NTNU – Trondheim
Norwegian University of
Science and Technology

Autonomous Unmanned Aerial Vehicle In Search And Rescue

Vegard B Hammerseth

Master of Science in Engineering Cybernetics (2)
Submission date: June 2013
Supervisor: Amund Skavhaug, ITK

Norwegian University of Science and Technology
Department of Engineering Cybernetics

Task Description

Title: autonomous unmanned aerial vehicle in search and rescue
Task given: 15.01.2013
Deadline: 11.06.2013
Supervisor: Associate Professor Amund Skavhaug, Department of Engineering Cybernetics, NTNU
Student: Vegard Hammerseth

The use of unmanned aerial vehicles can be useful for search and rescue operations. NTNU, ITK wants to pursuit this field by developing a robust and low-cost experimental prototype to support research and concept trial.

The student should:

1. Familiarize himself with similar previous problems and solutions.
2. Set up system requirements.
3. Suggest a solution which respects the project boundaries.
4. If time allows it, implement selected parts.
5. Evaluate the proposed system according to the derived system criteria.

Preface

The following master thesis is my final contribution to achieve the title Master of Science from the Norwegian University of Science and Technology at the Department of Engineering Cybernetics.

I would sincerely thank my supervisor, Associate Professor Amund Skavhaug for guidance and suggestions, but also for giving me the ability to define my own project, pushing me to perform better and being more constructive.

By choosing my own project, motivation has been higher and I hope that my motivation has spread to this paper. Hopefully, intrigued other individuals to carry on the overall goal and saving lives.

I would also like to thank Terje Daleng for taking his time, reading my drafts and providing me with remarks.

Sammendrag

Denne rapporten presenterer bruk av selvstendige droner til å forbedre søk og redningsoperasjoner og tar de første stegene ved å realisere et sånt system. Ved å bruke selvstendige droner kreves det mindre av redningspersonellet og personell med spesialkompetanse innen droner blir overflødig. På grunn av selvstendigheten kan en drone operere utenfor signalrekkevidde. Når signalet forsvinner vil dronen kunne forsette søket, mellomlagre informasjonen og sende informasojnen når signalet blir gjenopprettet. Ved å lage billige droner senkes tersekelen for å sende de ut i dårlig vær og andre oppdrag der droneinformasjonen er viktigere enn returnering av dronen.

Fordi dronene må søke over et stort område vil et fly være det beste skroget for en slik operasjon. For å generere mindre luftmotstand og øke stabilitet vil lange, slanke og tynne vinger være best. For å øke aerodynamisk stabilitet vil rette vinger med en positiv dihedralvinkel være fordelaktig. Vingene bør festes på toppen av en slank og tynnest mulig flykropp. På grunn av vanskelighetene ved landing og sårbarhetene forbundet ved dette kan en flyvende vinge som følger de samme kravene om utforming være et bedre valg. En prototype av en flyvende vinge konstruert i utvidet polypropylen ble satt sammen og testet. Den viste seg selv å være motstandsdyktig, i stand til å tåle mye misbruk, raskt falle tilbake til sin opprinnelige form og bli reparert i løpet av minutter. Ideelt for forskjellige landingsunderlag.

Et attityde og kompassvinkel referanse system er nødvendig for å kjenne orienteringen til et kjøretøy. En billig versjon ble realisert ved å bruke mikroelektromekaniske sensorer og en mikrokontroller. Siden flyet må kunne orientere seg innen for et søkeområde kan NAVSTAR Global Positioning System (GPS) brukes for selvposisjonering. Ved å forhåndsprogrammere dronene med koordinater kan dronene følge en bestemt rute ved hjelp av selvposisjoneringen. Dette har blitt utviklet sammen med et komplett flysystem.

Dronenes søkeområde må bli programmert automatisk fra et gitt søkeområde som søkearbeiderne kan bestemme. Dette krever programvare og kommunikasjonsam-

band mellom en bakkestasjon og dronene. Et intuitivt grafisk brukergrensesnitt har blitt utviklet og bekreftet å virke ved å markere et område i programmet og sende koordinatene over et kommersiell kommunikasjonsamband til en prototype drone.

Et termisk kamera kan brukes for å oppdage mennesker. Det vil få pattedyr til å skille seg ut i et miljø når en analyserer termogrammet som kamera lager. Sammen med en terskel kan operatørene bli varslet med koordinater når valgte terskel er nådd. Et eksperiment ble gjennomført ved å plassere en person på et bestemt sted. En drone med selvposisjonering utstyr ble sent over personen flere ganger, oppdaget personen automatisk og rapporterte sin posisjon til bakkeastasjonen. Konklusjonen ble at dronen oftest var innenfor 20 meter radius av hvor personen befant seg.

Generaliteten og de prisgunstige komponentene som flykroppen, attityde systemet og datasyn systemet kan være nyttig for universitetet i årene fremover. Terskelen for å realisere applikasjoner som bruker noen av disse systemene har derfor blitt redusert. Et dronebasert system som dette kan forbedre søk og redningsoperasjoner og bidra til samfunnet ved å spare tid, penger og redde liv.

Summary

This report presents a way of using autonomous drones to enhance search and rescue operations and takes the first steps in bringing the system to life. By using autonomous drones, less experience is required by the rescue personnel and drone specialists become excessive in this matter. Due to autonomy a drone can operate outside a valid radio link. Hence, when signal is lost, the craft can continue to search, buffer the information and send it when the link becomes active. By creating affordable drones the threshold decreases for deploying a unit in bad weather or other missions where the feedback is more important than drone return.

Because the drones must sweep a large area, an aeroplane is the best suitable airframe. To generate less drag and increase stability; long, slender and thin-as-possible wings are recommended. To achieve aerodynamic stability, non-swept wings and a small positive dihedral angle is also advised. The wings should be attached on top of a slender and small-as-possible fuselage. However, due to the difficulties in landing and vulnerabilities related to this, a flying wing which obey the same design requirements, may be a better choice. A prototype for a flying wing made in expanded polypropylene was put together and tested. It proved to be resilient, able to withstand significant abuse, quickly recover to its former structure and be repaired in minutes. Highly convenient for various landing areas.

An attitude and heading reference system (AHRS) is required to tell which orientation a vehicle may have. An affordable version has been realised by using micro electro mechanical sensors and a micro controller. Since the vehicle must orient itself within a search area, a NAVSTAR Global Positioning System (GPS) and way-point approach were drones are pre-programmed to follow a path has been developed together with a complete flight system.

The drones search path must be programmed automatically from the given search area by the rescue personnel. This requires software and active communication link between a ground station and drones. An intuitive graphical user interface has been developed and verified to work by marking an area in the program and send

coordinates over a commercial communication link to a prototype drone.

A thermal imaging camera can be used to detect humans. It will make mammals stand out in an environment when viewed in the produced image (thermogram). Together with an arbitrarily threshold limit, the operators can be notified with coordinates when the threshold is reached. An experiment was carried out by placing a human at known coordinates. A drone with self position equipment was sent over the human repeatedly, automatically locating the person and reporting its location. It was concluded that the drone would be within 20 meters radius of the person.

The generality and affordability of the airframe, AHRS and vision system can be useful for the university in the years ahead and the threshold for realising applications which uses any of these systems has therefore been reduced. A drone based system like this can enhance search and rescue and assist the majority by saving time, money and lives.

Table of Contents

List of Figures

List of Abbreviations

AGL	Above Ground Level
AHRS	Attitude and Heading Reference System
AI	Artificial Intelligence
ARG	Angular Rate and Gravity
AT	Hayes command set
AUAV	Autonomous Unmanned Aerial Vehicle
AUAVISAR	Autonoumous Unmanned Aerial Vehicle In Search And Rescue
AUV	Autonomous Underwater Vehicles
CA	Cyanoacrylate
CF	Carbon Fiber
CG	Center of Gravity
CPU	Central Processing Unit
CSS	Cascading Style Sheets
EPO	Expanded PolyOlefin
EPP	Expanded PolyPropylene
EPS	Expanded PolyStyrene
ESC	Electronic Speed Controller
FIFO	First In First Out

FPS	Frames Per Second
FPV	First Person View
FSM	Finite-State Machine
GF	Glass Fiber
GNSS	Global Navigation Satellite System
GPS	NAVSTAR Global Positioning System
GPU	Graphical Processing Unit
GUI	Graphical User Interface
HTML	HyperText Markup Language
I/O	Input-Ouput
IMU	Inertial Measurement Unit
INS	Inertial Navigation System
I^2C	Inter-Integrated Circuit
LiPo	Lithium polymer
LOS	Line Of Sight
LSB	Least Significant Bit
MARG	Magnetic, Angular Rate and Gravity
MEMS	Micro Electro Mechanical Systems
MSB	Most Significant Bit
NMEA	National Marine Electronics Association
OOP	Object Oriented Programming
OS	Operative System
PWM	Pulse-width modulation
RC	Remote Controlled

RPM	Revolutions Per Minute
SAL	Above (mean) Sea Level
SAR	Search And Rescue
SARD	Search And Rescue Drone
SPI	Serial Peripheral Interface Bus
UART	Universal Asynchronous Receiver/Transmitter
UAS	Unmanned Aerial Systems
UAV	Umanned Aerial Vehicle
UI	User Interface
VTOL	Vertical Take Off and Landing
XML	eXtensible Markup Language

Chapter 1

Introduction

Finding a missing person in a Nordic terrestrial using a robotic bird is the subject of this report. Hence, *autonomous unmanned aerial vehicle in search and rescue*. An unmanned aerial vehicles (UAV) is often refereed to as a drone and is usually a smaller and less important object than required to solve an overall problem. It is often customized for its task and work together with other drones. The term *autonomous* means ”acting on its own” and such crafts must therefore ”think” on its own being able to decide the best choice for all situations which may arise.

In the last few years, UAVs has gained momentum, both in military and private sectors. The large adoption of drone systems is far from its peak and next technological brake through in this area is autonomy. While military operations include enemy combat and surveillance, private applications usually consists of controlling a UAV using a first person view (FPV) for various tasks. Both applications open up the possibilities of finding humans in search and rescue (SAR), but demanding autonomy without any live image transmitting complicates the task. The main reason for autonomy is to avoid using pilots for manual control and all the air trafficking control this would introduce for a single SAR operation. In addition, less training is required for the operators, eliminating the human error factor, saving time and money.

This report aims to be a recipe for how a prototype can be conducted at low cost and focuses on explaining details for ease of further development and adoption. If the drones can be produced so affordable that the conclusion ”it is to expensive to loose a drone in this bad weather” become obsolete, then much is achieved!

For this reason, several issues has been developed from scratch such that the source code is available, free to any one contributing, bringing drone based SAR faster to live.

1.1 Motivation

Numbers from Department of Economic and Social Affairs [2012] indicates not surprisingly that population is increasing. With more urbanizing and the desire to wander forests and mountains, more people are likely to be reported missing. Not only due to population growth, but the loss of practice in travelling by foot in nature, possibly due to urbanizing.

Getting lost is easier than one should think, and each year, many Norwegians and tourists are victims of this. According to Hovedredningssentralen [2012], each year, over thousand persons are reported missing on land, just in Norway. Recent numbers pointed out by an.no [2012] shows that more people are getting lost in 2012 than ten years ago.

People are always ending up missing, hurt or dead - if not found in time. Thus, getting lost is a serious matter and the localization equipment should be as ideal as possible.

Several UAV projects have been conducted earlier at NTNU, but not with specific practical problems in mind. Because the undersigned is a remote controlled (RC) enthusiast and an experienced flyer and builder, the desire to use this passion for something practical is fairly great.

1.2 Other work

There has been constructed several UAVs in the past. Few of them are completely autonomous and often require manual input. Even fewer are explicitly constructed for SAR.

1.2.1 NTNU

The unmanned vehicles laboratory at NTNU is a test facility for hardware and software including inertial navigation systems (INS), global satellite navigation systems, autonomous underwater vehicles (AUV) and unmanned aerial systems (UAS). Several student projects has been completed and the most relevant which stands out is summarized below.

Design of instrumentation for autonomous fixed wing UAV (2006)

Høstmark [2006] constructed a small unmanned vehicle which was a cross between a conventional plane and a canard plane. Based on visual determination, the result was a design with low drag and noise reduction.

CyberSwan (2007)

Canards were removed from the plane constructed by [Høstmark, 2006] and baptized CyberSwan. The goal of this project was to create a general platform for UAV with intentions to recruit students to the Department of Cybernetics. The project was split in three theses where Eriksen [2007] created ground station hardware and used LabView for control software, Høstmark [2007] did modelling simulation and control, and Bjørntvedt [2007] constructed on-board hardware. The fourth and last test flight was concluded to perform stable and autonomous heading control.

General Platform for Unmanned Autonomous Systems (2010)

Skøien and Vermeer [2010] compared small computer boards, operating systems and created a general UAV framework. They argument for the choice of using BeagleBoard and Linux - several distributions were tested, but Ångström GNU/Linux was concluded superior due to popularity.

Overall Technical UAV Solution (2011)

Koteich and Rennæs [2011] created a single-board computer with UAV in mind. A project priority was to address computer vision and image processing. It was concluded that image processing had to be done in air by a computer in the UAV. In addition, the computer created was concluded to not be powerful enough for image processing which leaves the image processing issue unresolved.

Hardware-in-the-Loop framework for UAV testing (2011)

Christopher Stern made among other things a ground station software for UAVs aimed for developers. Because it is created using a proprietary framework, it is less compatible, customizable and more expensive than necessary. However, there was not any requirements to satisfy this nor was it aim to have a simple-as-possible interface for the operators either.

1.2.2 Collision avoidance UAV Declared a Success

NASA Dryden Flight Research Center [2012] developed an automatic ground collision avoidance system using their own unmanned Dryden Remotely Operated Integrated Drone and unknown sensoring. The collision avoidance software was implemented in a smart phone connected to its autopilot and successively guided the craft away from collision.

1.2.3 Supporting Wilderness SAR with Integrated Intelligence

Lanny Lin and Morse [2010] shows that its possible to find a missing person by analysing the terrain and finding likely walked-paths. By introducing a mini-UAV with a camera for terrain surveillance, the study concludes about the requirement of autonomy, but is not deduced. A real searcher manually flew the UAV after minimal training and successfully located the simulated missing person in a wilderness area.

Even though this research is mainly a FPV controlled UAV, the interaction with real searchers and their positive feedback is probably the most important conclusion for future systems.

1.2.4 Aerovision FULMAR

Aerovision [2012] FULMAR is a commerical mini-UAS teledetection system. Its claimed to be self-piloted with the capabilities to take and transmit real time video or infrared images to a ground control station. Possibly highly convenient for SAR.

1.2.5 Discussion

The completed projects carried out by NTNU students have been mostly theoretical and occasional table prototyping. Late projects has focused on which computer board to use and have been reconsidered again and again until a custom one was made. The custom board was originally made to handle computer vision expansions. However, the work addressed and realised during the CyberSwan project share many issues related to this project. Unfortunately, some parts are outdated and the system has been given away, unavailable for both the institute and author.

Høstmark [2006] did extensive research when building a UAV from scratch with low drag. Due to complexity and unavailability, the design will not be used. Even though the plane has canards to generate additional lift, it is more exposed for damage when considering the whole application. With long slender wings, the design share wing sweptness with a typical sail plane. This makes the plane more stable and have probably great range capabilities. On the other hand, during the CyberSwan project Bjørntvedt [2007] concluded that a new plane should be created for future projects. If the plane could be constructed in another material this design could be altered and used for SAR.

Since the drones must be autonomous, collision avoidance is a difficult, but a necessary subject to address which has not been solved at the NTNU UAV lab.

Considering the last two sections, it appears quite plausible to develop and utilize drones for real SAR.

1.3 Search and rescue

SAR operations are quite comprehensive and the following sub sections will explain how they usually are performed in Norway and how they can be improved from a vehicle perspective.

1.3.1 The way of today

SAR operations are still and will probably be highly dependent on human beings looking for missing persons in the future. This are often volunteers with much experience in nature and with more volunteers or helpers, the quicker the operation

can be completed. In addition, helicopters are much used in these operations and in some cases even planes. Its quite trivial to understand why aerial vehicles are used and why its reasonable to consider drones for this purpose. By equipping helicopters with thermal imaging cameras, it becomes a power full tool to detect mammals. These sensors are aimed at the ground and controlled by an operator whom looks for possible objects.

Even though thermal imaging cameras is an advantageous tool in SAR, it can only monitor a limited area and the question for parallelism arises. How many vehicles should be used for each operation, how many can we afford to maintain and how many pilots and co-pilots can we afford? the answer should be ”unlimited”, but in reality this is not the case.

1.3.2 How it can be

Imagine if we had a small optimized and autonomous fleet of helicopters and someone is reported missing. A UAV-corps is notified and drive as close to the relevant area in their specialized emergency vehicle with a trailer of drones. During the drive they remotely program the drones and ready them for action. A single operator choose a desired area for which the missing person is likely to be within and the area is automatically shared between number of drones as coordinates. The coordinates are set up such that launch, flying and landing is different for each drone to prevent any collision between the crafts.

When arriving on site, the vehicles are sent up in the air one by one. When airborne, the drone start its programmed flight fairly close to the ground. This means as high as the thermal camera allows for proper ground analysis. When a possible object is found, an image of the area and the NAVSTAR Global Positioning System (GPS) coordinates are sent back to the emergency vehicle which houses a computer with a digital map where the position gets marked. The base team can investigate the received image and if any signs of a person is present, a rescue team can be sent out to rescue the person.

Even though a single smaller vehicle will much likely be slower than a large one, they will gain rapidity in parallelism which also introduces the advantage of redundancy. In comparison, when a manned vehicle might be maintained, which is a serious and expensive matter, the drones can still operate, but with one less unit.

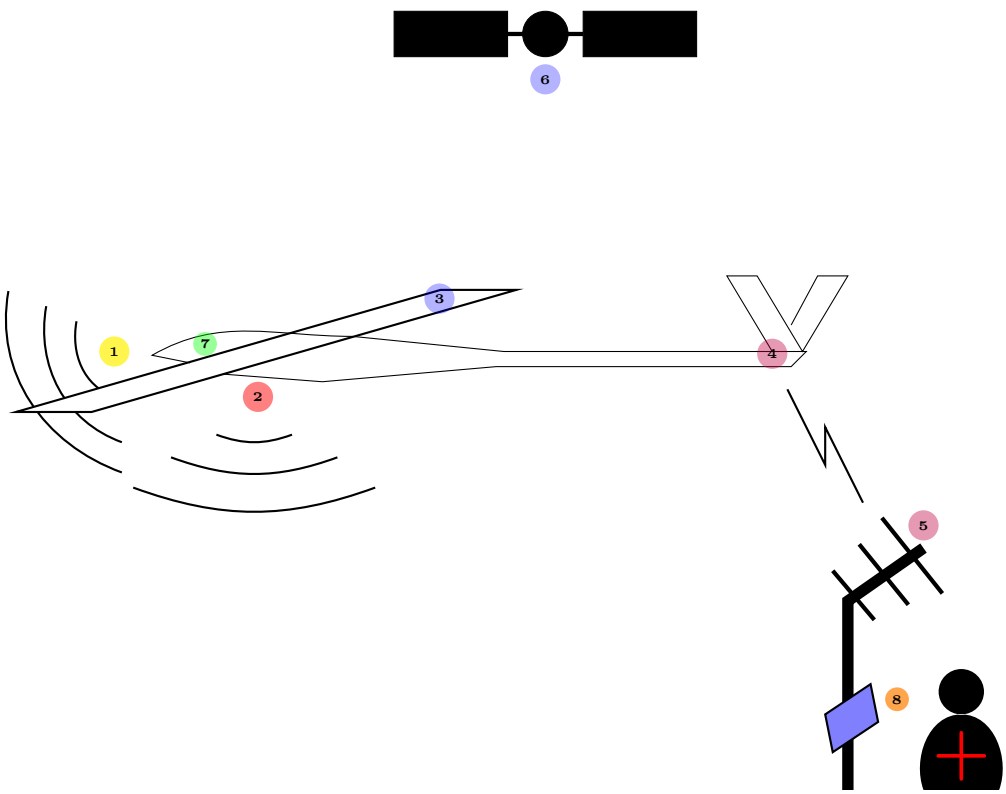


Figure 1.1: System overview, equal coloured numbers are linked and are scaled for illustration

Even when large helicopters are not allowed to be airborne due to bad weather, a smaller version can still operate without jeopardizing human lives. This does however put high demands on the system and fixed wings vehicle will most likely struggle in these conditions.

1.3.3 Alternate methods

Besides physically searching for an individual, there are other methods for finding a modern human. An ambulant traveller will bring with them a cellphone or some sort of traceable device. Because authorities can demand disclosure of the last connected cellphone towers, the travellers position can be estimated. Moreover, new smart phones are usually equipped with GPS receivers which can be used for self position. If the person is conscious and signals are available, the person should

be able to guide him or herself to safety.

1.3.4 Norwegian regulations

Unlike a full scale craft, an UAV will produce less damage it were to crash. Thus, an UAV will be unaffected by law or at least in a less strict manner than large craft would be. At least, a law should respect the craft size and the outcome if this were to happen.

The laws for drones is somewhat an untouched subject in the Norwegian law system. However, [Luftfartstilsynet] has compiled a few laws which affect autonomous aerial vehicles. According to this, there are at least two rules which breaks the prerequisites for this project. The first requirement is to always have the UAV visible in sight without use of binoculars. The second demand manual control of the vehicle. A third to mention is the height limit of 122 meters, but since a search and rescue drone (SARD) must be close to ground this limit does not affect planned operations. Clearly, an exception must be made for SARDs to be able to realise a system like this.

To protect national security, [Forsvarsdepartementet] reports that the Norwegian legislation does not accept any flight photo without notification in advance. A five-year valid license can be obtained, but require that the person applying can get proper security clearance. Thus, is not trivial to put a camera nor a thermal camera on any aircraft.

There are also regulations for communication frequencies and power output of these. From the regulations of Post- og teletilsynet [b], a regular person can transmit e.g. 2.4GHz signals at 100mW. With an amateur license, the same person can transmit 2.4GHz at 100W according to Post- og teletilsynet [a]. An AUAVISAR system will probably get a dedicated radio spectrum in this matter.

1.3.5 Environmental aspects

A positive side effect of using unmanned crafts are the smaller energy requirements. With a smaller vehicle, less weight is present, a smaller propulsion system is required and less energy is necessary to travel the same distance. This directly impacts fuel emissions.

When it comes to people working for SAR departments, a negative case is when staff becomes excessive. A large adoption of complete AUAVISAR systems can give full scale vehicles less air time. This may eventually mean that pilots are less required and must be reassigned for other tasks until drones are done sweeping or in worst cases lose their jobs. If the drones were to be manually controlled, then more pilots are actually required.

An indirect impact as a result of less air time results in less maintenance and less purchase of parts resulting in less income for part sellers and more could loose their jobs.

1.4 Main problems

To have a fully working system, the following topics must be addressed and resolved.

1.4.1 Artificial intelligence

All autonomous machines must have artificial intelligence (AI), simple or not. This would both be the overall glue to fuse all the listed problems and how they should work together. More or less a recipe for how each drone should react to each situation. To simplify this as much as possible, it could possibly be set up as a finite-state machine (FSM).

1.4.2 Collision avoidance

Animals, humans and even insects has highly sophisticated ”sensors” for collision avoidance. Insects usually has a good active collision avoidance as they react almost instantly when approached. Their passive system is not always equally good since they tend to fly into objects. While bats use sound based collision system, most creatures uses vision where two or more eyes are required for depth estimation.

