UNIVERSITY OF TRENTO

DEPARTMENT OF INDUSTRIAL ENGINEERING

MASTER DEGREE IN MECHATRONIC ENGINEERING

An high level architecture for UAV control and autonomous guide in gas leakage detection applications

Author:
Luca GEMMA

Supervisor:
Dr. Davide BRUNELLI

Assistant Supervisor:
Maurizio Rossi

Academic Year 2017-18

"Good morning, and in case I don't see you: good afternoon, good evening, and good night!"

Truman Burbank

Acknowledgements

All the remain part of this thesis is entirely redacted in English, but let me switch to Italian in this acknowledgement as the most important reason I achieved this thesis and went up to this point in my studies are my parents and they are both Italians.
Voglio dire un enorme GRAZIE alle persone più importanti della mia vita: i miei genitori. Perchè mentre io a volte vacillavo, loro hanno sempre creduto in me. Questo mi ha sempre dato enorme coraggio di fronte alle difficoltà. Mentre tutto intorno a me cambiava, si evolveva, ed io con questo, loro sono stati il mio faro. Perchè anche se mi trovavo ad una piccola distanza da casa, li sentivo sempre accanto per sorreggermi e darmi forza. Non ho mai chiesto favori nè mai "giocato sporco" per arrivare dove sono ora. Non lo dico con spocchia, perchè la mia strada è appena iniziata e di nozioni da imparare ce ne sono un'infinità ed altrettante sono le ore di studio e crescita che mi aspettano. Non lo dico con superbia, perchè questa è la migliore lezione che potessi ricevere: credere in sè stessi. Accettare le difficoltà e le sfide che il mio percorso mi porterà ad affrontare ed accoglierle come occasioni per crescere in conoscenza, in esperienza, diventare più forte. Da genitori, hanno fatto sacrifici per poter crescere un figlio. Dei sacrifici che nessuno ha chiesto loro, e so che in cuor loro sono fieri e felici di averli fatti per me. Questo è l'atto di amore più grande che un figlio possa ricevere: essere consapevoli dell'amore dei genitori. Non servono formule per capirlo nè teoremi per dimostrarlo, ma è qualcosa che rimane impresso per sempre in mente una volta imparato. Ora è tempo che vi ripaghi per questi sacrifici, che scelga la mia strada e costruisca il mio futuro. Perciò voglio dire GRAZIE a te mamma, ed a te papà, per tutto quello che avete fatto per me per 27 anni, crescendomi con giusti valori e rimanendo al mio fianco in ogni difficoltà. Ma GRAZIE anche a tutti i miei nonni, anche a chi non c'è più, perchè siete stati dei miei secondi genitori. Sono fiero di tutti voi, vi voglio bene e vi terrò sempre nel mio cuore. Un GRAZIE ed un saluto a tutti i miei parenti, da quelli vicini a quelli sparsi per l'Europa, ai miei cugini che, anche se lontani, sono stati un grande sorriso nella mia vita.
Vorrei poi dire un GRAZIE a tutti gli amici che nel corso della mia strada mi hanno accompagnato, anche solo per poco, nel mio percorso. A tutti gli amici, a quelli incontrati a Verona e a quelli incontrati a Trento, con cui ho condiviso una vita intera, anche solo per un anno o qualche mese. Mi avete mostrato come sia importante non tanto il viaggio, ma la compagnia. Quindi grazie a tutte le mie compagnie, vicine o lontane, grazie a tutti gli amici e a tutte le esperienze che insieme abbiamo fatto. GRAZIE, perchè siete stati dei buoni fratelli di un figlio unico, e come fratelli, legati anche quando separati.
Naturalmente un GRAZIE non posso che farlo anche al mio professore, così come ai ragazzi del laboratorio di Embedded Electronics and Measurement Systems dove ho costruito questa tesi. GRAZIE per questa stupenda occasione e per l'esperienza che ho potuto maturare. Per tutte le esperienze, per il magnifico viaggio in Germania e lo scambio di idee e menti che ho potuto "sbirciare" in quei giorni. GRAZIE all'Università di Trento per tutte le nozioni che ho potuto apprendere durante i corsi, GRAZIE a tutti i professori che ho avuto, a quelli "abbonati" con cui ho frequentato più corsi ed a quelli da cui ho potuto imparare anche solo per un corso. Mi avete insegnato che la conoscenza non è mai abbastanza e mi avete dato gli strumenti per approfondirla e per aumentarla.
Perciò, a tutti voi dico di nuovo uno, dieci, cento ed altri cento GRAZIE.

Contents

List of Figures

Introduction

Drones, or more specifically Unmanned Aerial Vehicles (UAVs), are increasingly drawing attention on the market, becoming from pure entertainment tools to subjects of the most various applications. From crop monitoring [1] to providing network in arduous sites [2] and up to emergency scenarios like avalanche rescues [3]. Several UAVs are already part of our society, like those for pipelines inspection [4] and many have landed recently to save lives, like those distributing vaccines and blood packs in Rwanda [5]. Long story short, UAVs are nowadays a powerful tools to raise applications to the next level. Technology now allows developing projects nearly impossible up to few years ago and UAVs are no less: thanks to more compact battery packs with higher capacity, to smaller electronic boards with higher computing power and to steps forward in the sensor side the assembling of UAVS with ever growing features it is now possible and accessible to a wide community of people, both hobbyists and professionals. Furthermore, measuring dangerous and potentially life-threatening gases, being them toxic or flammable, was of great interest and importance since beginning of the 20th century, when miners brought canaries to detect such threats. Few years later, a lamp would have aided workers to detect methane or lack of oxygen [6]. Since then, technology generated several different solutions up to the most recent gas detectors: Metal-Oxide (MOX) based, electrochemical and electrooptical.
Still, these technologies are relatively new and up to now not so many mobile gas sensing applications are available. Moreover, most of the tools allowing autonomous guide of UAVs are closed platforms and proprietary (like DJI), slowing the development of these projects. And still more, the use of gas sensor at a commercial level is often restricted to indoor or highly structured environment; in these scenarios gas detectors are deployed in well defined positions where external or environmental factors are well characterized and controlled. As some of many examples: monitoring gases in underground car parking [7], refrigerator plants [8] or in boiler plants [9]. In such scenarios, not only sensors are placed in fixed positions and impacting factors are well known but also such devices are more similar to expensive high-precision laboratory equipment than to the commercial ones. This precise devices ensure reliable results and stable behaviors thanks to rigid measures and constant calibration, quality checks and maintenance. For these reasons, precisely detecting gas concentrations with commercial detectors is not suitable and mostly unreliable,

[1] *Easy To Use Drone Enabled Technologies And Services, Purpose Built For Agriculture*
[2] *Inside Facebook's Ambitious Plan to Connect the Whole World*
[3] Silvagni et al., 2016
[4] *Drone Services for Oil and Gas Pipeline Inspections*
[5] *Blood Carrying, Life Saving Drones Take off for Tanzania*
[6] both canaries and the Davy's lamp were early methods coal miners used to detect dangerous gases, as shown in Chapter 2
[7] *Car Park Gas Detection*
[8] *KIMESSA gas detection systems for cold storage rooms and refrigeration systems*
[9] *KIMESSA gas detection systems for boiler plant rooms*

as inspected by the European Commission [10]. However, these commercial sensors lend really well themselves to detect the qualitative behavior of such concentrations, an information which, even if lacking the precise ppm value, can still be of huge importance. This is especially true in sites and scenarios where controlled and structured environments are not available and having bulky high precision equipment is not possible.
Another issue not yet solved is the chance to have an high mobility system for in situ measurement: gas measures are usually performed by human operators, thus potentially exposing them to dangerous and hazardous sites and with the sensible area limited to the one accessible by a human. In this application, an high mobility robot such as an UAV would allow to inspect and cover wider areas, reaching all directions and scanning even at different altitude levels.
That is where this thesis brings its contribution. Aim of this work is to build and give an architecture, both open-source software and hardware, for manage one or more UAVs, with a focus on qualitative sensing of potentially-threatening gases. Joining the high mobility and manoeuvrability provided by UAVs with the possiblity to sense some among the most common Volatile Oganic Compunds (VOCs) and hazardous gases, this thesis wants to provide a tool and a starting point for an high mobility gas sensing system.
In this thesis an open-source system based on the famous ArduPilot [11] flight controller is presented, using easy-to-find drone components to achieve an architecture able to monitor and manage multiple UAVs acquiring in-flight data and sending them MAVLink commands for autonomous navigation. Using DroneKit [12], a collection of Python APIs, it was possible to use high level functions for sending and receiving MAVLink commands, while thanks to Python's flexibility a structure to transmit to ground, elaborate and record data has been implemented. Some firmware to interface with both analog and digital sensors has been developed, using C/C++ language and an all-in-one solution for enclosing both the flight controller and the additional features is proposed using the BeagleBone Blue [13], a powerful portable electronic board suited for robotics. The architecture so assembled is able to pre-load a collection of GPS coordinates defining the path for each UAV, send command to them using the MAVLink protocol and even manage failures with the aid of the GeoFencing software feature. During flight, it is possible to command each UAV for relative movement or absolute GPS guided trails, while the vehicle senses for different target gases like Methane, Propane, Butane and Iso-Propane, some of the most common hydrocarbons, or other potentially toxic or dangerous gases like Carbon Monoxide and Ethanol. Data collected as UAV fly are transmitted to ground in real time through a TCP/IP communication channel over a Wi-Fi link and the algorithm running on the Ground Control Station (GCS) allows to access and display all parameters and states for each vehicle and acquired gas readings in almost real time along with generate maps to display them on their respective GPS position. Finally, the incoming MAVLink input connection stream is replicated on different channels in order to allow for accessing and controlling the UAV from a regular common GCS software like Mission Planner.
This thesis is organised as follows: on Chapter 1 a quick survey on related works is given, presenting papers and projects taken as a reference when deciding which directions to push this work, coming to the motivations of our solution. Chapter 2

[10] *Measuring air pollution with low-cost sensors*
[11] landing page: http://ardupilot.org/
[12] landing page: http://dronekit.io/
[13] landing page: https://beagleboard.org/blue

continues with an overview of the most common gas sensor types: MOX, electrochemical and Infrared detectors. Then the chapter ends by presenting our choice, a combination of a single channel analog MOX and a three-channel digital MOX sensors. In Chapter 3 UAVs are presented. First their main components are discussed, from propellers to the battery pack, then a peek at flight controllers is taken, showing our choice, the BeagleBone Blue with the ArduCopter flight stack. Finally, our setup is presented in details. Chapter 4 shows the main principles for the produced code, grouped into three sections: DroneLib, our custom library based on DroneKit, DroneSwarm, the multi-drone structure, and DroneSense, which manages data acquisition, transmission and elaboration. The chapter starts with a quick introduction on main packages and APIs used, a.k.a. MAVProxy [14] and DroneKit, and then illustrates DroneLib, our library which implements all high level navigation functions (take-off, land, movements, Return-To-Launch) as well as other functions such as loading a GPS path and raw wind-speed estimation. The chapter continues with DroneSwarm, going a bit deeper on its component scripts: Drone Interface, Swarm Manager, the logging utility as well as the main script. Chapter 4 ends with a quick overview to the Software-In-The-Loop simulator for DroneKit, the SITL. On Chapter 5 main results are presented showing the goodness and feasibility of the architecture. First battery tests for different speed and batteries are shown along with a mention on CPU and memory percentage usage on the BeagleBone Blue. Later on, the eCalc tool is presented: a poerful tool to evaluate and estimate vehicle performances depending on the chosen components. All results gotten by this tool are presented as well. Moving on, WiFi range tests and some qualitative results on acquired gas data are available and chapter ends with judgemental discussing about sensor positions and calibration. In Chapter 6 possible future improvements are presented, both on safety measures (e.g. moving to a distributed system) and on data acquisition (e.g. with data fusion and defining a sensing strategy). Finally, the thesis ends with some conclusions on the work done and with the Appendix section A where useful information on how to replicate this project are given. As common practice, last pages are for the Bibliography, where all details on consulted papers and documentations.

[14] landing page: http://ardupilot.github.io/MAVProxy/html/index.html

Chapter 1

Related Works

1.1 Autonomous UAV based systems

UAVs, and in particular fixed wings aircraft and multirotors, are nowadays a subject that is of particular interest for the scientific community. Zipline tested this year a fixed wing UAV based system to achieve fast delivery of critical medicines such as vaccines or blood in Rwanda *Blood Carrying, Life Saving Drones Take off for Tanzania*. This system is based on multiple fixed wing aerial vehicles and claims a payload of up to 1.8 Kilograms and an average delivery service radius of 80 Kilometers, with a fullfillment time of 30 minutes at most. Such impressive performance are possible thanks to both a fast packaging chain system and the top velocity of fixed wing drones, in this case up to 100 km/h.
A similar solution is the one created by Agribotix: this company realized a crop monitoring application through both fixed wings and multirotor UAV *Easy To Use Drone Enabled Technologies And Services, Purpose Built For Agriculture*. They embedded multispectral sensors and RGB imagery to provide farmers the capability to spot damaged area inside their fields and reduces crop losses. As for the Zipline solution, also Agribotix UAVs have autonomous flight capability and can take off and return to home without requiring human intervention. Both of them, however, based their systems to closed and proprietary hardware (e.g. one of the Agribotix solutions is based on the DJI Phantom 4 Pro) and no support or even mention to multi drones support is present.
Another interesting solution is the one proposed by the famous Facebook's founder Mark Zuckerberg: a fleet of passenger jet-sized UAVs who aim at delivering internet connection in infrastructure-lacking sites *Inside Facebook's Ambitious Plan to Connect the Whole World*. Following this project, several different types of devices, including satellites, will be involved, still the main concept remains unclear apart from requiring laser technology to be carried by huge UAVs flying at 2000 meters above the ground. This will bring a focused artificial intelligence software to estimate where citizens would need the most an internet connection together with the capability to manage large fleet of vehicles with a wingspan of a Boeing 737. Such drones would be able to stay aloft for even months with few pauses for maintenance and restore flight autonomy and this could be achieved by solar panel powered aerial vehicles. Still, designing and realizing drones able to fly at nearly 2 km of altitude and the need of a consistent backup plan for battling rain (lasers are hugely affected by obstales such as clouds and raindrops) are considerable obstacles that this project is facing and for these reasons, this solution represents more of a bet than an actual reference point for the work done in this thesis.

FIGURE 1.1: On top: Zipline drone for medicine delivery in Rwanda. In the middle: Agribotix agricultural drone. On bottom: Aquila, the passenger-jet sized Facebook's drone for providing internet connection in infrastructure-lacking sites. Courtesy of Zipline, Agribotix and Wired

1.2 Gas sensing systems

Moving to gas sensing related works, of particular interest is the research conducted by Eu Seng Kok et al, Eu and Yap, 2018. In that work the effects of propellers are inspected in relation to the sensitivity of gas detectors, examining different positions for them as well as compensating for induced turbulences by adopting new chemical plume tracing strategies. Despite such induced disturbances in the airflow, Kok et al. still promote multi-rotors, especially quadcopters, over other aerodynes such as fixed wing aircraft. The reason for this is that multi-rotor based UAV have greatly increased mobility and reactivity w.r.t. fixed wing ones and can allow for a wide pletora of sensing and tracing techniques based on the high degree of maneuverability only they can provide.
Moreover, they also suggest a sensing technique based on altitude changes in order to exploit for eddies airflow. They describe as eddies airflow the region where air pushed out from the propellers circulates back and into them again; such phenomenon is present only when the quadcopter, or in general the multicopter, changes its altitude. By exploiting such phenomenon they showed it is possible to increase the sensitivity of gas detectors of up to twice w.r.t. standard conditions and also suggested to place them as near to the propellers as possible, possibly even right under them. We based on Kok et al. work when choosing the position of the gas detectors as described in 5, even if we eventually moved a bit away from their proposed solution.
Another interesting work has been developed by P.Neumann et al. Bartholmai et al., 2013. In this work, Neumann developed a quadcopter based system able to localize gas concentration through wind speed estimation and particle filtering. Wind speed estimation is achieved by accessing the ground velocity (i.e. the velocity the quadcopter has from a ground observer point of view) from GPS data and extracting the wind velocity by detecting the air speed (i.e. the ground velocity the UAV would have if wind were not present). Air speed is estimated thanks to a set of airspeed-wind speed collection matches generated from precise wind tunnel experiments. Particle filtering is then applied to mimic insect-like gas localization methods, like one used by lobsters or moths. Although we did not go deep into gas localization in this thesis, the paper from Neumann et al. consolidated the idea of using a multicopter instead of a fixed wing aircraft based system and have suggested an intrinsic difficulty in matching exact ppm concentration of gases, going for a more general gas detection with qualitative based approaches.
In addition, Neumann showed in Bartholmai and Neumann, 2011 that gas detection with a quadcopter is highly possible, providing the gas source is not punctual but rather a cloud-like dispersion of significant gas concentration.

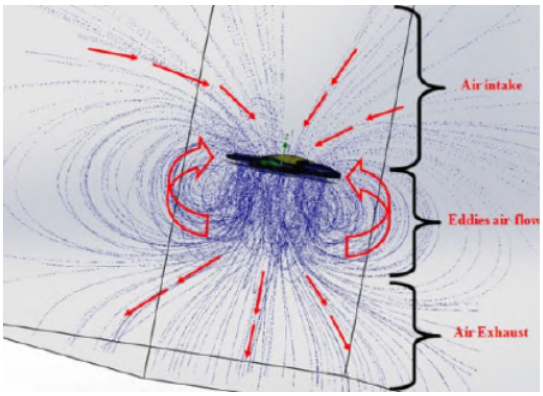


FIGURE 1.2: CFD study of a flying quadcopter from Eu and Yap, 2018 pointing out three different airflux region: air intake, eddies airflow and air exhaust. Courtesy of Kok et al.

1.3 Linux powered autopilots

Finally, of noteworthy relevance was the research from Vilches et al., 2014. In this paper, Vilches et al. discussed in detail the adoption of a Linux board as an autopilot flight controller for drones. In particular, they compared the BeagleBone Black, at that time one of the most flexible and most prone to robotics Linux boards, to standard autopilot options such as the APM 2.5 and Pixhawk. Standard autopilot options generally embedded only the flight control stack and needed an additional board, called companion computer, to achieve external features such as computer vision applications, thus increasing power consumption, payload and wirings. By adopting a Linux autopilot such as the BBBlack all these drawbacks can be overcome easily. They showed that the APM (AutoPilot Multiplatforms, which then became ArduPilot) autopilot was efficiently ported to Linux and Linux based autopilot options could achieve similar reliable performance to non-Linux ones.
This consolidated our choice to go for one of the most versatile Linux board up to now as our autopilot option, the BeagleBone Blue. This way, we could have opened this work not only to the huge ArduPilot community but also to the ever growing Linux one. In fact, we followed the documentation on adopting the BBBlue as a flight controller provided by Mirko Denecke in *Howto use BeagleBone Blue with ArduPilot* where he provides detailed information on how to properly tune and configure the board to adopt and run the ArduPilot flight stack.

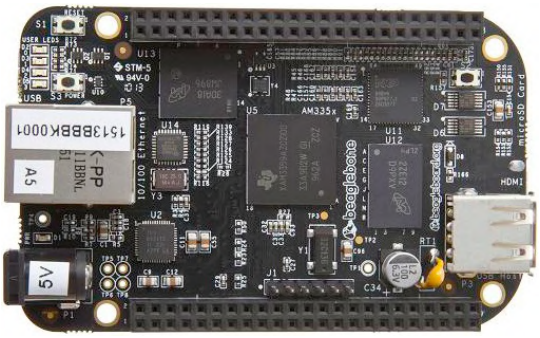


FIGURE 1.3: The BeagleBone Black: the Linux based autopilot option studied and analyzed in detail in Vilches et al., 2014. Courtesy of BeagleBoard.org

In conclusion, we analyzed UAV based systems from different solutions and as confirmed by Eu and Yap, 2018 and Bartholmai and Neumann, 2011 the most suitable aerodyne for gas sensing applications is the quadcopter, or more generally multicopters. In fact, thanks to their high maneuverability they are more prone to sensing and detection scenarios than fixed wing aircraft which still are a valid options for extremely wide area coverage or where high speed are required. Then we identified gas detection as our main scenario as measuring precise gas concentrations is usually unreliable and high demanding, as proved in both Eu and Yap, 2018 and Bartholmai et al., 2013, thus evidencing the need to go for qualitative over quantitative approaches. Finally we consolidated our choice of a Linux powered autopilot as it has been showed to be both reliable and efficient in Vilches et al., 2014.

Chapter 2

Gas detectors

2.1 General overview

Since 19th centuries where miners used canaries to detect dangerous gases such as methane and carbon dioxide, detecting potentially life-threatening gases has been always of crucial interest. Davy's lamp was another precursor of modern gas detectors: this interesting type of flame lamp was built as a composition of two stages, one containing the actual exhaustible object for lighting, such as a cotton part, and the other one being a sealing glass screen that had to be placed around the exhaustible part and a cage of dense metallic flame arrester mesh, which would have absorbed the heat through its precisely designed channels and avoid the ignition of a fire outside the lamp. Moreover, miners could detect the presence of methane or lack of oxygen by looking at the height of the Davy's lamp flame: rising for the former and lowering for the latter.
Since then, gas detectors evolved and exploited several principles, ranging from chemical infused papers which turned darker when exposed to the sensing gas (in the early 1990s) to more recent electronic detectors: gas sensors. A sensor is a device that receives as input a signal or stimulus and produces as output an electrical signal. The input may vary from physical, biological or chemical and the change of these stimuli is sensed and converted to electrical signal, mimicking human senses. Thus, gas sensors are devices which can detect the presence or concentration of one or more target gases through transforming this concentration into an electric signal, being it pure analog or, after an information extraction stages, digital.
Based on the gas-solid interactions and the physical principle used, gas sensors can be classified into three main categories: Metal Oxide Semiconductors, Electrochemical and Infrared. Shown below are the basic principles, advantages, disadvantages and some comparison for commercial solutions for each category.

FIGURE 2.1: On top: coal miners with a canary. On bottom: Davy safety lamp. Courtesy of the United States Mine Safety and Health Administration and daviddarling.info

2.1.1 VOCs and Flammable gases

VOCs and flammable gases are nowadays part of our lives as they are used from home heating to cars and protective coatings, thus monitoring the air pollution and detect critical gases is a crucial challenge. Generally, Volatile Organic Compounds (VOCs) are defined as organic compounds achieving a pressure equal or grater than 0.01 kPa at room temperature (20 deg C), where a compound is defined organic if it contains Carbon and any of the following elements in any combination:

- Hydrogen
- Oxygen
- Phosphorus
- Sulphur
- Silicon
- Nitrogen
- Halogens

Carbon oxides, bicarbonates, inorganic carbonates, Methane, Ethane and organo metallic compounds are not considered organic compounds. VOCs even if not highly

toxic can be a serious treat to health especially because they can be present in high concentrations even in indoor environments. The main source of VOCs are coatings such as paints and protective coatings. For sake of readability, list of VOCs is postponed at A.12.
Flammable and explosive gases are, instead, characterized by a lower and upper limit of flammability (also called explosive limits) indicated as LEL and UEL respectively. When such gases appears in concentrations within these two bounds, with an ignition source a flame can be triggered or an explosion can be generated. These thresholds change depending on temperature and pressure and are usually expressed in terms of volume percentage at atmospheric pressure and at 25 deg C. List of flammable gases is presented along with list of VOCs at A.13.

2.1.2 Metal Oxide Semiconductors

One of the most commonly used sensing materials, providing low cost and high sensitivity sensors. MOX-based semiconductors are mostly n-type, even if p-type versions are present; the main practical difference being that as temperature increases, the sensitivity of the former increases, while for the latter we have a decrease, resulting in lower operating temperature. For both p-type and n-type MOX sensors the detection is possible thanks to the internal redox reaction that happens when the target gas gets in contact with the oxide surface; by measuring the change in resistivity of the sensing element when such reaction happens it is possible to detect the gas concentration.
Common materials used as metal oxides are SnO2, CuO and TiO2, with Tin dioxide, SnO2, as the most broadly used due to its relatively high sensitivity. As the sensitivity is strictly dependant on the working temperature, MOX gas sensors usually embed a heated filament which allows to reach high temperatures. MOX based on different materials lead to different working temperatures and each target gas has its own most suitable sensing temperature. As a result, selectivity is a critical point as if multiple gases for which the material the MOX is based even partially overlaps their best sensing temperatures, the measure can be highly unreliable. On the other hand, if a MOX is reactive to multiple different target gases and they all are sufficiently discernible, the same sensor can be used to detect multiple gases through different heating cycles.
A common solution adopted to compensate for the selectivity issue is to adopt array of sensors reactive to different gases, this way, usually thanks also to a dedicated circuit, it is possible to increase selectivity and reliability of the measure without changing technology. To compensate for the high power consumption required by many heated MOX devices an alternation of measuring and heating stages are often used. Finally, MOX based gas sensors are more prone to detect inorganic gases such as Ammonia (NH_3) and some volatile organic compounds (VOCs) such as Propane (C_3H_8) and Ethanol (C_2H_5OH).

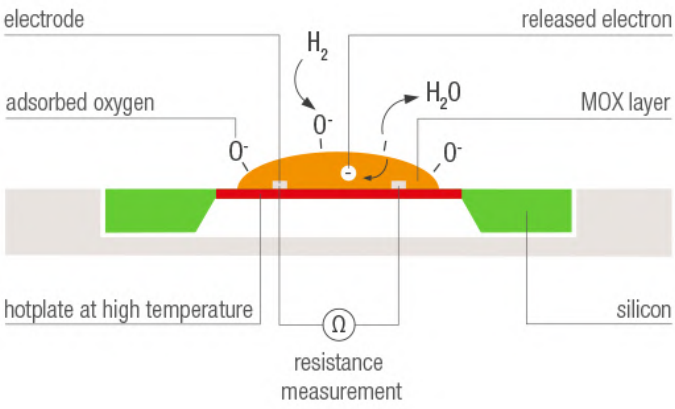


FIGURE 2.2: MOX Gas Detector

2.1.3 Electrochemicals

Electrochemical gas detectors are fuel cells which produces a current proportional to the measured gas concentration. Common to all fuel cells, they are made of two electrodes (sensing electrode and counter electrode) separated by an electrolyte. Due to oxidation or reduction reactions, when the target gas gets in contact with the sensing electrode, both protons and electrons are generated: the former can travel through the electrolyte and reach the counter electrode, while the latter moves along a wire which connect the two electrodes.
Shown at the end of paragraph is the schematic structure of a MEMBRAPOR electrochemical gas detector for sake of knowledge. From top to bottom: the gas is conveyed into the sensor passing through an anti-condensation membrane to avoid dust. By diffusing through a capillary, filter and then a hydrophobic membrane, the gas hits the sensing electrode, where an oxidation or reduction reaction happens, producing or consuming electrons, thus providing a current. The ionic current flows through the electrolyte, while the electric current moves on the wires to the pin connectors. The presence of a third electrode acts as a reference for the voltage, allowing to keep the potential of the upper, sensing electrode to a fixed value.
Of crucial importance is the design of the sensing electrode, leading to acceptable sensitivity to the target gas while simultaneously avoiding cross-sensitivity to other gases. As current is effectively produced by the very device during sensing operations, electrochemical detectors could require very little power to run; however, due to their strong sensitivity to temperature, many such devices embed also a temperature control circuit to keep the temperature value stable and controlled. Moreover, they require constant maintenance as the electrolyte has to be replaced after a shelf time of usually six months, with the life span getting shorter as the exposure to the target gas increases. Lastly, electrochemical gas sensors are more suited for measuring toxic gases such as carbon monoxide (CO) and hydrogen sulfide (H_2S).

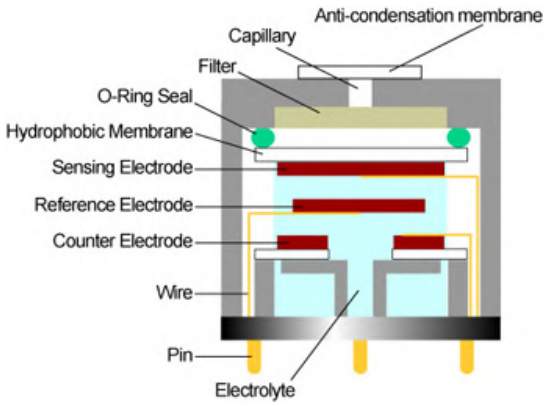


FIGURE 2.3: MEMBRAPOR Electrochemical Gas Detector

2.1.4 Infrared

Infrared gas detectors exploit absorption properties of some gases to certain electromagnetic wavelengths. In fact, almost all gases posses a unique absorption "fingerprint" to specific IR regions and IR gas sensors exploit this phenomenon: the gas enters a chamber where, thanks to both an IR source and an IR detector, the amount of absorbed energy is measured. A chopper can be present to modulate IR light at a specific frequency needed for IR detector and reading circuitry to work properly. To achieve desired selectivity, an IR filter is added to screen out all other radiations, which can be an optical bandpass one (for NDIR - nondispersive IR detectors) or a prism. The measure is taken by comparing the reading w.r.t. a reference scenario in which only fresh air is present and it is also possible to have IR detectors with an additional chamber containing the reference gas in order to compensate for environmental factors and increase accuracy.
Moreover, it is also possible to exploit temperature change of compounds inside the irradiation chamber to measure the gas concentration. In fact, when a gas absorbs radiations with a frequency near its natural one, its molecules start vibrating more intensely, leading to an increase in temperature. By detecting this temperature change it is possible to estimate the gas concentration. Although possessing higher sensitivity and stability w.r.t. other gas detectors, IR gas sensors are affected by cooling requirements, high complexity and a relatively high cost. Finally, some gases such as O_2, N_2 and H_2 can not be measured by IR detectors.

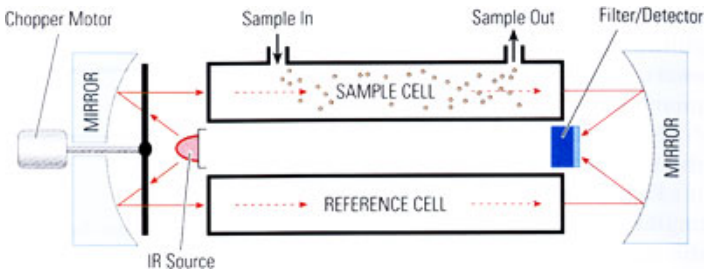


FIGURE 2.4: RKI Instruments IR Gas Detector

2.1.5 Other Technologies

Other technologies are available among gas detectors, although they are not suitable for portable devices. For sake of knowledge, they are briefly illustrated here.

Carbon Nanotubes

This upcoming technology promises higher sensitivity, quicker response time and better bandwidth w.r.t. MOX based detectors and the capability to run even at room temperature, with no need for a heater. Target gases for Carbon Nanotubes based sensors are Carbon Dioxide, Alcohol, Nitrogen Dioxide and Ammonia. However, this technology is not yet fully characterized and studied and MOX based detectors still remain a stabler and more suitable solution.

Polymer

Polymer based gas detectors work in a similar way MOX based ones do: when the target gas gets in contact with the sensing layer, polymer's physical properties (e.g. mass and dielectric insulation) change proportionally to the entity of the gas absorption. Generally, gas detectors based on Polymers have to undergo a doping process to enhance their conductivity, as without this process they are usually used only as membranes on MOX to increase selectivity. Finally, despite having high sensitivity and relatively short response time, the long time instability and poor selectivity make this technology an interesting future option more than a present solution.

Calorimetric Methods

Also called "pellistor" (pellet-resistor), calorimetric based gas sensors are the evolution of the Davy's lamp, exploiting changes in thermal conductivity as a detection method. Generally they are divided into Catalytic and Thermal Conductivity sensors, both measure resistively a temperature variation created by burning the target gas (for the former) or measuring its heat dissipation properties (for the latter). Thermal conductivity based detectors are a promising technology which posses good stability and relatively low complexity, but, as catalytic ones they may suffer from irreversible loss of catalysts when impurities are present in the sensed gas.

