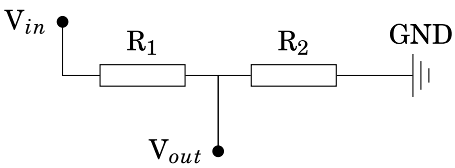


Figure 6.5: Schematic of a voltage divider

In the present case, for the voltage to drop from 5V to 3.3V, the resistors need to meet:

$$\frac{V_{in}}{R_1+R_2} = \frac{V_{out}}{R_2} \Rightarrow \frac{V_{in}}{V_{out}} = \frac{R_1}{R_2}+1 \Rightarrow \frac{R_1}{R_2} = \frac{5V}{3.3V}-1 \Rightarrow \frac{R_1}{R_2} = \frac{1}{2} \tag{6.1}$$

So finally, an ultrasonic rangefinder connected to GPIO pins 14 and 15, for instance, would be connected a shown in Figure 6.6.

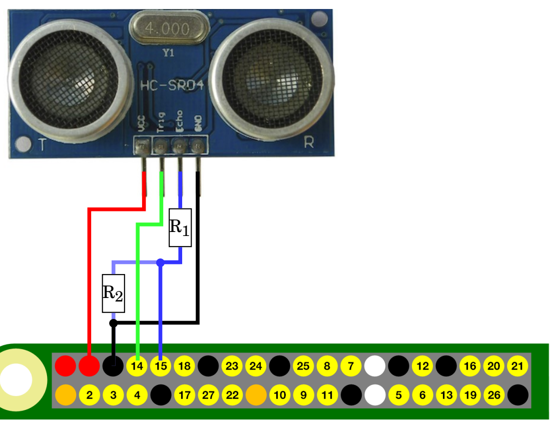


Figure 6.6: Connection of the HC-SR04 sensor to the Raspberry Pi

6.3 Software: Bringing everything together

Having several flows of information arriving to the Raspberry Pi, it is crucial to set up a system that acquires all the data before it can be processed. In the present section such a system will be described.

6.3.1 The Operating System

The first software layer on the Raspberry Pi (apart from the kernel) is the Operating System (OS). In this case, the linux OS is Raspbian Wheezy, which is a version of Debian adapted to be run on the Raspberry Pi's ARMv7 chip.

Raspbian is a complete OS, and as such its abilities are varied, being the most relevant for the project the network management tools and the capability of running external software applications. Within the OCAS, Raspbian will be used as container of the software subsystems specified in the Logical Decomposition phase of the design process (Figure 5.10), plus it will directly handle the functions associated to the GCS connector. An schematic of the relevant software architecture to the OCAS is represented in Figure 6.7.

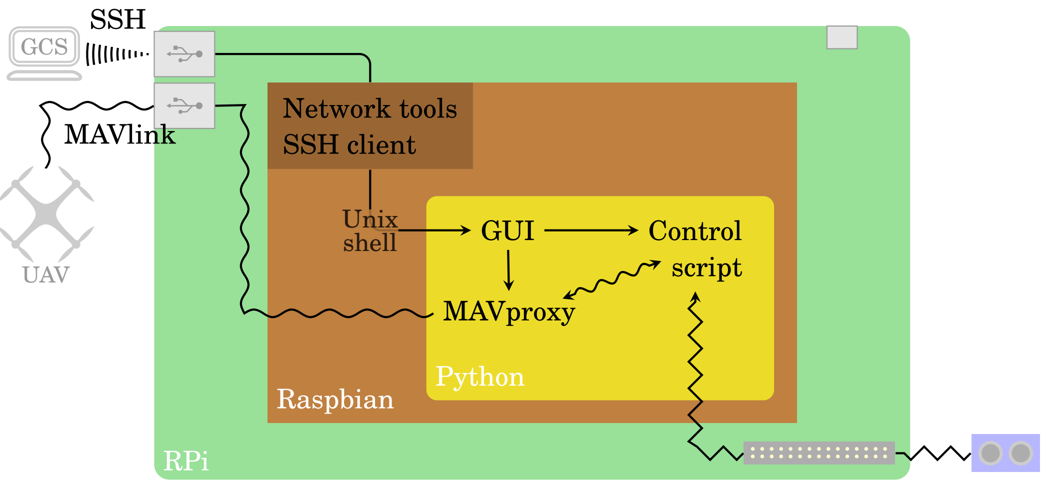


Figure 6.7: Software architecture of the OCAS computer

As it can be deduced from Figure 6.7, all the interaction from the Ground Control Station to the OCAS computer is performed through the Unix Shell, which only provides a command interface with the user. Nevertheless, the SSH connection allows an optional X Window System protocol forwarding (as mentioned in Section 6.1.2); and a Graphical User Interface (GUI) will be developed in response to Requirement 8.1, allowing the execution of MAVproxy as well as the custom control scripts in order to enhance the intuitive operation of the system by the pilot.