To prohibit collision with for example helicopters, power poles or trees, a similar technological sensing system must be developed or selected. This is a crucial subject and not trivial to solve.

1.4.3 Airframe

There should be chosen a frame which provides the best range, durability and simplicity for a SARD. Frame quality depends much on material used and a tough material must be used together with an easy to assemble frame. Range capabilities is not the most important property for a prototype.

1.4.4 Propulsion

When selecting a propulsion system, many factors are to consider. The most important ones include range, reliability and price. Efficiency, simplicity and noise are less important, but are indirectly connected to the first ones.

1.4.5 Determine attitude and heading

For attitude and heading determination, several sensors are required and must be selected. Several types of sensors are available, but price and weight limits the options. Known control theory can be used to fuse the sensors to form an attitude and heading reference system (AHRS).

1.4.6 Stabilisation and control

By using estimated attitude from an onboard AHRS, stabilisation of the vehicle is possible and several control laws can be used. The layer from stabilisation signal to actuator must be derived and also include guidance and heading such that vehicle will fly in the right direction.

1.4.7 Altitude

Altitude is usually referred to as above (mean) sea level (SAL) or above ground level (AGL). For this application it is the latter which is interesting. However, it is important that AGL include both terrain and objects. For this it is desired to use a sensor that can be aimed down and report measurements of current height.

The same sensor could be used for automatic landing procedures if the resolution is sufficient.

1.4.8 Navigation

A guidance system must be put together. This often includes pure GNSS or inertial navigation system/GNSS integration, way-points and path generation to be fed to the regulation controllers.

Path generation could easily be its own topic in this matter. E.g. where sweeping should start, in what pattern and respecting the terrestrial are just some problems to be solved for an optimal path generation. This computation can be done on a ground station before launch where additional computational power is available.

1.4.9 Communication

A wireless communication link between the ground station and drones must be established with focus on weight, range, reliability and power consumption. This link will mainly be used to send coordinates from the drones to the ground station and new way-points from ground station to drones. Being able to receive images over this link without having to over dimension the link in terms of size, weight and power consumption would also be advantageous.

1.4.10 Graphical User Interface (GUI)

There must be developed a computer software which displays a map where the operators can select an area to be searched. The map needs to know coordinates to be able to create way-points. Furthermore, the way-points has to be loaded to the drones which indicates that a stable link must be set up between the systems.

1.4.11 Recognizing humans

A sensor for detecting or separating a human from the environment from a distance is required. It must be selected or developed. With respect to power consumption, several sensors can be combined for higher accuracy.

1.5 Proposed system

With basis of section 1.3.2, this section will explain how the overall system is intended to work followed by subsequent sections listing the subjects which is a part of this.

1.5.1 Overall

If for each number in figure 1.1 relates to this list, the technological components can be summarized as:

1. Collision avoidance. Detect objects in planned path and alter trajectory.
2. The sensor to detect humans should be faced downward. If heavy, attach it at center of gravity (CG). Elaborated in section 9.
3. A Global Navigation Satellite System (GNSS) receiver should be placed on top of the vehicle away from disturbances, at least its antennas.
4. Transceiver. Send coordinates of spotted points. Receive confirmation and possible return to origin commands.
5. Transceiver. Receive coordinates of spotted points. Send confirmation when coordinates are received.
6. GNSS broadcast. Included for illustration purposes.
7. Compass, accelerometers, gyros, stabilisation- and navigation control.
8. Computer to program route or display coordinates to rescue personnel.

Note that the radio link will not transfer live images to base station due to the declared reasons and bandwidth limit addressed by [Koteich and Rennæs, 2011]. It is however more convenient to transfer crucial images for additional analysis, but only coordinates will be considered good enough at this stage.

1.5.2 Hardware

From figure 1.2 one can understand how the overall hardware should be connected to each other, but not necessarily what all the components do. For collision avoidance, several sensors may serve the purpose. This includes ultrasonic range finders, optical flow, laser distance sensors and more. A high level approach is the use of vision based systems, hence camera. In general, the latter needs to be stereo to be able to tell distance. However, with known vehicle velocity and frame rate, distance can probably be estimated sufficiently with a single lens.

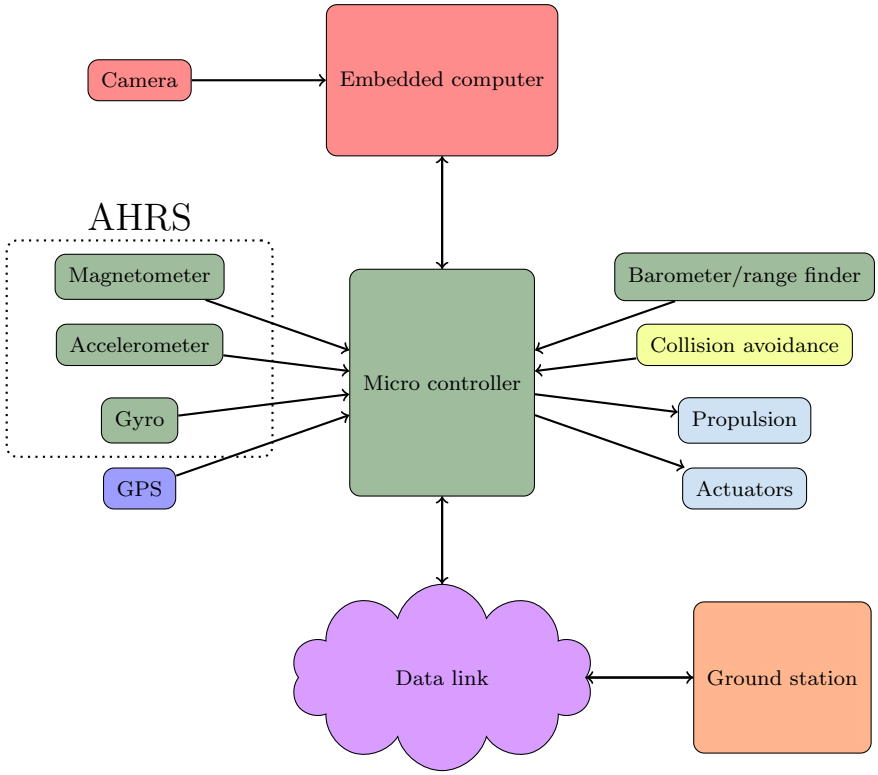


Figure 1.2: Hardware component outline

Even though figure 1.2 has its layout as shown, some re-organizing may be necessary. The main reason for having an embedded computer is to analyse images for missing persons. It may be omitted if camera vision is avoided. Collision avoidance has been placed as being its own sensor because an avoidance system should be able to sense collision quickly. On the other hand, if the embedded computer has enough computational power and the algorithms to detect objects in trajectory, the collision avoidance box could be connected to it.

1.6 Scope

An autonomous unmanned aerial vehicle (AUAV) system is designed to operate on itself without control input from humans during transit. The planning, designing and construction of such a system requires a vast combination of technology fields. This includes aerodynamic, collision avoidance, control theory, navigation theory, physics, electrical engineering and computer science. In addition, collision

avoidance systems may use radar or computer vision which indicate analyzing in real-time and statistics approaches.

A prototype require additional functions for simple debugging and manual control when needed. Hence, workload increases.

Even though the final is goal is to have a complete working system to be distributed, the budget and time for this project is not sufficient to achieve this. Due to the given budget, several topics has been developed from scratch to save money, reducing available time for each topic even further. It is therefore been decided to ignore the requirements of a collision avoidance system, altitude control and velocity control.

Beside collision avoidance, the other issues are pure cybernetics tasks and future students at Department of Engineering Cybernetics should not have any difficulties to complete and implement these subjects.

1.7 Report structure

Program code developed in this project will be presented in its written language and not pseudo code. This will mostly be object in the oriented programming (OOP) language C++. It was chosen to do it this way to ease the transition between algorithm presentation and realization. Several algorithm schemes are included for code snippets to aid code understanding. The header files defining classes and variables for the code is not presented, but rather included along with the code on the attached CD to skip tedious reading.

Because of the vast subject of this thesis, each chapter represent mostly its own subject. By doing this, each chapter can more or less be read by its own. The last section in each chapter is usually a discussion associated with its chapter.

Chapter 2 narrows down the available and most suitable structures for an AUAVISAR and presents different wing profiles and layouts with different properties for different use. Propulsion is investigated and is chosen together with a frame.

Chapter 3 derives an attitude and heading reference system from scratch.

Chapter 4 covers the navigation issue based on GPS and way-point tracking. It also derives software to realise this.

In chapter 5, several methods for sharing data wirelessly is discussed. Based on the best method, a system is put together and implemented.

Before the system is fused in chapter 7, chapter 6 suggest how the overall UAV handling should occur such that its easier to understand how every function is connected.

A simple graphical interface was put together and derived throughout chapter 8.

Chapter 9 presents ways of distinguishing humans from an environment. An experiment has been carried out to check how this can be done and another experiment takes the first experiment one step further.

Final thesis reflections is shared in chapter 10 and a future road map is discussed in chapter 11.

The final chapter 12 summarize the completed thesis and the need of this system.

Chapter 2

Airframe and Propulsion

Niu [1988] wrote that the mechanical structure of an aircraft is known as the airframe. It includes fuselage, wings and undercarriage. The airframe define the stability and behaviour of the aircraft when in air.

2.1 Rotorcraft

Vehicles where lift or vertical thrust is provided by rotating propeller(s) are known as rotor driven vehicles. Their main advantage is the ability for vertical take off and landing (VTOL) and maneuver during low airspeed conditions. This is typically the conventional helicopter, where the main rotor is a vertical axle with two or more horizontal blades attached on top. The horizontal-axle tail rotor compensates for the moment of inertia from the main axle. If this is broken, the helicopter will spin uncontrolled around the main rotor.

There exists however multi rotor copters. Typically dual-, tri-, quad- and hexacopters where the two latter ones have become very popular in UAV research these last years. Much due to the cost reduce of small efficient motors, regulators and battery technology.

The two main advantages with all multi rotor copter designs is their ability to lift heavy and not have a propelled tail to compensate for main rotor torque. Furthermore, copters with more than two rotors introduces the possibility for redundancy.

Because rotor driven vehicles uses most of its energy just to keep it airborne and long range vehicles are needed, they can be ruled out for both a prototype and a final solution. In the case where maneuverability is more important than range or stationary air operations are required, such as looking for persons in between trees close to ground, a small copter would be quite suitable.

2.2 Ornithopters

From [DeLaurier, 1994], an ornithopter can be defined as a plane where propulsion and lift are provided by hinged wings, allowing them to flap in similar manner to a bird. Actually, some of the first flight attempts used these principles and nearly equal many has failed. Still, there have been huge breakthroughs for unmanned ornithopters the last decade. Festo [2011] introduced SmartBird, a robotic ornithopter which simulates a seagull in appearance and size.

Ornithopters has the advantages of hiding all actuators and propulsion systems inside the fuselage and wings such that there are no propellers which induce drag making the airframe highly aerodynamic.

Ornithopters way of copying the nature to use as little energy as possible is very appealing for long range vehicles. Both aerodynamic advantages and the degree of blending with nature. However, in SAR, it might be quite a relief for the missing person to know that a UAV is nearby. This could be troublesome with a stealthy ornithopter. Furthermore, the technology which work similar to a bird is still new, proprietary and restricted. If an open version were available the choice had to be reconsidered.

2.3 Fixed wing planes

A definition for an aircraft is a vehicle capable of generate sustainable lift due to the vehicles forward velocity. The wings of a fixed-wing aircraft are not necessarily rigid. Kites, hang-gliders and planes using wing-warping are all regarded fixed. Grumman F-14 Tomcat illustrated in figure 2.1 is a plane with wing-warping. From the definition of an aircraft, it can be seen that a fixed wing partially fulfils demands for an AUAVISAR.

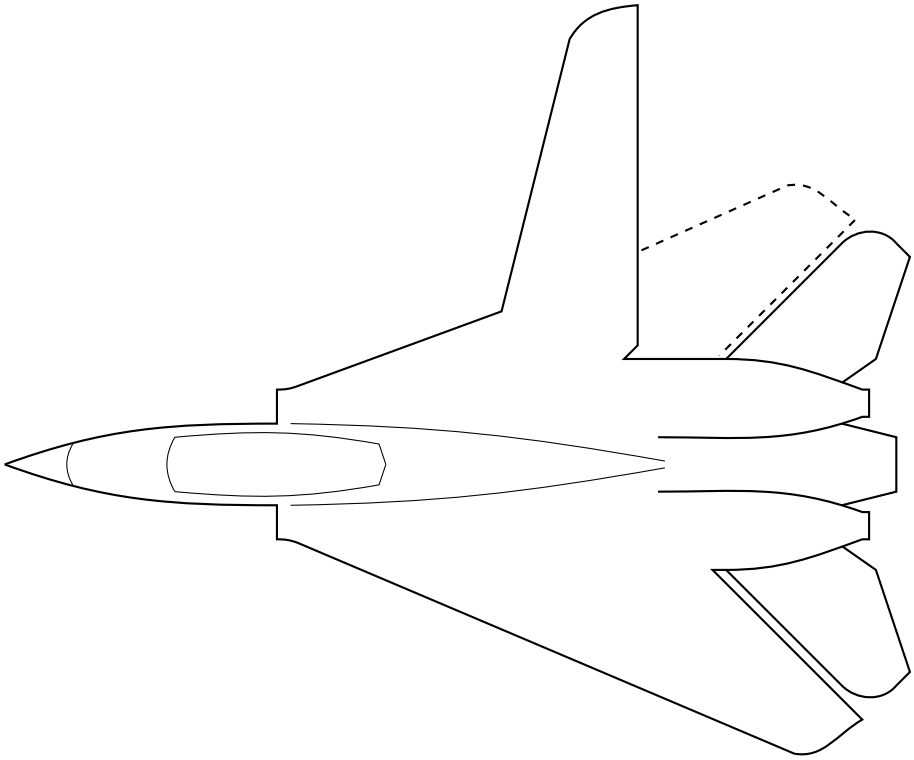


Figure 2.1: Grumman F-14 Tomcat with one wing swept backward and one straight outward

2.3.1 Conventional

With a slender body, wings about the middle and a T-tail, the conventional plane is over represented in the world. It provide a good and stable platform which can be manually flown without any necessary stabilization system where all orientations can be controlled.

Due to the slender body, additional weight can be distributed along the fuselage with respect to center of gravity.

2.3.2 Delta

A plane with delta wing differs from a conventional plane by not having a stand alone elevator tail. The main wing and tail are rather melted together such that the main wing runs along the fuselage creating a larger wing area. It often looks

like a stretched triangle and has its name from the Greek letter Δ. Most modern fighter jets are based on this design due to less drag when flying supersonic speeds [Kermode, 2006]. With both wings swept back, the plane in figure 2.1 has a delta layout.

2.3.3 Flying wing

A flying wing has very often a flat structure without any defined fuselage or tail, unlike a conventional plane. It does however sometimes have vertical fins for directional stability and yaw control. Since the wing is what generates lift of a plane, its trivial to understand that the whole body of this type generates lift.

2.3.4 Discussion

Even though a conventional plane with slender wings might be superior in the case of drag, it is much more exposed to damages than a flying wing when considering landing properties. This applies especially during prototyping. Thus, a rigid flying wing with large lift- and stability properties appears to be the best suitable solution at this point. Even though thick wings with large area may create more drag, the ability to lift more weight may be necessary when using bulky electronics for a prototype. With large enough wing area, large elevators and a low wing load, any small plane can perform a deep stall landing. It does however require a proper undercarriage and a rigid construction. With the ability of performing a controlled deep stall, only a small landing area is necessary.

2.4 Wings

With the choice of using a fixed-wing frame it falls natural to discuss wing-profiles, types and placement. The wings are the most important parts on an airplane and have major impact on the aerodynamic behaviour.

2.4.1 Airfoil

The surfaces that support the aircraft by the means of dynamic reaction on the air are called wings [Abbott and Doenhoff, 1959]. Profiles of the wings are called airfoils and there exists infinite types where each design has its own advantages and disadvantages, by the means of drag and lift. In figure 2.2, some common foils are illustrated.

(a) Thin airfoil ”high speed”

(b) Medium airfoil ”general purpose”

(c) Thick airfoil ”high lift”

(d) Transonic airfoil ”high subsonic speed”

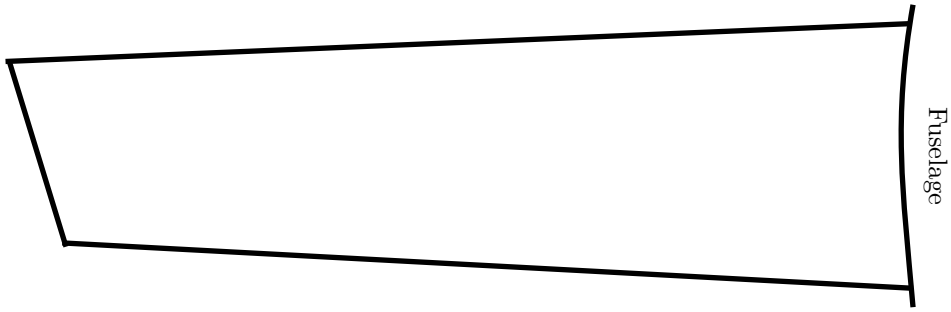
Figure 2.2: Thickness of airfoil sections

The general incorrect understanding for airfoils is the difference in pressure from upper and lower sides generates the lift. National Aeronautics And Space Administration reports that lift occurs when a moving flow of a gas is turned by a solid object. The flow is turned in one direction, and the lift is generated in the opposite direction, according to Newtons third law. Because air is a gas and the molecules are free to move about, any solid surface can deflect a flow. Unlike an airfoil a flat surface must be inclined to generate lift. For an aircraft wing, both the upper and lower surfaces contribute to the flow turning because of the slight upflow before reaching the foil. A wing with more chamber will induce more upflow. For instance, the thick wing in figure 2.2 has the most lift of those listed. Having more lift sounds good, but a thicker wing requires more force to be pushed through air

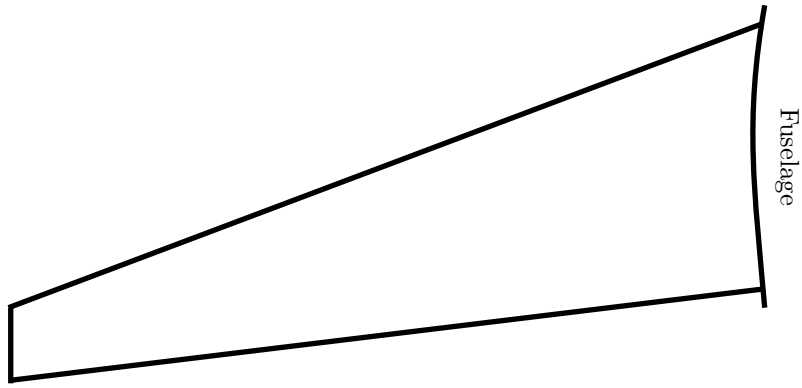
which indicate higher drag coefficients. To illustrate, this is why sail planes has very thin foils, but less lift for each square. For this reason, to generate sufficient lift, its wing span is much longer compared to a thick foiled wing.

2.4.2 Wing types

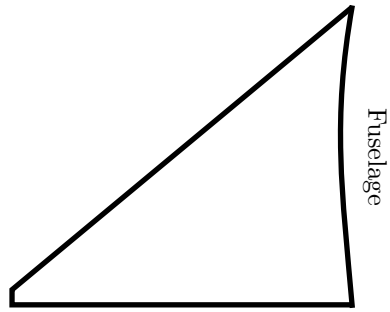
When considering wing types, it is the main wings which are to consider. Their main function is to generate lift and different layouts provide different properties for drag, lift and wake. From figure 2.3 the most popular wings with different sweep is presented.



(a) Straight wing



(b) Swept wing



(c) Delta wing

Figure 2.3: Wing layouts with different swept

A wing tip vortex generates a lot of drag and energy is lost in the process of making lift because of the airflow around the wing tips. Wings with wide wingspan has less wing tip vortex. Thus, an aspect ratio formula has been presented as

$$aspect\ ratio = \frac{(wing\ span)^2}{wing\ area} = \frac{length\ of\ wing}{width\ of\ wing} \tag{2.1}$$

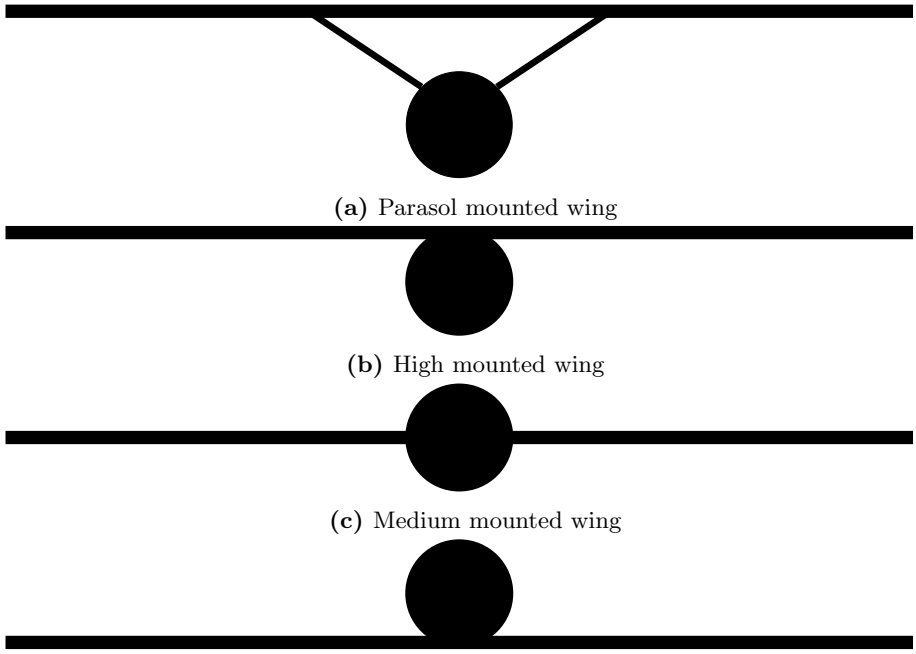
where higher aspect ratio indicate generation of more lift and less drag. Therefore straight wings are usually found on small, low-speed airplanes and especially on sail planes. Low aspect ratio wings like on a fighter jet have much more of this type of drag. Moreover, when the velocity is transonic, the shock waves from the nose starts to hit the wing tips producing additional drag [Abbott and Doenhoff, 1959]. For this reason, high aspect is not as preferable any more and more sweep is introduced. This is why fighter jets has highly swept wings, a delta layout.

A significant drawback of high sweep is the loss of stability at low airspeeds conditions. For this reason, fighter jets must land at higher airspeeds to perform a safe landing. In comparison, a commercial jetliner has wings with moderate sweep. This results in less drag while maintaining stability at lower airspeeds.

If one still consider the Grumman F-14 Tomcat from figure 2.1. In speed mode, the main wings are folded backwards, creating a delta wing. While during take off and landing the wings are folded outwards like the wing illustrated in figure 2.3a. This gives the plane longer wingspan and more stability at low airspeeds conditions. Thus, wing design depends upon its purpose to be used.

2.4.3 Wing position

With the choice of a craft without fuselage this section excess, but will be quickly considered for completeness.