Gas Chromatograph

Probably the best analytical technique for detecting and measuring the concentration of gases, it ensures one of the highest selectivity, separation and sensitivity performance. This technology however requires high cost and bulky devices and for this reasons it is used almost only as a laboratory technique.

Acoustic Methods

Finally, ultrasonic methods are possible and grants long lifespan and no secondary pollution. The most common detection method is time-of-flight, which compares the travel time of an ultrasonic wave at a fixed distance across the target gas (in the first chamber) to the one for the reference gas (in the second chamber). Attenuation, i.e. measuring the loss of energy during the travel of an acoustic wave through the gas, or acoustic impedance, i.e. estimating the gas density through its acoustic impedance, gas detectors are also possible although very rare among commercial

sensors. Moreover, for acoustic methods the impact of environment is probably even more critical w.r.t. other technologies, thus a system embedding this type of detectors have to highly take into account such drawback when deploying them.

2.2 Our choices

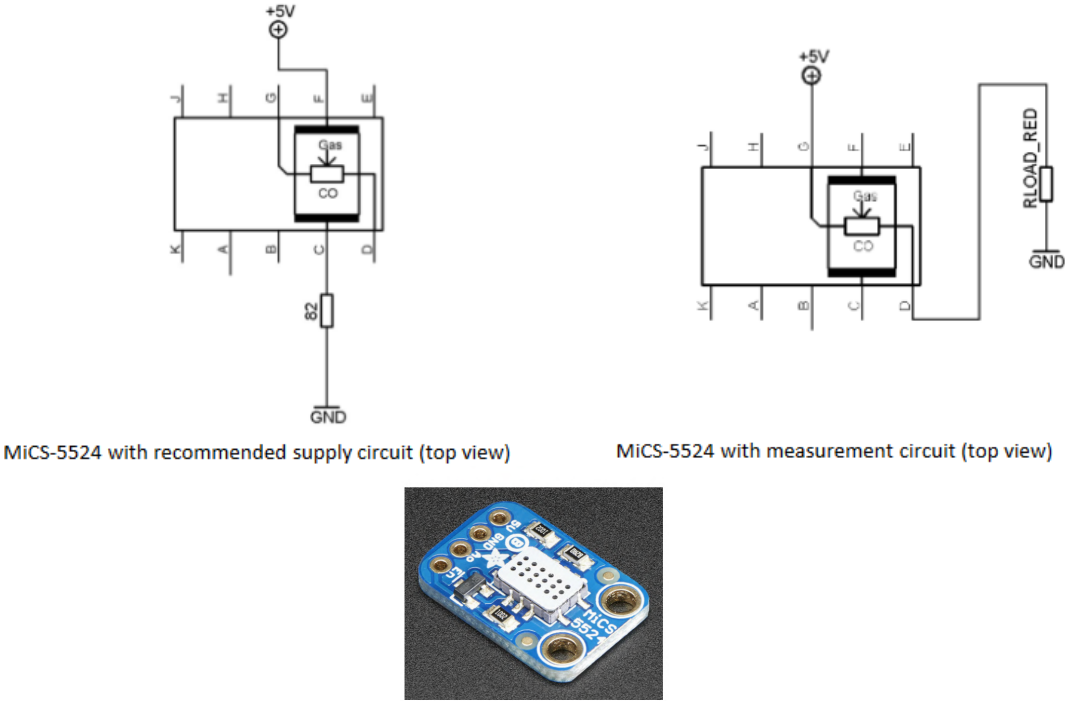
We chose to go for MOX based gas detectors as they are easily available on the market, cheaper, well-characterised devices with low power consumption and easy to interface with. They are cheaper and easier to interface than IR based devices and with a longer lifetime and less requirements of maintenance w.r.t. electrochemicals and come for a wide variety of target gases. Even if selectivity can be an issue because they are generally sensitive towards a family of gases rather than just one gas and can have cross-sensitivity issues, setting up an array of sensors can compensate for these difficulties with no increase in complexity. Shown below are our choices for both analog and digital sensors.

2.2.1 MiCS 5524 Analog Gas Detector

As a second type of analog sensor we chose the MiCS 5524 from SGX Sensortech. Being a MOX based detector as all of our gas sensors, the working principle is the same: its resistance changes accordingly to the oxidation or reduction reactions happening as soon as one of the target gases gets in contact with the detecting layer, namely an increase in resistance for oxidising gases (e.g. Ozone and Nitrogen) or a decrease for reducing ones (Carbon Monoxide and VOC's). Target gases for this device are:

- Carbon Monoxide 1-1000 ppm
- Ethanol 10-500 ppm
- Hydrogen 1-1000 ppm
- Ammonia 1-500 ppm
- Methane, Propane and Iso-Butane > 1000 ppm

We mounted the Adafruit MiCS 5524 Sensor module, a 19x13x3 mm module which requires 5 V to be powered and provides a straight analog output for the readings. Being analog and having only one output it is worth noting that this kind of device can sense the presence and, through proper calibration, the overall concentration of the detected gas among all the target gases but it cannot clearly distinguish which gas it has detected among all of them.



MiCS-5524 with recommended supply circuit (top view)

MiCS-5524 with measurement circuit (top view)

FIGURE 2.5: On top: SGX Sensortech MiCS 5524 Gas Sensor. On bottom: Adafruit analog module for MiCS 5524

2.2.2 MiCS 6814 I2C Gas Detector

We adopted another SGX Sensortech Gas detector, the MiCS 6814 featuring three independent channels for gas sensing. In fact, it embeds:

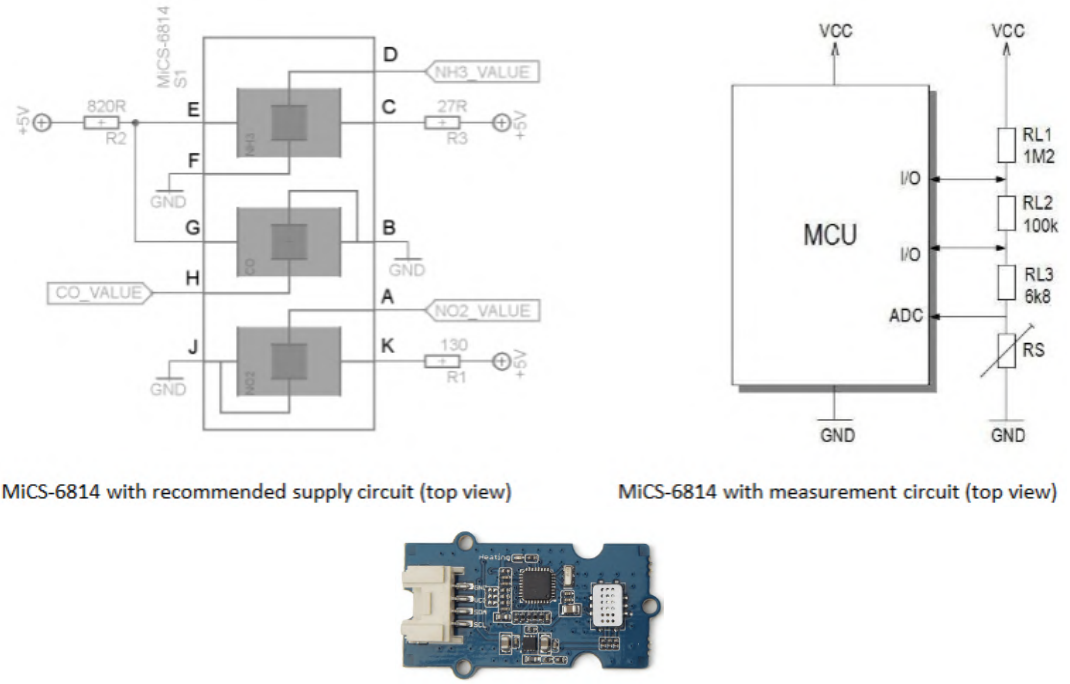
- one channel for reducing gases (e.g. Carbon Monoxide and VOC's)
- one channel for oxidising gases (e.g. Ozone and Nitrogen)
- one channel for Ammonia

This leads, differently from the MiCS 5524, to the capability to distinguish from different family of target gases and provides a readings for each of its target gases. Namely, target gases for the MiCS 6814 are:

- Carbon Monoxide 1-1000 ppm
- Nitrogen Dioxide 0.05-10 ppm
- Ethanol 10-500 ppm
- Hydrogen 1-1000 ppm
- Ammonia 1-500 ppm
- Methane > 1000 ppm
- Propane > 1000 ppm
- Iso-Butane > 1000 ppm

We mounted the Groove Multichannel Gas Sensor MiCS 6814 module, which accepts both 3.3 V and 5 V as voltage input and provides a digital output following the I2C communication protocol: first pin for GND, second one for VCC and two pins (SDA and SCL) for serial clock and serial data respectively. Being an I2C device, a

specific firmware has been developed based on the specification for I2C communication provided by the producer. This approach, as well as the firmware developed for the analog sensors is discussed in 4.



MiCS-6814 with recommended supply circuit (top view)

MiCS-6814 with measurement circuit (top view)

FIGURE 2.6: On top: SGX Sensortech MiCS 6814 Gas Sensor. On bottom: Groove I2C digital module for MiCS 6814

Chapter 3

Unmanned Aerial Vehicles

3.1 General overview

When talking about unmanned vehicles, we may think about Nikola Tesla as one of their main grandfathers and pioneers. Early 1900s were crucial years for wireless technology, as radio were becoming inspected by great minds such as Marconi, Hughes, Edison and Tesla itself. In particular, during an Electrical Exhibition in New York's Madison Square Garden in 1898 Nikola Tesla showed a 4-foot-long miniature ship controlled without any wires, through the aid of a radio-transmitting control box. Both the velocity and movements of the ship could be controlled as well as the switching of some light bulbs, useful to detect the ship in low light conditions. At that time many people pointed at that invention as some sort of magic, but, after a century, remote controlled vehicles are now broadly spread even among children. Up to now drones are rapidly increasing in popularity not only among hobbist but even big companies such as Amazon and Google are showing interest in developing toward these new promising type of vehicles. Commercial applications such as agriculture monitoring [*Easy To Use Drone Enabled Technologies And Services, Purpose Built For Agriculture*], providing internet connection [*Inside Facebook's Ambitious Plan to Connect the Whole World*] or safety related ones such as food and medicine delivering [*Blood Carrying, Life Saving Drones Take off for Tanzania*] and avalanche victims rescuing [Silvagni et al., 2016] are just the tip of the iceberg.

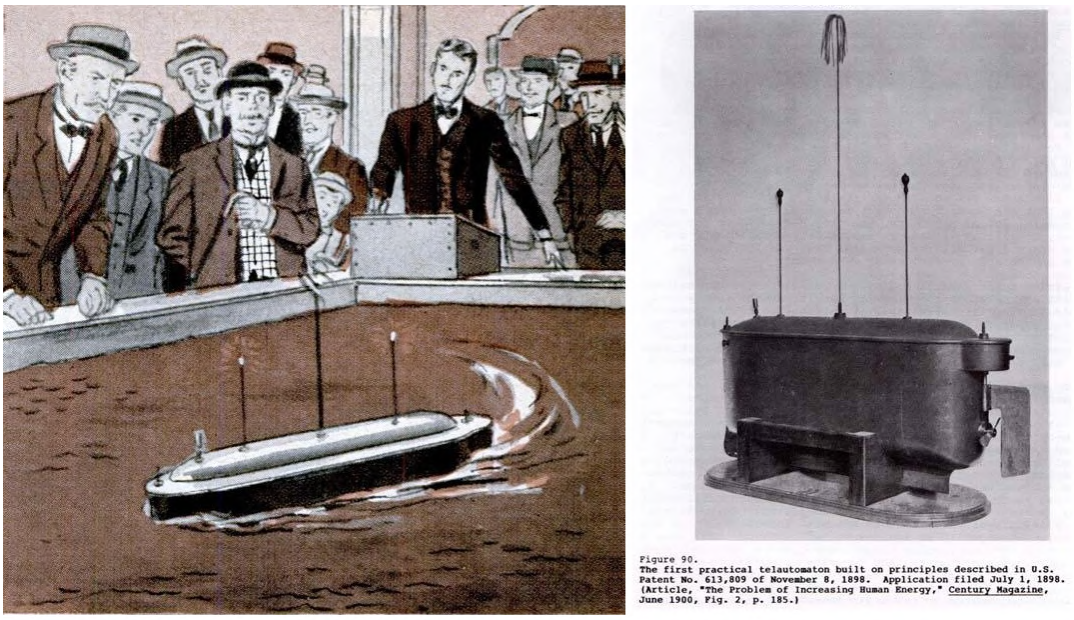


FIGURE 3.1: Paper article of Nikola Tesla's control boat first show (on the left) and closer look to the boat (on the right). Courtesy of quora.com

FIGURE 3.2: Amazon's Prime Air (on the left) and Google's Wing X delivery drones. Courtesy of Amazon and Google

3.2 How they work

Roughly speaking, a drone is composed of a frame, propellers, motors, a battery and electronic parts. The frame is usually made of carbon fiber, PCB, aluminum or plastic materials that can bear shocks induced by the moving rotors or by contact with ground or obstacles, without breaking. In addition, the material used for the frame has to be lightweight in order not to diminish considerably the flight time and allowing for greater payloads. Frame sizes for drones are classified by the distance from one motor to the opposite one and are measured in millimetres: Nano drones (size of 80-100 mm), Micro drones (100-160 mm) and Mini drones (up to 300 mm). There is no acronym for larger drones.
Propellers and motors are also of great importance for the vehicle dynamics. On every drones, half of the motors rotates clockwise and the other half rotates counter-clockwise, therefore propellers have to be mounted accordingly to generate thrust. Such alternating layout is needed in order to compensate yaw motion generated by

each propeller in flight, allowing the UAV to loiter (a.k.a. hovering on the same location and maintaining the same asset). Propellers can be found in different diameter and pitch, with as 'pitch' the distance travelled ideally by one single propeller during a complete rotation. Bigger diameters or bigger pitches lead to more energy to be spin and thus to more current drawing even if generally allowing for higher top speed. Conversely, by keeping constant the rotations per minute (RPM), the smaller the diameter and/or the pitch the faster the spin velocity and the more responsive and stable the UAV gets.
Furthermore, even the shape of a propeller impacts performance, with the most contribute given by the tip of the props. Tips are classified into pointy nose (PN), bull nose (BN) and hybrid bull-nose (HBN). PN are the most efficient of the three, thoutgh pulling the least thrust, while BN and HBN generate more thrust at the expense of an increased current drawn. Tip's shape also impacts, with smoother edges performing better than blunt ones. Finally, propellers exist in different number of blades too, basically increasing thrust at the expense of increased drag (the resistance the air opposes to the motion of the prop), even if it is a common practice not to get beyond three-blade props since from them on there are highly diminishing returns, thus a reduced efficiency. Aircraft speed is theoretically defined as: $MaxSpeed = MaxRPM \cdot Propeller'sPitch/60$ $[''/s]$. Following this, RPM and Pitch are tied when determining the top speed, thus bigger prop spinning at low RPM need to compensate with higher pitches to beat performances of smaller props with high RPM.
When dealing with props materials, several options are available: wood, carbon fiber and plastic. Despite being relatively cheap, wood propellers are the most unsuitable ones as they posses high stiffness. Carbon fibre ones, on the other hand, are probably the most performing, as they achieve the least vibrations and their light weight leads to more responsive behavior. Despite this, plastic propellers are a common and valid choice, as they are cheaper and can bring higher flight times w.r.t. carbon fiber ones.
Motors used on UAV can be grouped into brushed and brushless. Brushed motors exploit a spinning coil inside a fixed magnets case, while brushless do the opposite, spinning the magnets and keeping fixed the coils. Brushed motors are adopted only on small and cheap RC vehicles with brushless motor being the common solution for nearly all the UAVs. Brushless motors can be further divided into inrunner and outrunner. Inrunner motors have fixed coils on the outer casing and the magnets spin inside the casing; while outrunner ones have spinning magnets on the outer casing an coils on the inner region. Generally inrunner motors posses higher KV while outrunners provide more torque.
Being mentioned above, the KV is one of the main evaluation parameter when dealing with UAV motors: it detects the rotational speed of the motor from a given voltage. By multiplying the KV by the voltage fed to the motor, the rotations per minute is obtained. For example an 800 KV motor at 14.8 V (nominal voltage for a typical battery pack) will rotate at 11840 rpm. Notice that lower voltages tend to mean higher current draw. For most multicopter UAVs acceptable KV values are between 500 and 1000, since low values such as these contribute to stability. Higher KV values are mostly used only for acrobatic flight.
Along with the KV, the thrust is the other crucial parameter for motors. Thrust are often provided by manufacturers for different propeller choices and it has to be taken into account when computing the total UAV weight. As an example, for an hexacopter with motors providing a maximum thrust of 0.5 Kg with 11 inches propellers, the UAV will be able to lift 3 Kg at maximum thrust, including its own

weight. This means that if the hexacopter total weight (UAV weight plus payload) will be 3 Kg, the UAV will lift only at maximum throttle while it will not lift at all for bigger weights.
Coming to the end of the non electronics components of a UAV, batteries are also of great importance. Higher capacities not always lead to increased flight times, as usually they mean also heavier battery packs, thus more thrust required and more power consumed to fly the drone. Battery packs used in UAV are Lithium Polimer (LiPo) batteries, as they posses good discharge rates and a good weight-to-capacity ratio. Such batteries are made of LiPo cells with nominal voltage of 3.7 V for each cell. The number of cells in a battery pack is also quickly stated by the corresponding number before an 'S', thus a two-3.7 V cells battery pack is indicated as a 2S battery. Discharge rates, indicating the available burst and continuous discharge current over time, are also important as some motors may draw high current and damage the battery pack or cause crashes if mounting a low discharge rate battery. A final consideration on batteries for UAV: as they usually are one if not the heaviest component on a drone, they have to be placed very close to the UAV centre of mass to not generate unbalance and unequally stress motors.
Moving to the electronic parts, flight controller, ESCs, power module and, in some cases, a power distribution board are all needed to complete a UAV. The flight controller is the computational core of the very drone: enclosing the autopilot and keeping the vehicle stable are the two main tasks of such component, using the Inertial Measurement Unit (IMU) to assess the vehicle pose and states. IMUs are mostly composed of magnetometers, gyroscopes and accelerometers to sense magnetic field, velocity and acceleration along the three axis. Flight controllers also connect to both a GPS antenna for retrieving absolute positioning information and a barometer to get the altitude, though in some cases the altitude is extracted from both GPS and barometer data to have a better estimate. Finally, controllers may adopt distance sensors such as ultrasonic or LiDAR ones in order to increase the reliability of the altitude estimate and to acquire obstacle avoidance capability.
The ESC (Electronic Speed Controller) is the link between the flight controller and the motors, providing power and converting incoming velocity commands from the controller to a PPM-like control to move the motor. Usually EScs include also a back/boost circuit called BEC (Battery Elimination Circuit) which acts as a voltage regulator for the voltage fed to the motor. Other than the power input wires, an ESC also has a 3-pin R/C servo connector which accepts RC signals used to command the motor and has a 3 bullet output connectors which connect to the brushless motor itself.
Last of the three, the power module is needed in order to manage the voltage coming from the battery and regulates it to an acceptable value for the flight controller (usually 5V). The power module also feeds the ESCs either directly or through a power distribution board. In the former, the power module feeds a single BEC which powers all the ESCs while in the latter the positive and negative terminals of the power module are split into a number of other terminal equals the number of ESCs, usually soldering the cables on a board.

3.3 Flight Controller

3.3.1 The Hardware

We chose ArduCopter as our flight controller firmware as it is one of the most commonly used among both professionals and amateurs for its wide range of offered

features and its open-source nature. ArduCopter runs on several different autopilot systems: Pixhawk, NAVIO+, Erle-Brain, APM and BeagleBone. Pixhawk is the most used and recommended as it offers a wide array of peripherals such as UARTs, I2C and CAN as well as interesting features such as a failsafe co-processor and double accelerometers and gyroscopes. NAVIO+ and EErle-Brain are most suited for UAV vision applications as they both combines a Linux computer (Raspberry 2) and a daughter board containing several sensors, IO and power electronics. APM is one of the oldest system running the ArduCopter firmware and is based on the ATMega Arduino board; it is now discontinued.
Finally, the most recent addition to the ArduCopter supported boards is the BeagleBone. With one of its latest model, the BeagleBone Blue, it offers a really viable alternative to both Pixhawk and NAVIO+/Erly-Brain, as it places itself between the two options by embedding a several noteworthy electronics and peripherals into a powerful Linux board. Featuring an AM335x 1GHz ARM® Cortex-A8 processor, 512MB DDR3 RAM and 4GB eMMC flash storage, it represents an all-in-one solution for Linux-based robotic platforms thanks to its connectivity and sensors. In fact, it is equipped with a 9 axis IMU, as well as a barometer and thermometer, and provides additional buses and peripherals such including GPS, DSM2 radio, UART, SPI, I2C, 3V3 GPIOs and 1V8 analog inputs. Last but not least, support for bidirectional DC motors and quadrature encoders, 802.11bgn Wireless and 4.1 Bluetooth connectivity and the ability to power the board from both a 9-18V barrel jack charger input or from a 2-cell LiPo battery eventually monitoring its state of charge make this board a powerful and versatile tool not only for UAV platform but for several different robotic projects.
To conclude, we went over an all-in-one solution because as shown in Vilches et al., 2014 it represents a compact and less power demanding choice for embedding the UAV's intelligence into a single component, also easing cabling and power distribution.

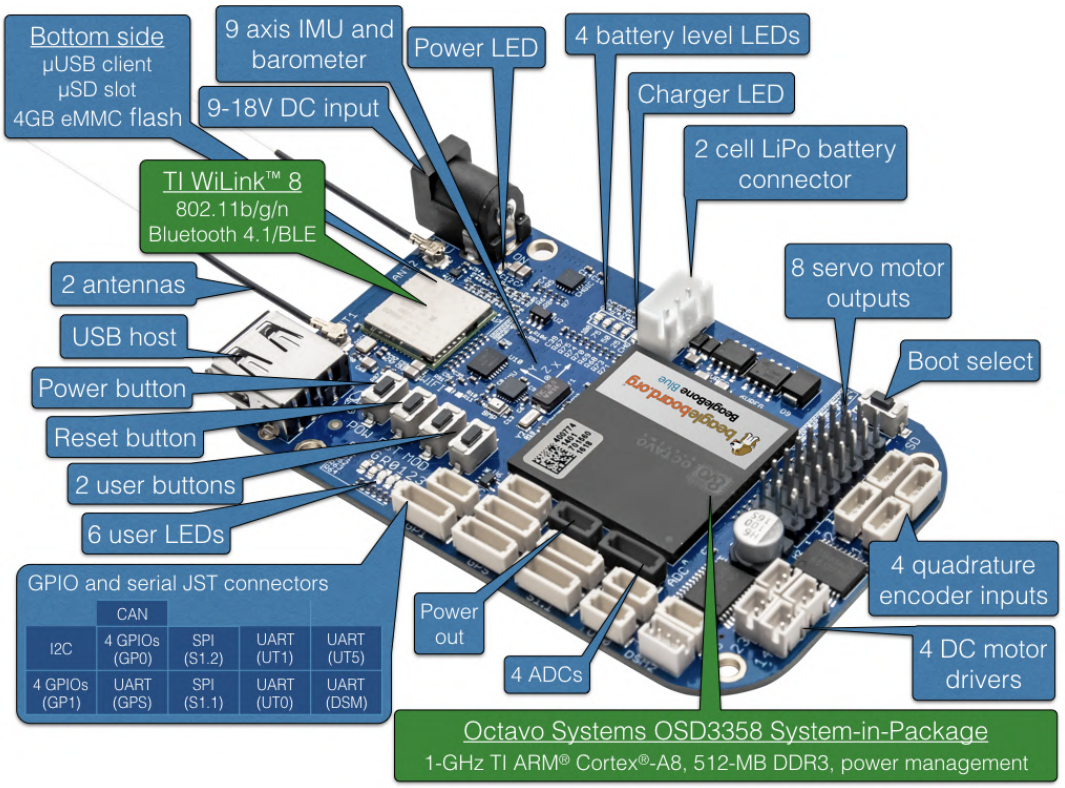


FIGURE 3.3: The BeagleBone Blue. Courtesy of BeagleBoard.org

3.3.2 The Firmware

Developing systems to autonomously control UAVs and extending their capabilities are pushing drone companies toward open source coding and sharing SDKs (Software Development Kits) and APIs (Application Programming Interfaces) to the developer community. DJI, one of the most famous and relevant UAV companies on the market has just released several SDKs allowing to achieve several tasks, ranging from extending on-board capabilities to fully customized mobile apps. However, for our system we chose one of the widest and most used APIs among both UAVs hobbyist and developers: DroneKit. DroneKit is a Python API which is fully compatible with the ArduCopter, ArduPlane and ArduRover firmware and grants a major level of compatibility to several other flight controller firmware. It allows for development of web service apps to manage and store data such as photos or live flight logs, building of mobile apps, general GCS (Ground Control Stations) and extends onboard computing by interfacing with the flight controller to extract all the flight data and send MAVLink commands (the type of commands used to communicate with a generic UAV). Furthermore, being a Python API enables to encapsulate the code into a powerful coding language as Python, further increasing its flexibility.

3.4 Our Setup

3.4.1 General overview

As our Ground Control Station (GCS) a laptop was used, running both DroneKit and Mission Planner, the former for sending commands to the UAV and manipulating

its incoming telemetry and the latter for visualizing telemetry data through a user-friendly interface. The UAVs are connected to the GCS through an 802.11g Wi-Fi network with two separate connections for each UAV: the first one for ArduCopter data and the second one for gas sensors data. Finally, two types of copters were used as UAVs: a quadcopter and an hexacopter. Both UAVs were assembled using a fiber glass frames (S500 for the quad and S550 for the hexa), 750kV brushless DC motors, 51 Ampere 2S-6S ESCs and 10 inches pointy-nose two-blades propellers. We used a series of 2 10400 mAh 2S LiPo batteries (producing a 4S battery pack) for both the hexacopter and the quadcopter. The series were needed as during our field tests we noticed motors were in need of higher voltages, thus we increased the maximum available voltage up to the one produced by a 4S pack (with nominal charge of 14.8V and roughly 16.8V when fully charged).
Both UAVs were equipped with a BeagleBone Blue board which run both the flight controller firmware (namely ArduCopter) as well as all the Python scripts and C executables required for data acquisition and ground transmission.

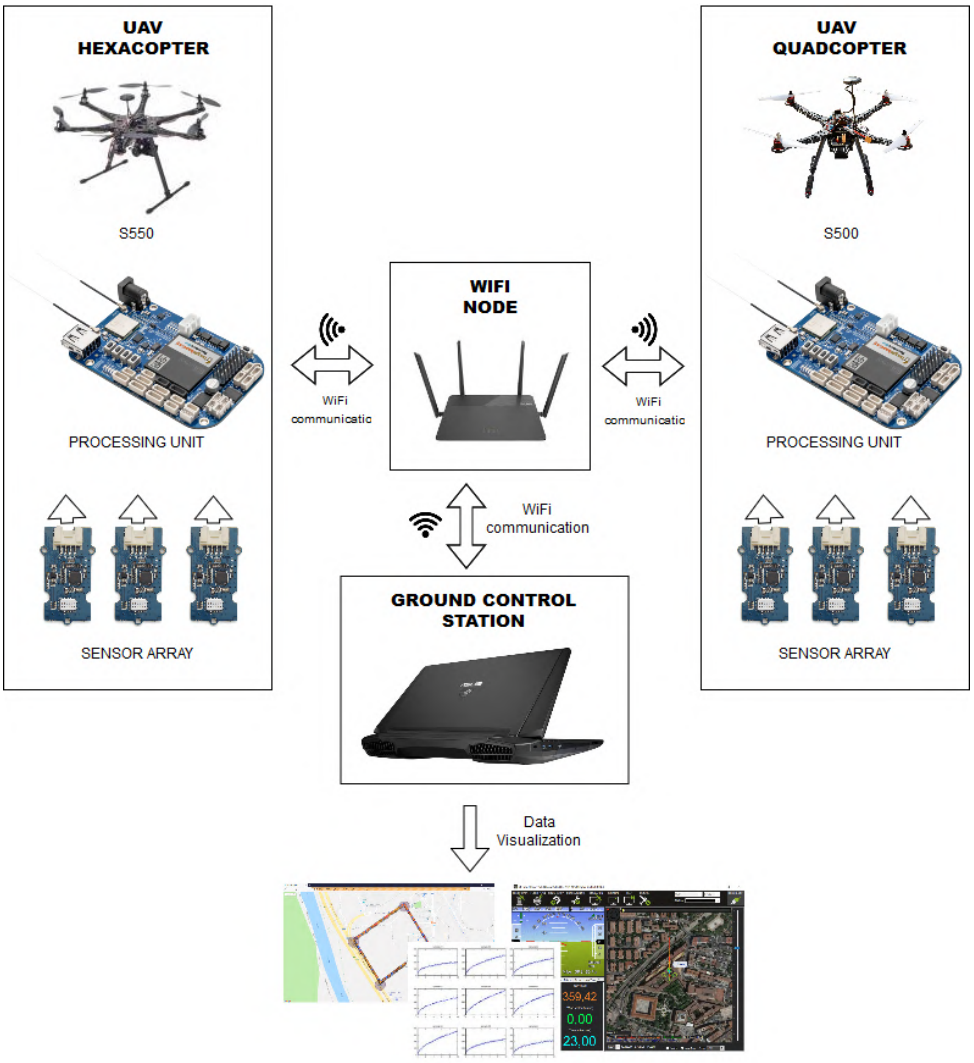


FIGURE 3.4: Our Setup

FIGURE 3.5: Our UAV ready to fly: the S500 Quadcopter and the S550 Hexacopter

3.4.2 Battery/Power

Each UAV was equipped with a 4S battery pack made of a series of 2 LiPo batteries with a capacitance of 10400 mAh. The battery pack was then connected to the Power Distribution Board (PBD) of the UAV through a two sided XT60 connector soldered on the board, with one side split into two XT60 connectors to achieve the series connection. Thanks to the PDB the power was distributed to each ESC through its power input wire soldered onto the board. We chose not to power our BeagleBone Blue from a different power source, thus we powered it through a center-positive barrel-jack connector soldered onto the PDB. The jack connector mounted on the BBBlue, as well as its AP1509 5V voltage regulator, accept up to 18 V as input, thus our 4S voltage was acceptable even when fully charged, reaching roughly 16.8V.

FIGURE 3.6: Our Battery: a Limskey 10400 mAh 2S. We built a pack with a series of two of these to achieve a 4S battery pack. Courtesy of Limskey

3.4.3 Data acquisition

We adopted two MOX gas sensor module for each UAV: the MiCS 6814 three-channels I2C digital module and the MiCS 5524 analog module. The former is a digital I2C interfaced module for the Amphenol SGX Sensortech sensor with three sensing elements sensing primarily Carbon Monoxide (CO), Nitrogen Dioxide (NO2) and Ammonia (NH3) and with good sensitivity to other relevant gases such as Methane (CH4), Propane (C3H8), Iso-butane (C4H10), Ethanol (C2H5OH) and Hydrogen (H2). It has an operating range of -30-85 C and can be powered with 3.3V or 5V. Being an I2C module, we connected the module to the I2C port on the BeagleBone Blue, which is a 3.3V I2C port.
The MiCS 5524 is an analog MOX gas sensor with high sensitivity to nearly the same MiCS 6814 target gases: Carbon Monoxide (CO), Nitrogen Dioxide (NO2), Methane (CH4), Propane (C3H8), Iso-butane (C4H10), Ethanol (C2H5OH) and Hydrogen (H2). Producing an analog output, it was connected to the ADC port of the BeagleBone and powered through the 5V line of the PWR port. As the 5V power line was shared with the radio receiver, a split of the line connecting in parallel both the radio and the gas sensor was realized with a small prototype board.