6.3.2 MAVproxy

MAVproxy is an open-source Ground Control Station piece of software that is distributed as a Python application. Thus, it can be run on any machine on which a Python distribution can be installed (virtually any operating system).

Its most significant difference compared with traditional GCS software such as Mission Planner or QGroundControl is that MAVproxy is built for the command line and does not need a graphical desktop environment to operate (although a small state window and map are also implemented), which means that it is the most adequate alternative for the operation of the UAV from the OCAS. Furthermore, another decisive feature is the ability of MAVproxy of forwarding the MAVlink messages that are received from and sent to the UAV, allowing the possibility of operating as an intermediate software layer between the UAV and other GCS pieces of software. That in particular is the chosen architecture for the OCAS in terms of communication between the UAV and the custom control scripts, which will be covered in Section 6.4.

Concerning the present section, the setup of MAVproxy will be explained.

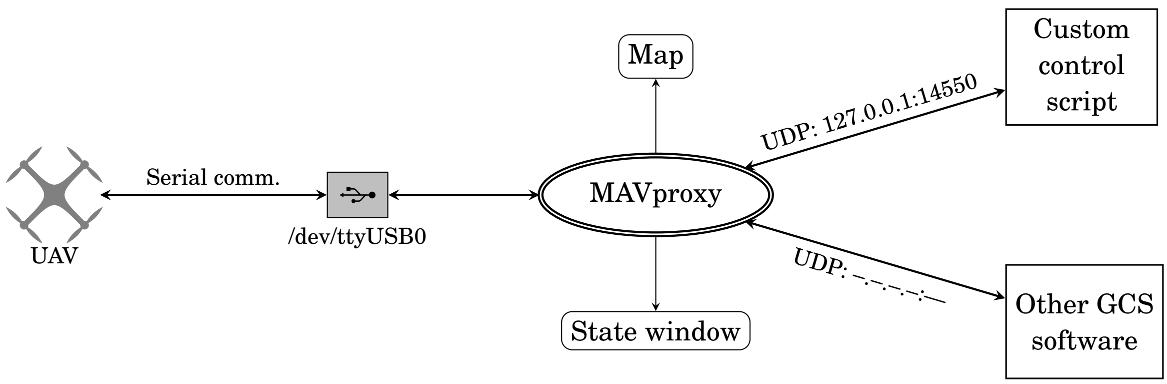


Figure 6.8: MAVproxy setup

As previously mentioned and shown in Figure 6.8, the connection from the UAV to the Raspberry Pi is done via a serial communication through USB cable. The Ardupilot software utilises a baud rate of 115200 bd/s by default, which is an important parameter that needs to be specified to MAVproxy before connecting to the vehicle (else the connection will fail). In addition, the address given by the OS to the USB port (/dev/ttyUSB0 in Figure 6.8) has to be provided too.

Finally, the redirection of MAVlink messages is done via the User Datagram Protocol (UDP), making use of the Internet Transport Layer. However, a remote Internet connection is not needed, since both MAVproxy and the intended target (the custom control script) will be running on the same machine; hence the local IP address can be used (127.0.0.1) together with any available port of choice. The optional second rerouting path is to be defined at the operator's discretion, and can be either a local or remote address.

The auxiliary Map and State windows are internally created by MAVproxy, using only Python function calls and libraries.

For completeness, the command to be executed in order to run the MAVproxy GCS software with the mentioned settings is:

```
$ sudo mavproxy.py --master=/dev/ttyUSB0 --baud=115200 --out=127.0.0.1:14550 --out=
127.0.0.1:14551
```

6.3.3 The Python environment

Python is an open-source general-purpose programming language built to be powerful and easy to use. Additionally, it is implemented to run on virtually any machine, providing interpreters and compilers for most of the operating systems available, which means that the source code can be seamlessly ported from one system to another without any modifications to the source code. Furthermore, Python has a considerable community of developers that contribute to the development of the language through a vast repository of libraries which allow for a greater abstraction and automation of common tasks.

These features and flexibility of Python make it ideal for the development of the OCAS. In particular, there exists a community of developers leaded by 3D Robotics who are creating an Application Programming Interface (API) that provides several useful tools for the communication and operation of Ardupilot-based UAVs. For instance, DroneKit API[1] creates a `vehicle` class upon connection to a MAVlink stream (both serial and UDP protocols are supported) which is automatically updated at a rate of 50 Hz and stores the instantaneous values of important state variables of the UAV, such as absolute GPS position (latitude, longitude, altitude), relative position with respect to the take-off location (north, east, down) or velocity in the body-fixed reference frame, among others. Moreover, it also provides some routines that translate commands like *take-off*, *change flight mode* or guiding instructions and reference states into MAVlink messages that can be readily sent to the UAV through the `vehicle` class.