(a) Parasol mounted wing

(b) High mounted wing

(c) Medium mounted wing

(d) Low mounted wing

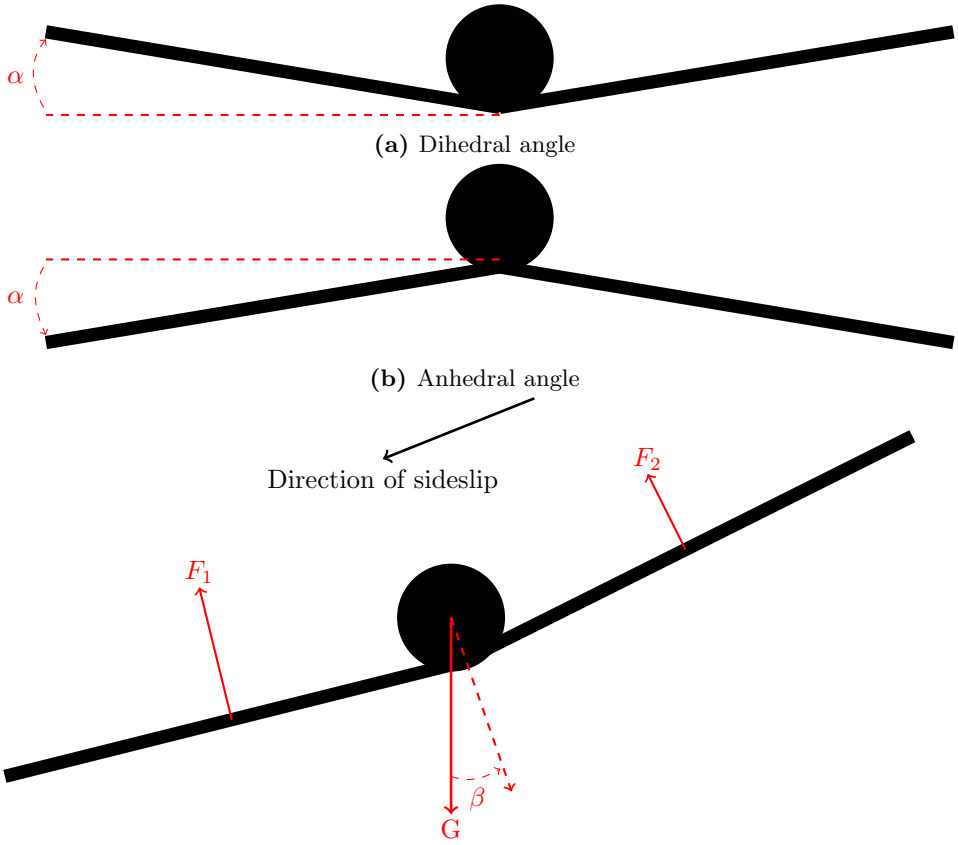
Figure 2.4: Wing positions where the circle is the fuselage

Parasoled wings are very common on micro planes. This is probably due to increase stability as smaller crafts tend to be more unstable. The top mount is the second most stable due to the nature of the design and is very popular for beginners and hobbyist. The two latter which are more unstable usually solve the instability issue by introducing dihedral angle, which will be addressed in section 2.4.4.

2.4.4 Dihedral angle

The most common method of obtaining lateral stability is by using a dihedral angle on the main wings. A dihedral angle is the angle between each wing and the horizon as illustrated in figure 2.5. Its negate, anhedral, has an angle making the wing point downwards [Kermode, 2006]. One might argue why anhedral should ever be used as it induces instability. For instance, on planes where Dutch roll tendencies

are present, anhedral angle is introduced. The same applies for planes where rapid roll is required or for large planes with high roll period.



(a) Dihedral angle

(b) Anhedral angle

(c) Dihedral effect, lift force $F_1 > F_2$ and β is side-slip angle

Figure 2.5: Flight behaviour with dihedral angle

When one of the wings of a plane with dihedral wings drops, the resulting lift on both wings will be slightly inclined. This causes the plane to side-slip resulting in a flow of air in opposite direction to the slip as illustrated in figure 2.5c. This increases the angle of attack of the lower wing. The lower wing will therefore produce more lift and a restoring moment will result Kermode [2006].

2.4.5 Discussion

Even though the search area should be swept over as fast as possible, it is not convenient to choose a delta wing with large toe when considering all the other limitations. A slightly swept wing will provide enough stability and speed to perform the action sufficiently.

If a conventional frame were to be used, the high mount would be suitable such that the undercarriage or fuselage would take the impact during landing. It would also provide sufficient stability as explained. When using a flying wing, the wing positioning is irrelevant unless one were to load the wing with bulky load which might act as a fuselage.

It is clearly better to have thinner wing as it will induce less drag. This is however not as easy for a flying wing where all components are to be put inside the wing itself. On the other hand, the wing does not have to be uniformly thick and a cockpit at center can be present. Even though the thicker wing will create more drag, less angle of attack is required to main the same altitude.

A flying wing is susceptible for a dihedral wing design which is often a good idea for lateral stability. It could be completely solved by using an electrical stabilization system, but by solving it aerodynamically less energy is wasted.

If the propulsion system is put behind the main wings or any control surfaces, little or none turbulent airflow can disturb the airflow and less drag is induced.

2.5 Fuselage material

For less drag, a streamlined fuselage and wings is desirable and can be constructed using various materials. When selecting material, integrity, weight, price and availability are to consider. The most commonly used materials for model planes are listed and explained below.

2.5.1 Ochroma pyramidale

Ochroma pyramidale is a tree type and is used to make a wood material commonly known as balsa wood. Due to its lightweight and stiffness it has been very much

used for model planes. The wood is only used for defining a skeleton for the plane and is later covered with thin lightweight film. It is not very expensive, but requires a lot of work to cut and assemble. On top of that, it is very fragile for impacts and time consuming to repair.

2.5.2 Glass fiber

Instead of covering the balsa built fuselage with film, sheets of glass fiber (GF) can be used in stead. It will add more weight, but is significantly stronger.

When damaged, fibers break and can not be reattached easily. It is common to apply a new sheet on top of the damaged area. If a significant damage occurs and many fibers are broken, it becomes hard to fix without altering the structure and adding additional weight.

2.5.3 Carbon fiber

Carbon fiber (CF) is newfangled GF, but stronger and lighter. To apply it, an airframe must be constructed first and sheets of CF would be attached similar to GF. The sheets are then soaked with epoxy and grinded to a even finish when dried. This is a hazardous task and proper protection is required.

CF is also susceptible for damage and fibers break similar to GF. Thus, repairing is equal.

2.5.4 Expanded PolyOlefin

Expanded PolyOlefin (EPO) is a molded plastic foam material that is created by engineering polyolefin into small beads and then using heat to mold it into different shapes. EPO is very durable and resilient to damage. When impacted, it has the ability to nearly return to its normal shape. Because of the polymer structure of EPO, it has an oily finish to its outer coating.

2.5.5 Expanded PolyStyrene

Expanded PolyStyrene (EPS) is often called Styrofoam which is a trademark. EPS is a plastic material composed of individual cells of low density polystyrene. EPS is very lightweight and can support many times its own weight in water. It is

commonly used in flotation devices, insulation, egg cartons, coffee cups and similar. It does however shatter easily.

2.5.6 Expanded PolyPropylene

[ARPRO] informs that Expanded PolyPropylene (EPP) is an engineered plastic foam material. By combining polypropylene resin with dust and applying heat, pressure and CO_2 in an autoclave, the material is formed into small plastic beads. These small, closed-cell foam beads are injected into a steam chest to create parts custom molded into complex shapes using steam heat and pressure. EPP is durable, light weight and recyclable. When put under force, the material will adapt and regain its former structure when force is removed. Of course, the material can be cut or teared, but is very resilient against breaking.

When damaged, only glue is needed for repairing. If the damage is to great to be repaired with glue or pieces are missing. The damaged area can be cut out and replaced by a new piece.

2.5.7 Discussion

The strongest material listed, in terms of force required before it gives is CF. It is probably the most expensive as well. If one ignore the price for a moment and consider the dangerous a carbon frame would introduce. A CF based frame would indeed be very robust, but a fast travelling thin wing in CF can be considered as a flying knife. This is necessary to obtain low drag, but during prototyping it might not be wise when many landings are required for testing with people nearby. Therefore, a foam based frame is advised for a prototype frame.

2.6 Propulsion

Everything put on planes outside the fuselage affects the aerodynamic of the plane. This highly applies to propulsion systems as its job is to use air friction to push it self forward. This does not only provide propulsion, but create turbulent air which may hit wings and induce drag. Moreover, airscrews and fans will induce drag if they are stationary while the plane is moving or the planes velocity is higher than the propelled speed. Thus, if rest of the airframe allows it, only a single motor is favoured.

In terms of reliability, efficiency, simplicity and noise handling for a small drone, an electrical propulsion system is advised. Due to its efficiency and high power/weight ratio; weight is spared. Furthermore, to convert torque to propulsion, an airscrew is a good choice as it has higher efficiency than a fan and can be folded such that less drag is induced [Houghton and Carpenter, 2003] when not powered. For what its worth, acceleration is usually higher on airscrews than on a fans in small scale.

2.6.1 Thrust vectoring

Thrust vectoring is the ability to manipulate the direction of thrust. In the last two decades, more and more fighter jets have been built with thrust vectoring capabilities. This give the planes additional rotational force when rapid movements are demanded, but also a way of controlling the plane in low airspeeds.

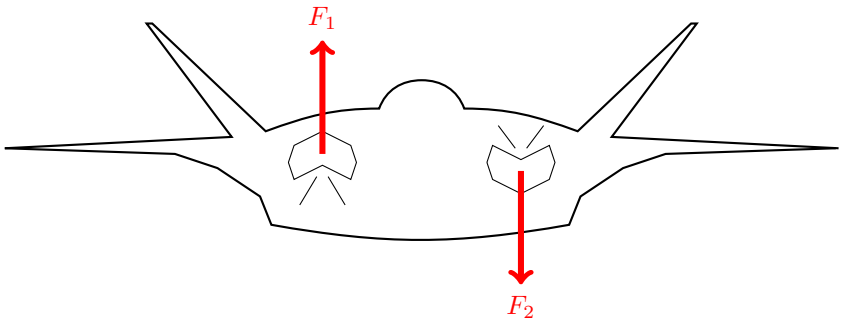


Figure 2.6: Lockheed Martin F-22 Raptor has thrust vectoring capabilities

One can argue whether thrust vectoring is frame or propulsion related as it incorporate control beyond surge direction. It is only mentioned in this context for completeness and to emphasise the possibilities this introduces if similar properties are required for a drone. If one consider long range aeroplanes, their airframe is constructed for its purpose and rapid movements is usually not a property. If this should become a property without sacrificing range capabilities, thrust vector is a valid option. It merely changes the plane by adding actuator on the thrusters without necessary changing the airframe itself.

2.6.2 Motor dimension

To be able to dimension the motor, Froudes momentum theory of propulsion will be used.

Froudes momentum theory of propulsion

Froude is given as

$$T = 2\rho S v^2 a(1 + a) \tag{2.2}$$

where T is thrust, ρ is air density, S is propeller circulatory area, v is velocity and a is inflow factor. To be able to use equation (2.2), a few assumptions will be made.

Choice of airscrew

Very often, a good way to design a model plane is by starting with the propeller type and size. Since the airframe already has been set, the propeller must be chosen to suit the plane. By using a two bladed folding propeller, less drag will be induced when gliding and less exposed for damage during belly landing.

Choosing an airscrew after plane size is hard because too small of a propeller require high revolutions per minute (RPM) and will make a lot of noise which is wasted energy. Too big propeller will induce moment of inertia which again can make the plane roll. Thus, practical experience or trial by error works and can save a lot of computations.

According to the plane size to be revealed in section 2.7, a propeller with diameter $d = 254mm$ will be suitable for this application. Propeller shape and pitch will not be elaborated due to its complexity.

Thrust

From appendix B, it can be assumed that total mass will be $m \approx 1500g$ (assumption 5.1). Calculating exact thrust required on an untested airframe is difficult. Assuming that the craft are to be pushed through air with an angle of attack $\alpha = 30°$ (assumption 5.2) and drag $C_D = 0.25$ (assumption 5.3), required thrust

can be estimated to

$$T = Gsin(\alpha)(1 + C_D) = mg\sin(\alpha)(1 + C_D) \approx 9.1969N \tag{2.3}$$

which is unrealistic because with air present, much of the planes weight would be "carried" by the air. Nonetheless, its a safe value to relay on.

Power

Furthermore, by assuming velocity to be $v = 20.0m/s$ (assumption 5.4) and that the craft is travelling at sea height where $\rho = 1.2928kg/m^3$ [Haugan, 1992] (assumption 5.5), Frodes momentum can be caluclated as

$$a^2 + a = \frac{T}{2\rho S v^2} = \frac{T}{\frac{1}{2}\rho d^2 v^2} \approx 0.5513 \tag{2.4}$$

and inflow can be found as

$$a \approx 0.3951 \tag{2.5}$$

Useful power is then given as

$$P_u = Tv = 183.9375W \tag{2.6}$$

and required power as

$$P_r = P_u(1 + a) \approx 256.61W \tag{2.7}$$

The actual power required by a practical airscrew would probably be about 15% greater than this [Houghton and Carpenter, 2003]. Therefore, true required power becomes

$$P_{tr} = P_r \cdot 1.15 \approx 295W \tag{2.8}$$

and with a cheap brushless motor usually having efficiency ratio of 80%, a motor with at least

$$P = P_{tr} \cdot 1.20 \approx 354W \tag{2.9}$$

is advised. However, brushless motors does not have a linearly efficiency ratio. Efficiency is not at maxima when using maximum power neither. Nevertheless, this was calculated when the motor is at 100% climbing 30° and when cruising, less power is required and efficiency ratio is most likely higher then.

Discussion

From this derivation, roughly 350W will be pulled from the battery when the craft is climbing with 30° angle at 72km/h. From a hobbyist point of view, the power estimation is sufficient to make a small UAV travel 72km/h. Due to the heat loss in an electronic speed controller (ESC) and the wires, a slightly higher power use is reasonable to assume. Still, several of the mentioned assumptions may not be valid and a constant use of 350W will not be sustainable for a long range plane. Nonetheless, the plane should not climb 30° continuously. Keep in mind that this derivation does not respect plane size, lift or propeller properties other than propeller size and may give surprises upon implementation.

For selecting brushed motors for a plane only based on its weight, several rule of thumb exists. Several decades ago, Dr. Keith Shaw created a rule for regular performance planes yielding 50-75W/lb and since brushless motors has nearly double efficiency, the rule can be derived to 25-38W/lb or 55-83W/kg. This indicate that a 125W motor should be sufficient for this application. This create a significant deviance with the previous found value. This is probably due to the assumption 5.3 and 5.4.

Even though electrical propulsion systems have many advantages, it is still beaten by petrol engines when it comes to range. If significant air time is required, a petrol engine will be a superior choice. As an example, the electrical UAV used by Lanny Lin and Morse [2010] claims to have an air time of two hours. [PC-Aero] claims their electric powered UAV has an endurance up to 8 hours which should be quite sufficient.

2.7 Build

Because some experiments had to be conducted for verification, an airframe was needed. Because of the assumptions from section 2.3.4, a frame advertised as ”Maxi Swift” was purchased. A superior frame could be constructed from scratch, but since EPP is more or less unavailable in Norway and appeared to be the best material, the purchased frame was rather modified.

Before the wings were glued, they were grinded down such that a small dihedral angle was present. Cyanoacrylate (CA), commonly known as super glue was used to melt the pieces together. The side fins was also glued with CA. Note that the CA used is foam safe meaning that if regular CA was used, the material would melt! One can argue about using epoxy in stead, but CA dries quick and was used in order to prevent wings from twisting during drying. Even though rest of the plane is somewhat flexible, it is desired to have a middle without flex. Which is why epoxy or CA is suitable.

Figure 2.7: All frame parts glued together. Notice how few frame parts there actually are

The glue area of the motor mount was rasped to stick better to the surface of the plane. Unfortunately, the motor mount is constructed weak and should eventually be replaced. In the mean while it has been augmented by a CF plate as illustrated in figure 2.8. It is also important to use thread lock on metal screws that enter metal. Thus, all metal in metal screws on this build is glued with thread lock.

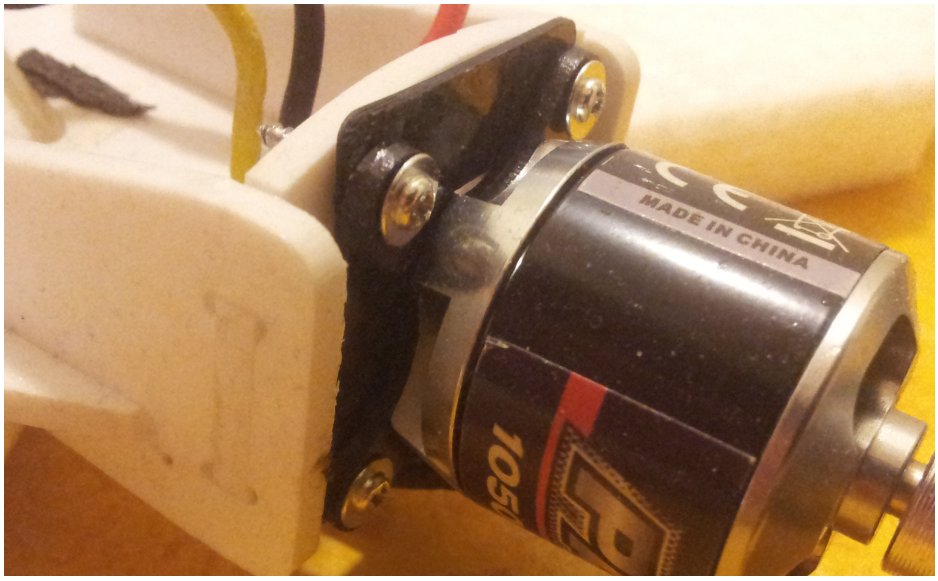


Figure 2.8: Motor mounted through carbon fiber plate

The slightly transparent part of figure 2.9 is a hinge of one control surface. Control horns must be mounted to each control surface to provide controllability. There are several ways of fastening the horns, but the horns delivered with the frame should be mounted vertically in the surfaces. This is done by cutting a line in the surface and gluing the horn in place. For this, epoxy glue is preferred over CA as it is thicker, stronger and not as stiff. It is important to have zero backlash for better accuracy and thus prevent flutter, but by using an epoxy which is slightly bendable when dry, the horn will be more secure when considering the entire application.

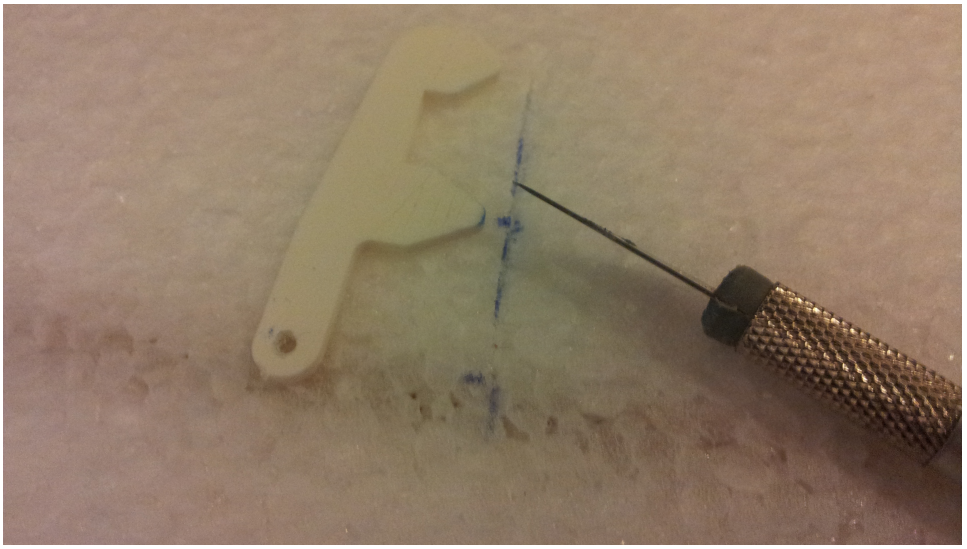


Figure 2.9: Mounting control horns, white horn shown to the left

Figure 2.10 shows battery compartment, servo slot and cutting of wire slot. A scalpel was used to cut these and a thin cut along the ruler is enough to house a servo wire. Both the battery and servo compartments was cut in desired size and depth before excavated. The servo link was attached to the servo using a turn able cylinder with a set screw where the screw was fastened with thread lock. The affordable servos was purchased with metal gear with robustness in mind. Unlike metal gear servos, common servos uses plastic gears which tend to break during various impacts.

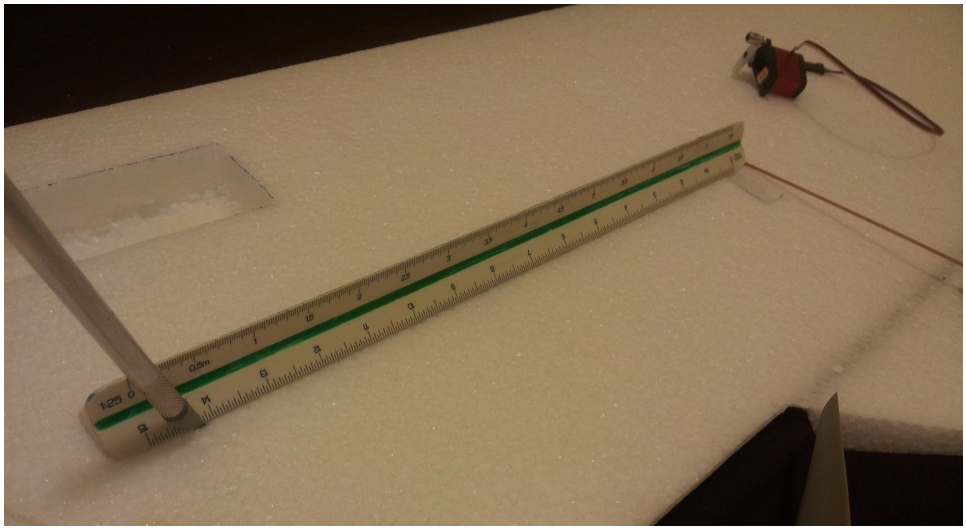


Figure 2.10: Installing actuator for right control surface

Figure 2.11 illustrate front of the plane with both batteries in place and the MARG sensor array. The sensor board is mounted using a double sided tape with foam to suppress vibrations. Ideally, it should be placed at CG, but due to disturbances from the motor it was placed between the batteries such that there was enough place in the nose to attach any desired camera. To cover a cut area

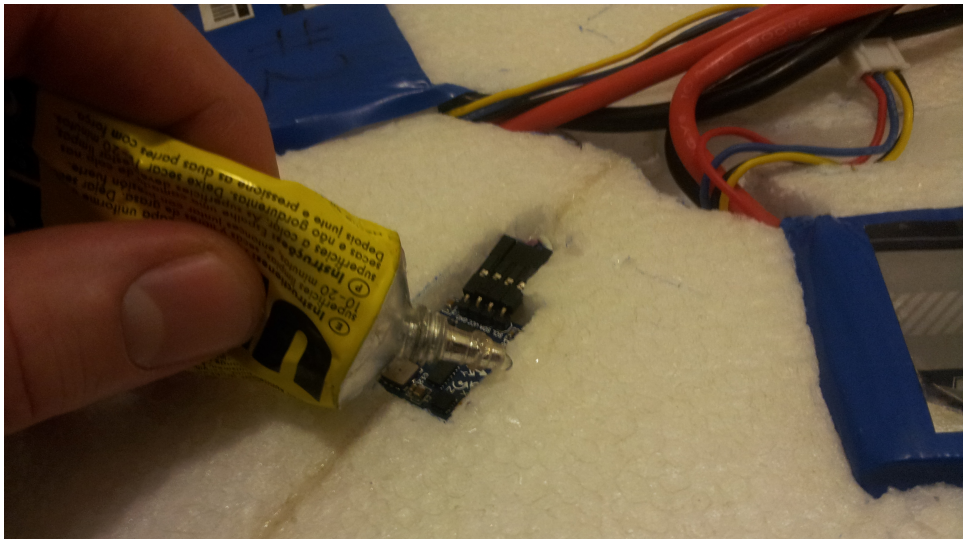


Figure 2.11: Covering MARG sensor

e.g. the MARG array in figure 2.11, it is common to use tape. Of course, other materials can be used, but tape is cheap, low weight, does not build any height and is simple to add and remove. However, attaching any common tape to EPP is tricky because EPP is a very dry and an uneven compound which is hard to attach to. Thus, by smudging glue around the sensor and let it nearly dry, tape can be added and will stick very good!