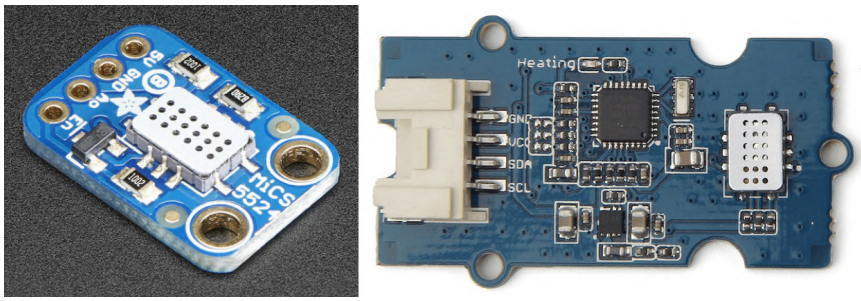


FIGURE 3.7: Our Gas Sensors: the single-channel analog MiCS 5524 and the three-channel I2C digital MiCS 6814. Courtesy of SGX Sensortech

3.4.4 ESCs, Motors and Propellers

Each UAV, both the quadcopter and the hexacopter, mounts Turnigy Multistar 51A 2S-6S ESCs, supporting a continuous current of up to 51 A and an input voltage suited for 2 to 6S LiPo batteries, thus fully compatible with our 4S battery pack. The two power input wires of each ESC have been soldered to the power distribution board, getting power straight from the battery pack, while the 3-pin R/C servo connector was connected to the servo motor output of the BeagleBone Blue through a parallel cable derived from an ATA one in order to reduce wirings, this way only one power and ground wire was connected to the motor output of the BBBlue.
Each ESC was then connected with its 3 bullet output connectors to an AX-2810Q 750 kV brushless DC motor. As usual in a UAV, to avoid unbalanced torques, motors with opposite spinning directions have to be placed alternated along the UAV; thus each counter-clockwise (CCW) spinning motor has to be placed near two clockwise (CW) spinning ones. According to this, each propeller has to be mounted following the same spinning direction of its motor, thus the leading edge (the one entering the air flux) has to be on the side the propeller is going to spin toward. We chose 1045 Pointy-Nose plastic propellers, meaning a diameter of 25.4 cm (10 inches) and a pitch angle of 11.43 cm (4.5 inches).

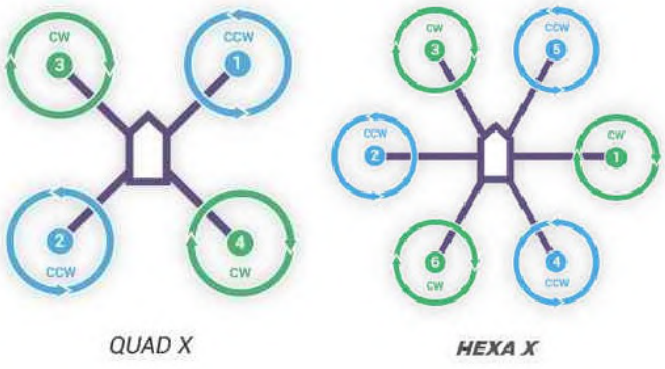


FIGURE 3.8: Propellers and Motors Layout for both an Hexacopter and a Quadcopter in X configuration (our choice). Courtesy of ArduCopter

FIGURE 3.9: Our Props (1045 plastic propellers), Motors (AX-2810Q 750 kV brushless DC motors) and ESCs (Turnigy Multistar 51A 2S-6S). Courtesy of Hobbyking

3.4.5 Communication

Two types of communication were used on each UAV: radio and Wi-Fi. The former was needed to couple each UAV with a radio command and used as a backup emergency tool when other communication channels would have failed; moreover, it was used to calibrate the ESCs with the maximum and minimum value for PWM commands for the DC brushless motors. We chose the FlySky FS-iA6B 2.4 GHz radio receiver and we used the Pulse Position Modulation (PPM) radio control protocol for the BBBlue supported it with no need for further expansions. In fact, we connected the PPM line wire to the pin 4 of the connector E4, a multi protocol pin compatible with PPM as well as with several other protocol such as S.BUS and Spektrum Satellite DSM. As previously enounced, the radio was powered through the PWR port 5V power line with a parallel connection with the gas sensor. Consequently, as a radio command we used the FlySky FS-i6.
The Wi-Fi connection was used to get telemetry data and control the UAV through an UDP socket and to collect gas sensors data through a TCP socket. As for the radio connection, there was no need for further expansion boards as the BeagleBone Blue offers an 802.11b/g/n connectivity.

FIGURE 3.10: Our RadioTransceiver along with its radio receiver (FlySky FS-i6 2.4 GHz). Courtesy of Hobbyking

3.4.6 GPS module

As a GPS module the Ublox NEO-M8N was used. This module offers good performances at a reduced power consumption of 50mA at 5V. The module offers also a compass, which was not used as the BBBlue embeds also an internal one. The GPS module was connected straight to the GPS port of the BBBlue board, achieving an UART connection and powering it with 5V. Shown below are the most relevant information about this uBlox module.

- GNSS GPS, GLONASS, QZSS, Beidou, Galileo ready
- -167 dBm navigation sensitivity
- 26 seconds cold start
- low power
- 38400 baudrate, 10 Hz refresh rate
- low noise 3V3 voltage regulator
- 60 mm diameter, 32 g with protective case

FIGURE 3.11: Our GPS module, the Ublox Neo-M8N. Courtesy of Hobbyking

3.4.7 Other peripherals

No other external peripherals were used, as the BBBlue embeds a 9 axis IMU (3 axis accelerometer, 3 axis gyroscope and 3 axis magnetometer) and a barometer, all suited for robotic applications. A small foam was placed on the barometer to filter and reduce air turbulence.

3.4.8 Cabling and Packaging

Finally, we managed to place all wires sufficiently far from propellers and reducing the wiring by collecting all 3-pin R/C servo connectors from ESCs onto one layer of the drone frame and into a parallel cable wire. Moreover, by taking all wires from radio, I2C and analog sensors onto the prototype board and to similar parallel cables allowed for more freedom in choosing the position of each sensor and a more compact layout. The GPS module was mounted with its stand on the top of the chassis, ensuring no propeller was in range for its wire and reducing electromagnetic interferences. Finally, to get a clean setup, a case was designed for the BeagleBone Blue, with screw holes to keep the board still and to fix the envelope to the drone frame. The case was then 3d printed and mounted on the top and center of the

chassis. Finally, all exposed electric terminals on the power distribution board were insulated with applications of hot-glue.

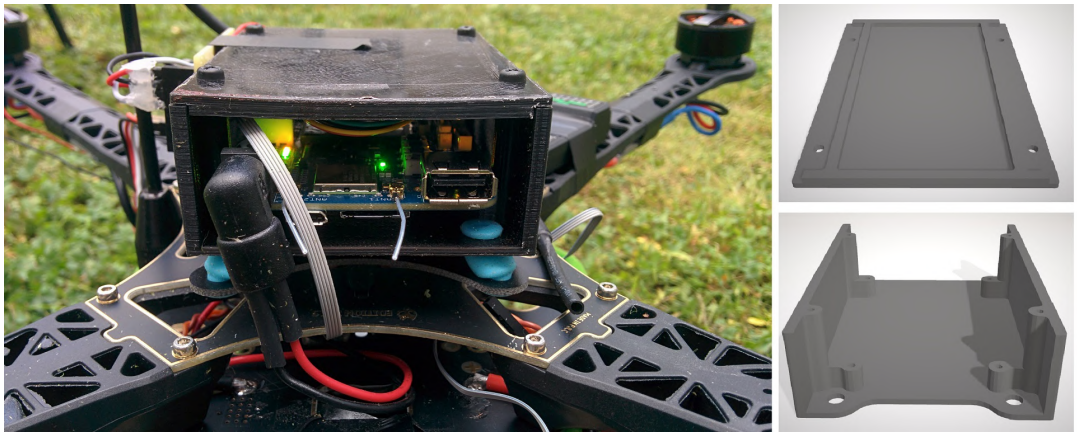


FIGURE 3.12: Our 3d Printed Case

3.4.9 Calibration

After the set up was completed, all calibration operations had to be performed using the Mission Planner software. Firstly, the compass had been calibrated using onboard calibration, a mode that allows the calibration routine to run on the flight controller, achieving more accurate results w.r.t. a typical offline calibration which relies on the ground control station software to calculate the compass offsets.
Secondly, the 3-axis accelerometer was calibrated, paying specific attention to the level position calibration, as it is usually the most important one as it is the attitude that the controller considers level when flying.
Thirdly, the radio control calibration had been performed, mapping each control signal (throttle, pitch, roll, yaw) and two additional toggle switches to the FS-i6 transmitter sticks and in turn to autopilot channels from the connected FS-iA6B receiver.
Then, the RC transmitter flight mode configuration was performed, mapping between switch positions on the transmitter and flight modes of the autopilot, binding the STABILIZE, RTL and LAND modes to one of the FS-i6 switches, while binding LAND and STABILIZE on the second switch for redundancy [Flight modes are described in detail in 4].
Finally, the ESCs were calibrated, mapping the minimum and maximum PWM values the flight controller can send them; the operation was achieved through a "all-at-once" calibration which allowed for the simultaneous calibration of all controllers.

Chapter 4

Software and Firmware

Two programming languages were adopted during coding: C/C++ and Python. C/C++ was chosen for its speed, low computational burden during execution (w.r.t. an interpreted language such as Python, even if normally it is pre-compiled) and because of RoboticsCape, a robotics hardware API for C/C++ available for the BeagleBone Black and BeagleBone Blue which implements several low level functions tested and optimized for the two boards, such as hardware initialization and check for hardware state. While C/C++ was used for retrieving data from gas sensors, being them analog or I2C, Python was used to exploit the powerful DroneKit library, allowing for an high level programming of UAV commands and telemetry manipulation. Finally, for its immediacy it has been used also to encapsulate the DroneKit functionalities in a customized simple Ground Control Station and to manage the TCP/IP communication over Wi-Fi which streams sensors data from each UAV to the ground, both coded in Python 2.7. The software and firmware was condensed into three section:

- DroneLib : all high level functions to move and control each UAV
- DroneSwarm : the core structure, it manage the drone swarm and send commands to each UAV
- DroneSense : all code related to data acquisition and display

4.1 Source Code Availability

As this work represents an open-source project, all the generated source codes as well as all the documentation, files needed to reproduce or modify this work and a pdf copy of this thesis to properly see and enjoy all hyperlinks can be found on GitHub at the following address:

https://github.com/LucaGemma/MasterThesis

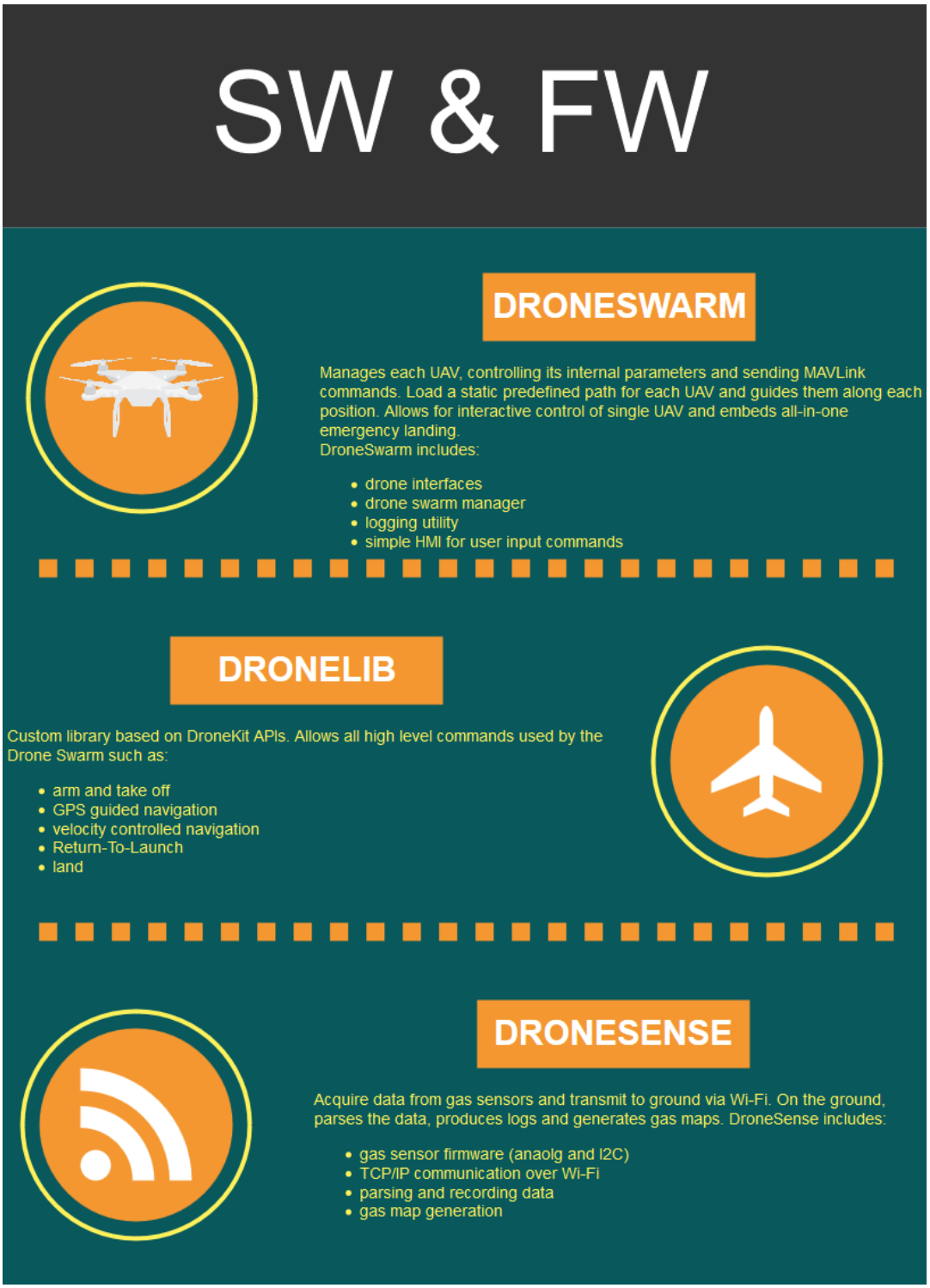


FIGURE 4.1: Software and Firmware overview

4.2 DroneLib

The entire DroneLib custom library was based on DroneKit. DroneKit is a collection of Python APIs compatible with Python 2.7 and built to manage the MAVLink communication protocol used in many UAV applications. Thanks to these APIs it is possible to connect to a UAV and get all the telemetry data it streams to the ground

as well as sending back MAVLink commands to set autopilot parameters or give several commands to the very UAV, ranging from GPS-position based movement to controlling velocities on each axis. In order to get a clear overview of what was coded thanks to DroneKit, some information must be given. First of all, the regular ArduCopter autopilot, as many other, allows to set the UAV to one among different flight modes. 19 flight modes exist, but we focused our attention on the following:

- Stabilize : for manual drive, self-levels the roll and pitch axis;
- Alt Hold : for manual drive, maintains a constant altitude while allowing roll, pitch and yaw control;
- Loiter : maintains the current location, heading and altitude;
- RTL : returns to the home location, i.e. the take-off location;
- Guided : to achieve autonomous flight, drives the drone autonomously following GPS coordinates;
- Land : lands the drone at the current location.

The first three modes were particularly useful for preliminary tests on the field of each UAV capabilities, ensuring a right overall behaviour during flight before proceeding with the autonomous flight. They were also required as emergency solutions: during tests, we equipped each UAV with a radio receiver coupled with a specific radio command and, in case of unexpected behaviours, the UAV was switched to a manual drive flight mode and recovered before crashing. The last three flight modes were used during autonomous flight, which makes larger use of the Guided mode, as in this mode the UAV can be set to fly toward a specific GPS position. Both RTL and Land modes were needed to land the drone, using the Return-to-Launch mode when GPS signal was available while recurring to Land mode to immediately make the drone land at the current position.
A noteworthy discussion has to be done for the available reference frame when locating the UAV and sending movement commands. Two main reference frame are implemented: Global frame and Local Frame. The Global Frame is based on the WGS84 CTS (Conventional Terrain System) used also for the GPS system: its centre is in the Earth centre of mass, the Z axis intercept the North Pole, the X axis is chosen such that the Greenwich meridian lays on the XZ plane and the Y axis is chosen such that it create a right-turning triad, i.e. pointing towards Asia. The global frame used in the ArduPilot software (and consequently by the Dronekit APIs) can adopt the sea level as its altitude starting point or the altitude of the UAV home location, in the latter it is called "global relative frame". Finally, the Local Frame is based on the NED – North East Down frame: commonly used in aviation, this frame places the X axis pointing toward the geographical North, the Y axis pointing toward the geographical East and the Z axis pointing down toward the Earth centre. For the relative frames implemented in the DroneKit APIs, the origin can be placed on the vehicle's home position ($MAV\_FRAME\_LOCAL\_NED$) or on the current vehicle position ($MAV\_FRAME\_LOCAL\_OFFSET\_NED$). It is also possible to get the X-axis of the local frame aligned with the current heading; following this, the two reference frames changes respectively to $MAVE\_FRAME\_BODY\_NED$ and $MAV\_FRAME\_BODY\_OFFSET\_NED$.
Having explained the main flight modes and reference frames used in this work, following there are discussed the main functions implemented with the DroneKit

API in the DroneLib library. Note that for almost every function it has been implemented a *stopaction* flag input. Each function constantly checks its value and as soon as it is set to *True* the function immediately exits. This parameter was needed by DroneSwarm code to be able to override current UAV actions with new ones.

4.2.1 Take Off

First, a check for armable status on the UAV is performed and the script is locked into a loop until the vehicle becomes armable. An UAV is armable when all the pre-arm checks are successfully performed, from the Extendend Kalman Filter (EKF) to the barometer and GPS fix. With an armable status, the UAV flight mode is set to Guided (necessary to take-off) and the drone is armed. When a drone is armed, motors start spinning at the default idle value (normally not enough to lift the UAV) and all its default parameters such as barometer, roll, pitch and yaw are set to their zero values, compensating their current offset values.
As none of the available already implemented DroneKit functions implement ACKs or similar techniques to ensure the message has been received from the UAV, it is strongly recommended to both check for a change in UAV status and send each command multiple times. Following this approach, the function does not continue unless both the status and the flight mode have effectively changed to armed and Guided respectively; if not, the command is sent again after a short sleep period. This "polling-like" approach has been adopted in every function from now on. Then, the $vehicle.simple\_takeoff(target\_altitude)$ DroneKit function is invoked multiple times within a loop to ensure it is received from the flight controller. Finally a check on the UAV current altitude is performed while the vehicle lifts and if it is within an acceptable percentage of the desired one, the UAV is considered taken off and the function exits.

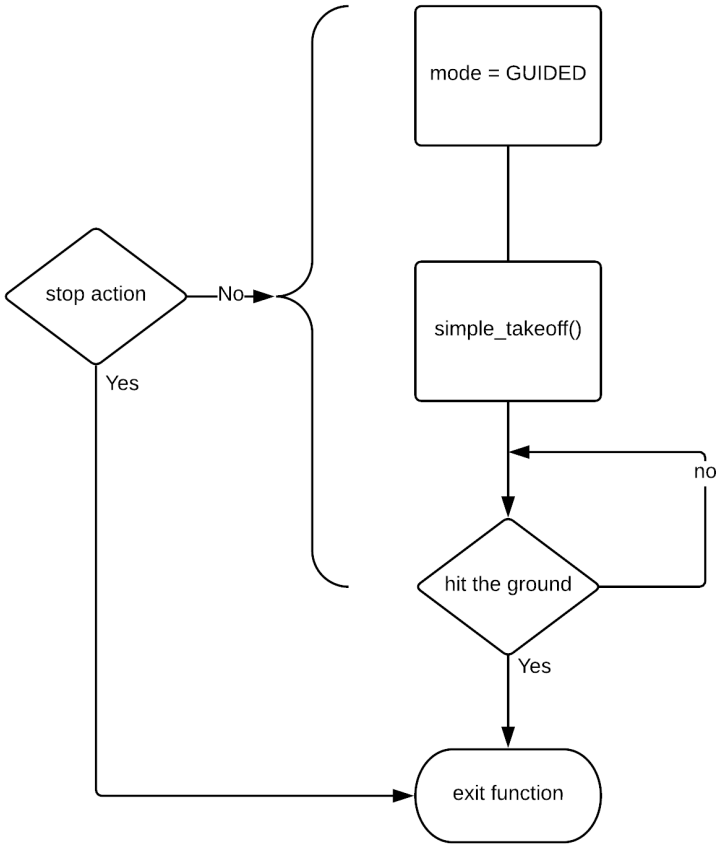


FIGURE 4.2: Arm and Take Off high level function

4.2.2 Land

First, the vehicle mode is set to LAND. As usual, the setting is performed multiple times through a loop in order to ensure the change has been performed. During landing, the UAV first descends to 10 m using the regular Altitude Hold controller, then it descends at a default rate of 50 cm/s. Both these parameters can be easily changed even with a regular GCS such as Mission Planner (and, of course, also by the DroneKit APIs). In this final stage, the UAV does not use the altitude to decide if it has landed, but instead uses its climb rate, detecting as a successful landing if the motors are at minimum and its climb rate falls between -20 cm/s and 20 cm/s. As the landing manoeuvre terminates with the disarming of the UAV (after a successful shutdown of the motors), the function enters a while loop, exiting only when the vehicle is disarmed and displaying periodically the current UAV altitude. A final printed output warns the user the UAV has landed with a LAND command.

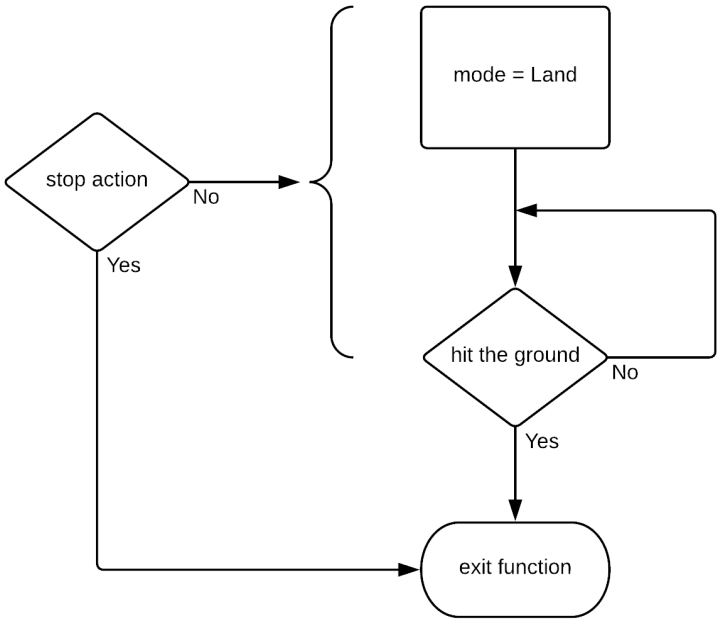


FIGURE 4.3: Land high level function

4.2.3 Return to Launch

This function is almost identical to the Land one, with the exception that the vehicle mode is set to RTL instead. The RTL feature is composed of two main stages: coming back to home location and landing. The first stage starts by raising the UAV up to a standard altitude (if above it, it will come back with its current altitude); then the vehicle returns to its home location, which is usually where the copter was first armed and taken-off. The landing stage is essentially the same as in the LAND mode (two different descending velocities and disarming the UAV when the climb rate falls in a certain threshold and the motors are at their minimum), as the RTL mode calls the same instructions performed in case of a LAND command. After the setting loop, the function enters the same while loop, constantly printing the current altitude in the global relative frame and exiting as soon as the vehicle is disarmed.

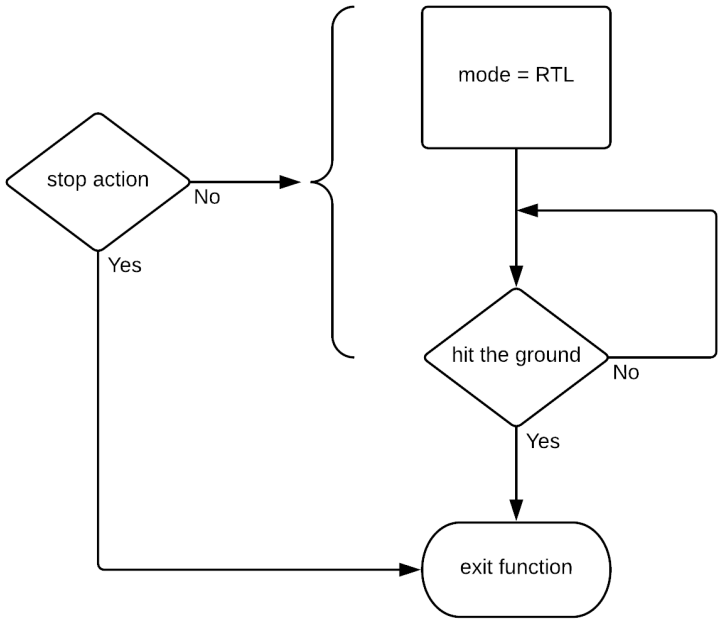


FIGURE 4.4: Return-To-Launch high level function

4.2.4 GPS guided flight

First of all, the distance from the target location is estimated thanks to DroneLib function 4.2.6, then by using the Dronekit $simple\_goto(location)$ function a command for autonomously guiding toward the desired location is sent repeatedly in a loop. The path the UAV takes is the simplest and shortest one, i.e. a straight line connecting the current UAV location and the desired final location. The function keeps track of the current UAV position and estimates the distance from the desired position and if this distance falls within an acceptable neighbourhood of the target position, the function exits.

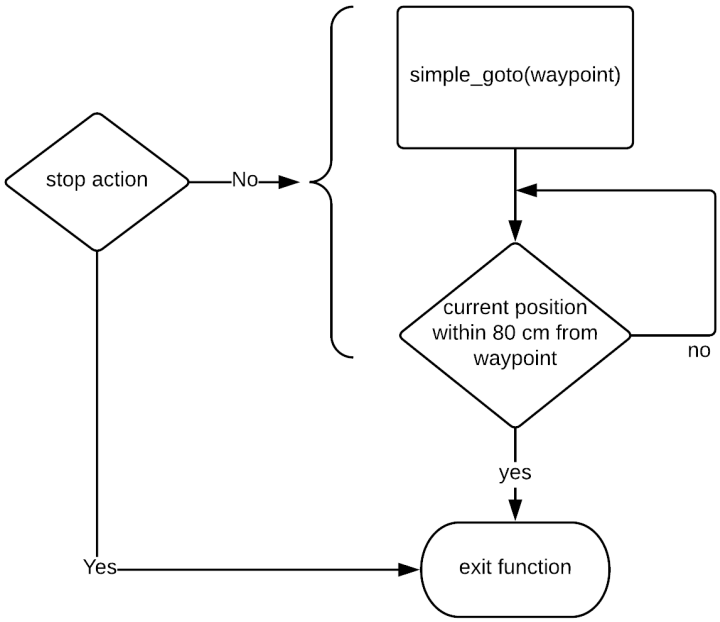


FIGURE 4.5: GPS guided flight high level function

4.2.5 Relative movement flight

The function is made of two parts: building the MAVLink command message for a velocity controlled movement and sending the message. Through the $message\_factory.set_position\_target\_local\_ned\_encode(time\_boot\_ms, target\_system, target\_component, frame, type\_mask, x, y, z, velocity\_x, velocity\_y, velocity\_z, x\_acceleration, y\_acceleration, z\_acceleration, yaw, yaw\_rate)$ function the message is built as follows:

- *time_boot_ms* as 0, currently not used
- *target_system* and *target_component* as 0, default values
- frame as $mavutil.mavlink.MAV\_FRAME\_BODY_OFFSET\_NED$, i.e. adopting a local NED frame with its origin on the current vehicle position and its X-axis pointing toward its heading.
- *type_mask* as 0b0000111111000111, i.e. enabling only speed controls
- x, y and z positions as 0, default not used values
- *velocity_x*, *velocity_y* and *velocity_z* as the desired corresponding velocities in the local NED frame

x, y and z acceleration as 0, not supported yet yaw and yaw rate as 0, not supported yet Then, the message is sent with the $vehicle.send\_mavlink()$ function.
As the relative movement flight function was designed by the author in order to guide the UAV for a certain amount of time, the velocity command is sent multiple times for a *duration* time interval, every 100 ms.

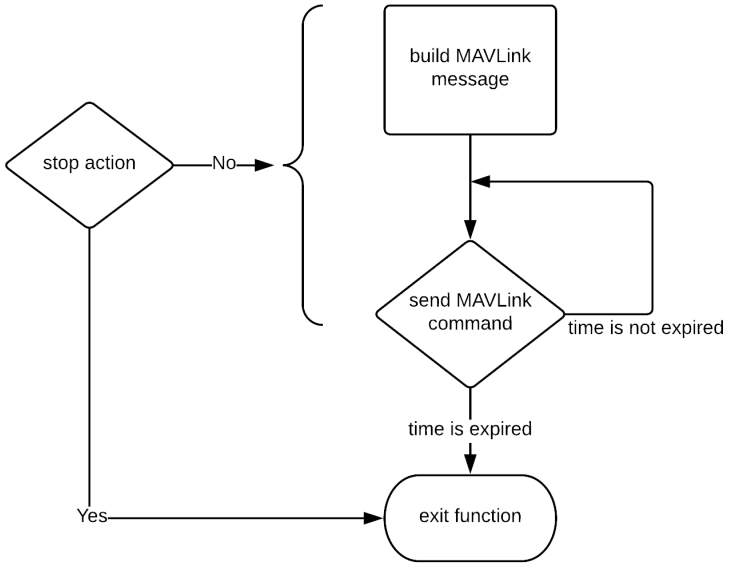


FIGURE 4.6: Velocity controlled flight high level function

4.2.6 Distance between two GPS positions

This function is used to get the ground distance in metres between two LocationGlobal objects, i.e. two positions in the Global Frame (WGS84) system. The difference between their latitude is computed, as well as the difference in longitude. Then the Euclidean distance is computed as the square root of the sum of the square of the longitude distance and the square of the latitude distance, multiplied then by a factor which compensate for the earth radius.

4.2.7 Find Wind

This is a primal DroneKit function prototype to detect wind direction and speed. No personal addition was made in this code. The concept is simple: let the UAV translate freely under the effect of wind for a fixed amount of time and compute the travelled distance. This is achieved by setting the UAV mode to *ALT_HOLD* which keeps the altitude fixed while allowing the UAV to move and translate freely on every other direction. For detecting wind direction, first a GPS guided command is sent in order to get back the UAV, then the heading of the UAV is taken. In fact, during a GPS guided command, the UAV points its nose to its target GPS position. The vector connecting its starting position to the point where the UAV ended up under the effect of wind have a slope w.r.t. the North (heading) equal to the wind direction.