In addition, DroneKit includes a branch of development that aims to provide other developers with an Ardupilot Simulator. The approach taken is to simulate the Arduino control board and other hardware on the UAV by means of software, hence the name Software-In-The-Loop (SITL) simulator. However, DroneKit-SITL does not aim to provide an accurate physical representation of the UAV [28], since the physical properties change from vehicle to vehicle; the main goal of the simulator is to emulate the Ardupilot firmware, so that MAVlink communication and commands can be safely tested prior to their implementation on the physical platform. Conveniently enough, DroneKit-SITL can be installed as a Python application, and outputs the MAVlink messages via the TCP Internet protocol, which is additionally supported by MAVlink as an input data stream, similarly to the Serial communication by the real UAV.

6.4 The Python script

At this point it can be useful to collect all the information generated in Chapters 5 and 6 do determine what functions and components have still not been covered and shall be implemented within the custom control script. To that end, an allocation matrix can be created as represented in Table 6.1. In this matrix, the functions as defined in Figure 5.3, the subsystems from Figure 5.10 (software branch) and the implementation in Chapter 6 up to this point will be related. Besides, a fourth column will represent the structure that the Python script will follow, and will be used during its development and coding phases.

As it can be noticed, there are some script components that have not been made modular (encapsulated in a class). The reason is that those actions are very dependent on the actual computational approach taken by the programmer, and can become significantly complex algorithms. Since the development of those algorithms is not within the objectives of the project, the coding phase has been simplified by inserting those functions directly into the “main” function as simple routines, even though if complex functionality is to be implemented, the most adequate approach would be to create classes to group all the related information to perform those tasks.

6.4.1 Script architecture

The tasks to be performed by the script are quite time dependent, since they have to be executed alongside the main mission. Thus, the nature of the script needs to be relatively sequential (following the functional

[1] Documentation can be found at `www.dronekit.io`

Function	Logical component	Actual component	Script component
GCS data-link	GCS connector	WiFi / SSH	
Start OCAS	GCS connector	GUI	GUI
Send data to GCS	GCS connector	SSH	
Log information	Program driver	Script	Logger
Stop OCAS	GCS connector	SSH	SSH
UAV communication	UAV connector	Script	DroneKit API
Monitor environment	Sonar operator	Script	Sonar class
Determine distance	Sonar operator	Script	Sonar class
Determine velocity	Sonar operator	Script	Sonar class
Confirm detection	Signal processor	Script	Sonar class
Level of threat	Signal processor	Script	Sonar class
Decide on action	Signal processor	Script	Auto class
Compute trajectory	Sonar operator	Script	Sonar class
Determine actuation	Actuation calculator	Script	main
Command actuation	UAV connector	Script	DroneKit API

Table 6.1: Functional and component allocation matrix

paths from the FFBD in Figures 5.4 to 5.9). Nevertheless, the Functional Diagrams also show that some tasks need to be performed simultaneously for the correct execution of the mission. Hence, for the diverging paths in Figure 6.9, a multi-threading processing approach has been implemented on the functions that are to be evaluated in parallel to each other.

Figure 6.9 represents the `main.py` file within the script, from which additional classes and methods are derived and used during execution. For instance, the "Observe state" blocks are indeed performed by the Sensor operator from the PBS, and has been implemented as a set of variables and methods within the `Sonar` class, allowing for example for the simplified operation of multiple rangefinders by creating multiple instances of the same class.

In the following subsections within Section 6.4, all the functionality of the Python script will be explored, starting from the auxiliary classes and functions to finally combine all the missing parts of the Obstacle Collision Avoidance System into the `main` Python script. The entire Python files will be included in the Appendices for completeness.

6.4.2 Multi-threading capabilities

As mentioned earlier, it is important for the OCAS to execute several critical tasks at in parallel, avoiding interference between them. Fortunately, Raspbian is a multitasking Operating System, and Python provides a `threading` library that manages the system calls invisibly to the user.

For achieving better accessibility, the `thrd` class has been developed to allow calls to the `threading` library to perform actions defined by functions which might or might not demand additional arguments. The complete code is displayed in Appendix D.