2.7.1 Wing test

The plane was first tested in air with only one battery pack (at middle) and appeared very light as it appeared to be floating around at times. By adding another battery pack it became more stable and somewhat a graceful flyer. More elevator input for increased angle of attack was however necessary to keep elevation. In both cases, the plane does appear to be suffering of noticeable drag.

Using a commercial Pitot-based telemetry system, maximum airspeed was measured to be 66km/h at 100% throttle over several levelled flybys.

2.7.2 Propulsion test

The plane is able to stay air born with about 60% throttle input. Using a wattmeter, static ground tests has shown that 60% throttle input results in 141W and 100% throttle in 265W. Since static tests can be seen as high drag, power consumption in air will most likely be lower than these measured values. With maximum airspeed close to assumption 5.4 of section 2.6, there is a noticeable deviation from the estimated power requirements.

2.7.3 Durability test

At least three EPP durability experiments was performed to test EPP capabilities. The first consisted of bending pieces of the plane to check breaking point and the ability to return to its former structure. The fins was bent nearly 180° and recovered without any signs of damage.

The second one which was conducted several times was belly landing. Every landing throughout the project was performed using belly landing expect cases when a camera was mounted underneath. The material did hold during all landings without any signs of injuries.

The final test was done by manually flying the plane about 20 meters above ground, pitch it down and turn off throttle. The nose hit the ground and the plane was shredded as illustrated in 2.12.

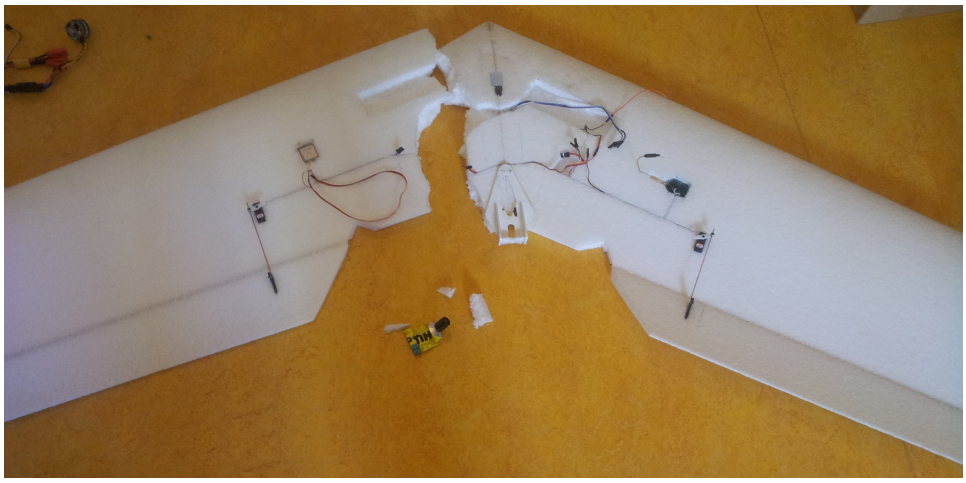


Figure 2.12: To be repaired

The plane was basically cut in half and three small additional pieces was separated from the main parts. Neither the nose or the plane show any signs of deformation which is quite surprising. The remaining pieces was an exact fit, fused perfectly together with glue and was back together in a few minutes.

2.7.4 Discussion

By knowing breaking point of this type of frame and material, durability and flexibility can be enhanced in several ways. When cutting routes and compartments in the foam it was noticed that the frame become more flexible an vulnerable. By adding glue in the routes after installing wires, the frame will most likely stiffen and become more rigid. Another reinforcement method would be to add carbon tubes inside wings. This keeps the outer durability, reduces the limit of flex and improves overall integrity. If this had been installed before test phases, it might have not been shredded at all. A final option is to cover the entire frame in lightweight tape as an outer coating preventing tearing of the material.

When considering the theoretical approximations and the actual values measured

it can be seen that the augmented brushless version Dr. Keith Shaw rule was off by only 16W (141W-125W). In contrast, the theoretical approximation was off by nearly 100W when ran at 100%. However, the approximated power requirement assumed 30° climb angle among other things. With 3S 11.1v 10Ah Lithium polymer (LiPo) batteries as power source a drone can stay airborne for about 45 minutes when motor is constantly pulling 141W. By adding more battery, thus weight, longer run times are expected, but would also require more energy and it becomes important to find the right battery-lift ratio for optimal travel distance. Disregard of these results and approximations, a final product with similar dimensions should use lesser energy to enhance range capabilities.

Of the shelf batteries was used in this project and by utilizing a supreme battery technology the flight range would increase. The research of Zhan Lin [2013] has recently developed a battery with four times the energy density compared to LiPo batteries which indicate four times longer flights for a SARD.

2.8 Discussion

A foam based flying wing design was easy to work with and EPP was proven resilient and easy to fix. Because the entire structure is made of foam, placement of components are simply cut in place. It is however a good idea to place the components with respect to CG and noise. Its toughness is a significant advantage when handled by floppy persons and for automatic landing procedures.

Choosing an airfoil with lesser drag and a slightly less swept design would be an overall better performing vehicle. For best possible range, long slender wings are preferred and can be constructed rigid enough by the mentioned ways.

Chapter 3

Attitude and Heading Reference System

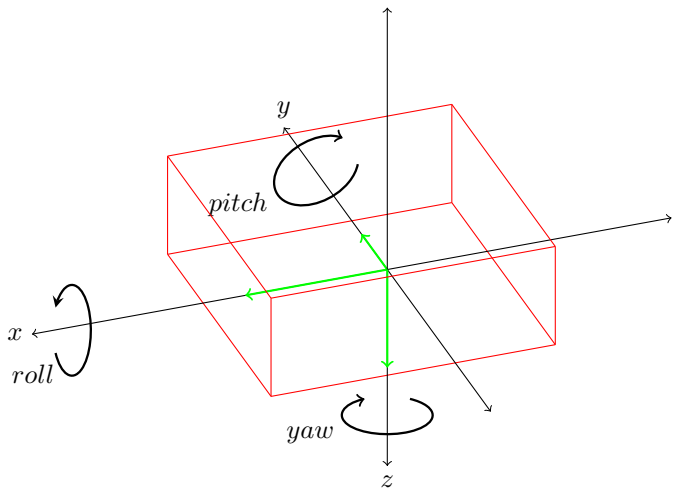


Figure 3.1: 6-DOF AHRS-box in an orthodox coordinate system

An AHRS is a system which uses sensors to tell its orientation in 3D-space (attitude) and how many degrees the vehicle is pointing away from the North pole (heading). Since this system tells the controllers which orientation a vehicle may have, accuracy and reliability must be significant. Such a system can be constructed in several ways; 6-DOF inertial measurement unit (IMU) with accelerometers and gyroscopes, 9-DOF IMU where additional magnetometers are added or GNSS with

three or more antennas. Of course, by combining all these, a better result can be achieved. Combining several sensors will affect weight, power consumption and price. High end sensors are usually large and expensive. Still, there exists micro electro mechanical systems (MEMS) which is less accurate, but lighter, less power hungry and much cheaper. The loss of accuracy is experienced in terms of noise and drifting. This is a tricky part and is basically solved with filters.

In both [Bjørntvedt, 2007] and [Skøien, 2011] carried out at NTNU, the IMU named Xsens MTi was used. Even though providing quality data Skøien concluded that 1750 Euro for a single unit was out of price range for a final product and that an own unit should be made. This is in totally agreement. Therefore, much time has been devoted to carry out this chapter where a MEMS based AHRS approach using magnetic, angular rate and gravity (MARG) sensor arrays will be derived.

3.1 Approach

By fixing a 3-axis accelerometer to a vehicle with zero translational velocity and let x-axis point forward (surge), y-axis to right (sway) and z-axis down (heave) a right-hand system is formed and its direction can be determined. Because all objects are accelerated towards the center of Earth a vector for the right-hand system can be computed. Imagine if the vehicle is perfectly level, x- and y-axes will have zero acceleration while z-axis will measure g ($9.81 m/s^2$). By changing attitude such that for example y- and z-axes have zero acceleration and x-axis is measuring g, the system knows that the vehicle is pointing downwards. Of course, the system is not this digital and in-between values are obtained such that the attitude can always be estimated.

Since acceleration is only along each axis, rapid movement around the axes are not measured. For this, a gyroscope is desirable. Unlike an accelerometer, a gyro measures zero value on all axes when perfectly still. When turned, angular velocity is measured.

A magnetometer measures the magnetic field vector relative to the sensor orientation. Since the earth has its own magnetic field, the measurement can be used as vector measurement to relate the body coordinate system to an earth fixed coordinate system. For enhanced bias drift for gyroscope and attitude reference, adding

a magnetometer to the mix may prove advantageous.

3.2 Equipment

To realize an AHRS, processor and sensors are needed. Due to a low budget, Teensy 3.0 by [Stoffregen, 2012] and a small 10 DOF breakout board was purchased. The latter connects all the sensors in an inter-integrated circuit (I^2C). Both these boards are open hardware which means that their schematic is freely available and can be found using appendix E.

The MARG array

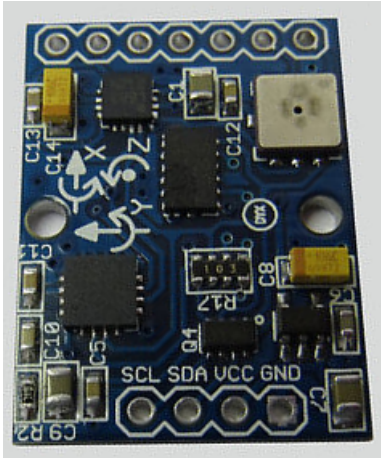


Figure 3.2: SD Card-size 10 DOF board

features the following sensors

- Analog Devices ADXL345 3-axis digital accelerometer
- STMicroelectronics L3G4200D 3-axis digital inertia sensor
- Honeywell HMC5883L 3-axis digital magnet compass
- Bosch BMP085 digital pressure sensor

which all operate in 3.3V range. The board does however accept an input voltage between 3-5V through an on-board regulator.

Teensy 3.0 features a 32-bit ARM Cortex-M4 48 MHz processor overclockable to

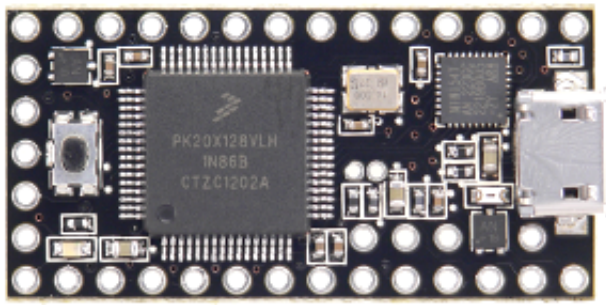


Figure 3.3: Teensy 3.0, Low-Cost 32 bit ARM-Cortex M4

96 MHz, 128 KB flash memory, 16 KB memory and 2 KB EPPROM. It operate on 3.3V logic level and has 34 input-ouput (I/O) pins where 12 accept analog and 10 are pulse width modulated (PWM). 3 universal asynchronous receiver/transmitter (UART), one I^2C and one serial peripheral interface bus (SPI) are present for external communication. Both circuits was mounted to a little cardboard box for a fixed mount, illustrated in figure 3.10 and used to test the sensors.

3.3 Sensors

The following sections derive generalized drivers with a template interface such that a new type of sensor can be changed with as little hassle as possible. Throughout the chapter, the code assumes that a physical connection between board and processor is present.

The driver routine is illustrated in figure 3.4 where the top most box may be acceleration, gyro or magnetometer.

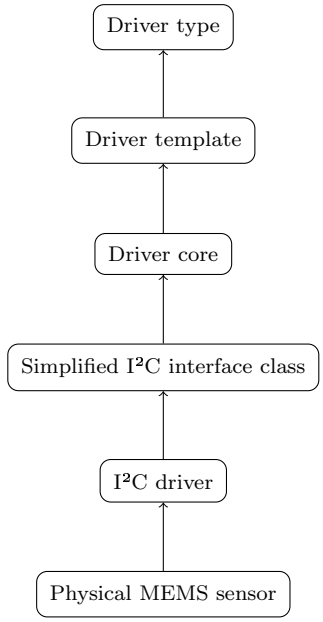


Figure 3.4: Driver routine

The continuing sections will derive a simpler I²C interface and sensor drivers. Driver cores is presented first followed by a generalised template driver.

3.3.1 I²C functions

For convenience, a common class for I²C actions was written. The class uses the basic functions from the I²C driver library TwoWire, written by Nicholas Zambetti.

The sensors are register based which means that data are read and written to registers inside the sensors. The TwoWire library implements an I²C protocol to read and write to these registers. The registers has a fixed address usually defined in hexadecimal and are found in the datasheet for the specific sensor.

When its desirable to change settings in the sensor or request changes, a package is created with the address of the sensor, register address and determined register content. This function is written as

Listing 3.1: I2C.cpp excerpt - write()

```
1 void I2C::write(uint8_t device, uint8_t address, uint8_t code)
2 {
3   Wire.beginTransmission(device);
4   Wire.write(address);
5   Wire.write(code);
6   Wire.endTransmission();
7 }
```

where sensor-address *device*, register address *address* and register contents *code* are given as arguments.

To read data from the registers the sensor address and register address is sent. Note that this time no additional content is sent which is perceived as a requesting action, telling the sensor to populate the registers with new data. It is common that a single register holds 1 byte of data. It is possible to read across registers by requesting data from *address* and then reading more than 1 byte. In listing 3.2 the variable *numberOfRegisters* determines the number of registers to be read.

Listing 3.2: I2C.cpp excerpt - read()

```
1  uint8_t * I2C::read(uint8_t device, uint8_t address, int numberOfRegisters)
2  {
3    /// tell sensor which address we would like to access
4    Wire.beginTransmission(device);
5    Wire.write(address);
6    Wire.endTransmission();
7
8    /// get register contents
9    Wire.beginTransmission(device);
10   Wire.requestFrom((int)device, numberOfRegisters);
11
12   int i = 0;
13   uint8_t buffer[numberOfRegisters];
14   while(Wire.available())
15   {
16     buffer[i] = Wire.read(); /// receive one uint8_t
17     i++;
18   }
19   Wire.endTransmission();
20
21   return buffer;
22 }
```

By calling the *read*()-function, data from a sensor is returned in an array pointer, ready to be processed.

3.3.2 Acceleration driver

The first function for ADXL345 is

Listing 3.3: ADXL345.cpp excerpt - initialize()

```
1 void ADXL345::initialize()
2 {
3   _i2c.write(_DEVICE, _DATA_FORMAT, 0x08); /// Put sensor in full resolution and +/-
        2g
4   _i2c.write(_DEVICE, _POWER_CTL, 0x0B); /// Put sensor in measure mode (turn on)
5 }
```

which sets resolution, how many g it is expected to measure and turns the accelerometer on by putting it in continuous measuring mode. It is common that the sensors boot into idle mode and must therefore be switched to measuring mode. The variables with capital letters and the object $\_i2c$ are defined in the corresponding C++ header. The capital letter variables contains register addresses in hexadecimal numbers.

To provide a simple interface for setting expected gravitational forces, the function in listing 3.4 takes integer input for the desired setting.

Listing 3.4: ADXL345.cpp excerpt - setG()

```
1  bool ADXL345::setG(int g)
2  {
3    /// values from datasheet
4    int gravity[4] = {2, 4, 8, 16};
5    uint8_t codes[4] = {00, 01, 10, 11};
6
7    bool ok = false;
8    int i;
9    for (i=3; i >= 0; i--)
10   {
11     if (gravity[i] == g)
12     {
13       ok = true;
14       break;
15     }
16   }
17
18   if (!ok)
19   {
20     return false;
21   }
22
23   /// write the desired rate to sensor
24   _i2c.write(_DEVICE, _DATA_FORMAT, codes[i]);
25
26   return true;
27 }
```

The input value is matched against $gravity[4]$ which holds supported accelerations. If the desired settings is not present in this array, the function returns false

to indicate that the setting did fail.

The final interface in this case is listing 3.5 which sets desired update rate according to array *hertz*[16]. Even though these values are hard coded, it is acceptable for a driver.

Listing 3.5: ADXL345.cpp excerpt - setHz()

```
1  bool ADXL345::setHz(float hz)
2  {
3    /// values from the datasheet
4    float hertz[16] = {3200, 1600, 800, 400, 200, 100, 50, 25, 12.5, 6.25, 3.13, 1.56,
           0.78, 0.39, 0.20, 0.10};
5    float bandwidth[16] = {1600, 800, 400, 200, 100, 50, 25, 12.5, 6.25, 3.13, 1.56,
         0.78, 0.39, 0.20, 0.10, 0.05};
6    uint8_t codes[16] = {1111, 1110, 1101, 1100, 1011, 1010, 1001, 1000, 0111, 0110,
         0101, 0100, 0011, 0010, 0001, 0000};
7
8    bool ok = false;
9    int i;
10   for (i=15; i >= 0; i--)
11   {
12     if (hertz[i] == hz)
13     {
14       ok = true;
15       break;
16     }
17   }
18
19   if (!ok)
20   {
21     return false;
22   }
23
24   /// write the desired rate to sensor
25   _i2c.write(_DEVICE, _BW_RATE, codes[i]);
26
27   return true;
28 }
```

Raw acceleration values are obtained by calling

Listing 3.6: ADXL345.cpp excerpt - measure()

```
1  int16_t * ADXL345::measure()
2  {
3    /// there are 6 registers for acceleration data, we start by reading from register
         DATAX0 and pluss 5
4    uint8_t * b = _i2c.read(_DEVICE, _DATAX0, 6);
5
6    this->_raw[0] = (b[1] << 8) | b[0];
7    this->_raw[1] = (b[3] << 8) | b[2];
8    this->_raw[2] = (b[5] << 8) | b[4];
9
10   this->_raw[0] *= -1;
11
12   return this->_raw;
13 }
```

which returns a three sized array pointer. Considering acceleration along x-axis on line 5., the most significant bit (MSB) $b[1]$ has to be shifted to the left because least significant bit (LSB) $b[0]$ is added afterwards. This is repeated for y- and z-axes.

The careful reader will notice that the raw values are not common acceleration since two bytes of data give maximum value of 65536. For this accelerometer, one g takes 8 bit. Therefore, a conversion will happen over in the generalized interface *Acceleration*, starting with listing 3.7 where the six parameters are in fact the conversions values.

Listing 3.7: Acceleration.cpp excerpt - initialize()

```
1 template <typename Driver> void Acceleration<Driver>::initialize(float xMin, float
      xMax, float yMin, float yMax, float zMin, float zMax)
2 {
3   /// Set calibration data. These numbers has been found by collecting raw data
        during slowly varying movements. 9.81 m/s^2 is earths gravitational
        acceleration.
4   _calibrationGainMin[0] = xMin; /// Smallest measured raw value
5   _calibrationGainMax[0] = xMax; /// Highest value
6   _calibrationGainMin[1] = yMin;
7   _calibrationGainMax[1] = yMax;
8   _calibrationGainMin[2] = zMin;
9   _calibrationGainMax[2] = zMax;
10
11  if (_g == 0)
12  {
13    _g = 2; /// default value
14  }
15
16  _driver.initialize();
17 }
```

Lets say the accelerometer is mounted with the axes in the same directions as in Figure 3.1. While stationary, the z-axis should measure 256 at maximum. Thus, the conversion value ($\_calibrationGainMin$) will simply be 9.81/256. Unfortunately, not all the axes are perfect and the maximum and minimum values must be found manually. If ADXL345 has been selected as core driver, then line 16. calls $ADXL345 :: initialize()$ and the accelerometer becomes ready.

For cases where the accelerometer is not mounted perfectly, biases must be added and manual bias setting can be done using

Listing 3.8: Acceleration.cpp excerpt - bias()

```
1 template <typename Driver> void Acceleration<Driver>::bias(float x, float y, float z
      )
2 {
3   /// Calibration found once(!) by positioning the accelerometer perfectly level and
        record readings
4   _bias[0] = x; /// X
5   _bias[1] = y; /// Y
6   _bias[2] = z; /// Z
7 }
```

and must have raw value input, not acceleration. The "template <typename Driver>" declaration define which driver core the *Acceleration* class should use as a driver object and is stored in *_driver*. This piece generalises the class such that only driver class name is required when creating an Acceleration object. It can be declared by called similar to listing 3.9.

Listing 3.9: Template decleration

```
1 Acceleration<ADXL345> myObject;
```

An automatic bias estimation routine has been written in listing 3.10 where several values are recorded over a time and averaged.

Listing 3.10: Acceleration.cpp excerpt - calibrate()

```
1  template <typename Driver> bool Acceleration<Driver>::calibrate(bool calibrateXaxis,
       bool calibrateYaxis, bool calibrateZaxis, unsigned int threshold, unsigned int
       samples)
2  {
3    int16_t sum[3] = {0,0,0};
4    int16_t prevVal[3] = {0,0,0};
5    int16_t * raw;
6    int moveCounter  = 0;
7
8    for(int i=0; i<samples;i++)
9    {
10     raw = _driver.measure();
11
12     /// Detect and count movement
13     if (abs(prevVal[0] - raw[0]) > threshold || abs(prevVal[1] - raw[1]) > threshold
           || abs(prevVal[2] - raw[2]) > threshold)
14     {
15       moveCounter++;
16     }
17     else
18     {
19       sum[0] += raw[0];
20       sum[1] += raw[1];
21       sum[2] += raw[2];
22     }
23     prevVal[0] = raw[0];
24     prevVal[1] = raw[1];
25     prevVal[2] = raw[2];
26     delay(1); /// must wait due to fast loop, sleeping is permitted here since this
           is an initation routine
27
28     /// if 10% movement has occurred, discontinue calibration
29     if ((float)moveCounter/samples >= 0.1)
30     {
31       return false;
32     }
33   }
34   if (calibrateXaxis) _bias[0] = sum[0] / (float) samples;
35   if (calibrateYaxis) _bias[1] = sum[1] / (float) samples;
36   if (calibrateZaxis) _bias[2] = sum[2] / (float) samples;
37
38   return true;
39 }
```

It is important that the sensor is stationary during this process. To prohibit movement in this phase, line 13. features a movement detector which triggers a counter when movement exceeds a given *threshold*. The calibration is aborted when counter reaches 10% of the desired *samples*. Additional three optional input arguments are given to decide which axes whom should be affected by the calibrated biases. It is common that the z-axis is not a part of the calibration routine. Notice that line 10. calls the driver core and extract raw acceleration data.