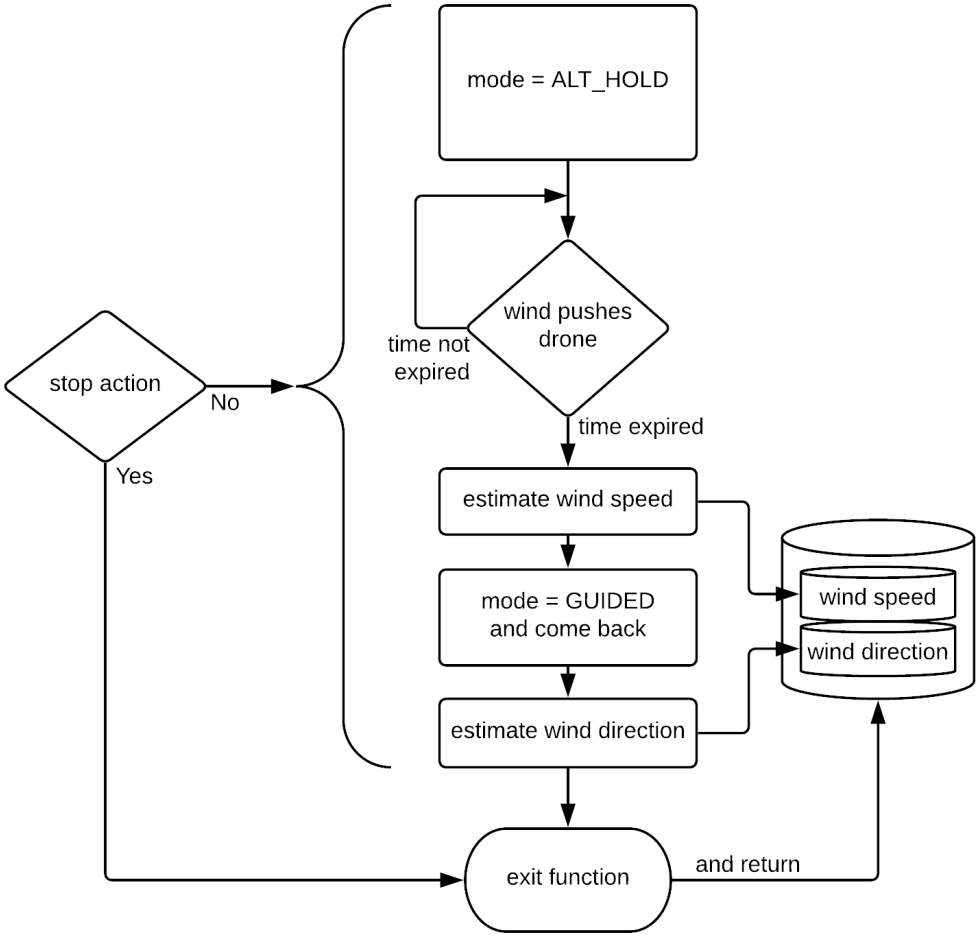


FIGURE 4.7: Find wind high level function

4.2.8 Load Path

Importing a predefined path, i.e. a collection of GPS positions, from a file is a useful way to command a drone to follow a trail. As a result, the *load_path*() function has been implemented. It was meant to be used with the output file generated from Mission Planner with the "Flight Plan" option. The Flight Plan produces a .waypoints file, which is, indeed, a collection of so-called "waypoints": a Mission Planner standard which defines a GPS position along with some other internal parameters such as the order of the waypoint in the path. As this file always has its first line for reference text, during the importing of the file the first line is always discarded. Next, the following line holds the home location. As the relevant information for the GPS position are in three specific columns ()for latitude, longitude and altitude respectively), and each column is by standard separated by a tab, the file is parsed one line at a time with the *line.rtsrip*() function and then each line is in turn split using the tab as a separation character, using the *line.split*() function. Finally, the latitude, longitude and altitude are separately appended in three lists, which at the end of the parsing operation are returned as the output of the function.

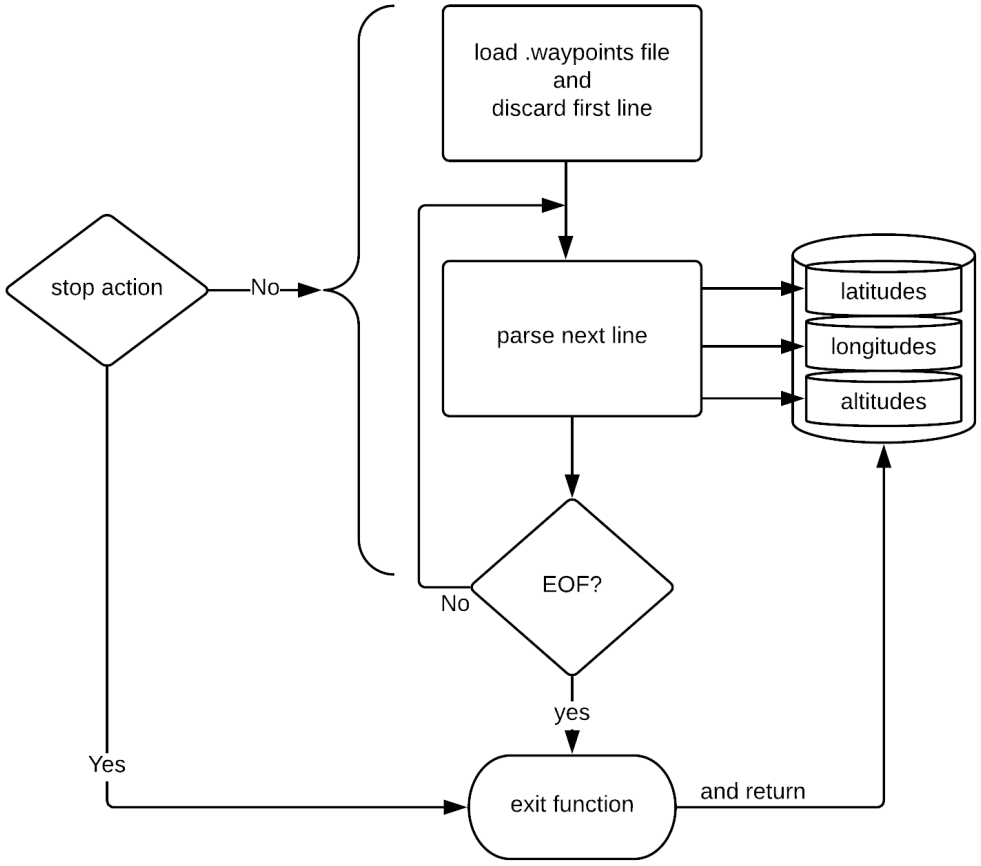


FIGURE 4.8: Path loading function

4.2.9 Getting and setting UAV parameters

DroneKit allows to access several state information, like speed and position, through python attributes. Most useful vehicle attributes are:

- location - which further includes global frame (WGS84) and local frame (NED) information,
- *home_location* – the home location for the vehicle, usually corresponding to the location the vehicle was first armed and taken off
- attitude – current vehicle attitude in pitch, yaw and roll
- velocity – current velocity in along x, y and z in m/s
- heading – current heading in degrees, where North is 0
- airspeed – current airspeed in m/s
- groundspeed – current groundspeed in m/s, i.e. the airspeed combined with the wind speed. It is the effective velocity at which the UAV is moving seen from a ground observer
- armed – indicating whether the UAV is armed or disarmed
- mode – current flight mode.

All these attributes can be read, but only the *home_location*, airspeed, groundspeed, mode and armed ones can be set. Moreover, DroneKit exposes also vehicle

parameters, i.e. variables containing information used to configure the autopilot for the vehicle-specific hardware/capabilities. It is also useful to know that all these parameters are populated when the first connection to the UAV happens and are kept updated by monitoring vehicle messages for changes, ensuring it will be always safe to read supported parameters and that their values will be consistent with what coming from the vehicle.
Parameters can be read through the *vehicle.parameter_name* attribute and, unlike attributes, they can all be set. For both attributes and parameters it is possible to observe changes thanks to listeners. Essentially, by adding callbacks to a specific attribute or parameter by using the *vehicle.add_attributelistener*() it is possible to invoke a $callback\_function$ every time the attribute/parameter is updated. The criteria for which an attribute is updated is different based on the type of observed attribute: for sensor values or other similar constantly streamed information, the value is updated (and thus the $callback\_function$ invoked) as soon as a message is received from the vehicle, while attributes related to the state of the vehicle (such as armed and mode) are updated only when their values change. Finally, parameters are updated as soon as a message is received from the vehicle.

4.2.10 MAVProxy

In order to being able to simultaneously connect both with a traditional Ground Control Station software (such as Mission Planner) and to a Python script running DroneKit, the MAVProxy tool was used. MAVProxy is a portable, light and fully functional GCS software which embeds in a command line interface all the commands required to control an UAV and manage its telemetry data. Command invocations such as the ones to arm, take off, go-to, land and RTL can be given to the UAV through MAVProxy and through a Graphical HMI it is also possible to see the UAV position over a map.
In this thesis, however, having Mission Planner as a user-friendly GCS we did not adopt MAVProxy as a GCS, but rather as a useful software tool to manage MAVLink streams. In fact, MAVProxy is capable of receiving a MAVLink stream and forward it onto one or more streams that can be further used by other GCS software or even to connect with a DroneKit script. Thus, we chose to take as input stream for MAVProxy the MAVLink stream coming from the ArduCopter autopilot software running on the UAV and splitting this connection into two UDP connection on different ports. The first would have been used as an input for the Mission Planner software, to easily visualize in a graphical way all the telemetry data as well as the status and the position over a map of the drone, while the second connection would have been taken by our Python DroneKit script. Being designed to control one vehicle per instance, we created multiple instances of MAVProxy each one for each UAV involved. As our focus is not on MAVProxy as a GCS alternative, we here inspect solely the data stream forwarding options in a brief manner.

4.3 DroneSwarm

As the project evolved, it was clear that a flexible, scalable architecture for managing and controlling multiple drones was necessary. Up to that moment, controlling a single UAV was possible as well as controlling and communicating with multiple ones separately and independently by running multiple instances of the same script, changing properly the starting parameters to reflect the different connection channels. Unless only a complete offline and static scenario would have been involved

as a result of this thesis, the algorithm we developed up to that moment was not complete and suitable for a multi-drone application. Thus we went toward a more process oriented solution, creating a simple macro-architecture to support and encapsulate our custom Ground Control Station. This code was written in Python 2.7 as well and organised in different files to keep each file dimension limited and ease the code readability. The code includes following files:

- aDroneIF.py – the single drone interface, which receives all the drone parameters from the MAVLink connection and run commands given by the swarm manager
- swarmManager.py – the structure that manages the entire drone swarm, knows each of their parameters and chooses which actions they have to perform
- swarmGlobals.py – declarations of global variables used inside each other file
- logger.py – a file defining how logs are created
- droneSwarmMng.py – the very main for this architecture, defining also the keyboard process for user input

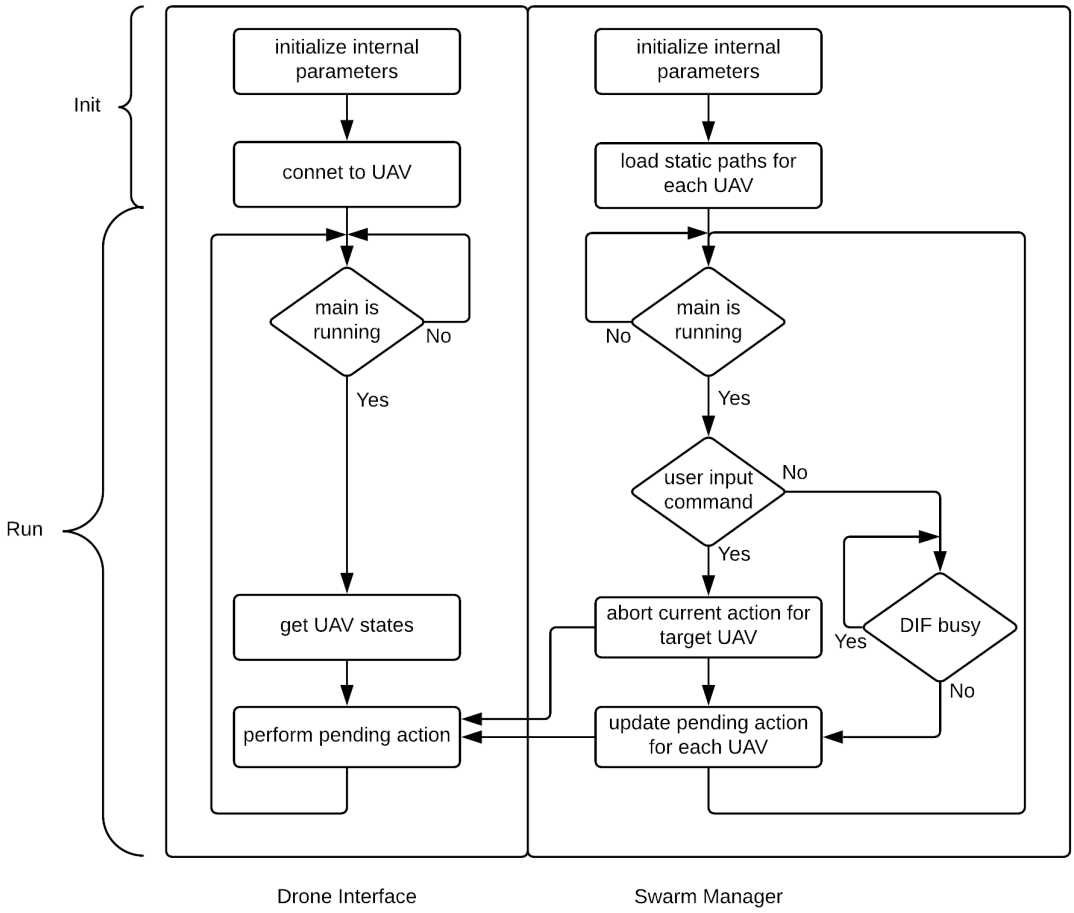


FIGURE 4.9: DroneSwarm code structure

4.3.1 Single Drone Interface (aDroneIF.py)

The aDroneIF class implementing the single drone interface is a child class of the *Process* class from *multiprocessing*, thus it inherits from its parent class. This way, we got a custom process class which can be run as a process and can exploit the same methods used for regular processes, such as the start() one. Through the__*init*__() method it is possible to get a peek at all the attributes of the aDroneIF class. Namely:

- log.l as the log level for output printing and file recording (see $swarm_logger$ later on)
- name as the name of the class
- myid as an univocal ID for the current instance of the aDroneIF
- myip as the IP address of the UAV connection of the current instance of the aDroneIF
- my_control_port as the UDP or TCP port related to the myip IP address
- my_sens_port as the port where the gas data stream related to such UAV is taken from
- ns as a global variable with several internal attributes, used for internal checks e.g. when exiting the run() method

- de_l as the drone_event flag, indicating wheter the UAV is busy with some operations or can accept a new command
- d1 and d2 as shared lists for internal parameters storage such as gas data readings
- action as a shared list of dictionaries each one structured as follows:
  - 'action':[next pending action],
  - 'altitude':[relative altitude from Home Location],
  - 'latitude':[GPS latitude coordinate],
  - 'longitude':[GPS longitude coordinate],
  - 'v_x': [desired tangential velocity],
  - 'v_y':[desired normal velocity],
  - 'v_z': [desired velocity normal w.r.t. ground]

  A dictionary so populated contains all the needed information about any type of action the UAV is required to do. Available actions are:
  - 'take-off' for taking off, the altitude is the one indicated with the 'altitude' key
  - 'land' for landing
  - 'RTL' for performing a Return to Launch
  - 'go-to' to move the UAV to the GPS position specified with the 'latitude' and 'longitude' keys
  - 'move forward' to move in velocity the UAV, following the specified 'v_x', 'v_y' and 'v_z'.
- stop_action as a shared global variable which represents a flag switched to True whenever the UAV has to immediately exit from the current action it is doing.
- sens_dict is a dictionary used for storing gas data readings of current UAV
- swarm_logger is a shared logger from the logging Python module. This helps keeping both the output printed on screen and the file log of limited dimensions by providing a flexible and efficient way to effectively choose what to display and record on file depending on the specified log level

Then the *__init__* method ends with a call at its base class by using the construct $super().\_\_init\_\_()$. This let us avoid referring to the base class explicitly and ensures that child classes that may be using cooperative multiple inheritance will call the correct next parent class function in the Method Resolution Order (MRO). Following the initialization method is the $run()$ one. This method manages all the operation required when the aDroneIF process will be run. During its starting stage, the method run a connection to the instance-related UAV following the IP and port stored as its class attributes. Then the method enters an endless loop and exits only when *ns.running* is set to False. Within this loop, it update the UAV states (such as Extended Kalman Filter Status, barometer and compass health and GPS fix), manages the current pending action for the UAV and passes to the next iteration of the loop. To perform a pending action the interface simply checks the type of action stored in

the pending action dictionary. Depending on the action, through our DroneLib custom library commands are sent to the UAV. Once the command is taken by the UAV and the action is performed, the pending action dictionary of the drone interface is updated.

4.3.2 Drone Swarm Manager (swarmManager.py)

As for the aDroneIF class, the swarmManager class is a child class of the *Process* class from *multiprocessing*, inheriting all the useful methods and attributes typical of processes in Python. The class is similar in its structure to the aDroneIF one, starting with a redefinition of the initialization method of its parent class. Even in this case, the internal attributes are initialized with the one passed as inputs during the creation of a swarmManager object:

- log_l – log level
- uav_n – number of UAVs belonging to the swarm
- ns – shared variable indicating whether to keep on running or stopping the swarmManager process
- de_l – shared list of flags for every UAV indicating whether or not the UAV is available to receive new actions.
- action – shared list of dictionaries resembling drone actions, as per the aDroneIF class
- stop_action – shared list of flag variables for each UAV indicating whether or not to interrupt the UAV current action
- name – name of the class
- drones_actions_list – list of n lists, where n is the number of UAV belonging to the swarm. Each single list contains all the programmed actions for a specific UAV.

Then for each UAV, the swarm manager loads all predefined actions from files (one for each UAV). These files contain ordered lists of actions each UAV has to perform sequentially. Such actions are then appended in lists of dictionaries, one list for each UAV, indicating its set of actions. Finally, as for the aDroneIF class, the *__init__* method ends with a call at its base class constructor. Following the initialization method is, again, the run() one. The function enters an endless loop, exiting only when the shared *ns.running* variable is set to False. Within such loop the swarm manager first checks if a keyboard command was typed for any UAV. If so, it aborts whatever action the target UAV is performing and update its pending action to the one specified by the user. If instead no keyboard command is present, it updates the pending action only for those UAV which have already completed their actions and are no longer busy; in this case the swarm manager pops the oldest action from the corresponding list of dictionaries.
As for the author's choice, the user input has the highest priority over any other command or pending action, thus as soon as the script receives a command typed by the user, this has to be executed immediately. This was meant for safety reasons, e.g. landing or RTLing a specific drone in case of issues or critical behaviours.

4.3.3 Global Variables (SwarmGlobals.py)

In this file are located all the global variables used by other files. Static variables such as class names, version of the code, file paths and regular expression precompiled

patterns are present, as well as default values for variables that may change during object instantiations or at run time like the log level, IP and port base addresses for GCS-UAV connection or gas data streaming.

4.3.4 Logging (Logger.py)

This file contains all is needed to create log files. First, a convenient function for converting system-time date into a string is defined. This function uses the useful *datetime* Python package, in particular the *datetime* class to get both a date and the time by accessing its attributes: year, month, day, hour, minute and second. Successively, a .log file is created indicating the filename, version and date of creation up to seconds; the file is conveniently placed in the log folder. Naming log files in this way makes for an easy way to quickly locate files checking both their date of creation and the file that generated them as well as keeping old files without the need to append on a single huge log file.

4.3.5 Main Script (droneSwarmMng.py)

This script contains the main loop and is the one invoked to get the Drone Swarm structure running. Firstly, it simply declares and initializes all shared variables used by each single drone interface (objects of the *aDroneIF* class) and the very swarm manager. Then it creates and starts both the swarm manager and a single drone interface for each UAV belonging to the swarm. The script then keeps stuck in a while loop exiting only when the user inputs an exit command from keyboard. In this pseudo-endless loop the script constantly read the keyboard input and passes it to the swarm manager which treats it properly as described in 4.3.2. Upon an exit command, the script exits the loop and closes all processes and connections.

4.4 DroneSense

4.4.1 Gas Map Generation

In order to get a graphical visualization of acquired data, the *gmplot* package for Python was adopted. Thanks to this powerful yet easy to use tool it is possible to render different type of data on top of Google Maps; it embeds all the code in order to generate the proper HTML and javascript files by exploiting Google APIs.
The generated code firstly creates both a map with markers of different colors following the average gas measurement and an heatmap proportional to the gas concentration. With the *gmplot.GoogleMapPlotter*() function the map is created at the latitude and longitude and zoom level specified. In order to draw the map, thus effectively generate the HTML file, the *gmplot.GoogleMapPlotter.draw*() function is called. On a first map a marker is added at regular fixed distance along the drone path using the marker() function. Gas data are coupled with their respective GPS positions, thus latitude and longitude are extracted from them. The marker color is decided with a thresholding operation: if at least one of the gas measurement is above an upper value (critical), then the measurement is classified as crucial and the marker color is set to red while with no measurement over the critical upper value but at least one above a lower value (safety), the point is registered as in need to be inspected and the marker color is yellow; finally, if all the measurement fall below even the safety value, the point is of no interest and the marker color is green.
The *marker*() function allows to associate a label text to each marker, displaying the

text as soon as the mouse cursor pass over the marker. Thus to further inspect the value of each measurement no more than a simple movement with the pointer over the interested marker is required, which displays the values that triggered the label and the GPS position related to that marker. As each call to the *marker*() function simply add a single marker, the computational burden is relatively low and it can be suitable for real-time applications, allowing to live displaying each gas data nearly as soon as it is available. Conversely, to get an heatmap the *heatmap*() function has to be invoked, which requires a collection of longitude-latitude points and at each calling.
The calling of this function redraw the entire collection of points each time it is performed, thus it is not suitable for real time application because as soon as the collection grows sufficiently large, the heatmap generation becomes unbearable for the system. Considering this, the heatmap is generated offline in a separate HTML file. The heatmap is a particular type of map with color zones proportional to their respective density of points, thus in order to get a concentration of points proportional to the measurement, area with higher gas concentrations have to be related to higher density of points. Considering this, the distance at which each point is drawn on the heatmap is dynamically changed with the same criteria applied for establishing the marker color: for a critical measurement, the drawing distance from two consecutive points is set to a minimum value, getting the higher density, while for middle and low value the distance is increased of respectively one or two steps in order to achieve different lower densities.
Finally, to get a live visualization of the marker map, the Selenium package for Python was adopted. In particular, thanks to the WebDriver extension it is possible to drive a browser natively as a user would do, either locally or on a remote machine. With the *webdriver.get*() function it is possible to open an html file and visualize it in a browser such as Firefox or Chrome. Thus, the *get*() function is called after the first generation of the marker map, in order to visualize it in a new browser tab. To periodically update the map and reload it in the browser tab, a *map_refresh*() function is created which instantiates and start a daemon Timer thread which periodically recall the *map_refresh*() function at fixed time intervals, then it redraws the marker map, updating the HTML file and finally it reloads the HTML file into the open browser tab using the *webdriver.refresh*() function. As a result, by invoking once the *map_refresh*() function after having opened in a new browser tab the HTML file it is possible to observe the marker map periodically refreshing and updating with the latest data. Results are shown on next page.

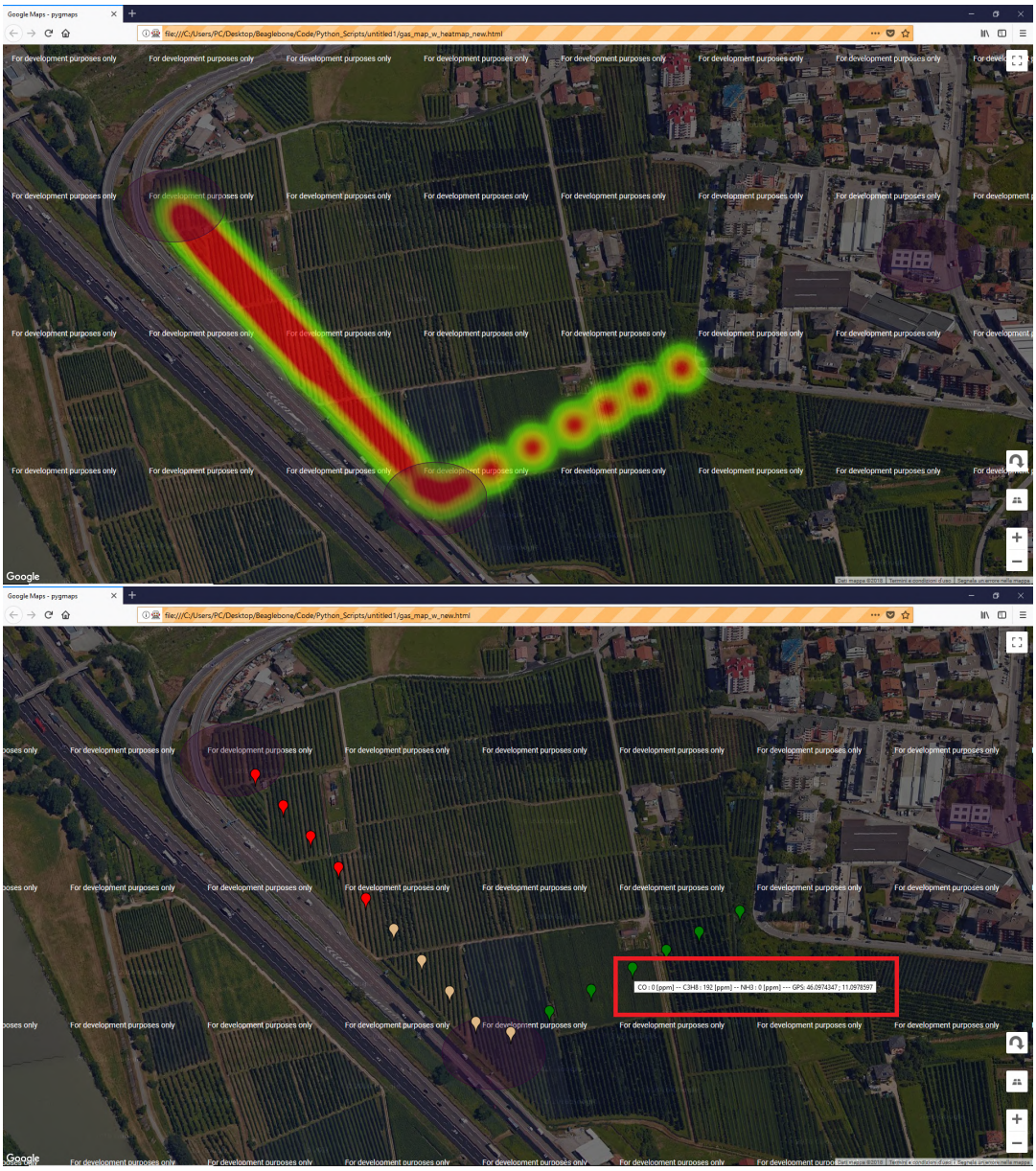


FIGURE 4.10: On top: heatmap from an increasing gas concentration scenario. On bottom: marker map for the same scenario. Note the gas concentration of each marker is displayed when inspected with the cursor

4.4.2 TCP Transmission

In order to transmit from the BeagleBone Blue board to ground the data coming from gas sensors, we chose to perform the communication via Wi-Fi with a TCP socket. Python was chosen as the coding language, as this choice simplified the ground side. In fact, due to DroneKit being a Python API, the Ground Control Station script was coded entirely in Python, thus being able to receive data through the Python's socket module was of great convenience. Firstly, a parsing section was implemented to be able to set the IP address and TCP port dynamically on each run of the script. Secondly, a socket to this TCP/IP address was created. Thirdly, the BBBlue receives from ground the system time and set it as its system time, thus achieving time synchronization. Finally, to get the stream of data produced by the

C executable file subprocesses were adopted. The Python's subprocess module allows to spawn new processes, connect to their input, output, error pipes and even obtain their return codes. In particular, by using the *Popen* class, we managed to execute a child program in a new process and the standard output was redirected into a pipe. The stream was then read one line at a time and sent through the TCP connection. To take into account interrupts in the form of SIGINT, a signal handler has been inserted, closing all instantiated process from Popen calls as soon as the interrupt would have been received.
On ground side, i.e. on the server, code was structured in the same way, with the only exception that received data are then parsed to extract each sensor readings from the communication stream. In addition, data are coupled to their relative GPS position.

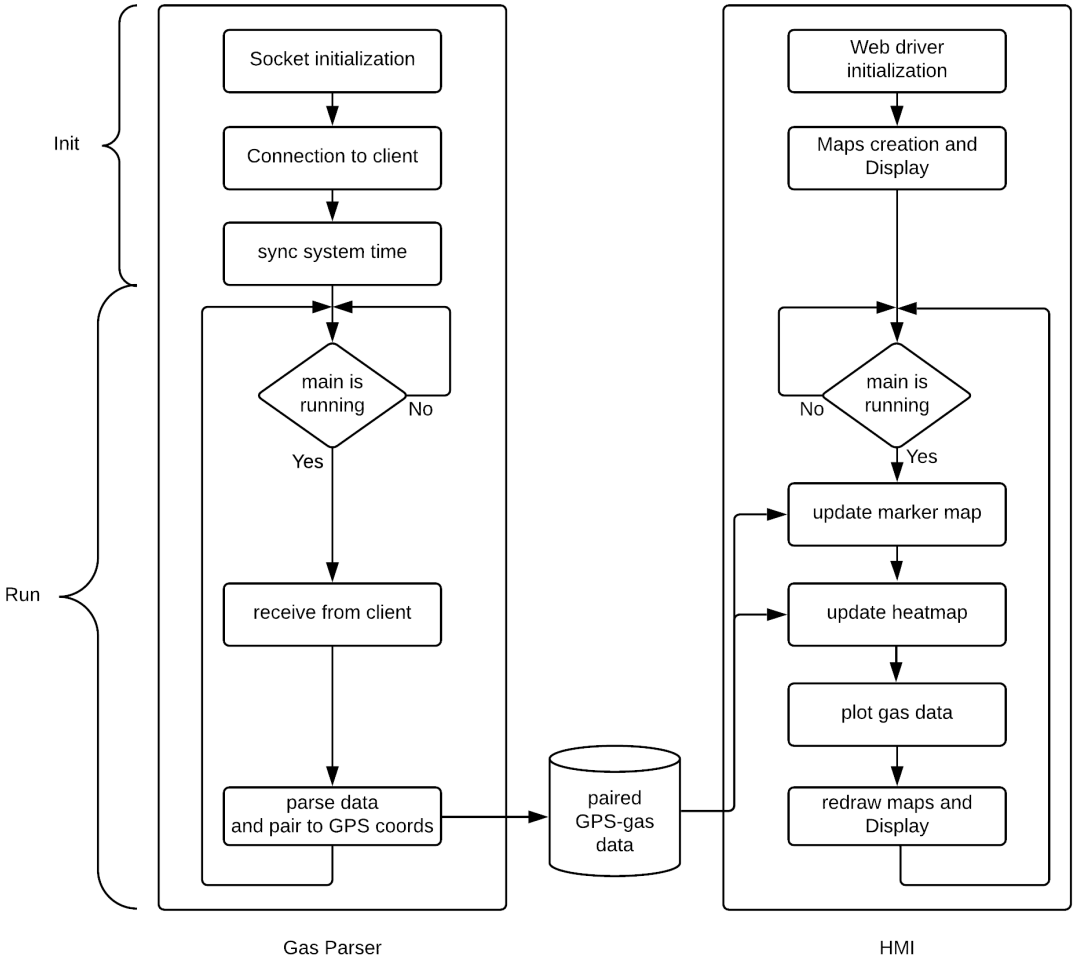


FIGURE 4.11: DroneSense (on GCS) code structure

4.4.3 Analog readings

The MQ-series and the MiCS 5524 analog gas sensors were connected to the inputs of the ADC, thus in order to retrieve their data, the ADC port was accessed. C/C++ language was used as well as the RoboticsCape library to initialize the hardware ($rc\_initialize()$), check for the hardware availability ($rc\_get_state()$) (ensuring no other process was trying to simultaneously access the peripheral such as the bus used to communicate between the ADC and the CPU) and get the readings from the A/D converter ($rc\_adc\_volt()$). For each reading a timestamp was produced along with

the very data, allowing for a better data manipulation on the Ground Control Station (GCS) side. A sleep of half a second was induced among the readings.