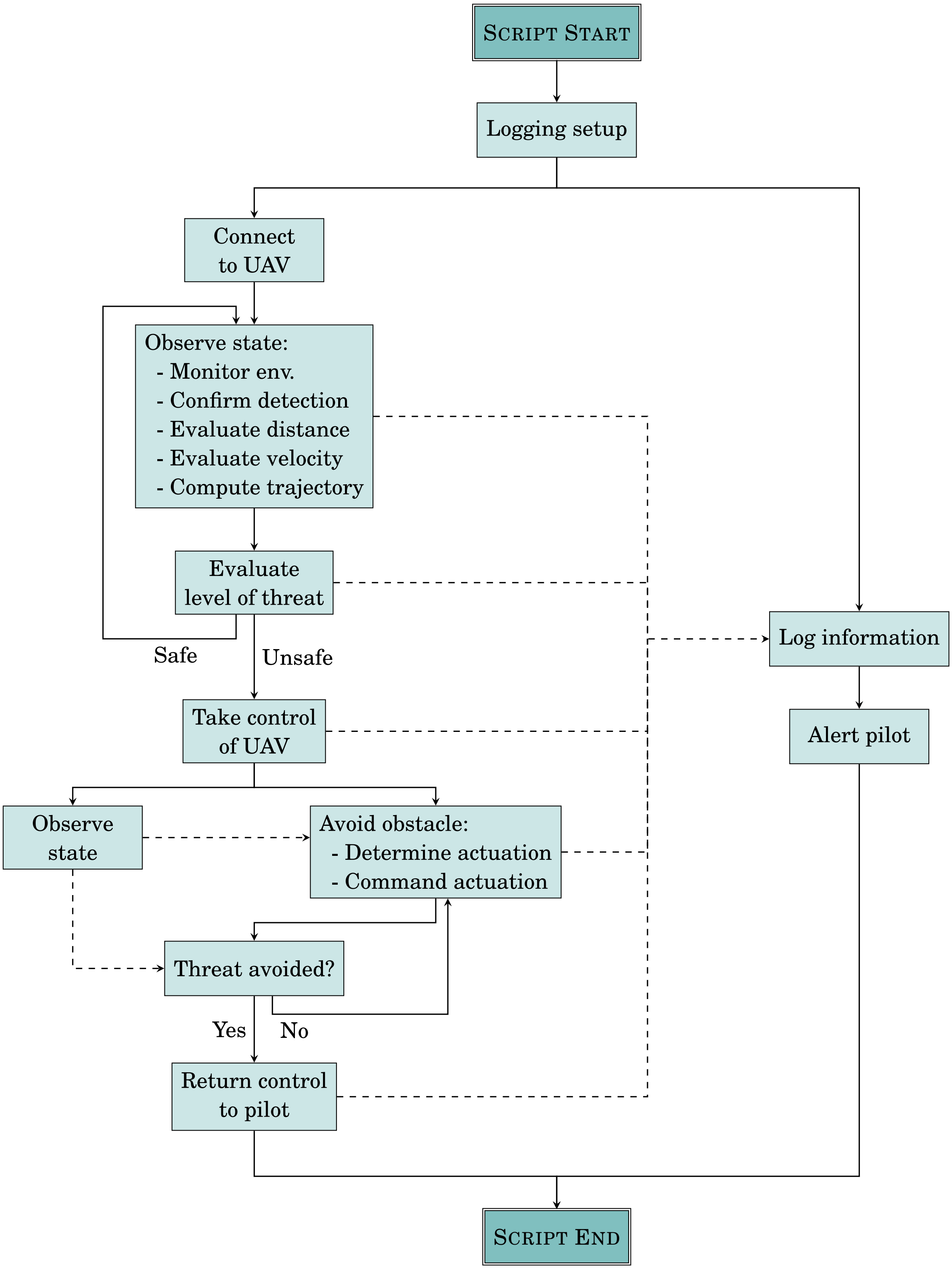


Figure 6.9: Functional flow diagram of the Python script

6.4.3 Log information

Information report is done from the script in two manners: text with relevant data is immediately printed on the GCS screen through the SSH connection, but additionally those same messages are stored in a dedicated log folder contained in the script's directory with precise information on the time each event is recorded. The logs are handled with the `logging` library, which is set up at the beginning of the `main` file as shown in Appendix E.

6.4.4 Connect to UAV

The file `connect.py` (see Appendix F) defines the function `Connect()` which conveniently encapsulates the function `dronekit.connect`[2] and makes the default arguments equal to the output settings of MAVproxy as defined in Section 6.3.2. In addition, the syntax `Connect(mode,address)` can be used for the cases when MAVproxy is not set up to work as an intermediate layer, accepting the Serial, UDP and TCP communication protocols on any local or remote address.

The output of the function is the `vehicle` class which handles all the communications with the UAV (through MAVproxy or otherwise) and updates the values of the state variables automatically on the background, as well as providing some convenient methods for interacting with it.

6.4.5 Observe state

The `Sonar` class performs the most important functions of the OCAS. It is not only responsible for operating the ultrasonic rangefinders, but also performs some basic signal processing to determine the speed and velocity of the UAV with respect to the detected obstacle, deciding if could become a threat for the flight and triggering the avoid manoeuvre. In the future, it might be interesting that these functions, which in principle could contain rather complex algorithms and processing techniques, would be developed in separate classes to enhance the modularity and upgradeability of the system.

The `Sonar` class defines three different methods which operate the sonar and calculate the distance to the closest obstacle, compute the velocity from distance measurements and evaluate the potential of a collision to trigger the avoiding manoeuvre, respectively.