For the enforcer, the *measure*()-function in listing 3.11 reads accelerometer data and cycles the first in first out (FIFO) stack.

Listing 3.11: Acceleration.cpp excerpt - measure()

```
1  template <typename Driver> void Acceleration<Driver>::measure()
2  {
3    int16_t * buffer = _driver.measure();
4
5    float x = buffer[0] - _bias[0];
6    float y = buffer[1] - _bias[1];
7    float z = buffer[2] - _bias[2];
8
9    /// Convert to m/s^2. Max and min values for each axis is different, therefore
         different gain will be applied.
10   x *= (x < 0 ? _calibrationGainMin[0] : _calibrationGainMax[0] );
11   y *= (y < 0 ? _calibrationGainMin[1] : _calibrationGainMax[1] );
12   z *= (z < 0 ? _calibrationGainMin[2] : _calibrationGainMax[2] );
13
14   /// do not accept values outside the configured gravity scope
15   float threshold = (float)_g * 9.81 * 1.1;
16   if (abs(x) > threshold) x = _xAxis[0];
17   if (abs(y) > threshold) y = _yAxis[0];
18   if (abs(z) > threshold) z = _zAxis[0];
19
20   /// FIFO stack
21   _xAxis[2] = _xAxis[1];
22   _xAxis[1] = _xAxis[0];
23   _xAxis[0] = x;
24   _yAxis[2] = _yAxis[1];
25   _yAxis[1] = _yAxis[0];
26   _yAxis[0] = y;
27   _zAxis[2] = _zAxis[1];
28   _zAxis[1] = _zAxis[0];
29   _zAxis[0] = z;
30 }
```

This stack has been implemented to feature a simple interface for filters. Bias and minimum/maximum conversions are executed here. Since à priori information is given through *setG*() a simple filter has been applied on lines 15-18. in case the accelerometer should experience unexpected noise.

To obtain the accelerations publicly, the functions

Listing 3.12: Acceleration.cpp excerpt

```
1  template <typename Driver> float Acceleration<Driver>::x(int index)
2  {
3    return axis(0,index);
4  }
5  template <typename Driver> float Acceleration<Driver>::y(int index)
6  {
7    return axis(1,index);
8  }
9  template <typename Driver> float Acceleration<Driver>::z(int index)
10 {
11   return axis(2,index);
12 }
```

are available where the paramenter *index* ranges from 0-2 depending on desire for acceleration, velocity or position. Do remember to integrate the values before using them as velocity and positions. *axis*() is merely a generalized function to access the stacks and can be found in the original file along with additional *roll*() and *pitch*() functions.

A simple realization was carried out such that only gravitational acceleration was measured. The transitions was rotated in 90° rotations always having the desired axis pointing towards earth. The recorded data is presented in figure 3.5. Notice the unexpected behaviour for z-axis about 4 seconds in. This value might have been greater, but has been filtered because expected acceleration was set to 2g using *setG*().

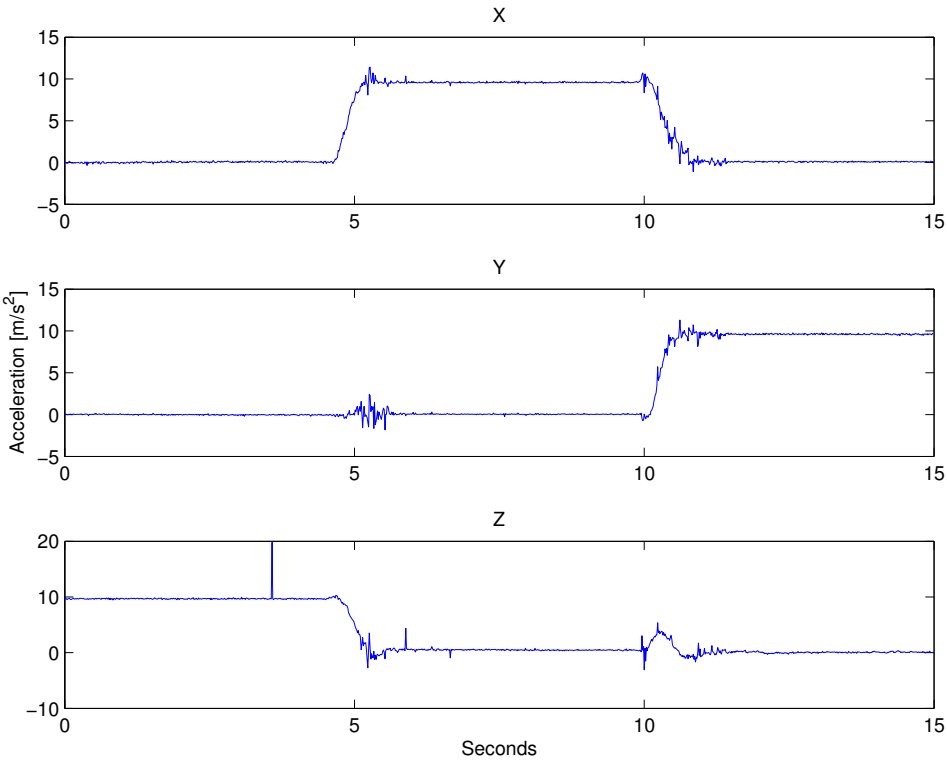


Figure 3.5: Acceleration while changing transition 90° from Z to X to Y

3.3.3 Gyroscope driver

The gyroscope L3G4200D differs slightly from the accelerometer driver and requires to be queried about address to start working. This is the *why* of lines 3-6. in listing 3.13 followed by the next lines which fills the registers with configuration data.

Listing 3.13: L3G4200D.cpp excerpt - initialize()

```
1  bool L3G4200D::initialize()
2  {
3    if (_i2c.read(_DEVICE, _WHO_AM_I, 1)[0] != 0xD3) /// Sensor must have WHO_AM_I
         before power on
4    {
5      return false;
6    }
7
8    _i2c.write(_DEVICE, _CTRL_REG1, 0x8F);  /// Power on, enable all axes and put in
         400 Hz mode
9    _i2c.write(_DEVICE, _CTRL_REG2, 0x00);  /// Disable all filters
10   _i2c.write(_DEVICE, _CTRL_REG3, 0x08);  ///
11   _i2c.write(_DEVICE, _CTRL_REG4, 0x30);  /// Set block mode, LSB, 2000 dps, disable
         self test and 4-wire I2C interface
12   _i2c.write(_DEVICE, _CTRL_REG5, 0x00);  ///
13
14   return true;
15 }
```

From the datasheet it can be verified that $CTRL\_REG1$ holds configuration for output data rate, bandwidth, idle/enable and which axes to be active. Secondly, $CTRL\_REG2$ holds internal high pass filter configuration. Thirdly, $CTRL\_REG3$ has miscellaneous external interrupt activation. Furthermore, $CTRL\_REG4$ selects endian type, resolution, self-test and interface (SPI/I²C). Finally, $CTRL\_REG5$ has switch options for reboot, FIFO, interrupts and high pass filters.

Listing 3.14 returns angular velocity in similar manner to the function for the accelerometer driver, but hard codes a simple orientation switch on lines 10-11.

Listing 3.14: L3G4200D.cpp excerpt - measure()

```
1  int16_t * L3G4200D::measure()
2  {
3    uint8_t * b = _i2c.read(_DEVICE, _OUT_X_L | (1 << 7), 6);
4
5    this->_raw[0] = (b[1] << 8) | b[0];
6    this->_raw[1] = (b[3] << 8) | b[2];
7    this->_raw[2] = (b[5] << 8) | b[4];
8
9    /// sort out orientation according to a right hand system
10   this->_raw[1] *= -1;
11   this->_raw[2] *= -1;
12
13   return this->_raw;
14 }
```

Because the gyro has z-axis pointing upwards, both z- and y-axes must be flipped according to a right hand system such that same orientation as in figure 3.1 is present.

The generalised driver has an initializing routine

Listing 3.15: Gyro.cpp excerpt - initialize()

```
template <typename Driver> bool Gyro<Driver>::initialize(float gain)
{
  this->_gain = gain;

  if (_driver.initialize())
  {
    return true;
  }

  return false;
}
```

where gain is an input argument. The gain is first applied in listing 3.16 and is selected such that output become degree per second.

Listing 3.16: Gyro.cpp excerpt - axis()

```
template <typename Driver> float Gyro<Driver>::axis(int axis,int index)
{
  /// do not get exceed array size
  if (index > 2)
  {
    index = 0;
  }

  if (axis == 1)
  {
    return _yAxis[index]*this->_gain;
  }
  else if (axis == 2)
  {
    return _zAxis[index]*this->_gain;
  }
  else
  {
    return _xAxis[index]*this->_gain;
  }
}
```

The rest of the available functions such as manual bias setting, calibration and more are merely functions identical to acceleration-class and therefore omitted.

Similar to the accelerometer, the code was realized and recorded data is plotted in figure 3.6, illustrating the performance of the gyro and driver. 90° transitions occurs from 2.5s for z-axis, 7.5s for x-axis and 12s for y-axis. Clearly, as the presented code did show, this driver does not feature any filter to prohibit unwanted

noise which is present in the plot. The peaks shown in figure 3.6 is most likely filterable with a low pass filter and should be added later.

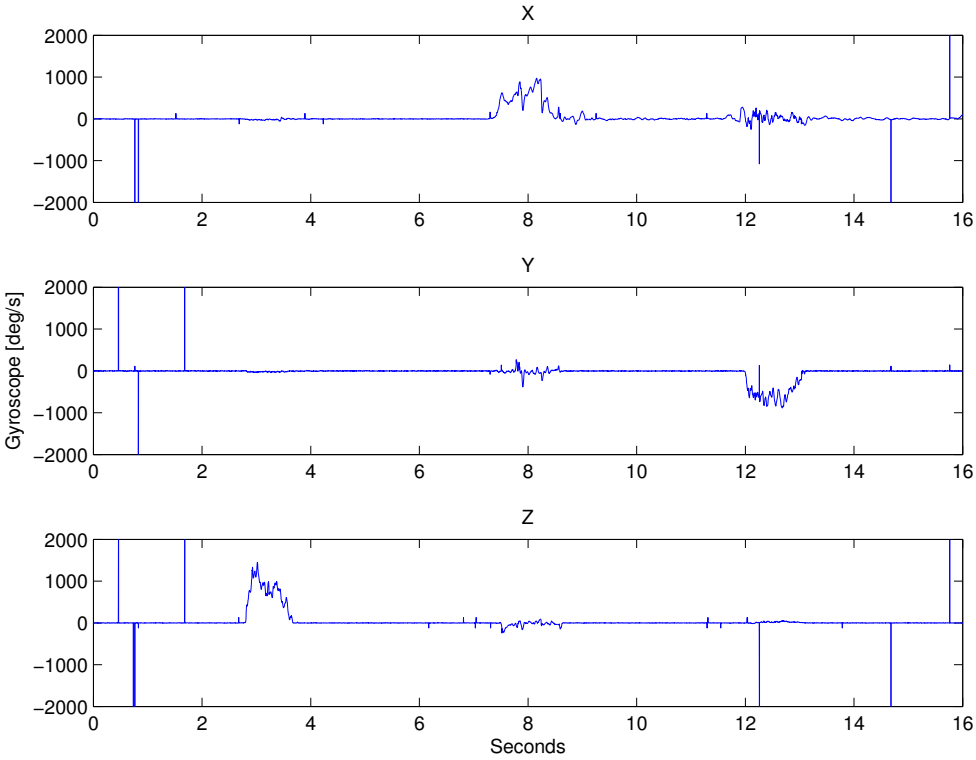


Figure 3.6: Values recorded over 16s while changing 90° orientation from Z to X to Y

3.3.4 Magnetometer driver

According to the datasheet and practical verification, the registers on the HMC5883L magnetometer has been flipped. In listing 3.17 it can be seen that z- and y-registers are switched.

Listing 3.17: HMC5883L.cpp excerpt - measure()

```
1  int16_t * HMC5883L::measure()
2  {
3    uint8_t * b = _i2c.read(_DEVICE, _DATA_OUTPUT_X_MSB, 6); /// read from
         DATA_OUTPUT_X_MSB and the next 5 registers
4
5    this->_raw[0] = (b[0] << 8) | b[1]; /// X
6    this->_raw[2] = (b[2] << 8) | b[3]; /// Z - note the order
7    this->_raw[1] = (b[4] << 8) | b[5]; /// Y
8
9    /// set orientation
10   this->_raw[1] *= -1;
11   this->_raw[2] *= -1;
12
13   return _raw;
14 }
```

The initiation routine in listing 3.18 basically exists of populating processor memory with settings and sending settings to the magnetometer.

Listing 3.18: HMC5883L.cpp excerpt - initialize()

```
1  void HMC5883L::initialize()
2  {
3    _i2c.write(_DEVICE, _REG_A, 0x70); /// what the hell does this do!?
4
5    /// have to populate this way to make sure it works in C++ and C++11
6    int a[8] = { 1370, 1090, 820, 660, 440, 390, 330, 230 };
7    uint8_t b[8] = { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07 };
8    for (int i=0;i<8;i++)
9    {
10     _gainGauss[i] = a[i];
11     _gainAddress[i] = b[i];
12   }
13 }
```

It is not ideal, but due to different compilers on different systems its a working way.

Several functions for settings measuring mode and flux-gain are present. The functions contains only a single line of code for settings and is commented good enough in the code to be left for the curious reader in appendix E.

Due to the difficulty when manufacturing digital magnetometers, there are introduced unwanted biases [Val´rie Renaudin and Lachapelle, 2010]. The sensor is therefore equipped with internal bias compensation, a self test. The test applies positive and negative biases to each axis in the physical world. Recording measure-

ments during the test, the true biases can be estimated and applied later. The self test implementation in listing 3.19 supports this, but has also an additional loop which iterate through the supported gains in case of compass becomes saturated.

Listing 3.19: HMC5883L.cpp excerpt - selfTest()

```
bool HMC5883L::selfTest(int samples, int startGain)
{
  _i2c.write(_DEVICE, _REG_A, 0x71);
  int gain;

  int16_t * buffer;

  /// loop through all gains in case of saturation
  for (gain = startGain; gain < 8; gain++)
  {
    long int lowLimit = _SELF_TEST_POSITIVE_LOW *_gainGauss[gain];
    long int highLimit = _SELF_TEST_POSITIVE_HIGH*_gainGauss[gain];
    long int total[3] = {0,0,0};

    setGain(gain);
    continuous();
    delay(6);

    applyPositiveBias();
    for (int i=0;i<samples;i++)
    {
      buffer = measure();

      total[0] += buffer[0];
      total[1] += buffer[1];
      total[2] += buffer[2];
      delay(1);
    }

    int sum[3];
    sum[0] = total[0]/(samples*1);
    sum[1] = total[1]/(samples*1);
    sum[2] = total[2]/(samples*1);

    if (sum[0] >= lowLimit && sum[0] <= highLimit && sum[1] >= lowLimit && sum[1] <=
        highLimit && sum[2] >= lowLimit && sum[2] <= highLimit)
    {
      removeBias();
      return true; /// saturation not found
    }

    /// saturation found, increase gain
  }

  return false;
}
```

Notice the delays in the code. These are present to pause the program such that the magnetometers get enough time to populate the registers with new and preferably different measurements. Note that this is not a complete calibration routine.

Calibrating magnetometers is not a trivial subject. Unlike accelerometers and gyroscopes, magnetometers must be mounted in their expected environment and

exposed to rotational movements such that all axes reaches minimum and maximum flux. Nearby objects often referred to as hard- and soft-iron error sources adds biases to the magnetometers. This is why the sensors must be in their expected environment (e.g. airframe) when calibrating. A calibration like this should be implemented in the generalised driver, but is not at this point.

The generalised template driver which lays on top of the core driver has even simpler routines than accelerometer and gyroscopes. Most of the functions are more or less directly routed through the template like the initiation routine

Listing 3.20: Compass.cpp - initialize()

```
1 template <typename Driver> void Compass<Driver>::initialize()
2 {
3   _driver.initialize();
4 }
```

and measuring

Listing 3.21: Compass.cpp excerpt - measure()

```
1  template <typename Driver> void Compass<Driver>::measure()
2  {
3    int16_t * raw = _driver.measure();
4
5    /// FIFO stack
6    _xAxis[2] = _xAxis[1];
7    _xAxis[1] = _xAxis[0];
8    _xAxis[0] = raw[0];
9    _yAxis[2] = _yAxis[1];
10   _yAxis[1] = _yAxis[0];
11   _yAxis[0] = raw[1];
12   _zAxis[2] = _zAxis[1];
13   _zAxis[1] = _zAxis[0];
14   _zAxis[0] = raw[2];
15 }
```

Setting continuous, periodic and idle measuring mode are also directly routed thus omitted here. Same goes for axes values for x, y, z which are called like functions similar to the other drivers.

The main purpose of magnetometers, or compasses are in fact to find the true heading angle. This angle can be found as

$$\psi = \arctan 2(y, x) + declination \tag{3.1}$$

where y and x is magnetic flux along y- and x-axes, respectively. This does however mean that the z-axis is pointing toward center of earth. The bias *declination* is really magnetic declination and is the angle between magnetic north and true

north. This is a fact because the magnetic poles are not lined up with the axis which the earth spins around. Since declination is not constant and varies over time, new updates should be obtained on a annually basis.

Declination compensation is naturally important for full scale planes which travel over large distances. For this reason, an almanac of declinations has not been implemented as the drones are not expected to be travelling very far.

With flux measurements available on y- and x-axes, equation 3.1 has been implemented in listing 3.22.

Listing 3.22: Compass.cpp excerpt - heading()

```
1  template <typename Driver> float Compass<Driver>::heading(float declination) /// psi
2  {
3    float heading = ( atan2(y(), x()) + declination)*(180/M_PI); /// convert to
         degrees
4
5    /// correct for when signs are reversed.
6    if (heading < 0.0)
7    {
8      heading += 360;
9    }
10   /// check for wrap due to addition of declination.
11   else if (heading > 360.0)
12   {
13     heading -= 360;
14   }
15
16   return heading;
17 }
```

To verify the sensors behaviour the driver was tested by turning the sensor 360° on the bench and recording its values. The recorded data has been plotted in figure 3.7 where it can be seen that this pairing creates unwanted spikes as well.

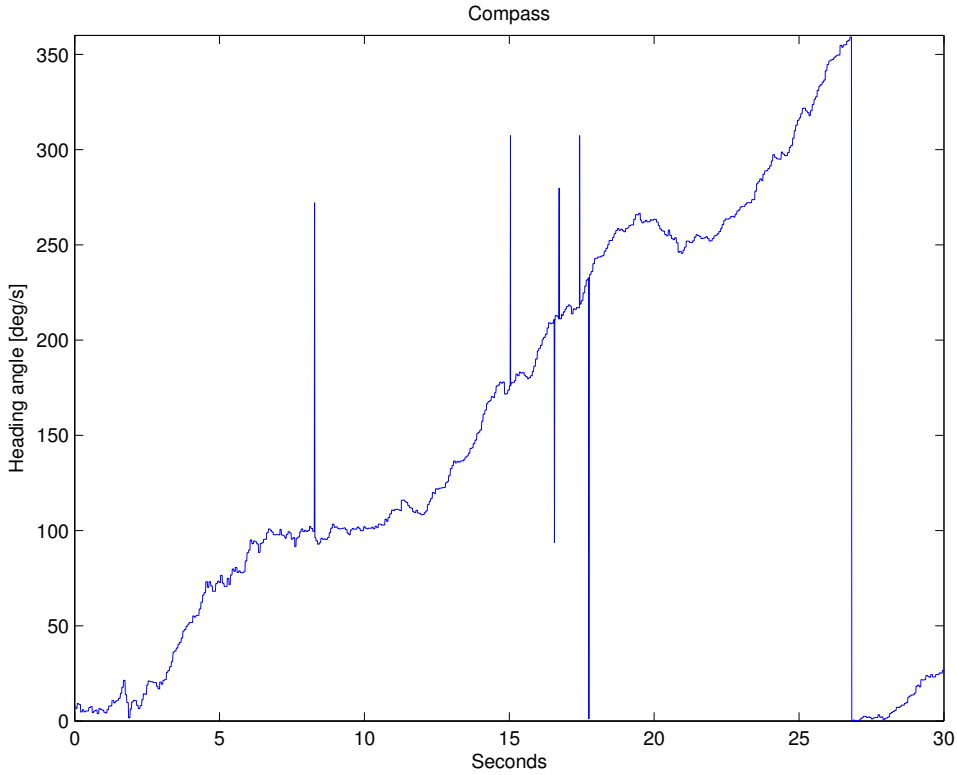


Figure 3.7: Turning sensor 360°

3.4 Orientation filter

It is very common to use a Kalman filter for fusing several sensors to form an AHRS. The computational power required to calculate the gain-matrix for the Kalman filter is not always appealing when using micro controllers. Therefore, the more modern adaptive controller algorithm by [Madgwick, 2010] will be used beyond. Both its algorithms will be expressed in C++ and used as main filter for this AHRS. The author claims that the filter support frequencies down to 10 Hz. [Madgwick, 2010] features two filters, one for tri-axis gyroscopes and accelerometers, and a MARG solution. The angular rate and gravity (ARG) solution will be presented first in 3.4.2.

3.4.1 Derivation

Theory for the filters presented in [Madgwick, 2010] will be briefly explained for completeness in this section.

With an available quaternion estimation and angular rate obtained from the gyroscope, the rate of orientation in earth frame relative to body (where gyroscope is fixed) can be expressed as

$$\dot{\mathbf{q}}_e^b(\omega, t) = \frac{1}{2}\hat{\mathbf{q}}_e^b \otimes \boldsymbol{\omega}^b(t) \tag{3.2}$$

where

$$\boldsymbol{\omega}^b = \begin{bmatrix} 0 & \omega_x & \omega_y & \omega_z \end{bmatrix} \tag{3.3}$$

is gyroscope measurements and

$$\hat{\mathbf{q}}_e^b = \begin{bmatrix} q_1 & q_2 & q_3 & q_4 \end{bmatrix} \tag{3.4}$$

is quaternion representation. The superscript ^{b} refers to the body frame where sensor is mounted and subscript $_e$ to the relative earth frame.

Running the filter at a known frequency, the sampling period Δt is known and orientation can be found as

$$\mathbf{q}_e^b(\omega, t) = \hat{\mathbf{q}}_e^b + \dot{\mathbf{q}}_e^b(\omega, t)\Delta t \tag{3.5}$$

Since quaternion representation requires a complete solution to be found, a optimisation problem can be defined as

$$\min_{\mathbf{q}_e^b \in R^4} \mathbf{f}(\hat{\mathbf{q}}_e^b, \mathbf{a}_b, \mathbf{d}_e) \tag{3.6}$$

$$\mathbf{f}(\hat{\mathbf{q}}_e^b, \mathbf{a}_b, \mathbf{d}_e) = {}^*\hat{\mathbf{q}}_e^b \otimes \mathbf{d}_e \otimes \hat{\mathbf{q}}_e^b - \mathbf{a}_b \tag{3.7}$$

where

$$\mathbf{a}_b = \begin{bmatrix} 0 & a_x & a_y & a_z \end{bmatrix} \tag{3.8}$$

is acceleration measurements and

$$\mathbf{d}_e = \begin{bmatrix} 0 & d_x & d_y & d_z \end{bmatrix} \tag{3.9}$$

is a predefined reference direction of a field in any direction.