4.4.4 I2C readings

The MiCS 6814 digital gas sensor was connected to the I2C port of the BBBlue and the I2C protocol was used to get its data. Both the RoboticsCape library and the BeagleBone Blue are quite new on the market, so no I2C examples were available for the board; thus we analyzed the I2C functions available in the *rc_i2c.c* library from RoboticsCape and modified as needed some functions in order to get things working. The MiCS Groove gas module sensor we used was fully compatible with the Arduino platform, providing even code examples on how to use it with such board. Consequently, in order to develop properly the I2C readings C code, we started from the logic of those code examples and adopted the functions available from the I2C library to replicate a custom code compatible with our BeagleBone Blue board. Code was structured as follows:
First the hardware is initialize (*rc_initialize*()), then the very I2C bus initialization (*rc_i2c_init*()) specifying the I2C address of the MiCS 6814. Secondly, the resistance of the sensor is heat. Afterwards, code enters an endless loop in which first the timestamp is generated, then each gas concentration is computed and finally data are printed. The endless loop terminates only upon generation of a SIGINT, e.g. by pressing Ctrl + C to abort the program. Upon loop exit, the code close all threads, pointers and pending files in order to avoid conflicts (*rc_i2c_close*() and *rc_cleanup*()). To compute each gas concentration, the ratio of current over default values for the proper sensing resistance is computed and then a resistance-to-ppm formula is applied, which is, as conceivable, different from gas to gas. To get a visual feedback of the sensing operation, the onboard MiCS 6814 LED is switched on and off at the beginning and end of each computing stage. To achieve I2C communication, the *rc_i2c_send_byte*() and *rc_i2c_read_bytes_void*() were used to respectively send bytes commands to and read a specific number of bytes from the digital sensor I2C address.

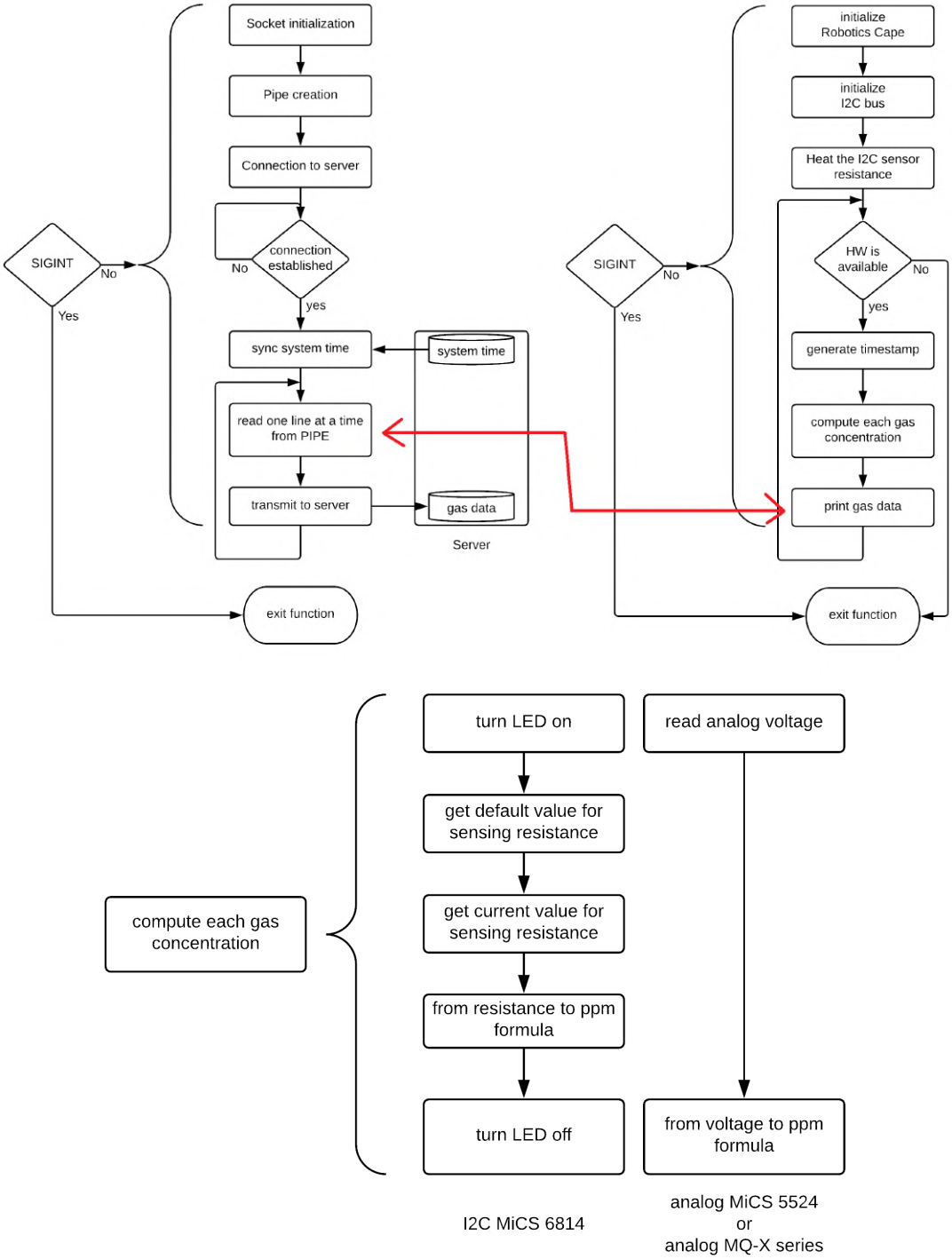


FIGURE 4.12: On top: DroneSense (on BeagleBone Blue) code structure. On bottom: focus on the *computeeachgasconcentrationblock*

4.5 SITL

Another powerful tool used in this thesis was the DroneKit Software In The Loop (SITL) simulator. Thanks to this tool, it is possible to create and test DroneKit-Python apps with no need for a real vehicle; in fact it allows to simulate an ArduPilot based autopilot and communicate with it using MAVLink over the local IP network. This not only considerably shortens time required to test new scripts and functionalities,

but also provides a safe way to effectively test the code and the ideal behaviour of a vehicle undergoing such commands. Moreover, being an embedded extension of DroneKit it can run on Windows, Linux and Mac OS X. It can be easily installed through pip tool on all platforms, however we adopted a closed environment which came with the WinPython distribution of Python for Windows along with all the necessary modules and libraries; the environment can be easily downloaded as a compressed archive and deployed with less to no additional steps a part from upgrading modules if needed. Then, in the WinPython command prompt by running a simple call to dronekit-sitl it is possible to start the SITL. DroneKit-SITL provides few simple commands to run pre-built vehicle binaries appropriate for the host operating system and can be even used to run binaries built locally by the user. Following there is a brief overview of its usage based on the github reference page for DroneKit-SITL. The simulator can be started by invoking the command $dronekit - sitl < copter(-version) > [parameters]$ with the specified autopilot version. Useful parameters are

- –home HOME to set home location in latitude, longitude, altitude and yaw
- –rate RATE to set SITL framerate
- –speedup SPEEDUP to set simulation speed
- –instance N to set multiple instance of the SITL.

Finally, other autopilot types can be used, such as plane, rover and solo for other kind of vehicles. We found of great usage the $\check{} - instance$ parameter to simulate multiple drones simultaneously, being able to test a multi-UAV setup in the same and easy manner as for a single UAV.

Chapter 5

Results

To get a better estimate of the entire system and any possible room for improvements, we performed several tests both on software and on the hardware. For the software, we test the CPU and memory percentage usage during under different conditions, while for the hardware we test the WiFi communication, the battery packs and the overall flight performance. Below are discussed one after the other all the performed tests along with the encountered issues and the asopted solutions.

5.1 Flight Site

We tested the flight capability of our UAVs at two different sites, both outdoor and with stable and good GPS fix. First site is on a park near Povo, Trento. In this site, the presence of a volleyball court helped in detecting the precision of the GPS guided movement, as we took each corner of the court as our waypoint for the track. Second site is near Lavis, Trento. We used this site as a wider outdoor environment with no obstacles to test autonomous guide on longer tracks and reaching higher altitudes. In addition, we used this site to perform our battery tests. Both sites are shown at the end of this section. On both site we achieved good if not optimal GPS fix and we confirmed the high precision of our GPS module, being able to achieve up to 60 to 90 cm of accuracy. We were able to set the perimeter of the volleyball court as in Figure 5.2 and saw the UAV carefully reach within 60 cm from each of its corner.
In addition we tested the GeoFence option of our flight controllers. GeoFence is a useful tool which let the user define a figure, either a predefined cylinder, square or an arbitrary polygon, which represents the flight zone. The flight controller will consider each zone outside of its edges, in each dimension, as a no-fly zone and trigger a predefined action, by default a RTL. If the UAV, for whatever reason, reach a predefined distance over the edge of the no fly-zone, by default 100 m, a LAND command is immediately send replacing the RTL. We were able to define this virtual fence as a cylinder with radius as 10 meters from the farthest point the UAV would have reached during its flight and 5 meters higher than the maximum altitude as its ceiling. In conclusion, the GeoFence proof itself as a reliable safety measure when dealing with critical errors which made the UAV fly away from its intended path. However, a remark has to be made on GeoFencing: because it relies heavily on GPS signal, quality of the GPS fix has to be good in order for it to work properly and, as conceivable, it cannot represent a safety measure against fatal errors caused by GPS glitches or malfunctioning.

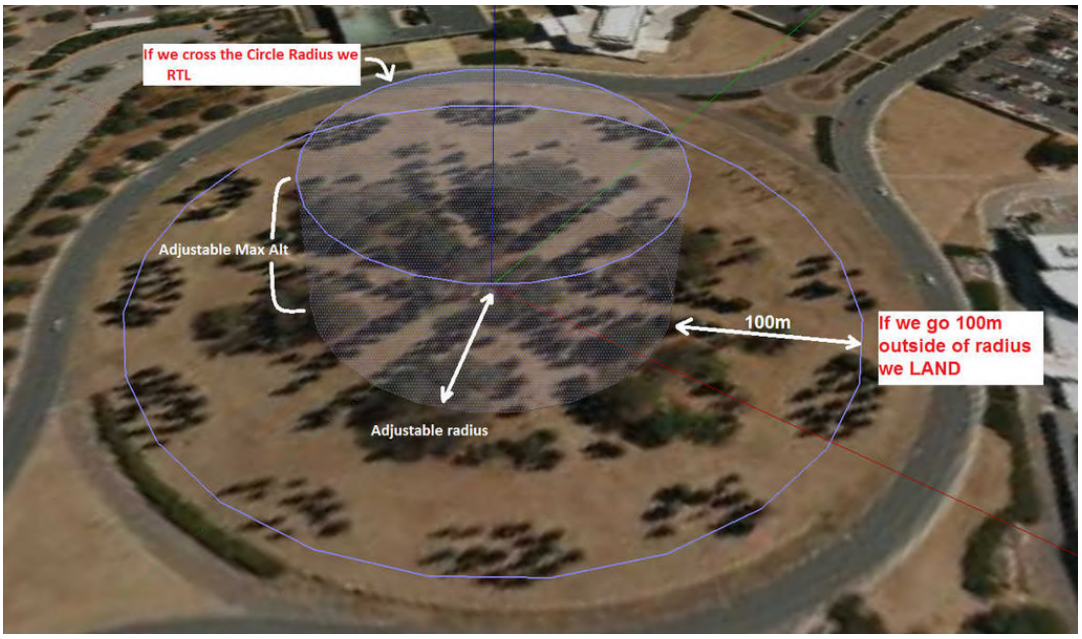


FIGURE 5.1: The GeoFence, a virtual area outside of which is considered no-fly zone

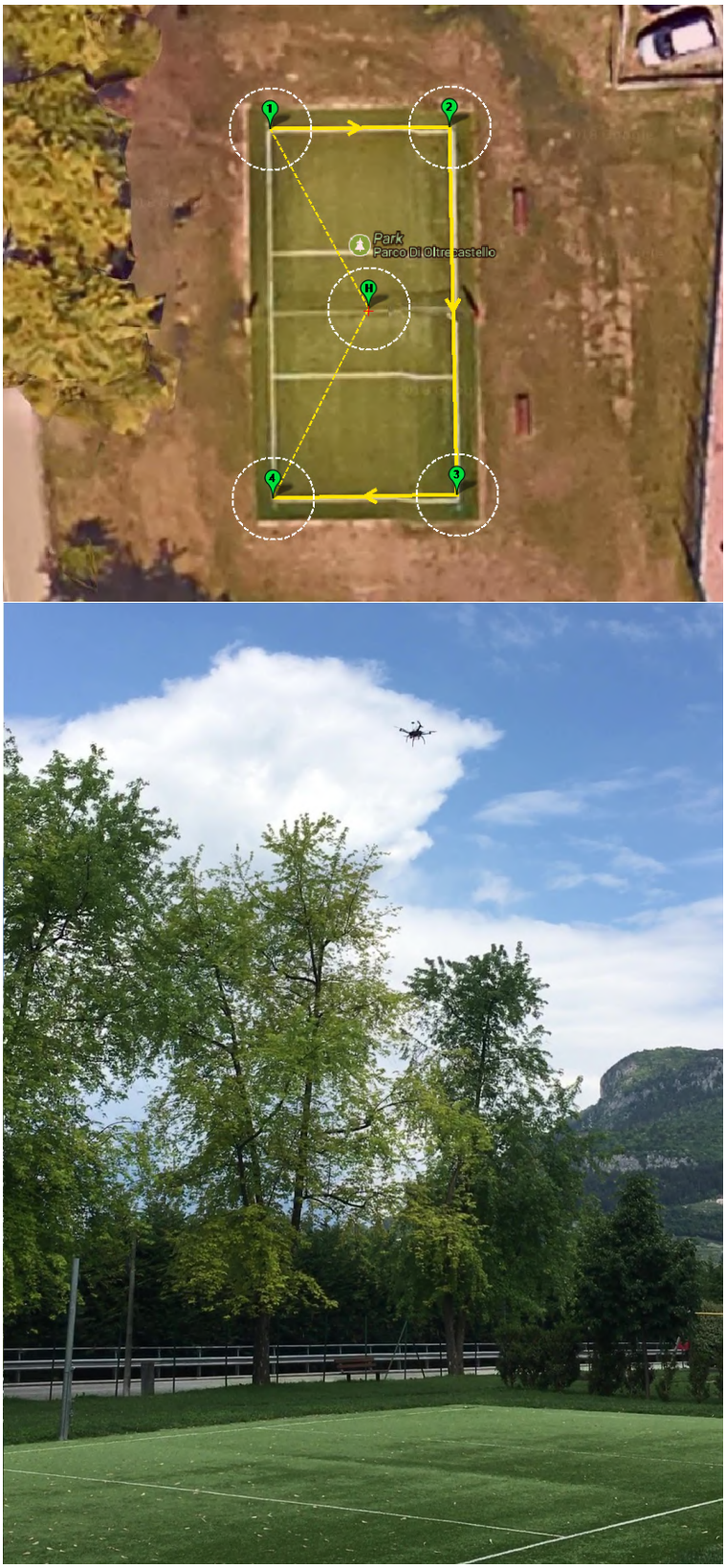


FIGURE 5.2: First site (yellow line: 20x10m). Volleyball court lines helped in detecting GPS accuracy

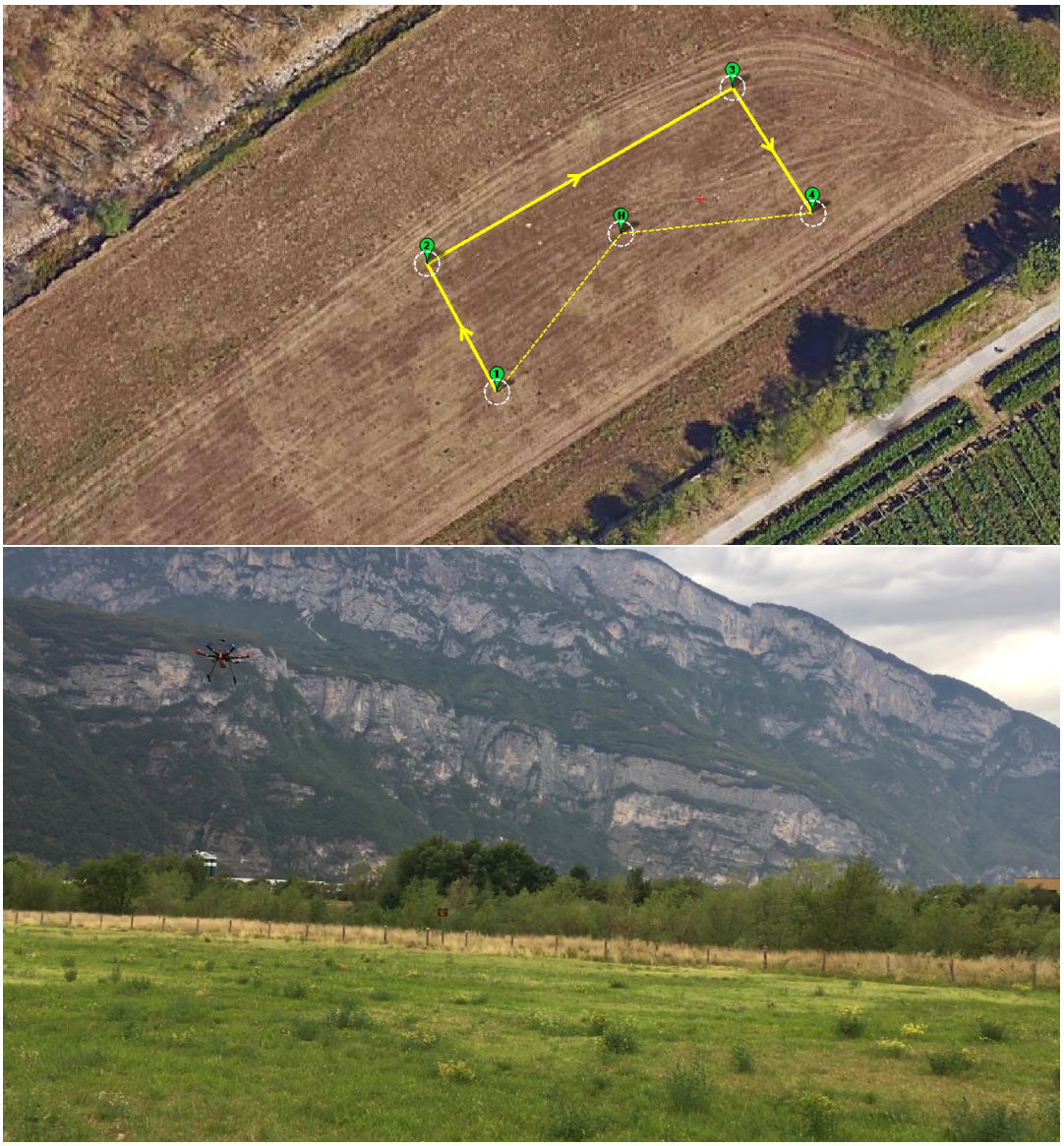


FIGURE 5.3: Second site (yellow line: 80x30m). The wider area allowed testing different speed

5.2 Battery Test

As the BeagleBone Blue can be powered even through its 2S LiPo balance charger input, we started from a set up of a single 2200 mAh 2S battery for our first flights. With this configuration, a single 2S battery powered both the ESCs (and consequently, the motors) and the board, by connecting the XT60 power connector to a female one soldered onto the Power Distribution Board of the drone frame and the balance LiPo charger to the 2S LiPo charger input of the BBBlue. Soon, we realized the voltage provided by a 2S battery, nominally 7.4 V, was barely sufficient to take off the quadcopter and flight performance drained and degaded quickly during flight, causing altitude loss and unstability.
Consequently, we upgraded our battery pack to a 4S one. The new pack was obtained by a series of two 2S 2200 mAh batteries and the connection was realized with a custom three-way XT60 connector. The battery pack was plug into the female XT60 connector of the frame PDB as usual, with no more LiPo charger connections

to the BeagleBone. In fact, we powered the board straight from the PDB by using a soldered barrel jack connector plugged into the DC jack connector of the board. This was possible thanks to the 2A 5V regulator mounted on the BeagleBone which regulates both the jack and the LiPo charger inputs. Following the BBBlue schematics, the regulator is the AP1509 from Diodes Incorporated. According to its datasheet, it can take up to 22 V as operating input voltage, thus a 4S battery pack, with a 14.8 V nominal voltage and a maximum voltage of 16.8 V when fully charged, is within the allowed operating conditions. As conceivable, with the higher voltage of the new battery pack, the quadcopter showed a completely different behavior, achieving a stable, reactive flight both in manual and in guided flights.
As we assembled the second UAV, an hexacopter based on the S550 frame kit, we run into the need for increased battery capacity, thus we mounted on our hexacopter a 4S 10400 mAh battery pack made with a serie of two 2S 10400 mAh batteries. We performed several tests to inspect the average flight time with both options, using the hexacopter for testing both the 2200 mAh and the 10400 mAh solutions. We tested battery performance under four conditions: hovering, slow movement (1 m/s), moderate movement (3 m/s) and fast movement (5 m/s). Hovering condition was achieved by taking off the drone and make it keep the altitude with no further commands, while for all movements conditions tests were performed by autonomously commanding the drone to travel back and forth over a fixed distance known a priori. Results are shown on next page.

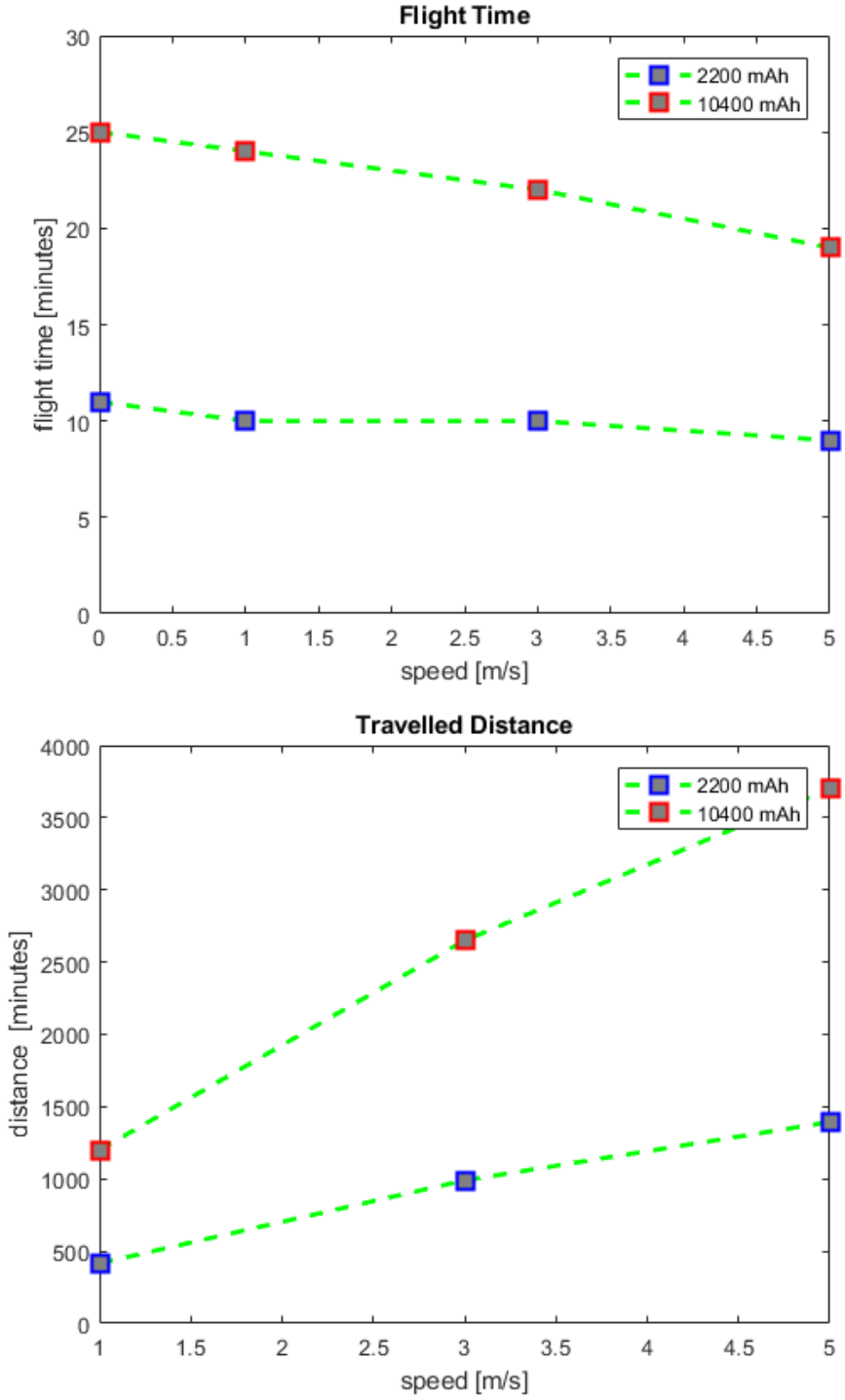


FIGURE 5.4: Battery Tests for 2200 and 10400 mAh 4S battery pack. On top: Flight time vs Speed. On bottom: Travelled distance vs speed

5.3 CPU and Memory Usage

We installed the htop interactive Linux process viewer on the BBBlue to quickly manage the CPU and memory percentage used and inspect what processes were requiring those resources. We tested the percentages on idle on a fresh new start, few minutes after the ArduCopter start and after an ordinary flight. Tests showed that the average percentage of CPU and memory usage both falls within acceptable

values not saturating the board resources and oscillates around a mean value without dramatic resource drainage as time passes. Nevertheless, it is worth saying that sporadically an increase in the CPU usage can be registered when the board heats up, that is after several minutes of work, typically twenty to thirty. This, however, can be ascribed neither to the ArduCopter flight controller code nor to the custom code developed during this thesis (sensor firmware or connection scripts), as their percentage remain almost the same during board operations.
Beyond doubt, tests showed that most of the increasing in resource usage is due to the Robotics Cape Battery Monitor process, which manages the battery charging, protection and monitoring circuitry. BeagleBone developers has claimed a possible spike in resource usage due to this process as it is in its early stages for the BBBlue, however this feature can be easily shut down in our setup as we do not require its functionalities.

5.4 Overall Flight Performance - the eCalc tool

An extremely useful tool we used during our project was the eCalc motor calculator. This online simulator has a huge database of motors and propellers from several different producers and giving a configuration of your UAV computes many helpful flight parameters such as the flight time, the available payload and the current drawn from the motors. It is based on common physics and mathematical model and through a comprehensive selection of parameters it evaluates the overall flight performance of your vehicle, helping your decision on what kind of hardware you should mount. Here is look at how the tool appears, all results and plots are produced with hardware similar to the one of our final configuration for the hexacopter. All entries and results are discussed reporting the eCalc help documentation.

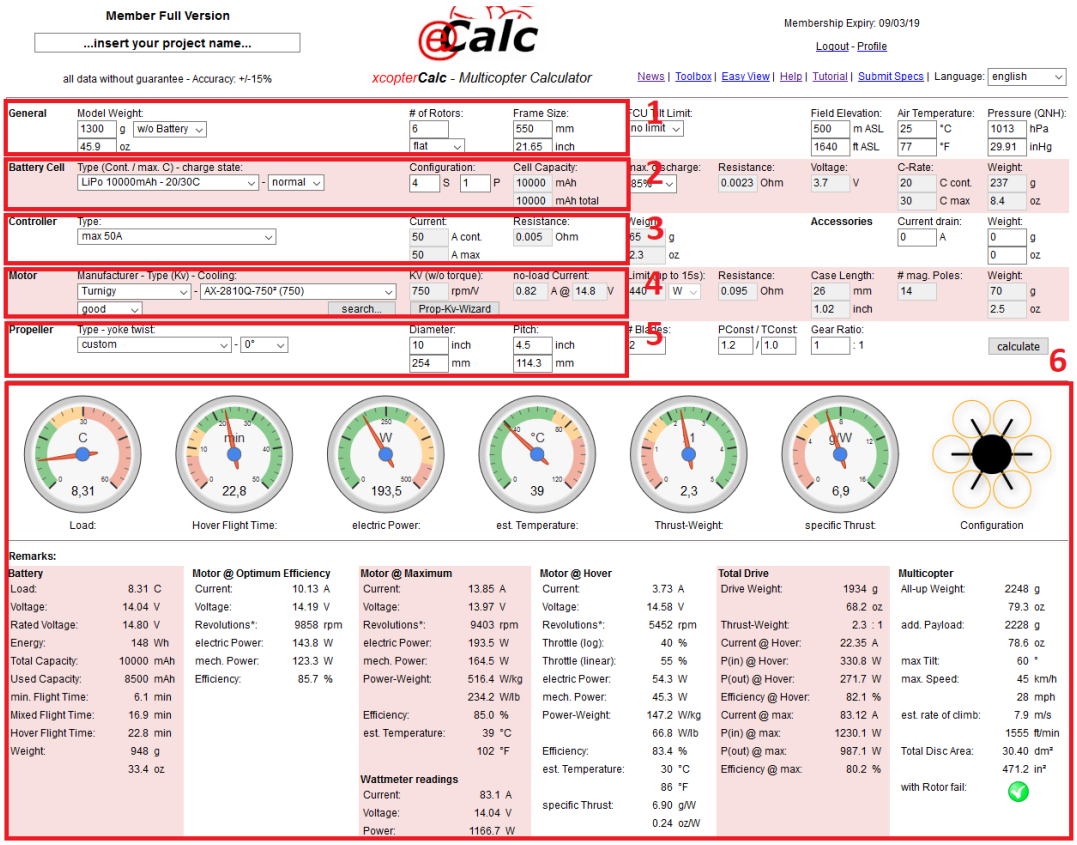


FIGURE 5.5: How eCalc appears

1 - General

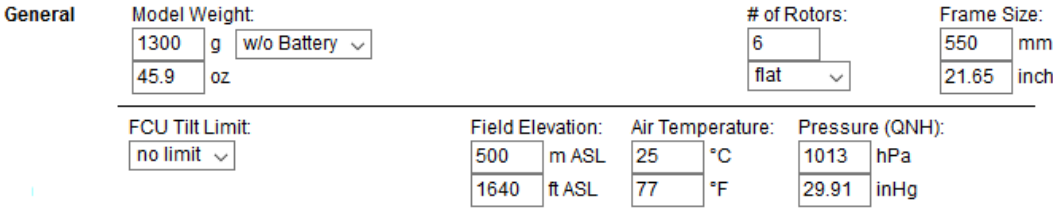


FIGURE 5.6: General Section

Here you can set the general frame parameters: weight of the UAV (with or without motors and/or battery pack), the number of rotors and the frame size. Additionally, you can also set the environmental data: elevation from sea level, typical air temperature and pressure of your test field.

Our Configuration : The hexa frame is a 550 mm diameter frame and its weight, exlcuding the battery pack, is of 1300 g.