6.4.5.1 Measure distance

The `measureDistance()` (Appendix G. Lines 41 to 91) method triggers the ultrasonic rangefinder defined at the initialisation (`__init__()`) of the class to take a measurement following the procedure from the technical documentation of the sensor (Appendix C), which specifies to start with the Trigger pin in LOW state, change it to HIGH state for at least 10 μs, and return to LOW state. These commands are sent in a parallel thread, created with the `thrd` class, to avoid timing issues with the `main` script. After the Trigger signal is sent, two system interrupts are set with the help of the Raspberry Pi's GPIO library to listen to the Echo pin for both the rising and falling edge, storing the times at which they happen. The distance to the obstacle can be calculated with the flight time of the ultrasonic signal, which has the same duration as the HIGH state time of the Echo signal returned by the sonar, with Equation (6.2) where d is the distance

[2] In Python, functions derived from libraries prepend the name of the library to the function itself, separated by the "dot" syntax

to the obstacle, t_{echo} is the time of the HIGH state in the Echo pin and $a(T)$ is the speed of propagation of sound.

$$d = a(T) \times \frac{t_{echo}}{2} \tag{6.2}$$

Unfortunately, the speed of sound depends on the temperature of the air it propagates through. Thus, if the most accurate results were to be achieved, a temperature sensor should be integrated to compensate for temperature variations during operation. Nevertheless, the effect that reasonable temperature fluctuations have on the final distance is relatively small, as shown in Appendix H, and can be effectively neglected, setting $a(T) = 340m/s$ as the standard and constant speed of sound value.

Occasionally, the rangefinders give incorrect measurements (probably due to multipath errors, cross-interference or noise) that can make the algorithms believe that the UAV is moving closer or faster to the detected obstacle than reality, which causes false-positive activation of the "avoid" procedures. To prevent these kind of errors, a rolling average with a default window of 5 measurements is computed on the distance, even though more advanced filtering techniques could be implemented in the future, ranging from a basic low-pass filter to the more complex Extended Kalman Filter (EKF) combined with IMU data [29], for instance.

6.4.5.2 Compute velocity

The `computeVelocity()` method in the `Sonar` class computes the speed of the vehicle with respect to its closest detected obstacle. Notice that since the ultrasonic rangefinders are non-directional (they provide the distance to the obstacle regardless of the direction it is found, as long as it lies within its field of view), the returned speed is effectively the normal component of the velocity vector of the UAV with respect to the obstacle.

Again, due to noise issues, the signal should be filtered before being fed to the decision module; but this time the rolling average does not seem appropriate since the velocity can be modified at a fairly high rate on quadcopter vehicles as is the case for the testing platform. Thus, a first-order, three-data-points backward difference approach is suggested [30], since it provides convenient damping properties, although higher-order or bigger stencil approximations would also be appropriate.

The mathematical derivation is as follows:

The normal component of velocity obeys the equation

$$v = \frac{dx}{dt} \tag{6.3}$$

where x is the distance measurement taken by the ultrasonic rangefinder.

Performing the Taylor expansion to that equation at both the previous data point and its preceding, that is, at $t_1 = t_0 - \Delta t_1$ and $t_2 = t_0 - \Delta t_2$ being t_0 the most recent data point, the expressions obtained, respectively, are:

$$x_1 = x(t_1) = x(t_0) - \left.\frac{dx}{dt}\right|_{t_0} (t_0 - t_1) + \mathcal{O}\left(\Delta t^2\right) \tag{6.4}$$

$$x_2 = x(t_2) = x(t_0) - \left.\frac{dx}{dt}\right|_{t_0} (t_0 - t_2) + \mathcal{O}\left(\Delta t^2\right) \tag{6.5}$$

Summing both equations together:

$$x_1 + x_2 = 2x_0 + \left.\frac{dx}{dt}\right|_{t_0} [(t_0 - t_1) + (t_0 - t_2)] \tag{6.6}$$

Since $\frac{dx}{dt}\big|_{t_0} = v$:

$$v(t_0) = v_0 = \frac{x_1 + x_2 - 2x_0}{2t_0 - t_1 - t_2} \tag{6.7}$$

From equation (6.7), v_0 is the returned value that is used to predict a potential collision.

6.4.5.3 Calculate collision

To successfully predict a potential collision, the future state of the vehicle shall be estimated. For the first prototype, the collision will be anticipated considering the present position and velocity of the UAV with respect to the closest obstacle, together with some intrinsic parameters.

The algorithm computes the parameter t_{safe} which encapsulates all the available information so that when $t_{safe} < 0$, a collision is expected to happen.

$$t_{safe} = t_{collision} - t_{reaction} - t_{stop} - t_{margin} \tag{6.8}$$

From equation (6.8), $t_{collision}$ is the estimated time to the obstacle computed with the actual velocity as computed in Section 6.4.5.2, $t_{reaction}$ is the time taken by the Python script to actually take control of the UAV after the obstacle situation has been considered as unsafe, t_{stop} can be estimated according to the avoidance procedure as the time it would take to completely stop the vehicle after control has been taken by the script, and finally t_{margin} is a figure representing the clearance to the obstacle after the avoidance manoeuvre is complete, also accounting for possible sensor errors, as shown in Figure 6.10.