If its assumed that there are only components within two axes, the expression simplifies after the gradient descent algorithm has been used. The equations can then be written as

$$\mathbf{f}_g(\hat{\mathbf{q}}_e^b, \hat{\mathbf{a}}) = \begin{bmatrix} 2(q_2 q_4 - q_1 q_3) - a_x \\ 2(q_1 q_2 + q_3 q_4) - a_y \\ 2(\frac{1}{2} - q_2^2 - q_3^2) - a_z \end{bmatrix} \tag{3.10}$$

$$\mathbf{J}_g(\hat{\mathbf{q}}_e^b) = \begin{bmatrix} -2q_3 & 2q_4 & -2q_1 & 2q_2 \\ 2q_2 & 2q_1 & 2q_4 & 2q_3 \\ 0 & -4q_2 & -4q_3 & 0 \end{bmatrix} \tag{3.11}$$

where $\mathbf{d}_e$ has vanished and $\hat{\mathbf{a}}$ is normalised accelerometer measurement.

Incorporating magnetometer measurements by considering that magnetic field of earth has components in one horizontal axis and the vertical axis, the expression

yields

$$\mathbf{f}_b(\hat{\mathbf{q}}_e^b, \hat{\mathbf{b}}_e, \hat{\mathbf{m}}_e) = \begin{bmatrix} 2b_x(\frac{1}{2} - q_3^2 - q_4^2) + 2b_z(q_2q_4 - q_1q_3) - m_x \\ 2b_x(q_2q_3 - q_1q_4) + 2b_z(q_1q_2 + q_3q_4) - m_y \\ 2b_x(q_1q_3 + q_2q_4) + 2b_z(\frac{1}{2} - q_2^2 - q_3^2) - m_z \end{bmatrix} \tag{3.12}$$

$$\mathbf{J}_b(\hat{\mathbf{q}}_e^b, \hat{\mathbf{b}}_e) = \begin{bmatrix} -2b_zq_3 & 2b_zq_4 & -4b_xq_3 - 2b_zq_1 & -4b_xq_4 + 2b_zq_2 \\ -2b_xq_4 + 2b_zq_2 & 2b_xq_3 + 2b_zq_1 & 2b_xq_2 + 2b_zq_4 & -2b_xq_1 + 2b_zq_3 \\ 2b_xq_3 & 2b_xq_4 - 4b_zq_2 & 2b_xq_1 - 4b_zq_3 & 2b_xq_2 \end{bmatrix} \tag{3.13}$$

where

$$\hat{\mathbf{m}}_e = \begin{bmatrix} 0 & m_x & m_y & m_z \end{bmatrix} \tag{3.14}$$

are magnetometer measurements and

$$\hat{\mathbf{b}}_e = \begin{bmatrix} 0 & b_x & 0 & b_z \end{bmatrix} \tag{3.15}$$

represent vertical component due to inclination of the magnetic field.

From figure 3.8 and 3.9 one can see the fused system for the ARG and MARG solution, respectively. The filter gains should be declared as

$$\beta = \sqrt{\frac{3}{4}}\tilde{\omega}_\beta \tag{3.16}$$

$$\zeta = \sqrt{\frac{3}{4}}\tilde{\dot{\omega}}_\zeta \tag{3.17}$$

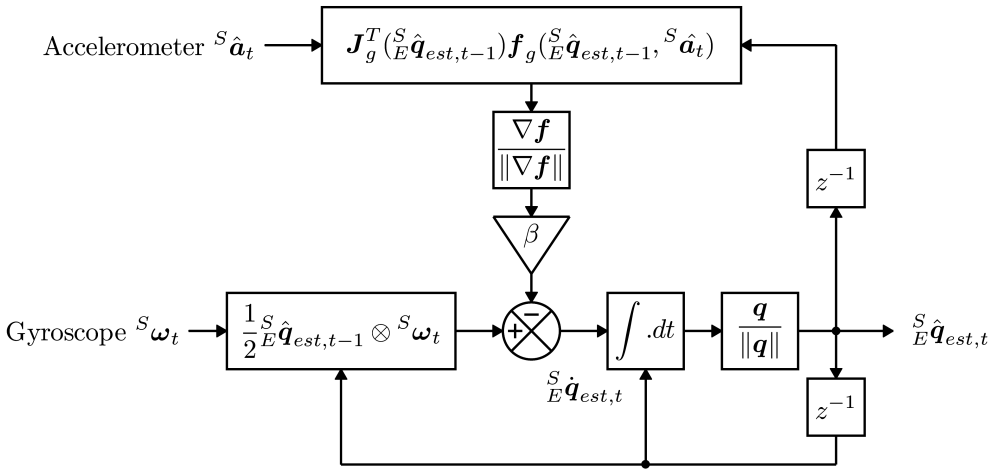


Figure 3.8: Block diagram representation of the complete orientation filter for an IMU implementation

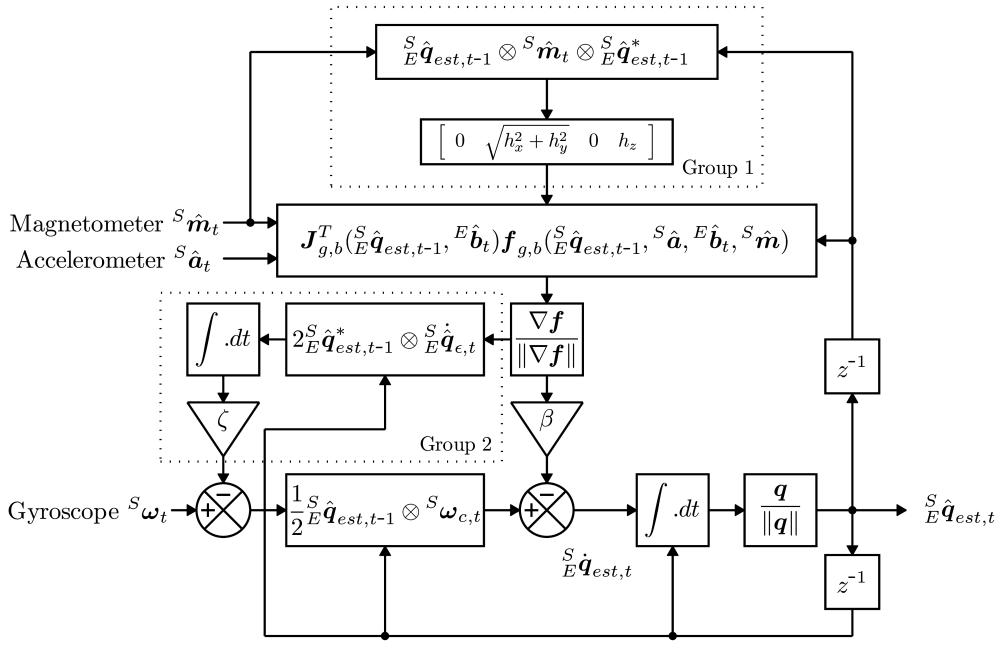


Figure 3.9: Block diagram representation of the complete orientation filter for an MARG implementation including magnetic distortion (Group 1) and gyroscope drift (Group 2) compensation

3.4.2 Implementation

With the formalities done, the practical implementation can be explained.

To initiate the filter the constructor in listing 3.23 populate the quaternion vector variable $\_q$, set gain and frequency.

Listing 3.23: AHRS.cpp excerpt - constructor

```
1  AHRS::AHRS(float frequency, float gain)
2  {
3    _q[0] = 1;
4    _q[1] = 0;
5    _q[2] = 0;
6    _q[3] = 0;
7
8    _gain = gain;
9    _dt = 1.0f/frequency;
10 }
```

The gain and frequency are used in feedback and integration, respectively. Where Madgwick executes line 9. for each update, this version calculates it once.

The first computation to be done is equation (3.2) which can be programmed as

Listing 3.24: AHRS.cpp excerpt - computeQdot()

```
1  void AHRS::computeQdot(float gx, float gy, float gz)
2  {
3    /// qDot = 0.5f * q (x) [ 0 gx gy gz ]; where (x) is sign for quaternion product
4    _qDot[0] = 0.5f * (-_q[1] * gx - _q[2] * gy - _q[3] * gz);
5    _qDot[1] = 0.5f * (_q[0] * gx + _q[2] * gz - _q[3] * gy);
6    _qDot[2] = 0.5f * (_q[0] * gy - _q[1] * gz + _q[3] * gx);
7    _qDot[3] = 0.5f * (_q[0] * gz + _q[1] * gy - _q[2] * gx);
8  }
```

where $\_qDot$ is $\dot{\mathbf{q}}$, $\_q$ $\mathbf{q}$ and $g_{x,y,z}$ is angular velocity.

To update the vector for next iteration, the function

Listing 3.25: AHRS.cpp excerpt - updateQuarternions()

```
1  void AHRS::updateQuarternions()
2  {
3    /// Integrate qDot
4    _q[0] += _qDot[0] * _dt;
5    _q[1] += _qDot[1] * _dt;
6    _q[2] += _qDot[2] * _dt;
7    _q[3] += _qDot[3] * _dt;
8
9    normalize(_q[0], _q[1], _q[2], _q[3]);
10 }
```

is declared where *normalize*() can be treated as a regular second norm.

Listing 3.26: AHRS.cpp excerpt - ARG()

```
1  void AHRS::ARG(float ax, float ay, float az, float gx, float gy, float gz)
2  {
3    /// Temporary variables
4    float f0, f1, f2, f3;
5    float _2q0, _2q1, _2q2, _2q3, _4q0, _4q1, _4q2 ,_8q1, _8q2, q0q0, q1q1, q2q2, q3q3
         ;
6
7    computeQdot(gx, gy, gz);
8
9    /// make sure acceleroemter data is valid or NaN will break further calculationsd
10   if(ax != 0.0f && ay != 0.0f && az != 0.0f)
11   {
12     /// Normalize accelerometer measurement
13     normalize(ax, ay, az);
14
15     /// Auxiliary variables to avoid repeated arithmetic
16     _2q0 = 2.0f * _q[0];
17     _2q1 = 2.0f * _q[1];
18     _2q2 = 2.0f * _q[2];
19     _2q3 = 2.0f * _q[3];
20     _4q0 = 4.0f * _q[0];
21     _4q1 = 4.0f * _q[1];
22     _4q2 = 4.0f * _q[2];
23     _8q1 = 8.0f * _q[1];
24     _8q2 = 8.0f * _q[2];
25     q0q0 = _q[0] * _q[0];
26     q1q1 = _q[1] * _q[1];
27     q2q2 = _q[2] * _q[2];
28     q3q3 = _q[3] * _q[3];
29
30     /// Gradient decent algorithm corrective step
31     f0 = _4q0 * q2q2 + _2q2 * ax + _4q0 * q1q1 - _2q1 * ay;
32     f1 = _4q1 * q3q3 - _2q3 * ax + 4.0f * q0q0 * _q[1] - _2q0 * ay - _4q1 + _8q1 *
           q1q1 + _8q1 * q2q2 + _4q1 * az;
33     f2 = 4.0f * q0q0 * _q[2] + _2q0 * ax + _4q2 * q3q3 - _2q3 * ay - _4q2 + _8q2 *
           q1q1 + _8q2 * q2q2 + _4q2 * az;
34     f3 = 4.0f * q1q1 * _q[3] - _2q1 * ax + 4.0f * q2q2 * _q[3] - _2q2 * ay;
35
36     /// Normalize gradient
37     normalize(f0, f1, f2, f3);
38
39     /// Apply feedback step
40     _qDot[0] -= _gain * f0;
41     _qDot[1] -= _gain * f1;
42     _qDot[2] -= _gain * f2;
43     _qDot[3] -= _gain * f3;
44   }
45
46   updateQuarternions();
47 }
```

Both listing 3.24 and 3.25 are included in the ARG filter in listing 3.26 which iterates to the orientation and outputs a quaternion vector. To ensure valid accelerometer data, line 10. validates accelerometer data and if valid, line 13. normalizes it. The subsequent lines is present to avoid repeated arithmetic and used in the calculation of gradient descent algorithm. The gradient is normalized and applied to the feedback step where the gain makes its entrance.

The MARG algorithm is similar, but with magnetometer measurements incorporated a verification is added to detect wrong inputs in listing 3.27.

Listing 3.27: AHRS.cpp excerpt - MARG()

```
1  if(mx == 0.0f && my == 0.0f && mz == 0.0f)
2  {
3    ARG(ax, ay, az, gx, gy, gz);
4    return;
5  }
```

When normalization of magnetometer measurements is not possible then the ARG-algorithm is automatically used in its place. Rest of the MARG implementation is found using appendix E.

To access the angular orientation representations, seven functions are added. Four to get quaternions and three for Euler angles.

3.5 Verification through visualisation

To verify the filters behaviour, it is wise to simulate or physical test its performance. Latter was naturally chosen here.

A program utilizing the derived drivers and filter was written where the sensors and filter was ran at 800 Hz. The micro controller allow faster execution, but the rate was found sufficient by far and is chosen to allow additional code execution for later implementations. Due to the sensor issue addressed in 3.3.3, a low pass filter has been added. Because the MARG filter will make the heading iterate to face north, the ARG filter is used for visualisation in this section. Unfortunately for the ARG solution, MEMS gyros tend to drift. The gyros used here is no different and yaw will suffer from this in the long run.

3.5.1 Visualisation

To visualise the movements in 3D, a small program was written in Processing [Reas and Fry, 2001], a Java-like language. It is free, open source, works on all major platforms and has OpenGL integration for accelerated 3D among other things.

To simplify programming for new users, Processing features a setup-routine which is run once and is listed in listing 3.28.

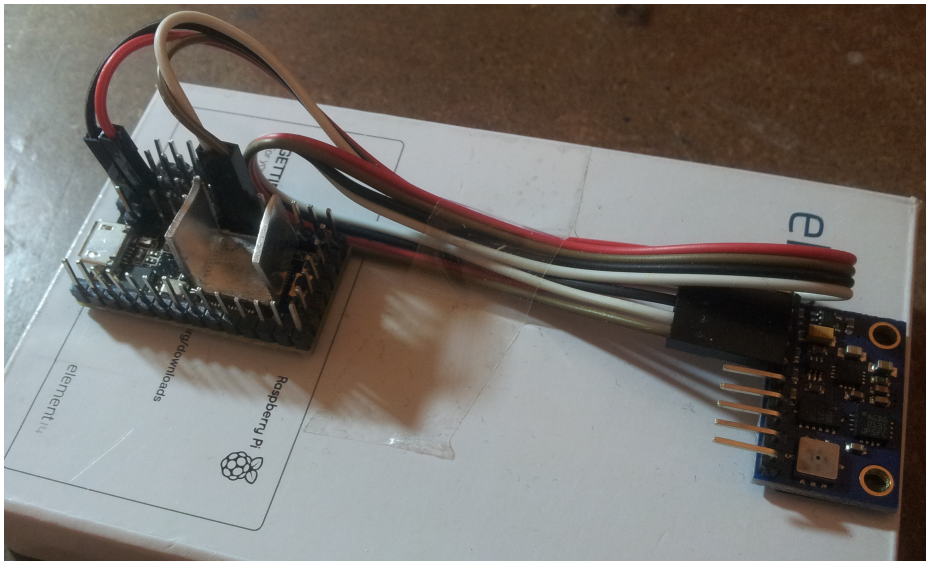


Figure 3.10: Micro controller to left and sensor array to right

Listing 3.28: visualisation.pde excerpt - setup()

```
1  void setup()
2  {
3    size(1000, 1000, P3D); /// Set window size on three axes
4    Serial serial;
5
6    /// This loop waits until a serial device is physical connected
7    while (true)
8    {
9      try
10     {
11       serial = new Serial(this, Serial.list()[0], _BAUD);
12       break;
13     } catch (Exception e)
14     {
15       print("No device present\n");
16       delay(1000);
17     }
18   }
19   serial.clear();
20   serial.bufferUntil('\n'); /// With this, serialEvent() is automatically run when
         newline is received
21
22   font = createFont("verdana.ttf", 32);
23
24   /// File writer
25   output = createWriter("output.txt");
26 }
```

Line 3. defines a window for 3D space. Lines 7-18. locks the program in setup mode until a valid serial serial connection is established. Line 11. is configured to connect to the first serial device with $\_BAUD$ as its baud rate.

To handle and parse data sent by Teensy, the serial handler in listing 3.29 is automatically triggered on data feed.

Listing 3.29: visualisation.pde excerpt - serialEvent()

```
1  void serialEvent(Serial serial)
2  {
3    /// Read input until newline
4    String input = serial.readStringUntil('\n');
5    if (input == null || input.length() == 0)
6    {
7      print(input);
8      return;
9    }
10
11   /// Split input data by tab and print resulting data when array is too small
12   String tmp[] = split(input, "\t");
13   if (tmp.length < 3)
14   {
15     print(input);
16     return;
17   }
18
19   for (int i=0;i<3;i++)
20   {
21     _EULER[i] = float(tmp[i]);
22   }
23
24   /// Write angles to file
25   output.print(input);
26 }
```

.readStringUntil() on line 8. automatically buffers the streaming data until a line feed character is read. The data is then shared into pieces using tabulator as a delimiter on line 16. Three values are required for euler representation of orientation. Unless this is received, line 20. abort parsing.

Line 25. assumes floats are received and are taken as Euler angles stored as ϕ, θ and ψ, respectively in $\_EULER$.

Listing 3.30 is ran continuously and incorporates the Euler angles.

Listing 3.30: visualisation.pde excerpt - draw()

```
1 void draw() /// this runs forever
2 {
3   background(#ffffff); /// white background
4
5   /// OSD _EULER angles
6   textFont(font, 20);
7   fill(#000000); /// text color
8   text("phi : " + degrees(_EULER[0]) + "\ntheta: " + degrees(_EULER[1]) + "\npsi : "
        + degrees(_EULER[2]), 10, 50);
9
10
11  /// comensation
12  float X = -_EULER[1] + 3.14/2;
13  float Y = _EULER[0] + 3.14;
14  float Z = -_EULER[2] + 3.14;
15
16
17  translate(550, 600, 0); /// Move object away from center of origin
18  scale(3,3,3); /// Scale all objects times 3
19
20  rotateX( X );
21  rotateY( Y );
22  rotateZ( Z );
23  buildWingShape();
24 }
```

Lines 12-14. adds biases to the angles to rotate the model to right plane. Because Processing does not assume that z-axis is upwards, but *buildWingShape*() does, the model will be rotated wrong according to the Euler angles. The latter function on line 23. define a flying wing in space through several point-to-point definitions. An excerpt of the complete model is written in listing 3.31 where five lines are present to define one red top of the model. A compilation of all these have been expressed to form the entire structure.

Listing 3.31: visualisation.pde excerpt

```
1   fill(#ff0000);
2   vertex(100, 0, -10);
3   vertex(100, 0, 10);
4   vertex(100, 40, 10);
5   vertex(100, 40, -10);
```

3.5.2 Simulation

With the micro controller connected to the computer over a USB interface, the processing code was run and the sensor was exposed to translational and non-translational motions. The 3D model in figure 3.11 was updated continuously when new data was received and represent the actual motion and proves a working system. The process can be seen in the attached *AHRS.mp4* video file located on the attached CD.

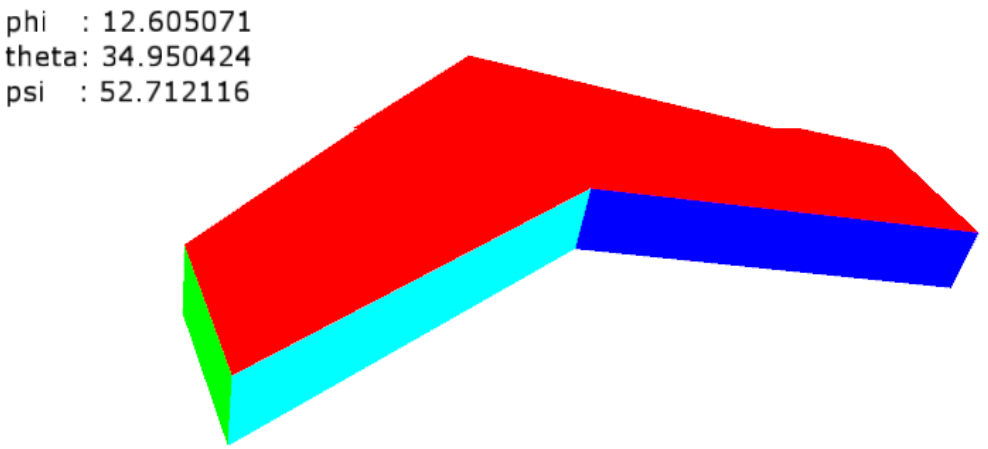


Figure 3.11: UAV visualization using custom Processing program

3.5.3 Plots

While continuously updating the visualisation model, the received data was written to a file as shown on line 29. of listing 3.29. This includes raw data from the sensors and the estimated angles from the orientation filter. Due to the length of the video, not all movements are included in the following plots for sake of readability. The careful reader will find additional plots which corresponds to the movements applied in the video on the attached CD. Its worth mentioning that acceleration has been augmented by a factor of 10 in the plots for visual purposes and that the plots are naturally coupled.