2 - Battery

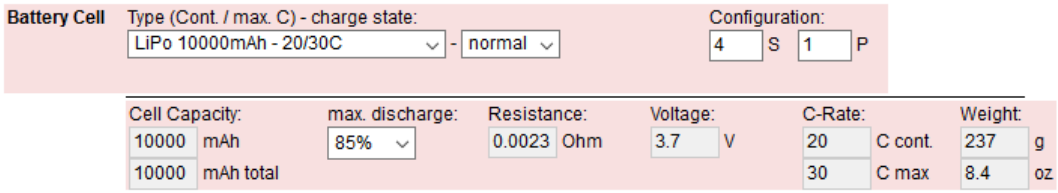


FIGURE 5.7: Battery Section

Stated here are the specs for your battery pack: battery type (capacity, continuous and burst discharge), battery configuration (2S, 3S, 4S and so on) and its maximum discharge percentage admissible.
Our Configuration : The hexa mounts a 4S 10400 mAh battery pack, the selected one resembled the most the pack.

3 - Electronic Speed Controller

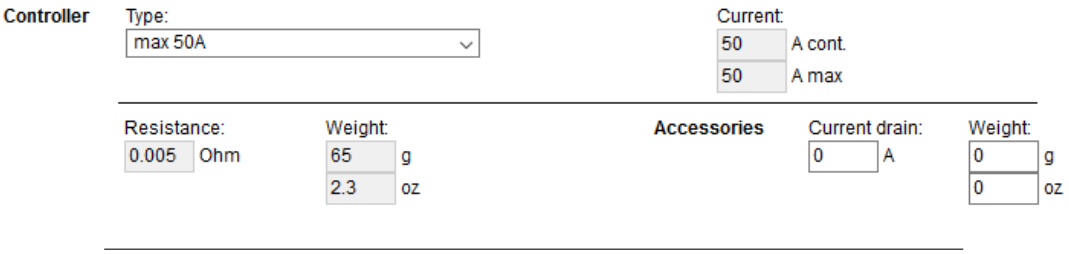


FIGURE 5.8: Electronic Speed Controller Section

The ESC specs, chosen among several different ESC brands or use a generic one, setting its maximum supported current.
Our Configuration : We equipped each copter with a 51 A 3S-6S race Electronic Speed Controller.

4 - Motor

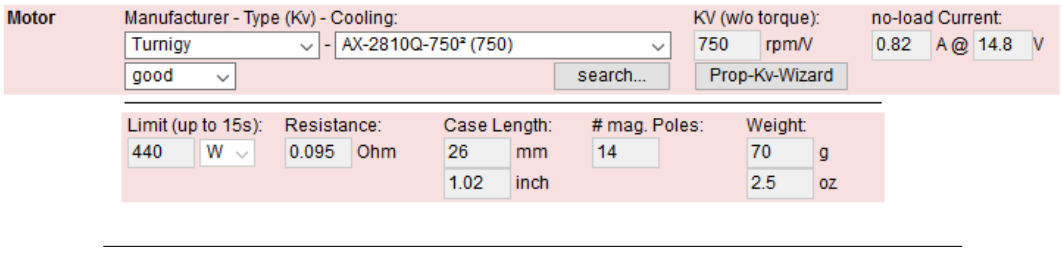


FIGURE 5.9: Motor Section

Here you may insert your choice for the adopted motors, selecting from an almost endless list of models and producers, the tool will automatically fill the remaining fields such as KV, current, power limit and weight.
Our Configuration : We chose the AX-2810Q 750 KV Turnigy motors.

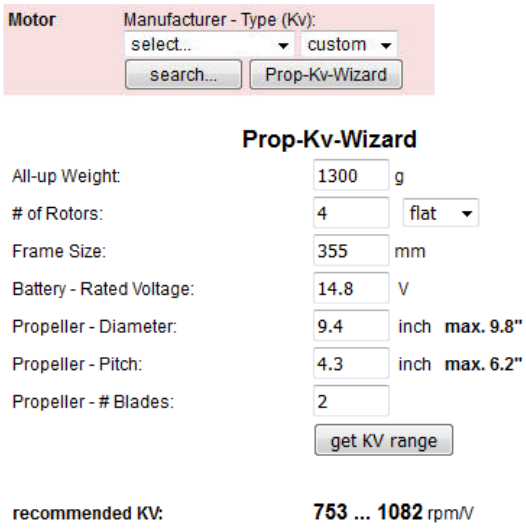


FIGURE 5.10: Prop-KV-Wizard

If you are looking for how many KV are needed for your configuration, the Prop-KV-Wizard helps you by computing automatically the recommended KV range from the total weight of your UAV, number of rotors, frame size and your propellers.

5 - Propeller

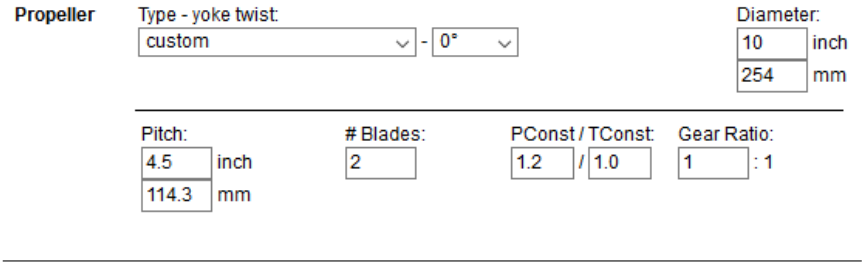


FIGURE 5.11: Propeller Section

The propellers mounted on board, selected from a list of brands or use a generic one and specify the prop diameter, pitch and number of blades.
Our Configuration : each of our UAV is equipped with two-blades 1045 propellers, featuring 10 inches diameter and 4.5 inches of pitch.

6 - Results

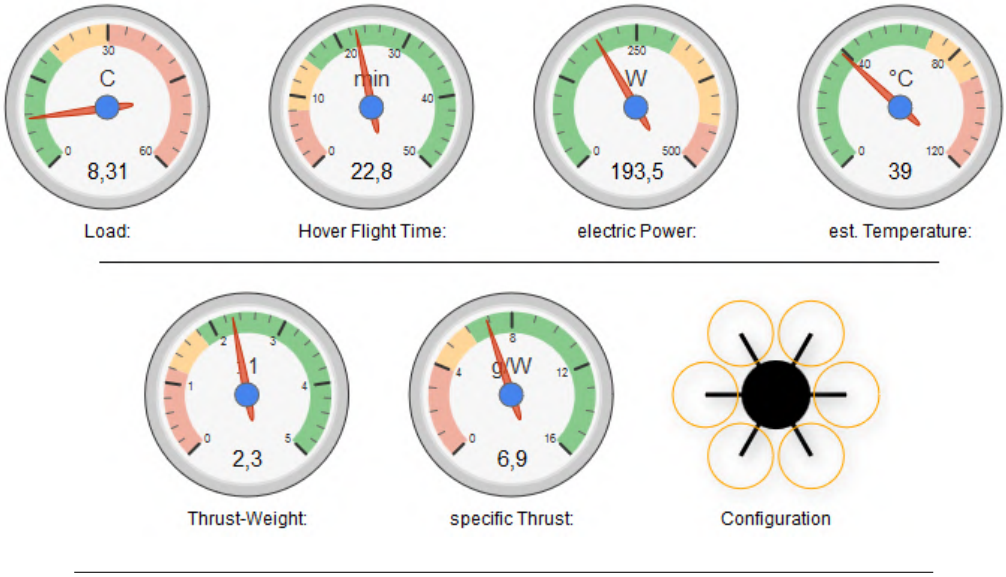


FIGURE 5.12: Quick Check Gauges

Remarks:

Battery		
Load:	8.31	C
Voltage:	14.04	V
Rated Voltage:	14.80	V
Energy:	148	Wh
Total Capacity:	10000	mAh
Used Capacity:	8500	mAh
min. Flight Time:	6.1	min
Mixed Flight Time:	16.9	min
Hover Flight Time:	22.8	min
Weight:	948	g
	33.4	oz

Motor @ Optimum Efficiency		
Current:	10.13	A
Voltage:	14.19	V
Revolutions*:	9858	rpm
electric Power:	143.8	W
mech. Power:	123.3	W
Efficiency:	85.7	%

Motor @ Maximum		
Current:	13.85	A
Voltage:	13.97	V
Revolutions*:	9403	rpm
electric Power:	193.5	W
mech. Power:	164.5	W
Power-Weight:	516.4	W/kg
	234.2	W/lb
Efficiency:	85.0	%
est. Temperature:	39	°C
	102	°F

Wattmeter readings		
Current:	83.1	A
Voltage:	14.04	V
Power:	1166.7	W

Motor @ Hover		
Current:	3.73	A
Voltage:	14.58	V
Revolutions*:	5452	rpm
Throttle (log):	40	%
Throttle (linear):	55	%
electric Power:	54.3	W
mech. Power:	45.3	W
Power-Weight:	147.2	W/kg
	66.8	W/lb
Efficiency:	83.4	%
est. Temperature:	30	°C
	86	°F
specific Thrust:	6.90	g/W
	0.24	oz/W

Total Drive		
Drive Weight:	1934	g
	68.2	oz
Thrust-Weight:	2.3	: 1
Current @ Hover:	22.35	A
P(in) @ Hover:	330.8	W
P(out) @ Hover:	271.7	W
Efficiency @ Hover:	82.1	%
Current @ max:	83.12	A
P(in) @ max:	1230.1	W
P(out) @ max:	987.1	W
Efficiency @ max:	80.2	%

Multicopter		
All-up Weight:	2248	g
	79.3	oz
add. Payload:	2228	g
	78.6	oz
max Tilt:	60	°
max. Speed:	45	km/h
	28	mph
est. rate of climb:	7.9	m/s
	1555	ft/min
Total Disc Area:	30.40	dm²
	471.2	in²
with Rotor fail:		

FIGURE 5.13: Remarks

Results are generated both graphically through speedometer-like gauges and numerically, condensed in multiple sections, here are reported the most significant ones. Gauges:

- Load: discharge rate of battery. Green if falling within the continuous C-rate, yellow for burst C-rate and red for beyond limits of the battery pack
- Hover Flight Time: estimated flight time in hover according to the computed total weight
- Electric Power: maximum value of power required in relation to specified motor limits. Green if falling within 70% of limitation, yellow for near-limitation and red for over limit
- Estimated Temperature: estimated maximum temperature. Green for 0..70 deg C, yellow for 70..90 deg C and red for overheat risk
- Thrust-Weight: thrust-weight ratio, showing the flight performance of the UAV, the higher the more agile the copter

- Specific Thrust: the specific thrust as an indication of the overall hover efficiency
- Configuration: quick look at the props configuration in relation to the frame, checking for colliding or overlapping propellers

For the Remark, looking at the Battery section:

- Voltage: battery voltage under expected maximum current
- minimum Flight Time: estimated flight time based on maximum throttle, independent of the weight
- mixed Flight Time: estimated flight time based on total weight and considering movement
- hover Flight Time: estimated flight time based on total weight and considering hovering only

Motor at Optimal Efficiency:

- Current: current for maximum motor efficiency with varying torque

Motor at Maximum:

- maximum Current: maximum current drawn, for each motor
- Revolutions: maximum revolutions
- Efficiency: efficiency at maximum current draw
- estimated Temperature: estimated temperature of the motor case, subject to the motor cooling.

Motor at Hover:

- Current: estimated current for hovering
- Efficiency: motor efficiency at hovering following the Power-Weight ratio
- estimated Temperature: estimated motor temperature at hovering

Total Drive:

- Thrust-Weight: Thrust-Weight ratio
- Current at Hover: total motor current when hovering
- Efficiency at Hover: total efficiency when hovering
- Current at max: total motor current at full thrust
- Efficiency at max: total efficiency at full thrust

Multicopter:

- All-up Weight: total weight without payload
- additional Payload: maximum additional payload possible to hover with 80 % throttle
- maximum Speed: estimated maximum attainable forward speed at max tilt and max throttle

- with Rotor fail: predicted controllability of the multicopter in case of a rotor or engine failure. Green for resistant to single rotor failure, i.e. hovering is possible with less than 80 % throttle, yellow for borderline controllable, i.e. controllable for an immediate emergency landing and red for uncontrollable, i.e. a flip-over is almost guaranteed.

Additionally, two other results are produced as plots: range estimation and motor characteristics. The Range Estimation represents a raw estimate of the expected maximum reachable total distance in still air on a level flight; the translational lift is not yet considered. The Motor Characteristics represents the full thrust parameters with increasing current with marked maximum values. These profiles (shown in the next page) are consistent with our battery tests in 5.2. In fact, because all tests were performed above the "best range" flight speed, from results produced in 5.2 we have not reached the peak of the Range over Air Speed curve (shown in 5.14). We can also notice that both 5.14 and our battery tests shows the flight time decreases almost linearly as flight speed increases.Results are shown on next page.

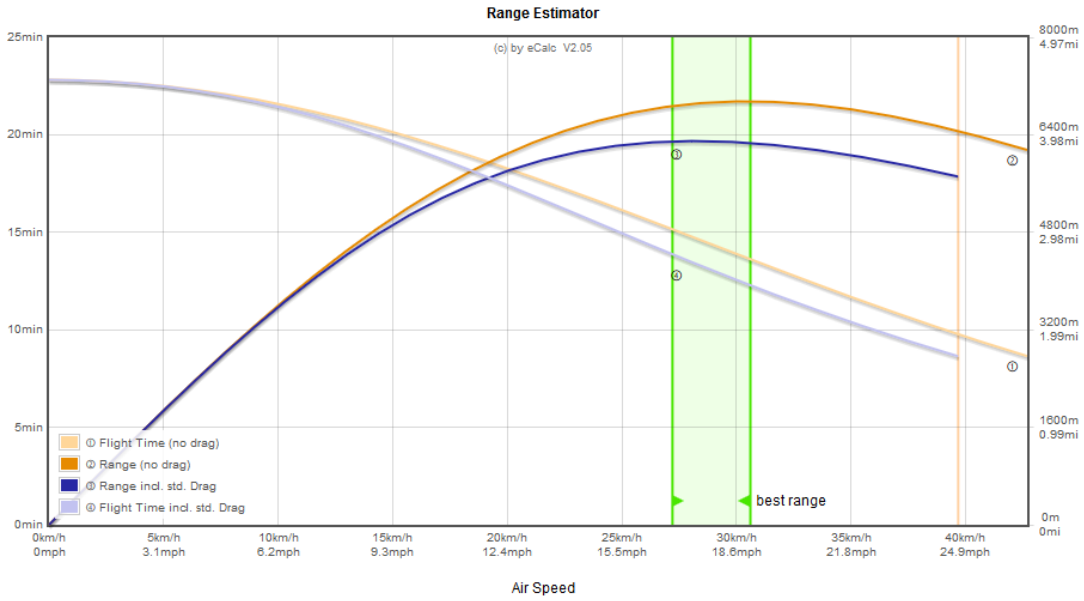


FIGURE 5.14: Range Estimation. Flight Time w/o drag, Range w/o drag, Range w. drag, Flight time w. drag

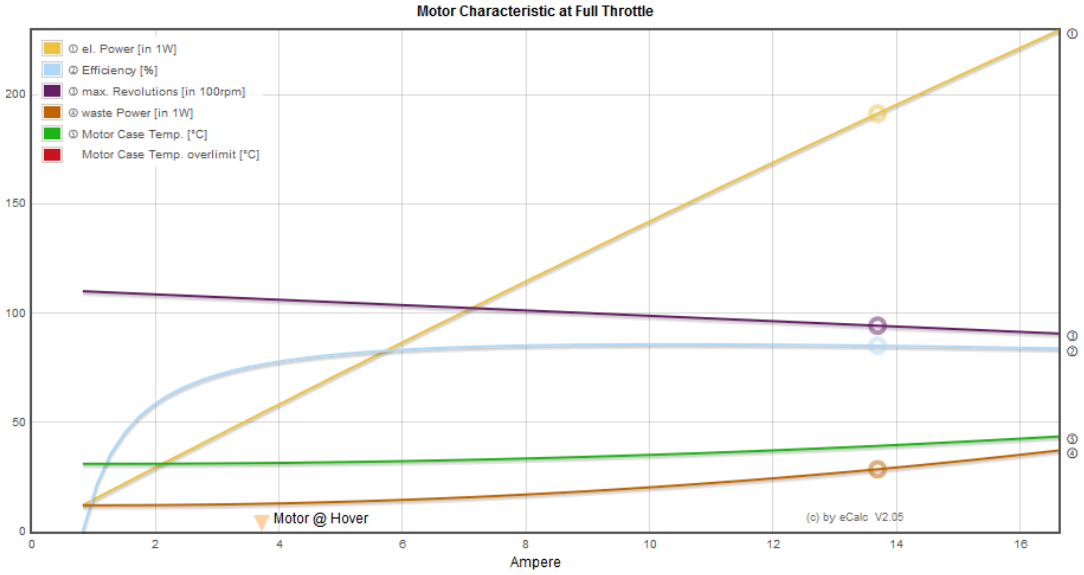


FIGURE 5.15: Motor Characteristics at Full Throttle. El. Power [in 1W], Efficiency [%], Max revolutions [in 100rpm], waste Power [in 1W], Motor case temp, Motor case temp overlimit

5.5 WiFi communication

In order to estimate the communication range of the BeagleBone Blue WiFi, we performed some range tests over a variable distance. After adopting a multi-UAV structure, we changed also the communication block, keeping the WiFi as our communication technology but adding a router between the UAV and the ground station. In fact, connecting to the WiFi access point of the BeagleBone was no more possible, as the server side had to be moved toward the ground station. In order to be independent from the platform the ground station run on, we chose a router as our server-side for the WiFi communication, getting a more versatile structure and the possibility to deploy the communication server even separated from the ground station. Thanks to this, it was possible to assign fixed IPs to each UAV and this simplified not only the connection stages but all the automatic services running on board which required IP addresses.
Having slightly modified the communication block, we tested the communication range capability with this final set-up, to have a raw estimate of the overall structure. The router was powered with a 3500 mAh 3S battery and its input voltage (it admitted 5V) was regulated by connecting a DC/DC step-down converter between the battery and the device. Tests were performed in park, thus an open environment, with some obstacles like trees and benches to be more similar to a real case scenario were some obstacles may be present between the UAV and the ground station. To detect a loss of connection, a simple script were made that periodically pinged the BeagleBone IP while the board simultaneously sent its parameters to the Mission Planner software to locate its position on a map. In fact, we set its starting position as the Home Location for Mission Planner and the last known location before the connection loss was taken as reference point on the map to measure the distance. During all tests, the distance matching the first triggering event between the ping and the streaming of MAVLink messages (i.e. the communication with Mission Planner) was taken in order to record the distance. On next page is shown the environment used for testing, along with the average range radius, based on 15 tests.

FIGURE 5.16: Wi-Fi range tests. H is where the router was placed. 1 is averaged upper limit for stable communication (70m). 2 is averaged upper limit for loss of connection (120m)

5.6 Gas Measurements

We performed several tests for defining an overall qualitative behaviour of both gas detectors mounted on board. Here are shown the most significant. For all tests we used both the analog MiCS 5524 and the digital three-channels MiCS 6814. Target gases for such sensors are:

TABLE 5.1: Target Gases

Gas	**MiCS 5524**	**MiCS 6814**
Carbon Monoxide	1-1000 ppm	1-1000 ppm
Ethanol	10-500 ppm	10-500 ppm
Hydrogen	1-1000 ppm	1-1000 ppm
Ammonia	1-500 ppm	1-500 ppm
Methane	> 1000 ppm	> 1000 ppm
Propane	> 1000 ppm	> 1000 ppm
Iso-Butane	> 1000 ppm	> 1000 ppm
Nitrogen Dioxide	not detectable	0.05-10 ppm

We considered two scenarios during our tests: flying drone on a controlled fire and flying drone on a gas leakage. For both scenarios the tests were performed on a controlled environment and by manual guiding the UAV at roughly one meter of altitude from the source, being it fire or gas tank. For the controlled fire a mixture of straw, wood and paper was used as fuel in order to produce a sufficient amount of carbon monoxide, the fire was triggered on a small traditional grill tower outdoor in presence of little to no wind and the UAV was manually guided to repeatedly pass over the fire swiping a length of 7 metres with the fire in its middle point. As fire was enclosed in the grill tower, the hot spot was enclosed in a 45 cm diameter circle. Shown on next page are the setup for the controlled fire site and the produced results.

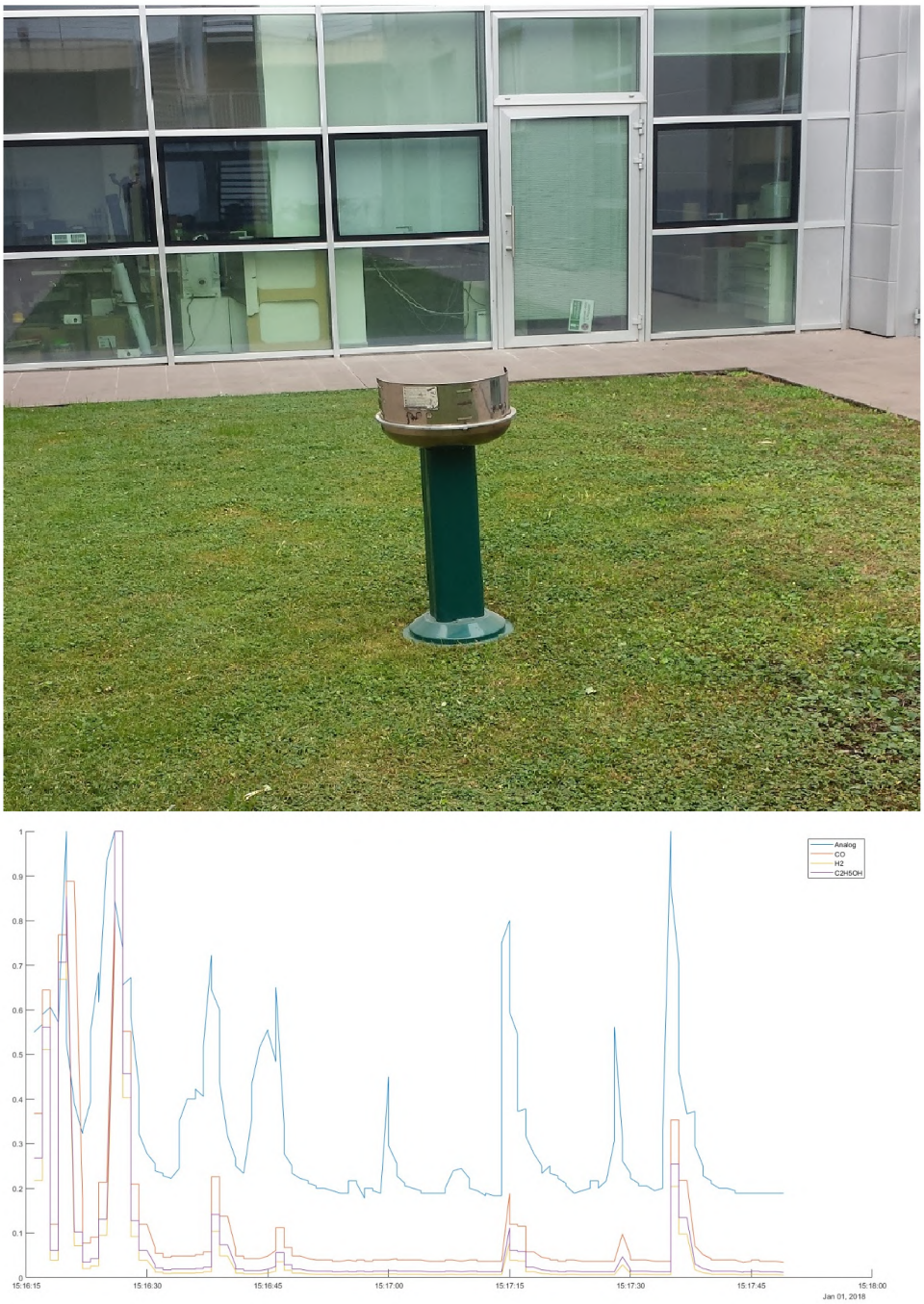


FIGURE 5.17: Controlled fire environment (Ethanol used as fuel). Analog sensor, Carbon Monoxide [I2C], Hydrogen [I2C], Ethanol [I2C]

For the gas leakage scenario, a small gas tank similar to the ones used on camping activities was used. Gas contained in such tank were mainly Propane, with other gases contained in minor quantities and tests on two setup were performed. Firstly tests with the gas tank placed straight on the ground and its gas-thrower pointing up, with the UAV thus flying at an average height of one meter from the ground and secondly tests were repeated with the tank placed at an height of roughly one meter above the ground. This was done to inspect for dramatic changes when the flux of air produced by the UAV and containing the detectable gases ricochets off

some obstacles, e.g. in such case the ground. Setup and produced results are shown below.

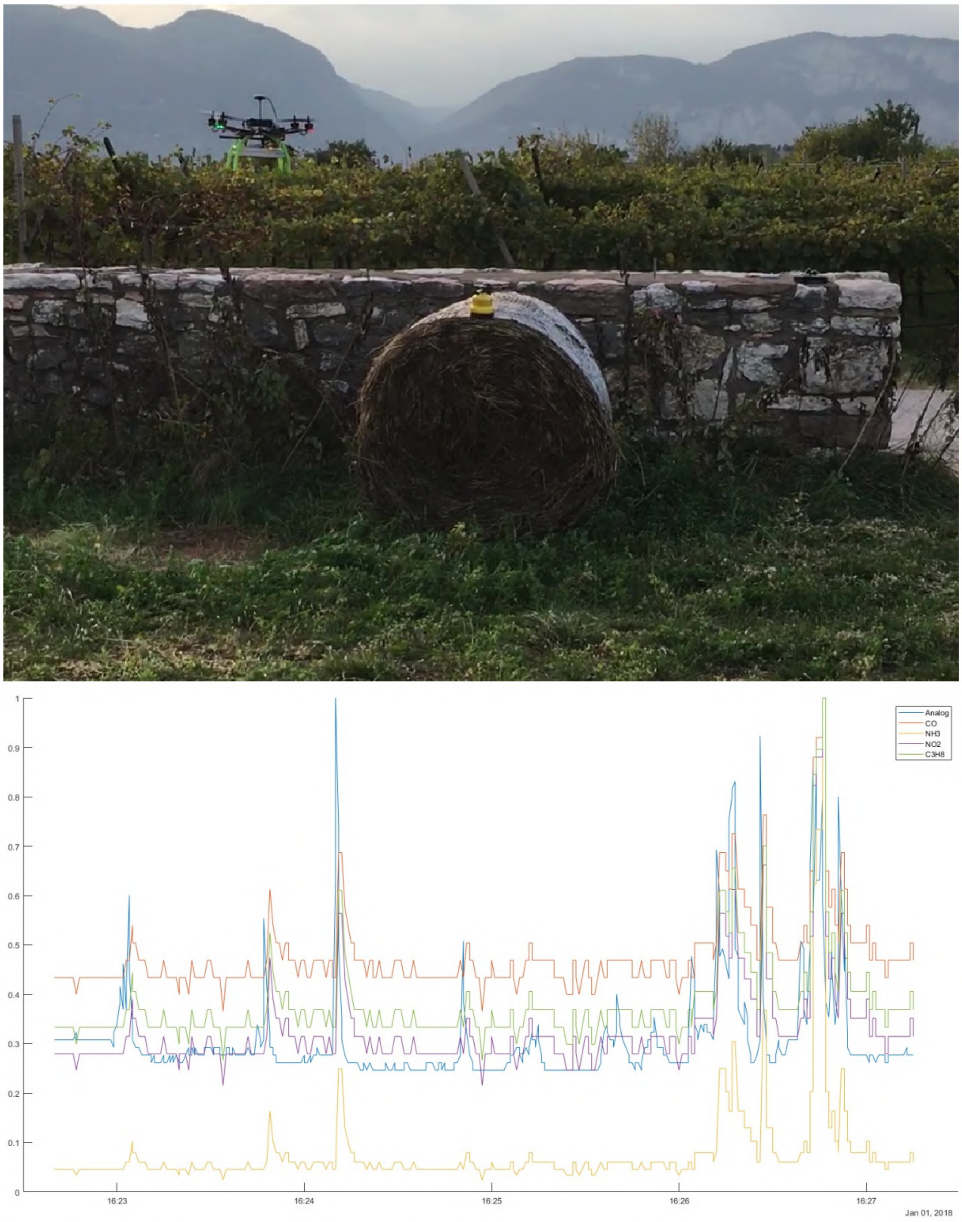


FIGURE 5.18: Detecting gas leakage from a camping gas tank (mainly Propane). Analog sensor, Carbon Monoxide [I2C], Ammonia [I2C], Nitrogen Dioxide [I2C], Propane [I2C]

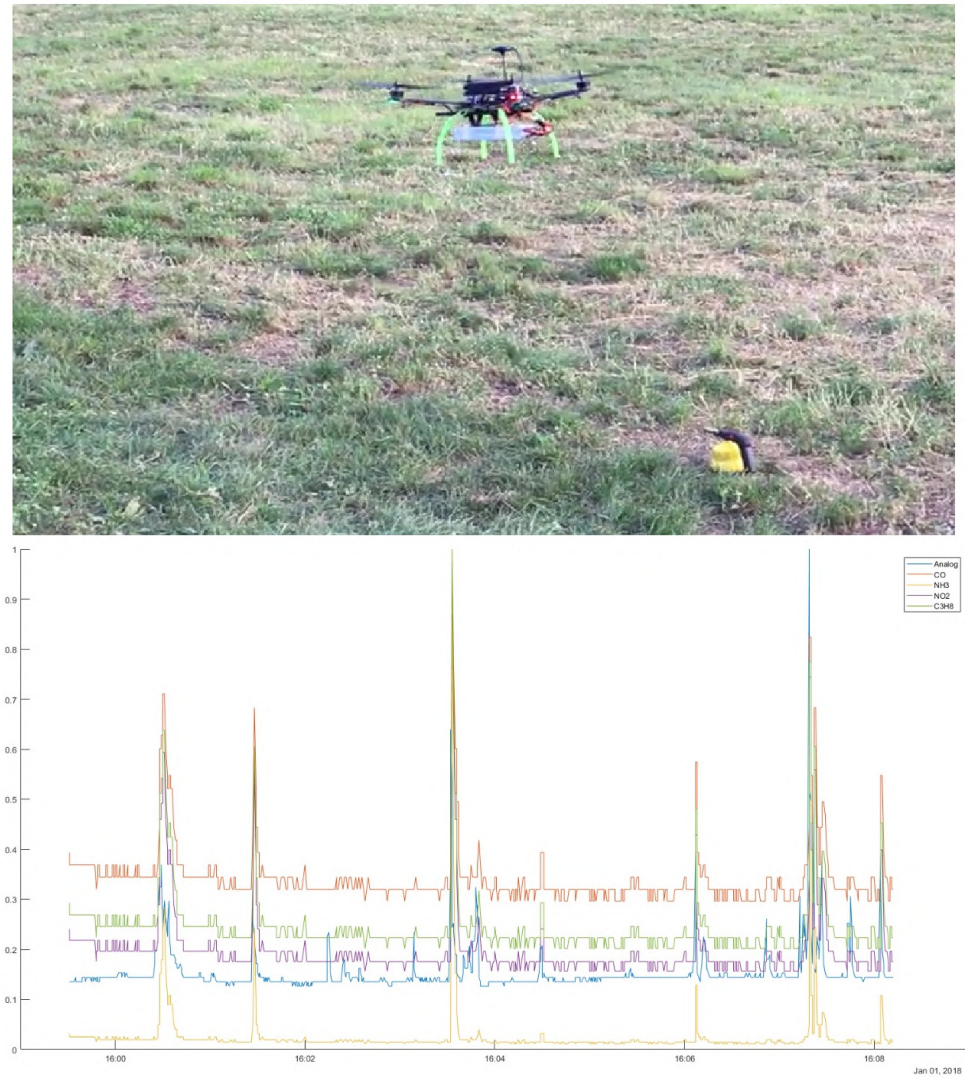


FIGURE 5.19: Replicating gas leakage near ground surface (mainly Propane). Analog sensor, Carbon Monoxide [I2C], Ammonia [I2C], Nitrogen Dioxide [I2C], Propane [I2C]. Gases are still detectable and trend is slightly modified by added turbulences

5.7 Few words on calibration

Throughout this thesis the main focus was on depicting qualitative behaviours of detected gases more than adopting a quantitative approach. This is due to the fact that almost all commercial gas sensors are used to detect the presence of target gases within upper and lower thresholds more than to exactly measure gas concentrations. Without professional gas cromatograph or laser spectroscopy devices, in fact, going for an exact ppm gas concentration measure is pointless, as cross sensitivity is an issue as well as temperature and humidity interferences. Furthermore, calibration operations for gas sensors are quite complex and demanding and our MOX based sensors made no exception.
As an example, quoting the Adafruit MiCS 5524 datasheet, the procedure for calibration requires "purchasing a calibration gas mixture with a known concentration of the target gas for each target gases. Such mixtures requires low-pressure aluminium cylinders with a regulator, a suitable tubing and an in-line flowmeter precisely set to

0.5 litre per minute. Finally a flowhood to place over the sensor allowing gas to pass across but not directly to the sensor is needed; such tool is not available on the market for all gases and for many of them has to be developed by the user. Moreover, for some gases such as Ozone which are so reactive they cannot be made and stored in cylinder, the gas has to be made in-situ using an electrical generator and even like this the concentration cannot be accurately selected by the generator, requiring at least an external analyser. Finally, to produce an adequate calibration result, the entire procedure has to be repeated at various points and produce a response curve for each gas." For these reasons, all the gas detectors we mounted were not calibrated, as we were more interested on the general trend for each device in presence of a target gas more than its actual value.