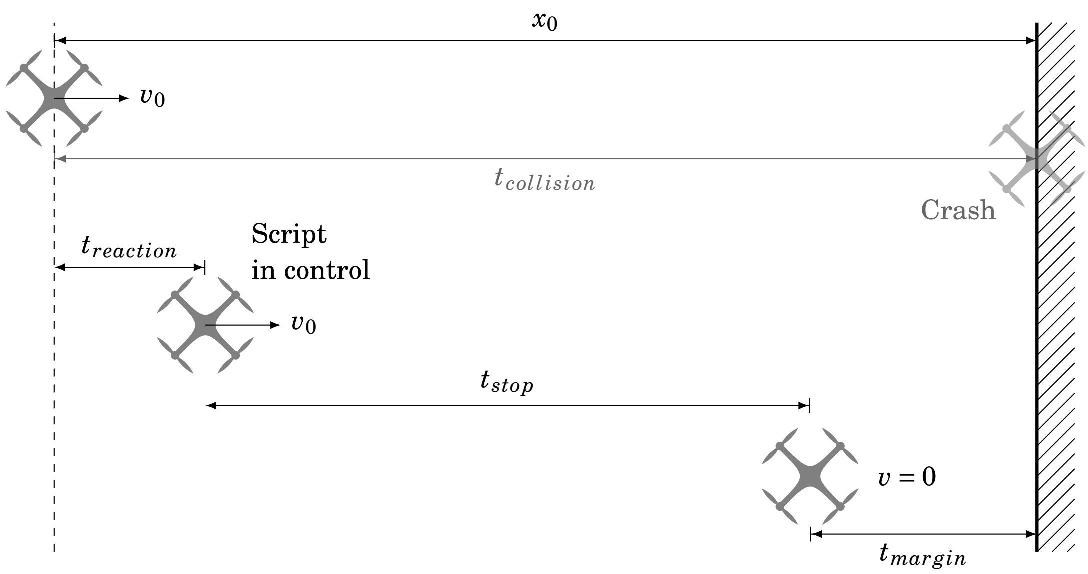


Figure 6.10: Prediction of a collision by the OCAS

APPENDIX A

Creation of GCS wireless network

Pending

Appendix B

SSH connection with the GCS

Pending

APPENDIX C

Technical documentation of the HC-SR04 rangefinder

Ultrasonic Ranging Module HC - SR04

Product features:

Ultrasonic ranging module HC - SR04 provides 2cm - 400cm non-contact measurement function, the ranging accuracy can reach to 3mm. The modules includes ultrasonic transmitters, receiver and control circuit. The basic principle of work:
(1) Using IO trigger for at least 10us high level signal,
(2) The Module automatically sends eight 40 kHz and detect whether there is a pulse signal back.
(3) IF the signal back, through high level , time of high output IO duration is the time from sending ultrasonic to returning.
Test distance = (high level time×velocity of sound (340M/S) / 2,

Wire connecting direct as following:

- 5V Supply
- Trigger Pulse Input
- Echo Pulse Output
- 0V Ground

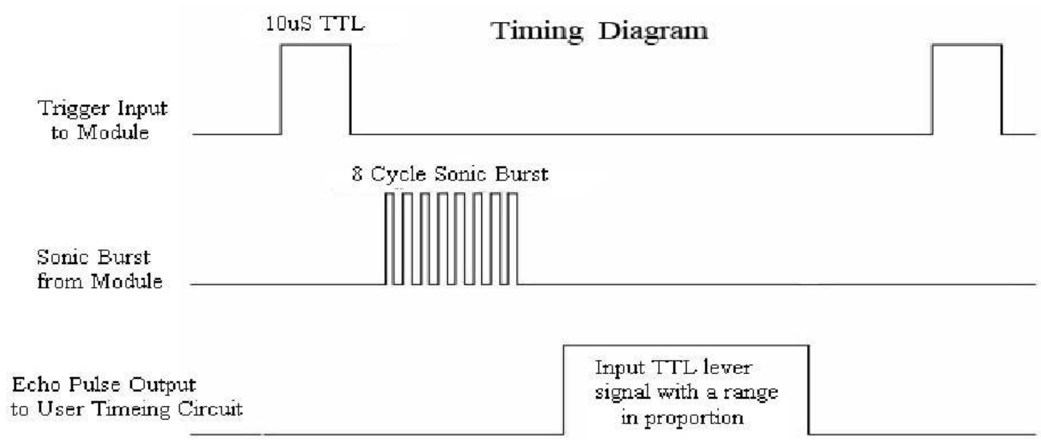
Electric Parameter

Working Voltage	**DC 5 V**
Working Current	**15mA**
Working Frequency	**40Hz**
Max Range	**4m**
Min Range	**2cm**
MeasuringAngle	**15 degree**
Trigger Input Signal	**10uS TTL pulse**
Echo Output Signal	**Input TTL lever signal and the range in proportion**
Dimension	**45*20*15mm**

Vcc Trig Echo GND

Timing diagram

The Timing diagram is shown below. You only need to supply a short 10uS pulse to the trigger input to start the ranging, and then the module will send out an 8 cycle burst of ultrasound at 40 kHz and raise its echo. The Echo is a distance object that is pulse width and the range in proportion .You can calculate the range through the time interval between sending trigger signal and receiving echo signal. Formula: uS / 58 = centimeters or uS / 148 =inch; or: the range = high level time * velocity (340M/S) / 2; we suggest to use over 60ms measurement cycle, in order to prevent trigger signal to the echo signal.