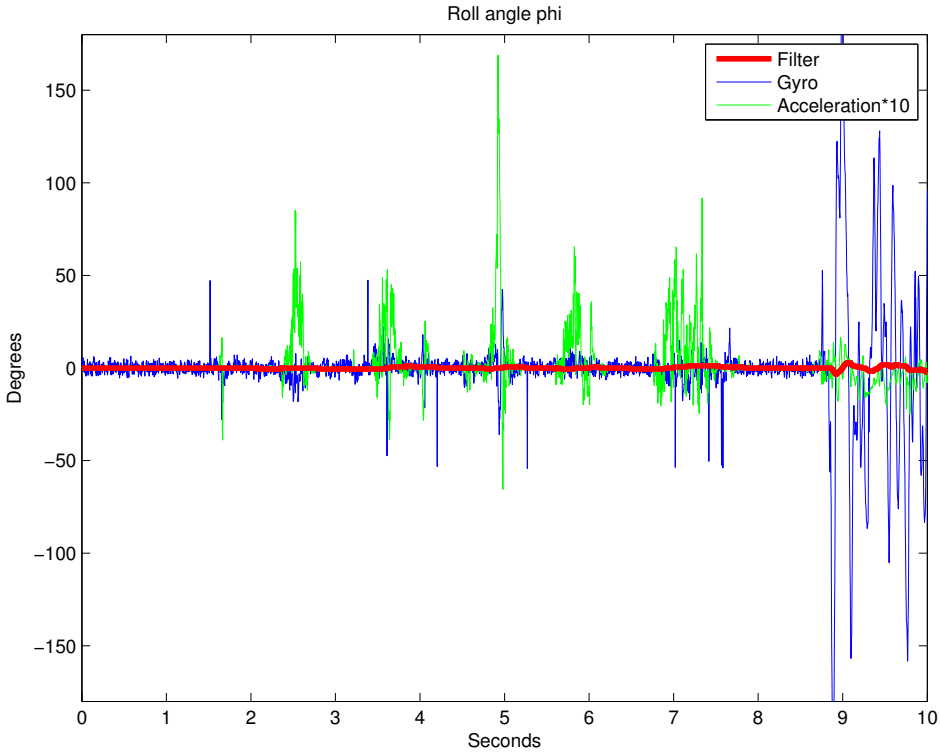


Figure 3.12: Roll movement the first ten seconds

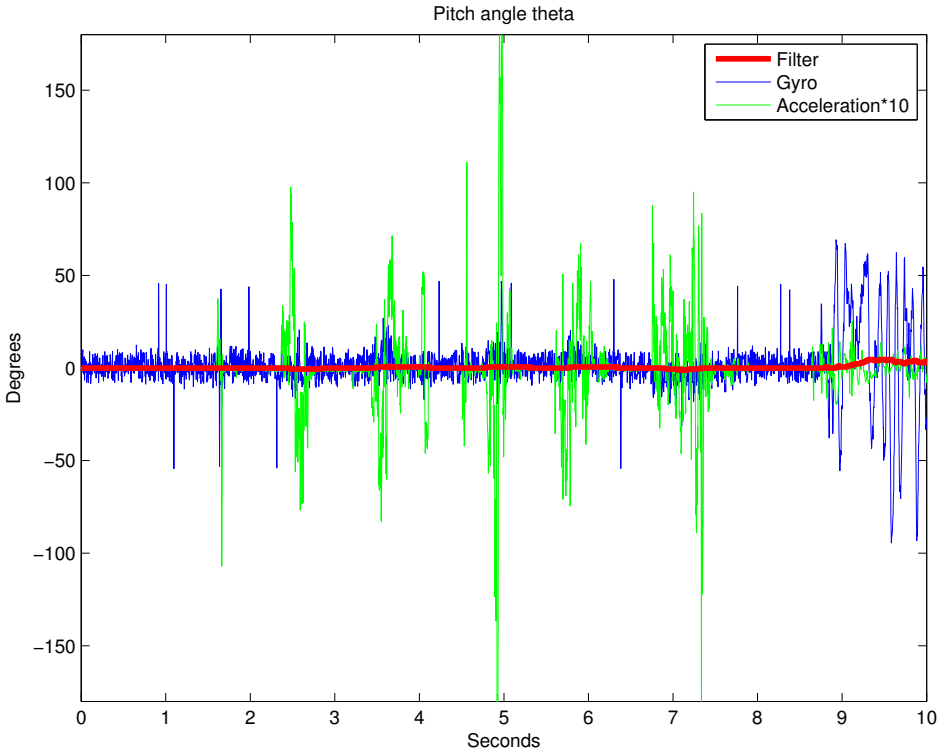


Figure 3.13: Pitch movement the first ten seconds

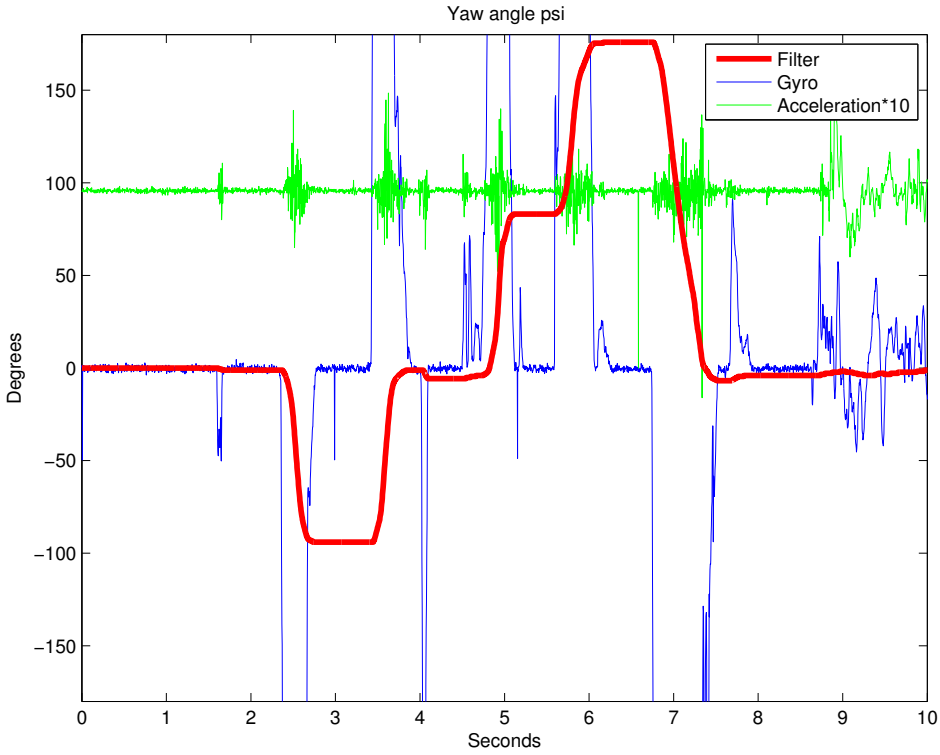


Figure 3.14: Yaw movement the first ten seconds

3.6 Discussion

The goal here was to realise an affordable AHRS to use on a drone. Due to the generality of the resulting system, this AHRS can be used on a wide range of systems. The abnormal activity of the gyroscope and the noise of the cheap sensors complicated the task. The total cost of this system was below $40 and was achieved by buying one piece of each. By buying a batch or even designing the PCB from scratch, the price will drop. In the time of writing, a similar commercial IMU unit start at 990 Euro before VAT. It is probably thoroughly tested, but the price range is beyond the budget for this project and affordable SARDs.

Recall that the acclerometer driver uses computation power to convert bits into acceleration. The detailed reader has noticed that the filter is normalizing accelerometer data and that the acceleration conversion is unnecessary. However, modularity has been prioritized as accelerometer data may be used for other applications on a drone than just for AHRS.

The micro controller used for this AHRS was chosen to allow for a complete flight system. If this is not necessary, a simpler version can be chosen and altered to work as a pure AHRS with UART interface similar to the commercial IMU mentioned earlier.

Chapter 4

Navigation

When drones are to sweep a specific area they must be able to orientate them self within it. There are several ways of doing this, such as using INS or feature based navigation similar to [W.Y. Kong and Cornall, 2006] and [Cummins, 2009]. It is however more convenient to use GNSS and then specifically GPS as it is the most common system. From the GPS national marine electronics association (NMEA) codes, ellipsoidal coordinates are given and used for self-positioning.

4.1 Way-points

Path following is the task of following a predefined path independent of time. This plan can exists of several way-points and can be defined using of coordinates in various formats. For convenience its decided to use geodetic coordinates latitude, longitude and height (μ, l, h) defining stepwise absolute positions for where the vehicle should be. This is mainly because NMEA codes contain geodetic coordinates. Furthermore, other way-point properties such as heading and speed can be defined [Fossen, 2011]. In other words, appending different speed settings for areas where more analyse time is needed can be programmed. Speed could also be decreased at places where collision is more likely to occur such that there is more time for evasive maneuver.

4.2 Way-point tracking

While GPS receiver provide geodetic coordinates of current position, the way-points have coordinates of the desired positions. It is therefore necessary to derive desired heading angle from this. To find the desired heading angle between two geodetic points, one can imagine that a vehicle located at A want to travel to B and that N is both z-axis and North in the earth centered earth fixed frame (ECEF). This frame is indicated by subscripts $_e$ in figure 4.1.

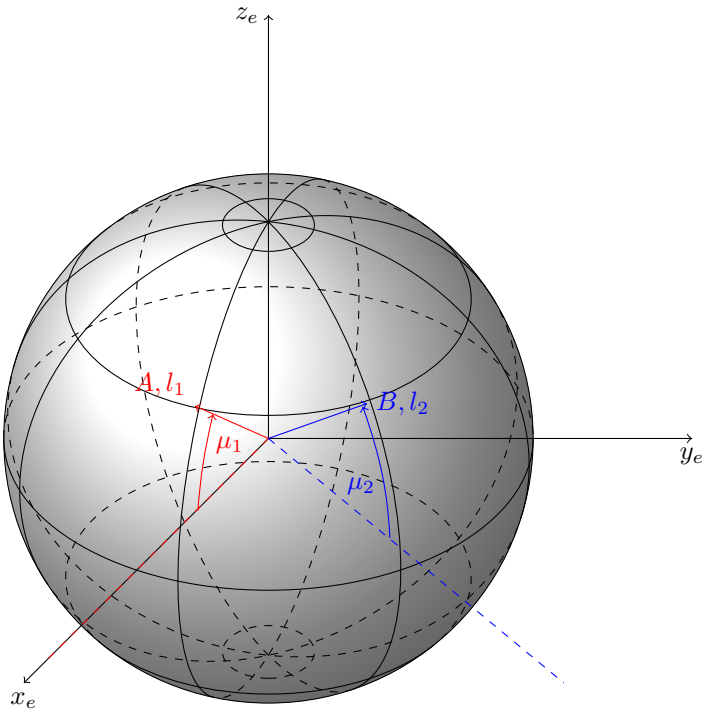


Figure 4.1: Spherical Earth model indicating A, B and N

The points can be set up as

$$A = \Big[\cos(\mu_1), 0, \sin(\mu_1)\Big]^T \tag{4.1}$$

$$B = \Big[\cos(\mu_2)\cos(l_2 - l_1), \cos(\mu_2)\sin(l_2 - l_1), \sin(\mu_2)\Big]^T \tag{4.2}$$

$$N = \Big[0, 0, 1\Big]^T \tag{4.3}$$

where μ_1 and μ_2 is latitude, l_1 and l_2 is longitude for A and B, respectively. N

is the normal for A and B.

The desired heading angle is between N-A and A-B plane. To find the angle between A and B the cross product rule can be used such that

$$N \times A = \left[0, \cos(\mu_1), 0\right]^T \tag{4.4}$$

$$B \times A = \begin{bmatrix} \sin(\mu_1)\cos(\mu_2)\sin(l_2 - l_1) \\ \cos(\mu_1)\sin(\mu_2) - \sin(\mu_1)\cos(\mu_2)\cos(l_2 - l_1) \\ -\cos(\mu_1)\cos(\mu_2)\sin(l_2 - l_1) \end{bmatrix} \tag{4.5}$$

where it can be seen that $N \times A$ is parallel to the y-axis. Using this fact shows that x- and z-components of $B \times A$ dived by the y-component yields

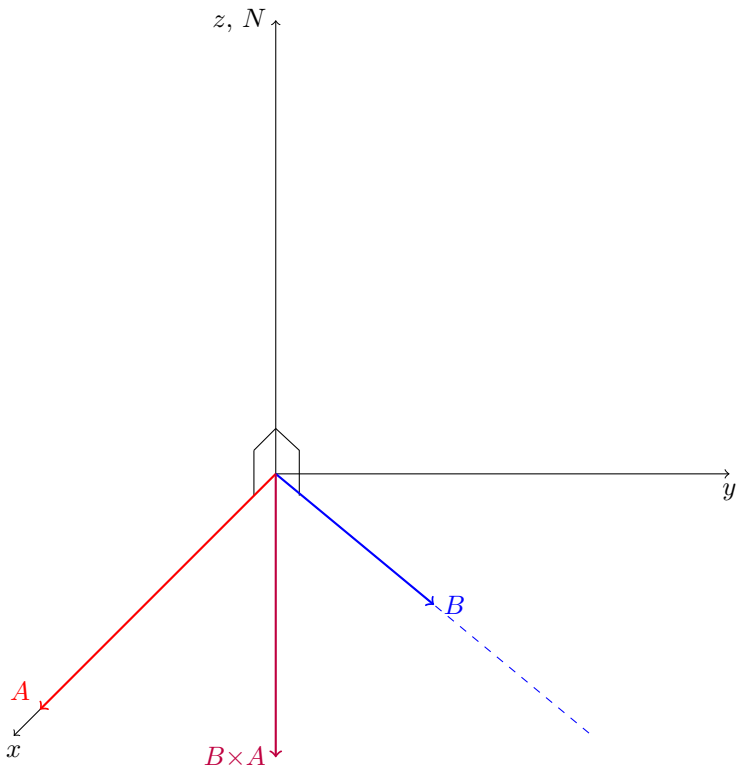


Figure 4.2: Graphically representation of a cross product

$$\tan(\Psi_d) = \frac{\sqrt{(B\times A)_x^2 + (B\times A)_z^2}}{(B\times A)_y} \tag{4.6}$$

$$= \frac{\sqrt{(\sin(\mu_1)\cos(\mu_2)\sin(l_2 - l_1))^2 + (-\cos(\mu_1)\cos(\mu_2)\sin(l_2 - l_1))^2}}{\cos(\mu_1)\sin(\mu_2) - \sin(\mu_1)\cos(\mu_2)\cos(l_2 - l_1)} \tag{4.7}$$

$$= \frac{\cos(\mu_2)\sin(l_2 - l_1)}{\cos(\mu_1)\sin(\mu_2) - \sin(\mu_1)\cos(\mu_2)\cos(l_2 - l_1)} \tag{4.8}$$

The desired heading angle can then be found from

$$\Psi_d = \arctan 2(\cos(\mu_2)\sin(l_2 - l_1), \cos(\mu_1)\sin(\mu_2) - \sin(\mu_1)\cos(\mu_2)\cos(l_2 - l_1)) \tag{4.9}$$

where all variables must be in radians.

The heading angle should inflict yaw moment, but roll can be used as a substitute for airframes underactuated in yaw.

In theory, to detect if a way-point is reached one can check if μ_1 equals μ_2 and l_1 equals l_2. To achieve identical points in practice is unrealistic for a moving object. As a consequence, a variant of Haversine formula can be used

$$d = 2r \cdot \arcsin(\sqrt{\sin^2(\frac{\mu_2 - \mu_1}{2}) + \cos(\mu_1)\cos(\mu_2)\sin^2(\frac{l_2 - l_1}{2})}) \tag{4.10}$$

where d is distance between the two coordinates and r is circle (earth) radius. By setting a threshold limit for d, one can iterate to next way-point when current position is within the threshold. Because earth is ellipsoidal r is not constant and the distance will be slightly off for large distances.

4.3 Implementation

The navigation software presented in this section has the responsibility to store way-points, load them at right time and to communicate with a GPS receiver over a serial interface.

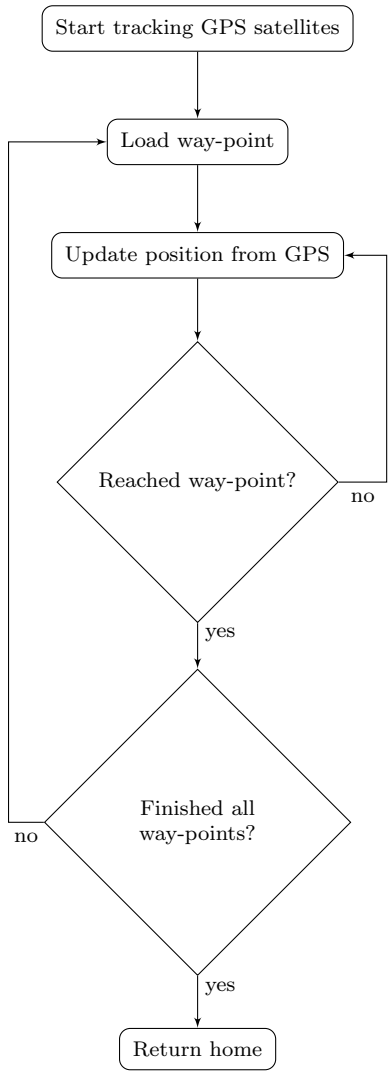


Figure 4.3: Navigation algorithm

To start UART communication with GPS receiver the function in listing 4.1 assumes that the receiver is connected at the pins corresponding to $Serial1$ which is a simple interface to low-level UART communication.

Listing 4.1: Navigation.cpp excerpt - begin()

```
1 bool Navigation::begin(int baudRate)
2 {
3   Serial1.begin(baudRate);
4   return true;
5 }
```

To add way-points, the function

Listing 4.2: Navigation.cpp excerpt - addWaypoint()

```
1 void Navigation::addWaypoint(float latitude, float longitude, float altitude)
2 {
3   _waypoint[_numberOfWaypoints] = { latitude, longitude, altitude };
4   _numberOfWaypoints++;
5 }
```

is used where $\_waypoint$ are defined as

Listing 4.3: Navigation.h excerpt

```
1   public:
2         struct Geodetic
3         {
4             float latitude;
5             float longitude;
6             float altitude;
7         };
```

and $\_numberOfWaypoints$ is an integer counting how many way-points which has been added. Accurate number must be known such that reading of $\_waypoint$ does not exceed its size. It is also used to know when the final way-point is reached such that the craft know when to head home.

To clear way-points the $\_numberOfWaypoints$ is simply reset with

Listing 4.4: Navigation.cpp excerpt - clearWaypoints()

```
1 void Navigation::clearWaypoints()
2 {
3   _numberOfWaypoints = 0;
4   _waypointCounter = 0;
5 }
```

and a complete new plan can be programmed. Note that this facilitate for programming of new trajectories aloft.

Reading new data from GPS receiver takes place in listing 4.5 and the function should be ran at least at the given baud rate.

Listing 4.5: Navigation.cpp excerpt - update()

```
1  bool Navigation::update()
2  {
3    if (!Serial1.available())
4    {
5      return false;
6    }
7
8    int c = Serial1.read();
9
10   #ifdef NMEA
11   Serial.write(c);
12   #endif
13
14   /// a NMEA line is ready to be parsed
15   if (_GPS.encode(c))
16   {
17     /// new data available - parse position from NMEA and save to latitude and
         longitude
18     _GPS.f_get_position(&_location.latitude, &_location.longitude, &_age);
19
20     return true;
21   }
22
23   return false;
24 }
```

Line line 3. of listing 4.5 checks whether new data is available from GPS. If so, the data is read on line 8. and printed on line 11. if the debug variable $\_NMEA$ is defined. Furthermore, the free library $TinyGPS$ is used to parse NMEA codes and the object $\_GPS$ is defined as a TinyGPS object in the header file. Since only one character is read for each iteration, TinyGPS have to buffer the values until a complete NMEA string is formed. When this happens, $TinyGPS :: encode()$ returns true and a new position is available, parsed and is saved to the struct *_location* on line 18.

The function in listing 4.6 must also be triggered frequently. It returns true when flight position is within a circle of the desired way-point. The circle radius is set using the variable *threshold* which implies that $distance()$ from line 4. is a Haversine function.

Listing 4.6: Navigation.cpp excerpt - next()

```
bool Navigation::next(int threshold) /// threshold in meters
{
  /// When distance between current position and desired waypoint is less than
       threshold
  if (_numberOfWaypoints > 0 && distance(_location.latitude, _location.longitude,
       _waypointTo.latitude, _waypointTo.longitude) <= threshold)
  {
      return true;
  }
  return false;
}
```

As long as the way-point counter does not exceed registered way-points, the function

Listing 4.7: Navigation.cpp excerpt - loadNextWaypoint()

```
bool Navigation::loadNextWaypoint()
{
  if (_numberOfWaypoints == _waypointCounter)
  {
    /// all waypoints has been loaded, loading next is unsuccessful
    return false;
  }

  _waypointTo = _waypoint[_waypointCounter];
  _waypointCounter++;

  return true;
}
```

will load next way-point. It returns a boolean value upon trigger such that the caller know if last point is reached and when to return home.

To return home, the function

Listing 4.8: Navigation.cpp excerpt - home()

```
void Navigation::home()
{
  /// Decrease waypoint counter in case home flying is aborted
  if (_waypointCounter > 0)
  {
    _waypointCounter--;
  }

  _waypointTo = _origin;
}
```

has to be called. The variable *_origin* contains coordinates of launch site and is set using the function in listing 4.9.

Listing 4.9: Navigation.cpp excerpt - setOrigin()

```
1 void Navigation::setOrigin(float latitude, float longitude, float altitude)
2 {
3   _origin = { latitude, longitude, altitude };
4 }
```

4.4 Verification

To verify that the software was working, a small program was written utilizing the navigation class and implemented on a micro controller. A GPS receiver was connected and five geodetic coordinates was loaded using *addWaypoint*(). The micro controller was connected to a laptop reporting distance to next way-point in meters, current latitude, current longitude, desired latitude and longitude. A threshold limit for the radius was set to 10 meters.

Figure 4.4: Way-point tester rig

On a clear day, the test was carried out by walking from way-point to way-point verifying that each point was reached and iterating to next one when within thresh-

old limit. The data was also logged and is plotted in figure 4.4.

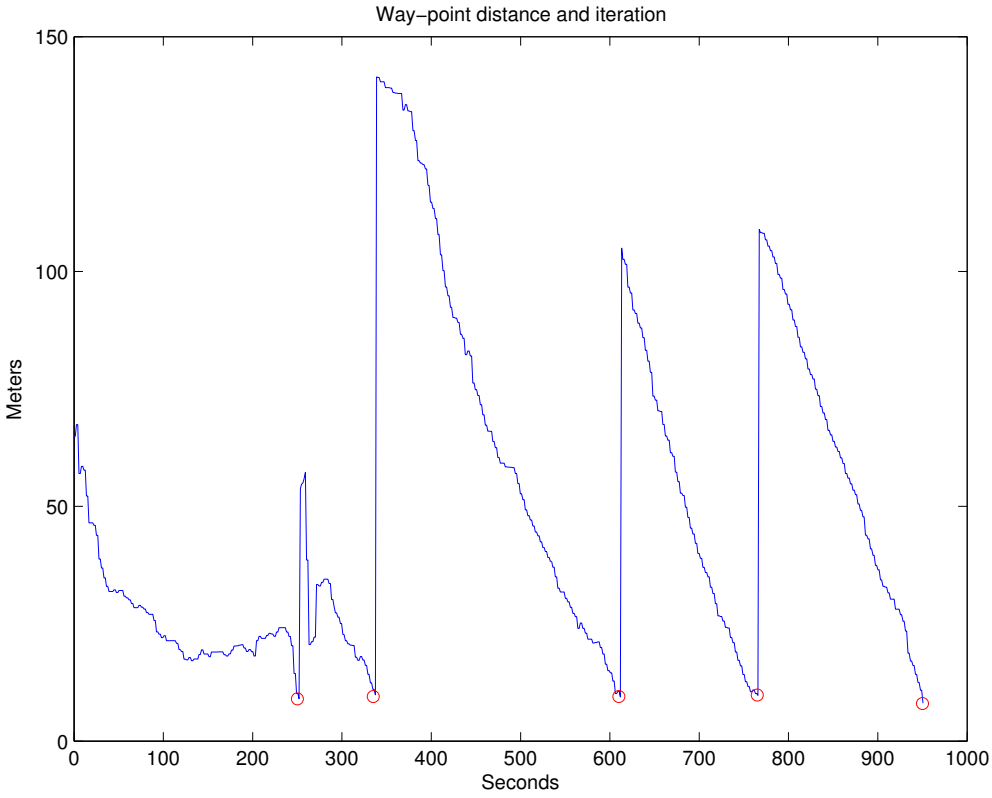


Figure 4.5: Distance to way-point, red circles indicate way-point iteration

4.5 Discussion

This realisation was only tested on ground and not in air. It did however work as expected and an aerial approach should not perform any worse. Indeed, there is no doubt that this system can work as a guidance system for a SARD. This method of navigating do rely on somewhat accurate measurements and that position is updated quick enough for a fast moving UAV, if not the threshold limit must be increased. Also, adding a filter (e.g. low pass) and INS integration can avoid iteration to next point if measurements were to jump which is an realistic case when few satellites are connected.