5.8 Choosing the sensor position

Choosing where to place each gas detector may be a crucial point when looking for sensitivity of the system, in fact turbulences and airflow induced by the propellers during flight can interfere even severely with detection and measuring operations. A recent study of Eu and Yap, 2018 proposed a CFD 3D modelling simulation to better inspect these phenomena. In such study the authors detect three regions in the induced airflow: air intake (upper region when air is sucked into the propellers), exhaust (where air flux is directed to generate lift) and airflow eddies (where air circulates back to the propellers, exists only during altitude changes). Studying these regions, they suggest mounting gas detectors near or straight under each propeller to be able to detect all conveyed air flux from the propeller, thus claiming higher sensitivity (up to twice w.r.t. mounting elsewhere).
Although agreeing with what proposed in that study, two remarks has to be made: sensor technology and sensing strategy. First of all, gas sensors we used are MOX based and required constant heating to achieve the optimal sensing temperature, that is because semiconductor based gas detectors do not work well at room temperature. By placing each sensor straight below a propeller we may incur in dramatic cooling, thus degrading sensor performance. Secondly, the sensing scenario and sensing strategy have to be evaluated as well. Eu and Yap themselves showed how, even with a perfect positioning of each sensor, false negatives can be significantly present if, e.g., the gas is not present in the air intake but only below the UAV. They suggest that to increase sensitivity position of each sensor has a moderate impact on the overall performance and may be irrelevant if the sensing strategy is not properly involved (e.g. by mapping at different levels the sensing scenario). Thus, the proposed solution by Eu and Yap represents a general recommendation more than a specific answer for every case and each system has to be considered in its entirety when sensitivity is addressed. Considering these reasons, we chose to mount our sensors in a neutral position close to the UAV PDB and, because we do neither soldered nor permanently fixed each sensor to a specific position of the frame, giving the chance to freely position each sensor depending on the sensing scenario.

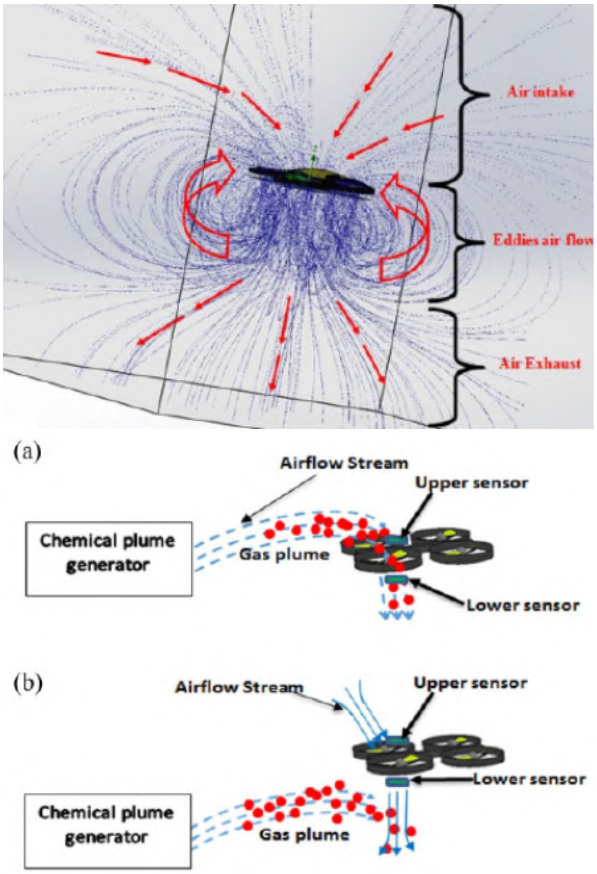


FIGURE 5.20: On top: CFD results of induced airflow of quadcopter with air intake, airflow eddies and exhaust. On bottom: even with right positioning of detectors, false negatives still happen. From Eu and Yap, 2018

Chapter 6

Future improvements

6.1 Adding safety checks

During this thesis, we took different solutions to keep a minimum acceptable level of safety. We ensured a radio connection were always present and stable between each UAV and its radio transceiver. This was an emergency channel and was extremely useful as a backup answer when, being for instrumentation glitches or for unwanted behaviours from code bugs, critical failures happened. Of noteworthy relevance was also the assignment to flight modes to the radio transceiver switches; thanks to that we could be able to override not only manually by taking direct control of the drone movements but also making the drone land or RTL with just a single toggle of a switch. An higher level of safety measure was the adoption of the GeoFence, a virtual fence defining the three-dimensional space where the UAV is allowed to fly. This is an autopilot feature of the ArduCopter firmware and no stable connection to the GCS is required as it is loaded as an internal software parameter: as soon as the UAV moves beyond the edges of this virtual confinement, the drone immediately return to the home location (alternatively, it can be set to land or even loiter). This is a further safety answer which, combined with the transceiver override, makes for a robust solution which prevents the UAV to fly away or into forbidden area, but more steps can be made to increase the robustness and safety of this project. First of all, a solid battery management system is strictly required to both manage live the flight autonomy and automatically detect the end of the flight.
A battery management system similar to the 3DR Power Module can be really interesting to live check both current and voltage and simultaneously avoid overvoltage and overcurrent issues. Up to now, no support for battery voltage checks is available for the BeagleBone Blue through ArduCopter, but the community is constantly developing and adding new features to the firmware and this is surely one of great interest. In addition, having the capability of live checking and monitoring both voltage and current from the battery pack allows to compute the average flight time left and helps in choosing even autonomously when to land or return to home location even if the current mission is not over.
Another feature not yet present on the system is obstacle avoidance and it is another high priority one. By adding a LiDAR or sonar to the UAV it is possible to avoid obstacles in line of sight and to dynamically adjust the drone altitude w.r.t. ground obstacles. No rangefinders were mounted on our UAVs at the moment for no direct rangefinder support is yet implemented on ArduCopter for the BeagleBone Blue; these missing features, however, are due to the relatively new board we used (the BBBlue) as a flight controller. Both hobbyists and professionals are knowing this board and its potentials right now and thus ArduPilot developers have just begun the porting operation and are constantly adding new features to its code. Finally, another useful safety measure that can be taken is monitoring critical variables

even straight on board, i.e. on the very BeagleBone. Adding a bit of intelligence to the UAV would let it detect when would be near the end of its flight autonomy or if some glitches or critical events occurred that required a land. This leads, more specifically, to the next point.

6.2 Move toward a distributed architecture

As being said, moving some intelligence from the central core, i.e. the ground control station, to each UAV can be useful to quickly react to critical situations and to increase safety. A drone losing its connection to the GCS would otherwise rely on pre-defined confinement measures like GeoFencing or manual override, but if it was capable of elaborating its measurements, both gas ones and internal ones such as temperatures and state of its internal parameters (GPS glitches, Extended Kalman Filter failures, bad compass variances) it could autonomously choose the action to take in relation to the gravity of the situation.
In addition to this, moving (at least some of) the intelligence on each UAV from ground to each board would incur in more robustness and reliability, for these reasons a distributed architecture represents the natural evolution of this project. To achieve this, it is necessary to get DroneKit on each UAV in order to access internal flight controller parameters and send it MAVLink commands. Following this architecture type, the GCS will have less and less tasks, ending in displaying on screen data as a HMI, keep overriding capability and user input commands for safety reasons, while all the critical decisions will be made on boards, sharing collected info among all boards and thus achieving a system more robust to critical failures.

6.3 Increasing communication range

Achieving and optimizing a long range transmission was not among the goals of this thesis, thus as our communication channel we used Wi-Fi, which for its specifications extends itself up to a relatively short range, usually falling in a couple of hundreds of meter at most, with the majority of Wi-Fi devices communicating within a hundred of meters outdoor and with few to no obstacles. As our purpose was to project and assemble a first solution for gas detecting flying agents we did not inspect further communication range upgrades, but here we suggest two possible solutions: establishing a network of access points or installing telemetry radio antennas. The former keeps Wi-Fi as a communication channel, extending its range through a network of access points which replicate the network over a wider area. There are several hardware solutions which allow for at least three to four hundred meters in radius of outdoor communication capability, clearly depending on the number and entities of obstacles and the location of each network node.
Even if this solution can be sufficient and enough to compensate for the low communication range, it may be hard or even impossible to deploy several network nodes and power them over a wide area, thus, depending on the morphology of the area where this project has to be used, a more compact and general solution has to be taken: switching from a short-range communication protocol to a longer one. Following this approach, a telemetry radio antenna can be the solution. Installing a long range antenna like the 3DR 433 MHz 915MHz telemetry radio allows for a communication range of few kilometres.

6.4 Define a sensing strategy

Up to now, UAV paths are given by collections of waypoints generated by hand from the user, even if with the help of a mapping tool such as the Flight Data tool of Mission Planner. Although this works for simple scenarios, things get more complicated in case of complex and more general ones. This may lead to the need of generating automatically the UAV path giving just the edges of the area. An interesting possible feature can be, in fact, to automatically generate a collection of waypoints from an area enclosed in given edges (e.g. GPS coordinates) and exclude some critical points were obstacles were a priori known to be. Clearly, this may involve optimisation strategy to find a good if not the best solution which minimise travelled space or flight time depending also on the number of drones belonging to the swarm. A possible starting point is to divide the area into subsections of well-known shapes, e.g. rectangles or trapezoids, and then cover this areas in a "serpentine-like" path spaced by a span computed on the number of drones belonging to the same "search spanner" (e.g. supposing a drone can sense up to 3 meters away from its centre, 6 meters in total for left and right, with a spanner of 3 drones each serpentine segment will be spaced by at least 18 meters).

6.5 Increasing measurement reliability

This thesis focused on building the architecture for a swarm of autonomously guided UAVs, not going deeper on what kind of sensors can be optimal for detecting at best gas leakages. Other than mounting an array of gas sensors, even with different sensing principle (MOX, NDIR and so on), the information entity can be increased by data fusion. An interesting approach will be to fuse measurements coming from gas sensors (and their variances) to information extracted through temperature or pressure readings, achieving a more precise result. In addition, by monitoring air flow it will be possible to compensate the drift in those measurements induced by wind and get a closer and more accurate map of gas leakages.

6.6 Add a GUI

Considering all the noteworthy features listed above and the different gas readings available, to have a compact and immediate view of all parameters and states of the system a Graphical User Interface (GUI) can be of great interest. A GUI realized in C sharp or QT can condensate in a more intuitive and readable way all the useful data as long as embedding an HTML viewer to periodically show marker and heat maps generated by the system along with each UAV location on a map as if it was through a graphical GCS. A good first draft solution could be a window with different tabs, e.g. one for showing all the gas reading live as well as the gas map and one for displaying all the UAVs over a map and check their parameters as well as sending navigation commands through button-events.

Chapter 7

Conclusions

In this thesis an high level architecture for UAV control and autonomous guide in gas leakage detection applications has been developed.
First we illustrated our references and potential commercial solutions for different UAV applications, moving from fixed wing aircraft to multicopters and showing why they are preferable for gas sensing systems.
Then we presented the state of the art for gas sensors, illustrating the main sensor types: MOX, electrochemical and electrooptical. We presented a sneak peek on other gas sensor technologies and showed why MOX based solutions are more feasible, finally illustrating our choices: the MiCS 5524 single-channel analog sensor and the MiCS 6814 I2C three-channel digital gas sensor. Following this discussion we presented our choices for gas sensors.
Then we gave an introduction to multirotor vehicles, discussing their frames, electronic speed controllers, motors, propellers and battery pack, ending the discussion to the very flight controller, landing to the main solutions for a nowadays hardware for flight controllers. We discussed in details our choice for the flight controller, both for hardware (the BeagleBone Blue) and for firmware (the ArduCopter flight stack).
Later on we presented in details our UAV setup, illustrating and discussing its sections: the power, the engine, the data acquisition, the communication and its final cabling and packaging.
We continued with a big chunk of this thesis: the software. We presented DroneLib, DroneSwarm and DroneSense, three main sections the entire generated code was grouped into. We discussed all the high level functions used for guiding and controlling our vehicles along with all the algorithms used to acquire, transmit and manage the data, all through easy-to-catch flowcharts for an instant readability. We mentioned the Software-In-The-Loop simulator we used (SITL) and provided a quick introduction to both MAVProxy and DroneKit, respectively the Python 2.7 package and APIs we used to build our architecture.
We then moved to discussing results from all tests we made: gas measurements, communication range, flight time and CPU/memory usage for our flight control board. Those results suggested the high feasibility and goodness of the realised system, providing cues for improvements. We mentioned both the calibration and the sensor positioning crucial issues, convalidating our choice of going towards a more qualitative description of sensed gases and we moved to a section dedicated entirely to show possible room for improvements, proving again the flexibility of this work.
This thesis wanted to be a tool and a starting point for an high mobility gas sensing system, thus we tried to realize an architecture as accessible as possible by assembling our UAVs with commercial easy-to-find and almost ready-to-fly solutions and using open-source software. Throughout all the thesis hyperlinks are placed to instant taking the digital reader to our references and useful links, from citations to

datasheets. In the following appendix A we gave all the information on how to reproduce this work, reporting both the code documentation and all the useful links to guide through all the procedure.
We focused our discussion on the very Software to achieve a multi-UAV structure, presenting and discussing the main algorithms with brief introductions on all the relevant Python modules adopted along with immediate flowcharts to instant read through all the code.
We then proposed some future improvements like going toward a more distributed structure or to use data fusion to enhance data information.
To conclude, in our opinion this architecture succeeded to create an open-source flexible system not only able to harvest and record gas data from different mobile stations, but also to manage and control one or multiple UAVs simultaneously without neglecting the basilar safety measures such as manual override and GeoFencing capability.

May this work be helpful for all who want to replicate or deviate
from the purpose of this thesis but build a system based on
such powerful and promising tools as UAV are.

Luca Gemma

Appendix A

Help in replicating this work

A.1 Getting ArduCopter and Robotics Cape on the BeagleBone Blue

Up to now, the Robotics Cape Installer software changed its name to Robot Control Library, but it keeps all the features described in this thesis. To get the Cape from a fresh new board, the first step was to update the BeagleBone Image; two options are available: booting an image from an SD card or flashing the built-in eMMC memory of the BBBlue. We went for the latter, to achieve better performance and updating the bootloader (this last step was not necessary, but it is required if the board comes with an outdated one). During this procedure, refer to this link for detailed info. Here we briefly show how to do it: Download the BeagleBone Blue Flasher Image, extract it and write it onto a microSD card of at least 4 GB using Etcher. Then, with the BBBlue powered off, insert the SD card. Now apply power (either through USB or the DC barrel jack). After few seconds, you should notice the USR LEDs blinking in a regular pattern, back and forth in a Knightrider pattern. The operation may take several minutes. When the flashing operation is done, all 4 LEDs should stay on for a minute before turning off. Pay attention that if the board powers on with the SD card inserted but the LEDs flashing pattern does not start, you may have an outdated bootloader. In this case, force the booting from SD by powering off the board and then applying power while holding the SD button. Once the eMMC flashing is complete, power off the BBBlue and remove the SD. With an updated image for the BBBlue, it was then time to access this appealing device. Fortunately, no need for external hardware, such as a network router or a display, were needed, as the board includes both a USB and a WiFi network gadget, allowing to communicate straight with TCP or SSH. We used PuTTY to SSH into it, as it is a light and reliable SSH client which allows even to easily log the output produced by the connection. Thus, you can connect to the board either via USB or via its WiFi access point. For the former, the static IP address is 192.168.7.2 (and the one of the device connected to it is set to 192.168.7.1), while for the latter one you have to first connect to the BeagleBone-XXXX access point (usually the password is BeagleBone) and then SSH to the static IP address 192.168.8.1 (and usually the one of your device is set to 192.168.8.72). Check for your username and password, but commonly they are respectively “debian” and “temppwd”. Alternatively, the BeagleBone Blue comes with the Cloud9 IDE installed, an open-source web based programming platform that supports several programming languages and features code completion, drag-and-drop functionality, SSH, FTP and many other. To access to this powerful IDE just connect either through USB or through its WiFi and type on a web browser address bar http://192.168.7.2:3000 for USB connection or http://192.168.8.1:3000 for wireless connection.

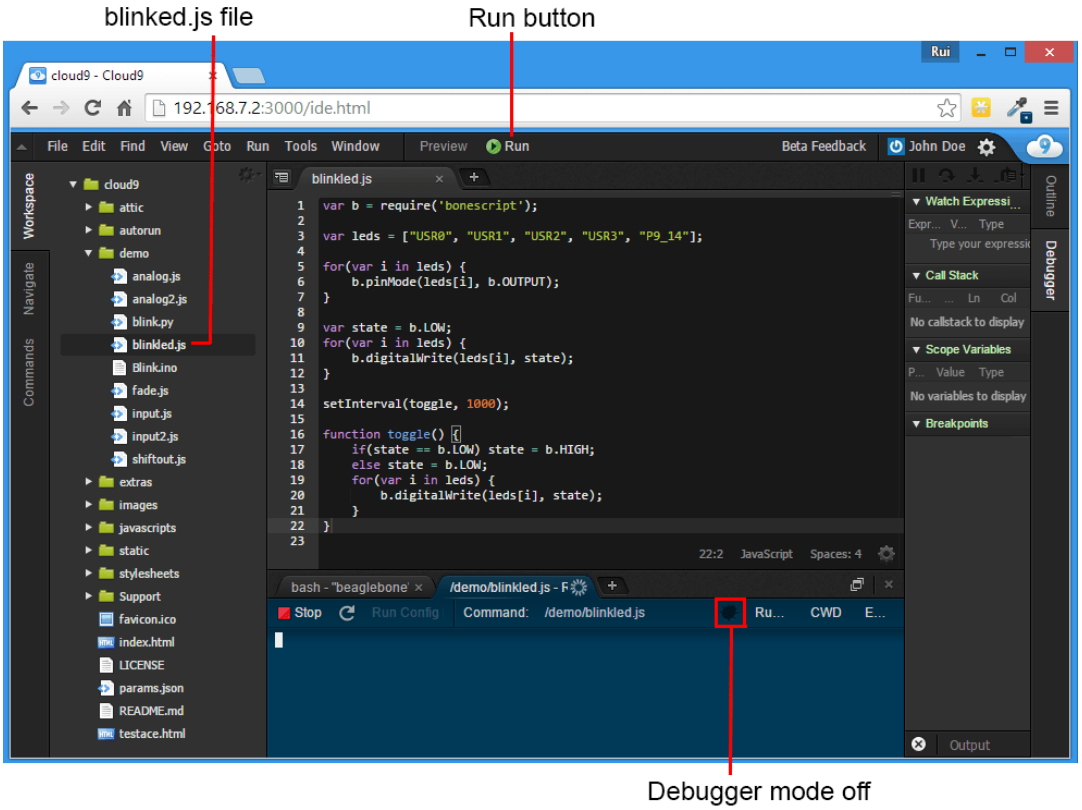


FIGURE A.1: The Cloud9 IDE hosted on the BeagleBone Blue. Courtesy of BeagleBoard.org

Even if in our project we did not make extensive use of cloud9, this great IDE is one of the key features that makes this board perfect for programming, including Node.js, a JavaScript based tool for easy coding of fast and scalable network application. Once you have access to your BBBlue, the next step is to connect the board to a network with Internet connection, and here is how to do it. Once you have SSHed into the board (PuTTY or Cloud9), type:

- connmanctl
- enable wifi
- tether wifi disable
- scan wifi
- services (this should show you the available network with their names. copy the name, including wifi_" of the network you want the board to connect to)
- agent on
- connect [paste here the name you have copied]
- if any, enter the password
- quit

You can then ping google.it or another website to check if the connection to the network has been established successfully. for more info, land to this video Finally, you can get the Robotics Cape Installer by creating a new directory and then

```
git clone$ $https://github.com/StrawsonDesign/Robotics_Cape_Installer.git
```

In the examples folder you can find useful easy readable code examples for testing several peripherals such as the ADC (*rc_test_adc.c*), buttons (*rc_test_buttons.c*) and LEDs (*rc_test_leds.c*) or even calibrate the IMU and access the barometer. Next step we took was to prepare the BeagleBone for ArduPilot, and main steps to achive it are (refer to the following webpage for detailed info: https://github.com/mirkix/ardupilotblue).

- Update the software
- Install the software
- Set the clock frequency to 1GHz
- Update scripts
- Install the RT Kernel 4.4
- Reboot the system
- Download the ready compiled ArduPilot file from http://bbbmini.org/download/blue/
- Copy file via SCP or microSD on your BeagleBone Blue

Once you have completed all these steps, ArduCopter should be ready on the board and, depending on the peripherals you want to use, you can run it with several different start parameters by typing

```
sudo [path to your arducopter hex file]/arducopter-X\_Y\_Z [start parameter] [address]
```

and the start parameter are, for Telemetry switch:

- -A for serial 0 (usually the console, default baud rate of 115200)
- -C for serial 1 (telemetry 1, default baud rate of 57600)
- -D for serial 2 (telemetry 2, default baud rate of 57600)

and for GPS switch:

- -B for serial 3 (first GPS option, default baud rate of 38400)
- -E for serial 4 (second GPS option, default baud rate of 38400)

Thus, taking the most common case (which was also our case), if you want to run ArduCopter connecting to a MAVLink groundstation with IP 192.168.8.72 (the one of your PC when connected to the BBBlue access point) and port 14560 using an UDP connection and the BBBlue have a GPS connected to its GPS port (which is its /dev/ttyS2), simply type and enter:

```
sudo [path to your arducopter hex file]/arducopter-X\_Y\_Z -C udp:192.168.8.72:14560 -B /dev/ttyS2
```

you can type the "–help" option to inspect available start options. In addition, to help choosing which address you should give for the start parameter here is a sneak peek at the first page of the BBBlue schematic, where almost all the useful information about ports are, as well as the board pinout table. You can find the

complete files respectively here for schematics and here for the pinout table. foto di pagina 1 dello schematic BBBlue foto di pinout table BBBlue With ArduCopter now running, the BeagleBone Blue is effectively a flight controller and you can manage its parameter, flight modes and counterparts with any MAVLink based software, the easiest one is a regular Ground Control Station such as Mission Planner. You can find the latest Mission planner installer from `http://firmware.ardupilot.org/Tools/MissionPlanner/MissionPlanner-latest.msi` (pay attention that it runs on Windows, on Ubuntu and Mac OS the process to get it is quite involving). Once Mission Planner is running, simply check for the right transmission protocol (UDP or TCP) and baudrate and hit the Connect button. We highly recommend Mission Planner as a visual GCS as it is probably the best groundstation available, embedding everything a UAV owner needs to get it fly fast and efficiently, from guided calibration procedures to full customization of all the internal parameters and attributes of the flight controller. foto di Mission Planner con evidenza delle maggior aree di interesse Before ending this section, we took one last noteworthy action: configuring ArduCopter to run as soon as the BeagleBone Blue starts. This may be particularly useful in case of an automatic solution which requires the least human intervention possible, as this thesis purpose. There are two ways to achieve it: through the rc.local file or by creating a service. Below are reported both of them, but we choose to go with the rc.local file as it was trivial and fast.

A.2 logging Python package for Logging facility

Of great interest for a clean, clear and modular debugging is with no doubts the logging Python package. Thanks to this module it is possible to set a flexible event logging system which can include messages from third-party modules as well. Three main objects can be instantiated to set such a system: loggers, handlers and formatters. Loggers create the logging interface used by the code. Loggers are create by the getLogger(name) function with the specified name. Handlers manage the log output to the desired destination. Handlers are created by the FileHandler(f) function for the specified file f and are added to a logger with the addHanlder(handler_name) function. Formatters modify the layout of log outputs. Formatters are created by the Formatter(pattern) function, following the specified pattern, as an example:

```
("[%(asctime)s][%(levelname)s]: %(message)s")
```

This pattern tells the formatter to print [date/time] [logging level]: logging message. Date and time are displayed together in the ISO8601 format and the logging level represents "the severity of the events they are used to track". Logging messages all have a logging level, ranging from DEBUG to CRITICAL (ordered from least to most severe: DEBUG, INFO, WARNING, ERROR, CRITICAL) defined by the function called to print them as following:

Logging level can be set for Handlers and Loggers separately and is set through the setLevel(Logging Level) function. Logging messages are printed on output (being it the terminal or a file specified for an Handler) only if their respective logging level is of a severity level equal or greater than the one set (for the logger or the FileHandler respectively).

TABLE A.1: Printing functions for different logging levels

Logging Level	Print Function
logging.DEBUG	logging.debug()
logging.INFO	logging.info()
logging.WARNING	logging.warning()
logging.ERROR	logging.error()
logging.CRITICAL	logging.critical()

A.3 re Python package for Regular Expressions

Regular expressions are frequently used to manipulate strings and extract information out of certain patterns. They are widely used throughout all the code in the form of input argument parsing and managed internally with little to no action required from the user thanks to the argparse Python package. Here we briefly show how to use such package along with a general introduction to regular expressions. ArgParse Python package The ArgParse module allows for straight and easy command-line parsing out of sys.argv, thus increasing the flexibility of the code by catching and parsing input arguments. First step to achieve it is to create a parser through the ArgumentParser() function. Secondly, by using *add_argument*(), you can specify which command-line options the program is willing to accept. Finally, the *parse_args*() has to be invoked to effectively parse all the added arguments. Here is a code snippet to better show the use of the module.

```
{parser = argparse.ArgumentParser(description='Ground Control Station')
parser.add_argument('--connect1',
help="vehicle 1 connection target string. e.g:--connect1 192.168.1.15:14550.")
parser.add_argument('--sport',
help="vehicle 1 streaming port. e.g:--sport 46101.")
args = parser.parse_args()
connection_string_1 = args.connect1
sport = args.sport}
```

In line 1 the parser is created, adding a useful description, then arguments "connect1" and "sport" are added along with details on their use that can be viewed by typing the "–help" command. From line 4 on, the input arguments are parsed and assigned to variables for further use on the code. Note that to do that, you have to access the attribute named after your added argument, e.g. to access what typed after "–connect1" you have to access the "connect1" attribute. Finaly, assuming the script where this code is placed is called GCS.py, you can execute it at command line with the newly added arguments by using e.g.

```
python GCS.py --connect 192.168.1.15:14550 --sport 46101
```

Check https://docs.python.org/3/library/argparse.html for further details. Following the official documentation of the re Python module for regular expressions (https://docs.python.org/2/library/re.html), "A regular expression (or RE)

specifies a set of strings that matches it. [...] Regular expressions can be concatenated to form new regular expressions; if A and B are both regular expressions, then AB is also a regular expression. In general, if a string p matches A and another string q matches B, the string pq will match AB. This holds unless A or B contain low precedence operations; boundary conditions between A and B; or have numbered group references. Thus, complex expressions can easily be constructed from simpler primitive expressions like the ones described here. [...] Regular expressions can contain both special and ordinary characters. Most ordinary characters, like 'A', 'a', or '0', are the simplest regular expressions; they simply match themselves. You can concatenate ordinary characters, so last matches the string 'last'. [...] Some characters, like '|' or '(', are special. Special characters either stand for classes of ordinary characters, or affect how the regular expressions around them are interpreted. [...] Repetition qualifiers (*, +, ?, m,n, etc) cannot be directly nested. This avoids ambiguity with the non-greedy modifier suffix ?, and with other modifiers in other implementations. To apply a second repetition to an inner repetition, parentheses may be used. For example, the expression (?:a6)* matches any multiple of six 'a' characters."
Shown below are the most frequently used special characters for regular expressions:

- r"\n" : string of two characters containing '\n' and 'n'. 'r' is needed to disable escape characters
- "\n" : string of one character containing a newline
- ur"this is a unicode string" : unicode string
- . : each character except for newline
- [a-z] : each letter from a to z
- a. : match "ax" where x is each character except for newline
- seriali[sz]e : match both "serialize" and "serialise"
- gray | grey : match both "gray" and "grey"
- x? : 0 or 1 occurrences of the preceding element x
- x* : 0 or more occurrences of the preceding element x
- x+ : 1 or more occurrences of the preceding element x
- xn : exactly n occurrences of the preceding element x
- min,max : at least min, at most max occurrences of the preceding element x
- [^x] : each character except for x
- ^: starting position in the string or in the line
- $: end of the string or of the line
- [0-9] : each digit character
- [^0-9] : each non digit character
- [A-Z a-z 0-9] : each alphanumeric character, i.e. from capital A to capital Z, from small a to small z and from 0 to 9
- [[[\t]]] : space and tab characters
- \w : each alphanumeric character including "_"
- \W : each non alphanumeric character, except for "_"
- \s : whitespace character (e.g. tab, line feed, form feed, carriage return, space)
- \S : all except for whitespace characters

- \d : each digit character
- \D : each non digit character
- (?=pattern) : positive lookahead assertion. Match if after the match there is the pattern
- (?<=pattern) : positive lookbehind assertion. Match if before the match there is the pattern

Considering all of what have being discussed, here is a code snippet of one of our uses of the re Python module. This code allows to parse and manage the incoming data of our communication stream catching the gas measurements for propane (C3H8) gas. Our incoming stream is of the type: CH4 : 200.0 12:10:07 C3H8 : 350.2 12:10:08 C2H5OH : 120.0 12:10:08 where all gas measurements are placed into such a string, separated by spaces and announced by its gas type along with its timestamp. All listed values are for illustrative purposes only. Thus, to get the propane measurement we had to be able to catch the value listed after "C3H8 : " and we achieved it by compiling the following regular expression pattern_gas_C3H8 = re.compile((?<=C3H8 :)[-+]? d+. d* d+[:] d+[:] d+) This looks for only characters after the C3H8 : string ((?<=C3H8 :)), with the optional presence of a plus or minus sign ([-+]?) and which matches the expression of a floating number (d+. d*) and a timestamp of the type hours:minutes:seconds (d+[:] d+[:] d+) separated by a space. Note the presence of the backlash escape character to indicate the dot; putting just the dot would have generated a different regular expression. Then, to detect such precompiled pattern into a string and treat it, use the findall(pattern, string) function. Here it is how we managed it:

```
result_gas_C3H8 = re.findall(pattern_gas_C3H8, str(gas_data_stream))
for p in result_gas_C3H8:
split0 = p.split()
gas_value_C3H8 = split0[0]
timestamp_C3H8 = split0[1]
print("C3H8: " + str(gas_value_C3H8) + " at time:  " + str(timestamp_C3H8))
```

In line 1 we find all occurrences of our precompiled pattern into our incoming stream string and then print each one of them along with their timestamps. Note that following our pattern the expression matched includes also the timestamp, thus we separated the measurement from the timestamp by using the split() function. Doing so, it is possible to split a string into multiple ones using as a separator character the one specified as its input; no input means to use the space as a separator, as in our case.