Attention:

- The module is not suggested to connect directly to electric, if connected electric, the GND terminal should be connected the module first, otherwise, it will affect the normal work of the module.
- When tested objects, the range of area is not less than 0.5 square meters and the plane requests as smooth as possible, otherwise ,it will affect the results of measuring.

www.Elecfreaks.com

Appendix D

Thrd.py

```
import threading

class thrd(threading.Thread):

    def __init__(self,fun,*args):
        threading.Thread.__init__(self)
        self.function=fun
        if len(args)!=0:
            self.arguments=args

    def run(self):

        try:
            self.arguments
        except:
            self.function()
        else:
            self.function(*self.arguments)
```

APPENDIX E

LOGGING SETUP

```
## Set-up logging ##
logFilename=os.path.dirname(os.path.realpath(__file__))+"/logs/"+str(time.strftime(
"%Y%m%d-%H%M%S"))+".txt"

fid=open(logFilename,"w")    # Open and then close to create a new file
fid.close()

logging.basicConfig(filename=logFilename,level=logging.DEBUG,format='%(asctime)s %(
name)s:%(levelname)s %(message)s')
```

Appendix F

Connect.py

```python
import dronekit

def Connect(mode="udp",address=["127.0.0.1",14550]):
    """ Connects to the vehicle defined in the arguments and returns its class
        Admissible modes: udp (default), serial or tcp """

    if mode=="serial":
        connection_string=address[0]
        baudrate=str(address[1])
    elif mode=="udp":
        connection_string=str(address[0])+":"+str(address[1])
    elif mode=="tcp":
        connection_string="tcp:"+str(address[0])+":"+str(address[1])
    else:
        raise Exception('Connection mode has to be "serial", "udp" or "tcp"')


    print "Connecting on: %s" % connection_string
    if mode=="serial":
        vehicle=dronekit.connect(ip=connection_string,wait_ready=True,rate=50,baud=baudrate)
    else:
        vehicle=dronekit.connect(ip=connection_string,wait_ready=True,rate=50)

    print "Vehicle connected"
    return vehicle
```

Appendix G

Sonar.py

```python
import RPi.GPIO as GPIO
import time
import signal
import numpy

from threads import thrd

class Sonar():

    def __init__(self,trigPin,echoPin,bufferLen=5):

        GPIO.setmode(GPIO.BCM)

        self.echoPin=echoPin
        self.trigPin=trigPin

        GPIO.setup(self.trigPin,GPIO.OUT)
        GPIO.setup(self.echoPin,GPIO.IN)

        self.distance=100
        self.distanceBuffer=[100]*bufferLen
        self.avgDistance=100

        self.velocity=1e-5
        self.velocityBuffer=[1e-5]*bufferLen
        self.avgVelocity=1e-5

        self.initialTime=time.time()
        self.timeArray=[time.time()-self.initialTime]*bufferLen

        self.Tcollision=100
        self.Treaction=0.5
        self.Tstop=1
        self.Tmargin=0.5
        self.Tsafe=100

    def __del__(self):
        GPIO.cleanup()


    def measureDistance(self):

        time.sleep(0.05)    # Wait a bit to avoid interference from previous measurement

        def triggerSonar():
            GPIO.output(self.trigPin,False)
            time.sleep(2e-6)    # 2 microseconds
            GPIO.output(self.trigPin,True)
            time.sleep(1e-5)     # 10 microseconds
            GPIO.output(self.trigPin,False)
        thrdTriggerSonar=thrd(triggerSonar)
        thrdTriggerSonar.start()

    #   while GPIO.input(self.echoPin)==0:  # Overwrite pulseStart until pulse is detected
    #       pulseStart=time.time()-self.initialTime
            # Performing rolling average over the buffers to reduce noise-related errors

    #   while GPIO.input(self.echoPin)==1:  # Overwrite pulseEnd until pulse has ended
    #       pulseEnd=time.time()-self.initialTime
```

```python

        GPIO.wait_for_edge(self.echoPin,GPIO.RISING,timeout=100)
        pulseStart=time.time()-self.initialTime
        GPIO.wait_for_edge(self.echoPin,GPIO.FALLING,timeout=100)
        pulseEnd=time.time()-self.initialTime

        try:

            pulseDuration=pulseEnd-pulseStart

            sonarDistance=(pulseDuration/2.0)*340

            if sonarDistance<4: # Sensor not accurate for higher values
                self.distance=sonarDistance

                # Update buffer
                for b in range(len(self.distanceBuffer)-1,0,-1):    # Shift position of the old values
                    self.distanceBuffer[b]=self.distanceBuffer[b-1]
                self.distanceBuffer[0]=self.distance    # Include latest measurement

                # Update filtered distance
                self.avgDistance=numpy.mean(self.distanceBuffer)

                # Update time array
                for t in range(len(self.timeArray)-1,0,-1):
                    self.timeArray[t]=self.timeArray[t-1]
                self.timeArray[0]=(pulseEnd+pulseStart)/2

                return self.distance

        except:
            print "Error reading the distance. Trying again"


    def computeVelocity(self):

        try:    # To avoid divisions by 0 from throwing an error

            # Backward differences with a three-data-points stencil
            self.velocity=(self.distanceBuffer[1]+self.distanceBuffer[2]-2*self.distanceBuffer[0])/(2*self.timeArray[0]-self.timeArray[1]-self.timeArray[2])

        except:
            pass

        else:
            for v in range(len(self.velocityBuffer)-1,0,-1):
                self.velocityBuffer[v]=self.velocityBuffer[v-1]
            self.velocityBuffer[0]=self.velocity

            self.avgVelocity=numpy.mean(self.velocityBuffer)

            return self.avgVelocity


    def calculateCollision(self):

        self.Tcollision=self.avgDistance/self.avgVelocity
        self.Tsafe=self.Tcollision-self.Treaction-self.Tstop-self.Tmargin

        return self.Tsafe
```

APPENDIX H

Temperature sensitivity effect on ultrasonic rangefinder

Pending

BIBLIOGRAPHY

[1] A. Wisniewski and M. Mazur, “Clarity from above pwc global report on the commercial applications of drone technology,” tech. rep., 05 2016.

[2] D. H. Ballard and C. M. Brown, *Computer vision*.
1982.

[3] I. UK, “Why invest in systems engineering,” 07 2016.

[4] J. Daily, “Dull, dirty, dangerous - it’s robot work,” 02 2015.

[5] U. aerial vehicle systems association applications, “Uas applications,” 2016.

[6] G. Aguado, Á. Melgosa, A. de Miguel, Á. Ordax, J. Perales, J. L. Sáez, and L. C. Sánchez, “Uav application in search & rescue at sea,” UC3M, 05 2016.

[7] Airbus, “Airbus demonstrates aircraft inspection by drone at farnborough,” 07 2016.

[8] K. P. Valavanis and G. J. Vachtsevanos, *Handbook of Unmanned Aerial Vehicles*.
New York: Springer Reference, 2015.

[9] ICAO, “Manual on remotely piloted aircraft systems (rpas),” 2015.

[10] AESA, “Ley 18/2014, de 15 de octubre, de aprobacion de medidas urgentes para el crecimiento, la competitividad y la eficiencia,” 10 2014.

[11] EASA, “Regulation (ec) no 216/2008,” 02 2008.

[12] “Civilian unmanned aerial vehicles ready for takeoff,” tech. rep., 04 2012.

[13] “Unmanned aircraft systems: Perceptions and potentials,” tech. rep., Arlington, 05 2013.

[14] P. Bergqvist, “Drone jobs: What it takes to fly a uav,” 06 2014.

[15] INCOSE, “What is systems engineering?.”

[16] N. Aeronautics and S. Administration, *NASA systems engineering handbook*.
Washington: US National Aeronautics and Space Admin, 1 ed., 12 2007.

[17] Á. Melgosa, “Internship memoir: Centum solutions,” tech. rep., UC3M, 05 2016.

[18] “Disposición 16097 de boe núm. 274 de 2011,” 10 2011.

[19] D. I. Insights, “Top20 drone company ranking q2 2016,” tech. rep., Hamburg, Germany, 06 2016.

[20] “Dronecode,” 04 2016.

[21] K. Ogata, *Modern Control Engineering*.
Prentice Hall, 5 ed., 2010.

[22] A. S. of Defense for Research and Engineering, “Technology readiness assessment (tra) guidance,” tech. rep., USA, 04 2011.

[23] C. Hulsmeyer, “Verfahren, um entfrente metallische gegenstände mittels elektrischer wellen einem beobachter zu meiden,” 04 1904.

[24] J. Krolik, “Radar and sonar signal processing: Similarities and differences,” 05 2005.

[25] M. Tooley and D. Wyatt, *Aerospace engineering e-mega reference*.
United Kingdom: Butterworth-Heinemann, 03 2009.

[26] R. Beasley, “Systems engineering - what and why?,” 02 2015.

[27] M. Arteta, “Assembly, modeling, simulation and control of a quadcopter for application to solar farm inspection.” BSc Thesis, 2015.

[28] R. V. Astorga, “Simulation of an unmanned aerial vehicle.” BSc Thesis, 07 2016.

[29] J. Sasiadek, Q. Wang, and M. Zeremba, “Fuzzy adaptive kalman filtering for ins/gps data fusion,” in *Proceedings of the 2000 IEEE International Symposium*, 2000.

[30] R. Asselin, “Frequency filter for time integrations,” *Monthly Weather Review*, pp. 487–490, 06 1972.