A.4 Automatic startup with rc.local

In the /etc/ folder it is located a file called rc.local where can be put all the instructions the user wants to run when the BeagleBone Blue starts. Doing this is as simple as inserting the instruction in a file with a text editor: sudo nano /etc/rc.local add the instruction used to start ArduCopter with the proper parameters, followed by &, e.g.:

```
/home/debian/arducopter/arducopter-3_5_4 -B /dev/ttyS2 -C udp:192.168.8.72:14560 \&
```

the ampersand & is used to fork processes, thus causing the command it is put after to be forked and run in the background. This prevents the board to be stuck forever in the execution of the ArduCopter and not executing anything else. Pay attention that it may be needed to set a proper state for some multiplexed pins of the BBBlue if you are using them; in our case, as we were using the GPS, we had to enable them by adding the three following lines before the execution of the *arducopter* $-$ *X_Y_Z* file.

```
/bin/echo uart > /sys/devices/platform/ocp/ocp\:P9_21_pinmux/state
/bin/echo uart > /sys/devices/platform/ocp/ocp\:P9_22_pinmux/state
/bin/echo pruecapin_pu > /sys/devices/platform/ocp/ocp:P8_15_pinmux/state
```

We added a sleep time before all these instructions in order to let the BBBlue WiFi run before doing anything else. You can do it by adding before all of your instructions: /bin/sleep [number of seconds you want to wait] Once you are done, just save and exit nano with respectively Ctrl + O and Ctrl + X. At the next startup of your board, ArduCopter will start automatically. If you want to close it, just kill the process by first checking its PID with ps aux | grep arducopter and then sudo kill -9 [PID of the arducopter process]. Here is how it looks our rc.local file: immagine di rc.local file

A.5 Automatic startup with a service

Although running at startup through rc.local gets the job done, it may be useful to have a cleaner way to do it. By creating a service and then make it run at startup you essentially achieve the same goal, but in a more efficient and supported way; in fact, it seems the startup via rc.local is coming to an end in a near future and replaced by systemd tasks. To define a service, in this case ArduCopter.service:
sudo nano /lib/systemd/system/arducopter.service and fill the file with

```
[Unit]
Description=ArduCopter Service
After=bb-wl18xx-wlan0.service
[Service]
ExecStartPre=/bin/echo uart > /sys/devices/platform/ocp/ocp\:P9_21_pinmux/state
ExecStartPre=/bin/echo uart > /sys/devices/platform/ocp/ocp\:P9_22_pinmux/state
ExecStartPre=/bin/echo pruecapin_pu > /sys/devices/platform/ocp/ocp:P8_15_pinmux/state
ExecStart=/usr/bin/ardupilot/blue-arducopter -A udp:192.168.0.27:14550 -B /dev/ttyS2
StandardOutput=null
[Install]
WantedBy=multi-user.target
Alias=arducopter.service
```

Note that it is possible to have complete control over the moment the service starts by specifying it in the After field in the [Unit] section. As in the rc.local case, multiplexed pin 21 and 22 of port 9 and pin 15 of port 8 are set before running arducopter. For further information on services see the examples of .service files

in /lib/system/system of your BBBlue. Now that the service file has been created, enable it by:

```
cd /lib/systemd/system
sudo systemctl enable arducopter.service
and start the service with:
sudo systemctl start arducopter.service
```

arducopter is now saved as a startup service and runs every time the system boots. You can stop it with:

```
sudo systemctl stop arducopter.service
and disable it by:
sudo systemctl disable arducopter.service
```

A.6 Installing DroneKit

To take the system to the next level, i.e. controlling the UAV with a script, a tool able to manage MAVLink messages, both receiving and sending them, is needed. This tool in our project was DroneKit, a collection of Python 2.7 APIs to connect and communicate with devices supporting MAVLink protocol based communications although ArduPilot based UAV platforms are recommended as they are fully supported. DroneKit is available for Linux, Mac OS X and Windows and it is easily installed from pip on all platforms. As a GCS we used a laptop with Windows 10 installed, getting the job done with the WinPython 2.7 64bit Python distribution, as it is the most tested version w.r.t. DroneKit. For installation on other OSs, refer to this link (http://python.dronekit.io/develop/installation.html). To get WinPython 2.7 64bit, install the Microsoft Visual C++ 2008 Redistributable Package x64 for WinPython 2.7 and run the WinPython 2.7 installer extracting the folder in the desired destination folder. Once done, you can run Python 2.7 scripts using the WinPython Command Prompt executable file. Note that you may need to upgrade or even install new packages depending on your needs. To add a new package, first ensure the package is not already installed on the system: dpkg –l | grep [name of the package] If the package is installed already, ensure it is the version you need. If the package is an older version, update it using the Advanced Packaging Tool (apt) with: sudo apt-get upgrade or, if the package is not already installed on the system: sudo apt-get install This command also automatically install all any dependencies for the related package. Note that you may also update all packages on the system (or on your BeagleBone Blue) by simply running sudo apt-get update sudo apt-get upgrade

A.7 A short introduction to Linux commands

As in this thesis we found ourselves heavy turning to Linux commands, both on the BBBlue and on our Python environment, it may be worth doing a quick review of the most useful Linux commands.

A.7.1 Advanced Packaging Tool – APT

- dpkg –list : show all installed packages
- dpkg –status <package> : check if the <package> is installed and list its details
- sudo apt-get update : update the list of package versions
- sudo apt-get upgrade : update all packages to their latest version
- sudo apt-get install <package> : install the specified package
- sudo apt-get update && sudo apt-get install <package> : install and update the specified package
- sudo apt-get remove <package> : remove the specified package, but leave config files and logs
- sudo apt-get purge <package> : remove the specified package, including config files and logs
- sudo apt-get autoremove : seek and remove all unnecessary packages and files automatically installed to satisfy dependencies for some packages and that are no more needed
- sudo apt-get autoclean : clears out the local repository of retrieved package files, only removing packages that can no longer be downloaded and are largely useless
- sudo apt-get clean : clears out the local repository of retrieved package files entirely

Note: as good habit, you should only use the apt-get remove <package> command for packages you have installed. To remove other packages, use the apt-get autoremove, autoclean or clean commands.

A.7.2 General Linux commands

- sudo <command> : use superuser privileges to execute the specified command
- shutdown –h now : securely shut down the system
- reboot : reboot the system
- cd /desired/path : change directory to the specified path
- pwd : show the path to the current working directory
- ls : list the current directory content. Try the option –l to get more details.
- chmod [OPTIONS] <permissions> <file name> : change permissions for specified file, such as executability, readability and writability
- chown [OPTIONS] <user> <file name> : change ownership for specified file
- cp <file> /desired/path : copy the file into the desired path. Try –r to recursively copy a folder and all files contained.
- mv <file> /desired/path : cut and paste the file into the desired path. Try –r to recursively copy a folder and all files contained. It can also rename files.
- rm <file> : remove the file. Try –r to recursively remove a folder and all files contained.
- mkdir <directory path> : create a directory
- rmdir <directory path> : remove a directory

- cat <file 1> <file 2> ... <file n> : dump file contents, sequentially reading one file after the other and printing them out
- less <file> : progressively dump files
- vi <file> : open the file in a complete text editor
- nano <file> : open the file in a simple text editor
- head <file> : trim dump to top, printing the first 10 lines of the file
- tail <file> : trim dump to bottom, printing the last 10 lines of the file
- lsusb : list all the USB buses and detected USB devices
- echo : display messages on screen
- export : set environment variables
- env : dump environment variables
- history : dump command history
- grep : search dump for string
- man : get help on command
- apropos : show list of man pages
- find : search for files
- tar : create or extract file archives
- gzip : compress a file
- gunzip : decompress a file
- du : show disk usage
- df : show disk free space
- mount : mount disks
- tee : write dump to file in parallel
- hex dump : readable binary dumps
- ifconfig : list all network interfaces
- scp source_host_username@source_host_address:/source/path/file destination_host_username@destination_host_address:/destination/path : copy files to, from or between different hosts using SSH for data transfer and providing the same authentication and same level of security as SSH
- top : show current running processes and system resource usage
- htop : more detailed and user-friendly tool to list running processes and system resource usage

A.8 Getting Linux on Windows with Bash on Windows

Running Windows but dealing frequently with Linux commands may seem a conflict, requiring solutions such as running a Linux OS on a virtual machine. Fortunately, for easy tasks, Bash on Windows comes in handy, being able to run Linux commands inside Windows with no further need for external solutions. Bash on Windows provide a Windows subsystem in which Ubuntu Linux runs atop of, allowing you to run the same Bash shell you use on Linux. Here we briefly show how to get it, for more info check this link. Go to Start menu, search PowerShell and run it

as administrator. Then, activate the Windows-Subsystem-Linux Windows Optional Feature and reboot your system. Once rebooted, land to the Windows Store and search for Linux. Choose the distribution (Ubuntu is recommended) and install it. Once done, search for the Linux distro you installed and run it like a normal application. After some time the installation is done and you can setup your username and password. A final note: to get to your regular computer folders, type from within the bash shell the following:

```
cd /mnt/c/Users/Your_user_name
```

A.9 Creating batch files in Windows

Batch files are text files which contain a sequence of commands meant to be run by the command-line interpreter. Such files may include any command understandable by the interpreter and even conditional branching and looping by using IF, FOR or GOTO statements. Batch files can be of great help by introducing automation of certain common regular tasks the user would otherwise perform manually. Batch files we used were extremely simple and no more than a collection of script run one after the other, at most including a delay on the execution. Here we introduce only the commands we used in our batches, for more info on other commands check https://en.wikibooks.org/wiki/Windows_Batch_Scripting.

- @echo off – for batch reloading. the interpreter reload the content of a batch after each execution of a line.
- REM this batch file run the test1.py Python script located in the Downloads folder – adding REM before a line ensures the interpreter skip entirely such line, thus it is a way to add useful comments to the batch file.
- echo Hello world – print "Hello world" on screen
- TIMEOUT /T X – wait for X seconds and display them on screen, allowing user to interrupt the waiting by pressing a key. Add the option /NOBREAK to ignore user input other than Ctrl + C. Add the >NUL option to suppress the output.
- start "" python C:\Users\Default\Downloads\test1.py – run with python the script test1.py located at the specified path. If Python is not in your PATH or default path, you may have to write it as
- start "" ""%WINPYDIR%\python.exe"" %userprofile%\Downloads\test1.py – by writing the environment variable %WINPYDIR% the interpreter looks for a Python installation within the path indicated by the %WINPYDIR% variable. To change this path, simply write
- set WINPYDIR=C:\path\to\your\Python\installation

before invoking it. Note that the %userprofile% environment variable was written in order to get the user's home directory location, thus being somewhat independent from this path, unifying the procedure for different users. Here is what one of our batch files look like; the file simply run the mavproxy Python script to share an input MAVLink based connection into three connections over three different UDP ports.

```
@echo off
REM Run MAVProxy to split an UPD incoming connection over three UDP ports
echo Connecting to drone and sharing output into 3 ports
TIMEOUT /T 1 /NOBREAK
start "" ""%WINPYDIR%\python.exe""
%userprofile%\Downloads\WinPython-64bit-2.7.13.1Zero\python-2.7.13.amd64
\Lib\site-packages\MAVProxy\mavproxy.py
--master udp:192.168.0.2:14560 --out udp:127.0.0.1:14551
--out udp:127.0.0.1:14553 --out udp:127.0.0.1:14552
echo now run first_flight_and_graph_part_two.bat
```

A.10 The SITL tool

After having set up our BBBlue board and built confidence with Linux commands, we took our next step by going deeper with DroneKit and its SITL simulation tool. As previously mentioned, SITL can simulate any kind of mobile robots supported in the ArduPilot software, from copters to rovers. Here is how to install and run it.

A.10.1 Installing and Running SITL

DroneKit-SITL is available for Windows, Linux (x86 only) and Mac OS X. Be aware that is still in experimental stage and up to now it is tested on Windows 10, Ubuntu Linux and Mac OS X "El Capitan", no ARM builds (e.g. Raspberry) are supported and binaries are available for x86 architectures only. The installation is easy and straightforward through Pytonn's pip tool: pip install dronekit-sitl –UI This downloads and pre-built vehicle binaries appropriate for the host operating system. To run a simulation through SITL, first check the available vehicles by running

```
dronekit-sitl --list
```

and then simply run

```
dronekit-sitl [vehicle_type]
```

e.g.

```
dronekit-sitl copter
```

for multirotor UAVs The SITL will directs its output through localhost (127.0.0.1) and TCP port 5760. Usually you may want to place your vehicle at a specified starting location, i.e. setting the Home Location to some GPS coordinates. You can achieve this by adding

```
--home=[LATITUDE],[LONGITUDE],[RELATIVE ALTITUDE],[HEADING]
```

Finally, to simulate multiple vehicles of the same kind, add the option

```
--instance X
```

this way, X instances of a simulated vehicles will be created, connecting at 127.0.0.1 (localhost) and at port number starting from 5770 and spaced by 10 (e.g. the ones from first vehicle starts at 5770, second vehicle starts at 5780 and so on).

A.10.2 Connect to a Ground Control Station

It is useful to be able to connect the vehicle to a GCS in addition to having its MAVLink message stream managed in a DroneKit script. The easiest and most reliable way to do so is by sharing the output on additional ports using MAVProxy.

MAVProxy is a complete, light and cross-platform groundstation which can also share an incoming MAVLink connection over multiple communication channels. You can do so by running in the WinPython Command Prompt:

```
%userprofile%\Downloads\WinPython-64bit-2.7.13.1Zero\python-2.7.13.amd64
\Lib\site-packages\MAVProxy\mavproxy.py
--master [PROTOCOL:IP:PORT] --out [PROTOCOL:IP:PORT] --out [PROTOCOL:IP:PORT]
```

where PROTOCOL is UDP or TCP, and IP and PORT are respectively the IP address and the port related to the incoming connection (–master, the IP usually is 127.0.0.1 if you are running a SITL) or the desired output connection (–out). Note that, thanks to the %userprofile% environment variable it is possible to uniform the command for different user paths, providing the path after the variable is the same. With the command listed above, you can split the MAVLink stream of your UAV, being it simulated through SITL or a real one, and connect an arbitrary number of scripts and/or additional GCSs, just ensure there is no port collisions.

A.10.3 Connect multiple UAV to Mission Planner

Once able to run multiple instances of a simulated vehicle and to arbitrarily share their outputs, next step was to manage and display all of their flight states and parameters through an easy and understandable way, such as a graphical GCS. As we chose Mission Planner among all the GCS options available, we kept it also for displaying multiple UAVs and here we show only how to manage multiple UAVs with such a groundstation. Connecting multiple UAVs to Mission Planner is as easy as connecting a single one. Just right click on the black area near the upper buttons and hit “Connection Options”, then select the right communication protocol and baudrate, click the Connect button, then insert the local port for the first UAV and smash “Ok”. Repeat the process for every other UAV you want to connect to Mission Planner. Once done, you should see on the map the icon of every UAV you have connected. Mission Planner treats them as separate UAVs and display only the parameter of the currently selected one. Change the selected vehicle by selecting a new one from the dropdown menu on the left of the Connect button in the upper-right corner of the window.

A.11 APM as a Flight Controller board

During the early stages of this thesis, the first examined option for the autopilot board was the APM 2.6 (ArduPilot Mega). This compact and professional IMU autopilot is based on the Arduino Mega platform; it is fully compatible with the ArduPilot firmware and can control multi-rotor helicopters (drones) as well as traditional helicopters and fixed-wing aircraft or even ground rovers. It is capable of autonomous stabilisation, way-point based navigation and two way telemetry. The board presents no onboard compass, as it is optimized for vehicles where the compass should be placed far from power and motor sources to avoid magnetic interferences; it is intended to be used with a 3DR GPS uBlox with compass module. Namely, the uBlox GPS family of products is a warranty for quality GPS localisation, as it generally ensures a quick and stable GPS fix. Available features for the APM 2.6 include (from the ArduCopter APM Overview):

- 9 analog input pins, suitable for analog applications such as sonar. they can be configured as digital outputs, e.g. for camera shutter or relay-like applications
- 8 digital input pins for RC radio connection, compatible with both a multi-channel receiver and a PPM-SUM receiver (which uses a single channel for radio communication) and includes a power rail to provide power to the receiver straight from the APM
- 8 PWM compatible digital outputs for controlling motors; placed on a rail separated by a jumper, it is possible to power other external high power equipment such as servos through the power line of the ESC's BEC
- a 6-pos connectors power port, for powering the board with an APM Power Module such as the 3DR Power Module. Thanks to this module it is possible to get a stable 5.37V and 2.25A power supply and perform monitoring of battery's voltage and current, unlocking a return-to-launch trigger when voltage drops or the total power consumed gets close to the battery's capacity. The board embeds a 6V Zener diode and a 500mA fuse which create an over-voltage, over.current, reverse-polarity protection.
- an I2C port for interfacing with I2C digital equipment such as an external compass
- a GPS port
- a robust mini-USB port connector, to easily upload firmware on the board, with no need for additional external hardware
- a wireless telemetry port, suitable for 3DR Radio or MinimOSD OSD external module
- status LEDs, which indicate the regular running of the autopilot firmware as well as provide help during early set up operations such as calibration ones

Although not offering many expansion ports, the APM represents a valid option for building a solid UAV capable of both manual and automatic flight, because of this it was our first choice and option to realize this project. Following, the initial configuration for the system is illustrated:

- S500 quadcopter fiberglass frame, an excellent strong and lightweight 480mm frame with landing gear and a built-in Power Distribution Board (PDB) to have an easy connection from the battery to each ESC. Each arm has a carbon fiber rod through the center, achieving a rigid and extremely strong structure. With an height of 170mm and a weight of 405g, this frame makes for a compact UAV solution which can be easily upgraded with additional equipment such as a camera gimbal.
- 4 2212 920KV DC Brushless motors, 22x12mm, 53g and admitting a 7-12 V DC input
- 4 30A SImonK formware Brushless ESCs
- 4 1045 plastic rotors
- uBlox NEO-M8N GPS with compass, a low power and high precision GPS module achieve a precision of 0.9 m and a power consumption of 23mA at 3V and up to 5mA at 3V when in power saving mode. The GPS + compass module is enclosed in a plastic case and can be placed with its anti-vibration adhesive foam onto the proper support.

- a 3DR Power Module, which connect both to the 6-pos connector power port of the APM providing a stable 5.37V and up to 2.25A and to the Power Distribution Board embedded in the S500 frame.
- a 3S 3500 mAh 30C/Discharge and 60C/Burst LiPo battery, which could be connected via its XT60 connector straight to the Power Module.
- a 3DR 433 MHz 915MHz telemetry radio, coming in a pair of modules, one easily pluggable into a USB port and another with the proper pos connector to be connected to the wireless telemetry port of the APM

The following step was documenting and increasing our knowledge about the autopilot firmware and get a way to access the flight parameters and, in turn, control the UAV. As the APM can (and is intended to) run the ArduPilot firmware, the choice for the autopilot was immediate. Other platforms were not ever an option, as the ArduPilot is one, if not the most valid, solid and most suitable solution for a complete, real-time UAV application which demands scalability and modularity without giving safety up. As it is an open-source project, source codes are constantly and periodically upgraded and tested by a huge and responsive community of both hobbyists and experts. In order to inspect and load the firmware into the APM board, a modified version of the Arduino IDE has to be used (additional info following this guide); the source code for the APM board can be downloaded via git and can be found at the following GitHub repository: `https://github.com/ArduPilot/ardupilot`. Next, in order to stream our gas reading, a change both in the software and in the hardware had to be done. The MQ MOX gas sensor was connected to the A0 analog input pin of the APM board, connecting the GND, VCC and AOUT to pin -, pin + and pin S respectively of the A0 port. Then, as for the software side, some changes has been performed in the ArduCopter code:

- in UserCode.cpp, in the *userhook_SlowLoop*() function the proper analog reading function for getting analog value from the MQ sensor has been inserted, exploiting the AnalogRead() function to get the raw value and converting it to the corresponding voltage value with the usual formula:
  $floatvoltage = raw\_value * (5.0/1023.0)$
- in sensors.cpp, the so computed voltage value is assigned at the rangefinder_state.alt_cm variable
- in the *APM_Config.h* header file, the define USERHOOK_SLOWLOOP userhook_SlowLoop() has been uncommented, in order to run the so modified user loop, which runs periodically at 3.3 Hz.

This changes represented the simplest yet effective way to use an unsupported device in conjunction with the ArduCopter firmware. In fact, the ArduCopter, and more generally, the ArduPilot code encapsulates all of the UAV parameter states and variables into MAVLink messages. The source code involves more than a hundred of code files and several hundred of thousands of lines of code, thus adding a brand new functionality to the code is not a trivial task, considering all mutex-like structures and checks the code performs and, thus, was beyond the goal of this thesis. Considering this, encapsulating our gas data into a well-defined and tested structure was acceptable as a solution to get the job done, even if not the cleanest one. Thus, because we were not using a rangefinder, we encapsulate it into the rangefinder structure, achieving the transmission of the effective measured data as a "fake altitude measurement" from an inexistent sonar. The value was observable

from a regular Ground Control Station software, such as Mission Planner, and could be inspected in Python using the Vehicle.rangefinder DroneKit parameter.
This solution may seem good and more than enough for what was our purpose (streaming to the Ground Station readings from a single analog gas sensor), but as the project grew, this answer became more and more inadequate to our needs. First of all, this solution required ArduCopter source code modification, and consequently recompiling and re-uploading it into the board as well as relying on pieces of a constant evolving code. The ArduCopter code is managed by a vibrant and responsive community, which constantly adds new features, fixes bugs and changes several pieces of the source codes at a time. For this, the solution was not transparent to the autopilot firmware. Secondly, as previously said, exploiting other structures, although effective, is not recommended, as the very code relies on several status flag and parameters coming from almost each structure. "Faking" a parameter may lead to undesired and unpredictable behaviours (e.g. if specific defines are uncommented, the autopilot firmware uses also rangefinder distance measurements to estimate its altitude). Moreover, even not considering this, at the time of the adoption of this solution the code "offered" few fields for hiding unsupported parameters, transmitting more than three sensor measurement at a time was not possible. In addition, the limited available ports, both in terms of protocols (no I2C or SPI port ready to be used) and in terms of raw numbers made the APM board not suitable for future expansions of our project. Finally, as our knowledge of DroneKit APIs grew, we realized a radio connection, such as the one we used with the 3DR telemetry radio, could not coexist with DroneKit. In fact, we had not troubles with a serial USB communication, i.e. when the APM board was connected to the device running DroneKit with an USB connection; but as soon as we switched to a wireless communication, the code shown itself as unreliable, having hard times keeping the connection stable. This was due to the fact the APM was slowly becoming an obsolete board and thus DroneKit did not provide support for it. All these considerations quickly made the APM to the end of its life, at least for this thesis.

A.12 List of VOCs

This list reports all VOCs classified by the National Research Council of Canada as grouped by D.Won et al in Won, Lusztyk, and Shaw, 2005 and represents one of the most comprehensive list for volatile organic compounds.

- Aldehydes
  - Acetaldehyde
  - Acrolein
  - Benzaldehyde
  - Butanal
  - Decanal
  - Formaldehyde
  - Furfural
  - Heptanal
  - Hexanal
  - Nonanal

  - Octanal
  - Pentanal

- Ketones
  - Methylethylketone
  - Acetone
  - Acetophenone
  - Cyclohexanone
  - Methylisobutylketone

- Alcohols, Glycols, GlycoEthers
  - 1,2-Ethanediol
  - 1,2-Propanediol
  - 1-Butanol
  - 1-Methoxy-2-propanol
  - 1-Propanol
  - 2-Butoxyetahnol
  - 2-(2-Butoxyethoxy)ethanol
  - 2-Ethoxyethanol
  - 2-Ethyl-1-hexanol
  - 2-Methoxyethanol
  - 2-Methyl-2-propanol
  - 2-Propanol
  - Ethanol
  - Phenol

- Esters
  - 1-Methylethylacetate
  - 2-Ethoxyethylacetate
  - Butylacetate
  - Ethylacetate
  - TM-PD-DIB

- Halocarbons
  - 1,2-Dichlorobenzene
  - 1,4-Dichlorobenzene
  - Dichloromethane
  - Trichloroethylene

- Aliphatic Hydrocarbons
  - 2-Methylpentane
  - 3-Methylpentane
  - Decane
  - Dodecane

  - Heptane
  - Hexane
  - Nonane
  - Octane
  - Pentadecane
  - Tetradecane
  - Tridecane
  - Undecane

- Aromatic Hydrocarbons

  - 1,2,4,5-Tetramethylbenzene
  - 2-Ethyltoluene
  - 3-Ethyltoluene
  - 4-Ethyltoluene
  - 4-Phenylcyclohexene
  - Benzene
  - 1,2,3-Trimethylbenzene
  - 1,2,4-Trimethylbenzene
  - 1,2-Dimethylbenzene
  - 1,3,5-Trimethylbenzene
  - 1,3-Dimethylbenzene
  - 1,4-Dimethylbenzene
  - Isopropylbenzene
  - Propylbenzene
  - Ethylbenzene
  - Naphtthalene
  - 4-Isopropyltoluene
  - Styrene
  - Toluene

- CycloAlkanes

  - Cyclohexane
  - Butylcyclohexane
  - Ethylcyclohexane
  - Propylcyclohexane
  - Decahydronaphthalene

- Terpenes

  - alpha-Pinene
  - alpha-Terpinene
  - beta-Pinene
  - gamma-Terpinene
  - 3-Carene
  - Camphene

  - Limonene

- Other

  - 2-Pentylfuran
  - 1-Methyl-2-purrolidinone
  - Acetic acid
  - Hexanoic acid
  - n-Butylether
  - Pentanoic acid

A.13 List of flammable gases

This list presents all flammable gases as reported in Products, 2001.
LEL and UEL are omitted for readability.

- Acetone
- Acetylene
- Acrylonitrile
- Allene
- Ammonia
- Benzene
- 1,3-Butadiene
- Butane
- n-Butanol
- 1-Butene
- Cis-2-Butene
- Trans-2-Butene
- Butyl Acetate
- Carbon Monoxide
- Carbonyl Sulfide
- Chlorotrifluoroethylene
- Cumene
- Cyanogen
- Cyclohexane
- Cyclopropane
- Deuterium
- Diborane
- Dichlorosilane
- Diethylbenzene
- 1,1-Difluoro-1-Chloroethane
- 1,1-Difluoroethane

- 1,1-Difluoroethylene
- Dimethylamine
- Dimethyl Ether
- 2,2-Dimethylpropane
- Ethane
- Ethanol
- Ethyl Acetate
- Ethyl Benzene
- Ethyl Chloride
- Ethylene
- Ethylene Oxide
- Gasoline
- Heptane
- Hexane
- Hydrogen
- Hydrogen Cyanide
- Hydrogen Sulfide
- Isobutane
- Isobutylene
- Isopropanol
- Methane
- Methanol
- Methylacetylene
- Methyl Bromide
- 3-Methyl-1-Butene
- Methyl Cellosolve
- Methyl Cloride
- Methyl Ethyl Ketone
- Methyl Mercaptan
- Methyl Vinyl Ether
- Monoethylamine
- Monomethylamine
- Nickel Carbonyl
- Pentane
- Picoline
- Propane
- Propylene
- Popylene Oxide
- Styrene

- Tetrafluoroethylene
- Tetrahydrofuran
- Toluene
- Trichloroethylene
- Trimethylamine
- Turpentine
- Vinyl Acetate
- Vinyl Bromide
- Vinyl Chloride
- Vinyl Fluoride
- Xylene

Bibliography

AgribotixSolutions. *Easy To Use Drone Enabled Technologies And Services, Purpose Built For Agriculture*. URL: https://agribotix.com/ (visited on 10/14/2018).

Bartholmai, M. and P. Neumann (2011). "Adaptive Spatial-Resolved Gas Concentration Measurement Using a Micro-Drone". In: *Technisches Messen*, pp. 470–478.

Bartholmai, M. et al. (2013). "Gas Source Localization with a Micro-Drone using Bio-Inspired and Particle Filter-based Algorithms". In: *Advanced Robotics*, pp. 725–738. URL: https://www.researchgate.net/publication/263008758_Gas_Source_Localization_with_a_Micro-Drone_using_Bio-Inspired_and_Particle_Filter-based_Algorithms.

Commission, European. *Measuring air pollution with low-cost sensors*. URL: https://ec.europa.eu/jrc/en/publication/brochures-leaflets/measuring-air-pollution-low-cost-sensors (visited on 10/14/2018).

Denecke, Mirko. *Howto use BeagleBone Blue with ArduPilot*. URL: https://github.com/mirkix/ardupilotblue.

Eu, Kok Seng and Kian Meng Yap (2018). "Chemical plume tracing: A three-dimensional technique for quadrotors by considering the altitude control of the robot in the casting stage". In: *International Journal of Advanced Robotics Systems*, pp. 1–20. URL: https://www.researchgate.net/publication/322957994_Chemical_plume_tracing_A_three-dimensional_technique_for_quadrotors_by_considering_the_altitude_control_of_the_robot_in_the_casting_stage.

IGDSolutions. *Car Park Gas Detection*. URL: https://www.internationalgasdetectors.com/applications/car-park-gas-detection/ (visited on 10/14/2018).

Kimessa. *KIMESSA gas detection systems for boiler plant rooms*. URL: http://www.kimessa.com/applications/boiler-plant-rooms/ (visited on 10/14/2018).

— *KIMESSA gas detection systems for cold storage rooms and refrigeration systems*. URL: http://www.kimessa.com/applications/refrigeration-plants/ (visited on 10/14/2018).

LandPoint. *Drone Services for Oil and Gas Pipeline Inspections*. URL: http://www.landpoint.net/drone-services-for-oil-and-gas-pipeline-inspections/ (visited on 10/14/2018).

Products, Matheson Gas (2001). "Lower and Upper Explosive Limits for Flammable Gases and Vapors (LEL/UEL)". In: *Matheson Publication Library*. URL: https://www.mathesongas.com/pdfs/products/Lower-(LEL)-&-Upper-(UEL)-Explosive-Limits-.pdf.

Silvagni, Mario et al. (2016). "Multipurpose UAV for search and rescue operations in mountain avalanche events". In: *Geomatics*, pp. 18–33. URL: https://www.tandfonline.com/doi/full/10.1080/19475705.2016.1238852.

Vilches, Victor M. et al. (2014). "Towards an Open Source Linux autopilot for drones". In: *LibreCon 2014: Business and Open Technologies Conference*. URL: https://github.com/BeaglePilot/beaglepilot/blob/master/files/APM_Linux.pdf?raw=true.

Wired. *Inside Facebook's Ambitious Plan to Connect the Whole World*. URL: https://www.wired.com/2016/01/facebook-zuckerberg-internet-org/ (visited on 10/14/2018).

Wired2017. *Blood Carrying, Life Saving Drones Take off for Tanzania*. URL: https://www.wired.com/story/zipline-drone-delivery-tanzania/ (visited on 10/14/2018).

Won, D., E. Lusztyk, and C. Y. Shaw (2005). "Target VOC list: Final Report 1.1". In: *CMEIAQ-II Report 1.1*. URL: https://nparc.nrc-cnrc.gc.ca/eng/view/object/?id=b5da1c23-89a1-4f8e-a888-18194cf73173.