The most important requirement for this application is to not deviate so much from the planned trajectories that ground areas left out. When its assumed that an area is scanned and marked negative, it would be quite disadvantageous if a person was in fact present. This depends much on GPS receivers and integration with other system may be advantageous. It can however be enhanced in several ways such as programming trajectories closer together or to fly higher above ground such that areas overlap.

A practical problem with GPS is when loss of signals occur. Then the vehicle is suddenly flying blind which is extra bad for autonomous vehicles. Therefore, it is general a good idea to estimate position with a compass and velocity sensor and/or with INS/GPS. The latter can be implemented in several ways; navigating using GPS and when signal loss occur, the vehicle can rely on INS. It would also be advantageous to use INS or additional navigation methods to prevent spoofing (vehicle takeover). Shepard et al. [2012] has recently carried out experiments showing that GPS relying vehicles can be rerouted by a third party. Thus, anti-spoofing methods should be considered at a later stage.

Chapter 5

Communication

When a missing person is found or a probabilistic object is located, the drones (peers in this context) need to notify the operators. There are several ways of doing this, but sending a message over radio frequency to be received in the base station is a natural choice.

5.1 Methods

Norway has a very strict policy when it comes to radio frequency and output power. This complicates the task of selecting a suitable data link. Still, by using commercial communications protocols, the range can be found sufficient by far.

5.1.1 Cellular

When considering the range capabilities, the descending prices for data transfer and a broad selection of power efficient transceivers, a communication protocol over a cellular network is a safe choice. On the other hand, this require a stable connection and can be a severe issue when the search area is outside established cellular towers. Since most people carry with them a cellular phone as explained in section 1.3.3, it is very likely that loss of signal can be the reason for why the person is lost. It should therefore be assumed that cellular network is not always available.

Another established approach could be to use a direct link with satellite to transfer the data. A downside with this is the slow speed and higher costs.

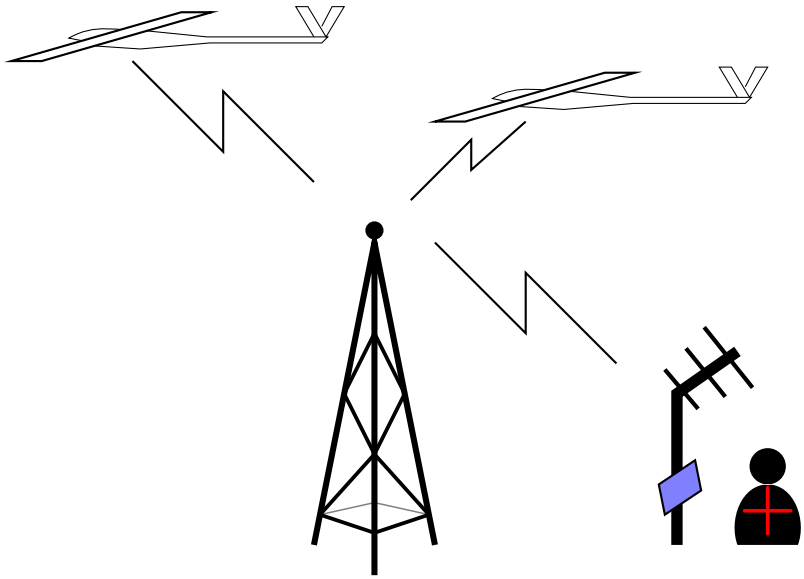


Figure 5.1: Data link using commercial cellular towers

5.1.2 Direct link

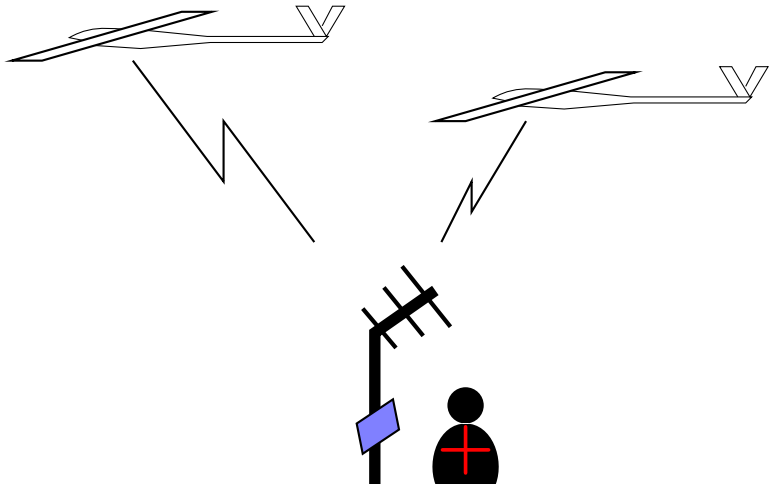


Figure 5.2: Data link using direct signals

By establishing a direct link with the ground station, a secure, fast and reliable connection can be attained. Each peer must carry a transmitter, antenna and power source. The size of this equipment is dependent on the range capabilities

each drone should have, because a short range drone do not require the transmit power a longer range may have. The reason for this is the problem of sending a direct signal from a moving drone, therefore the signal must be broadcasted in all directions. Besides the obvious energy waste this entails, a direct link solution do require a direct line of sight (LOS). This means that if an object intersect the link loss of bandwidth can occur and in worst cases the link goes down. The latter is highly relevant for when a peer flies around a ”corner” like a mountain.

5.1.3 Mesh network

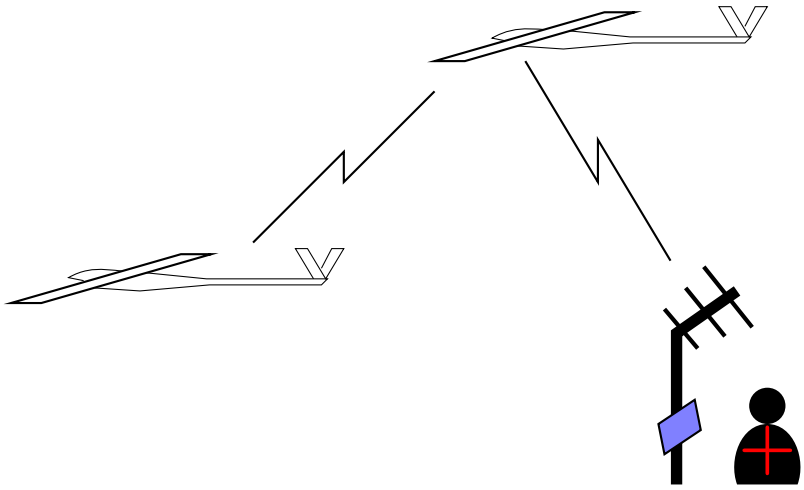


Figure 5.3: Data link using mesh network

With several peers aloft, a mesh network can be set up between them. This reduces the required signal strength and the demands of LOS with ground station. Imagine if a drone flies out of range from the base station and it finds a potential missing persons. With no active link, the operators can not be notified. Without knowing that this is in fact is a persons, the drone can not simply fly back into communication range. It would therefore be advantageous if a another peer is within reach which has an active link with the ground station. If each peer only needs enough power to broadcast a signal to the next peer, both weight and power is saved. By transmitting less power, radio frequency policies addressed in section 1.3.4 may be easier to obey.

5.1.4 Discussion

Using GSM for data transfer is probably the best choice in most situations, but due to the addressed concern in section 5.1.1, other methods must be used. Either as a step in system to replace GSM for cases where cellular towers are unavailable or replace it completely. Thus, a system which enables both direct and mesh network is recommended.

5.2 Data link

Two Zigbee devices was purchased and set up for data transfer between ground station and the prototype drone. The devices support both direct and meshed network and features several interface protocols, but ships default with Hayes command set (AT). Since the latter is also the default protocol for GSM devices, drop-in replacement for GSM devices should be possible.

5.2.1 Set up

While one device was set up as a client and mounted on the drone, the other was set up as the base station creating a direct network. The devices are interfaced over UART and by writing data serially to one device, the other device will receive it wirelessly. Because data is sent wirelessly and without timestamps, it may appear in random order. As a consequence, the data must be sent in chunks with a short delay in between to ensure correct stacking. Since the data is received one character at a time, the data must be buffered before it can be parsed.

5.2.2 Buffering

Due to the addressed buffer requirements in section 5.2.1 a C++ class was written to handle this. Listing 5.1 buffers characters until a given character is received.

Listing 5.1: Buffer.cpp excerpt - bufferUntil()

```
1 bool Buffer::until(char c, char delimiter)
2 {
3   if (c == delimiter)
4   {
5     _buffer[_offset] = '\0';
6     _offset = 0;
7     return true;
8   }
9
10  if (_offset < sizeof(_buffer)/sizeof(_buffer[0])) /// Overflow protect
11  {
12    _buffer[_offset] = c;
13    _offset++;
14  }
15
16  return false;
17 }
```

When received, the string is terminated on line 5. and accessible through listing 5.2 and used in the application.

Listing 5.2: Buffer.cpp excerpt - get()

```
1 char * Buffer::get()
2 {
3   return _buffer;
4 }
```

The buffer has a limited size, but can be increased in the header file.

5.2.3 Test aloft

Naturally, the link was tested on the bench and was found to be working sufficiently. The bench area is a controlled environment and there should not be any surprises. It could have been stress tested, but since it was going to be used on a frame with electrical current flowing nearby, within scope of an additional 2.4 GHz control link and noises from an electrical motor, it was rather tested in its expected environment. There was not put much effort in testing this link, but merely verify that it worked at sufficient testing distances while the drone was aloft and that received data was not corrupted.

5.2.4 Discussion

As expected of a commercial product like this, the system worked without any problem and was found sufficient for prototyping and the experiments carried out in this project. There was however not any attempt to see how far the actual transmit distance was active before losing connection. Neither was there any attempt on

mesh networking which is interesting to address. The link has been put to more use and is indirectly mentioned in section 9.6.

5.3 Control link

Even though a data link is set up between base station and drone, it will not be used to control the drone manually. For prototyping, it is wiser to use a common RC system for manual control inputs. These systems uses various protocols and frequencies from transmitter to receiver. The signals out from the receiver to the actuators are standardized and are PWM. These signals has a fixed voltage from 3.0V to 5.0V and a pulse length of 20ms. The high-pulse ranges from 750μs to 2250μs. To be able to use these systems in a micro controller such that one can switch manual control on and off, a driver has to be derived.

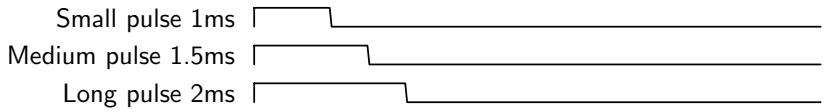


Figure 5.4: Typical RC signal pulse-width modulation

5.3.1 PWM driver

The driver can accept unlimited different signals, but external interrupt routines in the micro controller is however a common ceiling.

The two functions in listing 5.3 informs the processor which pin(s) to enable interrupts on.

Listing 5.3: PWM.cpp excerpt - add()

```
1 void PWM::add(int pin, functionPointer customFunction, int initialValue, int
      filterLowPass)
2 {
3   _signal = initialValue; /// initialize a signal value
4   _interruptStartTime = micros(); /// initialize start time such that high pulse does
        not become too big!
5   _filterLowPass = filterLowPass;
6   pinMode(pin, INPUT);
7   attachInterrupt(pin, customFunction, RISING);
8 }
```

This is done by writing values to the pin registers in the processor. The codes for doing this is not standardized and differs from vendor to vendor. The function

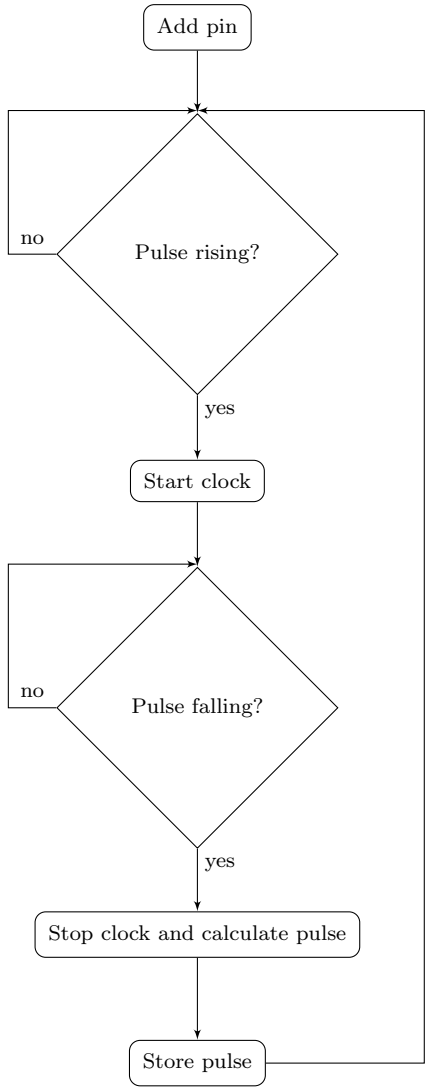


Figure 5.5: Basic PWM driver algorithm without low-pass filter

on line 7. in listing 5.3 fixes this for us since its authors has included support for several processors. The attach-routine on line 7. tells that the *customFunction* will be triggered whenever the signal is rising. A small disadvantage is that the custom function can not accept arguments and must be static.

Whenever a positive flank is perceived on the pin, listing 5.4 is triggered and stores micro seconds since boot.

Listing 5.4: PWM.cpp excerpt - handleRising()

```
1 void PWM::handleRising(int pin, functionPointer customFunction)
2 {
3   _interruptStartTime = micros(); /// Store start time to be used in fall()
4   detachInterrupt(pin); /// Detach..
5   attachInterrupt(pin, customFunction, FALLING); /// ..and set it to falling
6 }
```

The size (bytes) of this time is not important as long as it does not overrun size of the variable definition and counts evenly until its called again. Since a high signal is perceived, the pin interrupt is disconnected and reattached to listen for falling flanks.

The actual high pulse calculation happens on line 3. in listing 5.5 where the pulse is stored in the global variable *_signal*.

Listing 5.5: PWM.cpp excerpt - handleFalling()

```
1  void PWM::handleFalling(int pin, functionPointer customFunction)
2  {
3    int pulse = micros() - _interruptStartTime; /// calculate signal pulse
4
5    detachInterrupt(pin);
6    attachInterrupt(pin, customFunction, RISING);
7
8    /// if the pulse is outside expected area, ignore it
9    if (pulse > PWM::_maxPulseWidth || pulse < PWM::_minPulseWidth)
10   {
11     return;
12   }
13
14   /// simple optional filter for large changes
15   if (_filterLowPass != 0 && abs(pulse - _signal) > _filterLowPass)
16   {
17     return;
18   }
19
20   _signal = pulse;
21 }
```

Again, pin interrupt is disconnected and reattached to listen for rising flanks on lines 5. and 6. In case of noise or other undesired actions, a saturation filter is added on line 9. An optional filter was added in case a signal were to change rapidly due to noise or faulty interrupt handling which infrequently occurred while developing.

Because the global variable *_signal* holding high pulse is restricted within this class (private) public function is present making it publicly available.

5.3.2 Connecting receiver

The receiver class uses four PWM-driver objects. Because the *customFunction* has to be static, eight separate static functions must be declared for four signals. To omit equal functions, lets consider two which are distinct. From before it can be perceived that there must be one function for high flank and one for low flank.

Listing 5.6: Receiver.cpp excerpt

```
1 void Receiver::throttleRising()
2 {
3   _pwm[_throttlePin].handleRising(_throttlePin, Receiver::throttleFalling);
4 }
5 /// Function is called when a pin state changes from high to low
6 void Receiver::throttleFalling()
7 {
8   _pwm[_throttlePin].handleFalling(_throttlePin, Receiver::throttleRising);
9 }
```

The functions for throttle is declared in listing 5.6 where $\_ppm[4]$ is an array holding four PWM objects. Together with $PWM :: handleRising()$ and $PWM :: handleFalling()$ these two functions toggle between each other detecting the signal present. Six equal functions are declared for aileron, elevator and the mode pin. Additional functions must be declared if more signals are added. This rises the desire for recursion.

Listing 5.7: Receiver.cpp excerpt - attach()

```
1 void Receiver::attach(int throttlePin, int aileronPin, int elevatorPin, int modePin)
2 {
3   _throttlePin = throttlePin;
4   _aileronPin = aileronPin;
5   _elevatorPin = elevatorPin;
6   _modePin = modePin;
7   _attached  = true;
8   _lostLink = false;
9
10  _pwm[throttlePin].add(throttlePin, &Receiver::throttleRising, 900, 500);
11  _pwm[aileronPin].add(aileronPin, &Receiver::aileronRising, 1500, 500);
12  _pwm[elevatorPin].add(elevatorPin, &Receiver::elevatorRising, 1500, 500);
13
14  /// The mode pin is not required to be attached
15  if (modePin > 0)
16  {
17    _pwm[modePin].add(modePin, &Receiver::modeRising, 1000, 500);
18  }
19 }
```

The function in listing 5.7 connects all the components and should be called in a start-up routine in the main program. The pin-arguments are the physical pins on the processor to be connected to the corresponding pins on the receiver. The variable *modePin* is an optional input such that the interrupt routines for the other channels can be disconnected by for example controlling using different

signals through this variable.

The detach routine in listing 5.8 features an optional disconnect-able mode pin.

Listing 5.8: Receiver.cpp excerpt - detach()

```
1  void Receiver::detach(int throttlePin, int aileronPin, int elevatorPin, int modePin)
2  {
3    _attached = false;
4    _pwm[throttlePin].remove(throttlePin);
5    _pwm[aileronPin].remove(aileronPin);
6    _pwm[elevatorPin].remove(elevatorPin);
7
8    if (modePin > 0)
9    {
10     _pwm[modePin].remove(modePin);
11   }
12 }
```

In most cases the mode pin should not be disconnected since it is in fact used to trigger receiver connectivity. One may argue why the detach routine should ever be used. The fact is that computational power can be delegated to other parts by disconnecting unused functions.

Lastly, a safety function is present in the receiver class. The function

Listing 5.9: Receiver.cpp excerpt - lostLink()

```
1  bool Receiver::lostLink(int pin)
2  {
3    /// This function may be very custom depending on RC-system. On Spektrum systems,
         the same throttle position you used when binding to the receiver will be
         applied when signal is lost.
4    /// It is therefore wise to push throttle signal below normal operating range when
         binding. By doing this, a throttle signal was measured to be about 924
         microseconds, hence 1000 value.
5
6    if (_pwm[pin].signal() < 1000 || _pwm[pin].signal() > _pwm[pin].maxPulseWidth())
7    {
8      _lostLink = true; /// use recoverLink() to change state
9      return true;
10   }
11
12   return false;
13 }
```

detects if the transmitter is either off or out of range. This does however require that the receiver has the feature where one of the active signals drops to be outside the normal operating range such that it can be detected.

5.3.3 Discussion

Because it was chosen to rely on *attachInterrupt*() in the PWM-driver, recursion becomes difficult. The callback function which is required by this routine does

not provide any argument or variable which indicate on which pin the interrupt were triggered. Thus, hard coding several functions like this is necessary. However, by taking control of the external timer routines and registers on the processor, a recursive method can be created without too much hassle. Pretty recursive code was therefore sacrificed for modularity.

5.4 Discussion

There are still communication issues which has not been addressed in this matter. This includes required power, best frequency in terms of stability and speed, and antennas. There exists omnidirectional antennas, directional antennas and other designs and it has therefore not been concluded which one its better for this application.

Until now, the base station has been referred to as the ground station, but if the station were in air in e.g. a full scale helicopter, the LOS to each drone could be better and drone reports could be verified quicker.

The range capabilities of the data link used for prototyping may not be sufficient for a final system. To save additional costs and enhance complexity, the construction of a custom transceivers would be advantageous. The system has and will however serve the purpose for proof of concept.

Chapter 6

Flying

Previous topics has discussed and put together technologies required by the drones. This section explains how the programmed vehicle will behave by using these components.

6.1 Deploy

For fixed wings without VTOL capabilities, wheels are the most common method for take off and landing. Otherwise, they are dead weight and not recommended for lightweight constructions. Thus, wheels are not considered in this text. This does rise an issue about landing which will later be addressed in section 6.5.

After deployment, drones should start their programmed trajectories.

6.1.1 Rail launch

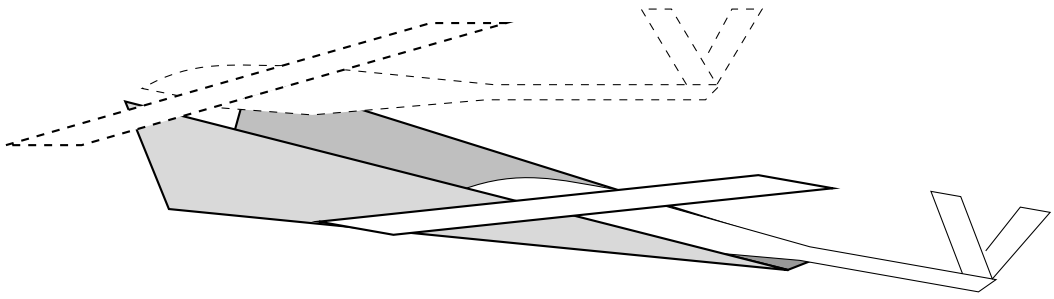


Figure 6.1: Rail launching

Utilizing a fixed-wing without wheels is difficult since fixed-wings generate lift by forward speed. Without wheels, velocity is not built up and lift is absent. Nonetheless, using a rail system where the plane lays on a wagon which generate substantial speed, the plane would be able to launch. By using materials with small friction the same functionality can be achieved without the wagon if the plane is able to propel itself.

A simple approach could be to use a full scale car as wagon. By placing a custom cradle on top of the car, the car can provide enough speed for the craft to gain sufficient lift to take off. This do require a somewhat straight road.

6.1.2 Hand launch

In addition, if the drones are small and manageable, hand launch can be performed. This require less equipment and space, but it also gives the operator more flexibility in form of launch angles and speed. Hand launching may sound easy, but getting a UAV airborne at right velocity without self injury is not trivial. Thus, experience is highly advised since a propeller will cut flesh.

6.1.3 Propulsion

Just before or right after launching, the vehicle must turn on its propulsion system automatically. By using INS, the vehicle can be programmed to start the motor when the desired acceleration is reached if attitude is correct. Otherwise, a timed start-up can be used. E.g. 10 seconds after a button has been pressed.

If launched by hand, the speed may be low and if propulsion is done wrong the initial moment produced by the propeller can force the plane to roll. With little air flowing over its control surfaces the plane will be uncontrollable and unable to stabilize itself. Thus, proper initial routines are important.

6.2 Turning

If the vehicle has a camera faced downwards in a stationary position, the planned ground sight can be lost when turning. Surely, the plane has to turn without rolling when searching below. By contrast, it is possible to keep ground sight if large rudder and roll inputs are applied. However, this can make the plane stall if

it is ”banked” to hard. Therefore, its a good idea to perform turning outside the intended area. This will add delay to the necessary flight, but skips the requirement of a rudder. Thus, additional weight and drag is avoided.

In fact, many of these issues can be avoided by using a gimbal mounted camera which will make the camera naturally more exposed. Consequently, it will make landing more difficult without damage. Still, a stabilized gimbal camera would be able to take better pictures when exposed to turbulence and vibrations.

6.3 Cruising

When flying, the craft should follow its programmed path as good as possible, keeping the cross-track error small. The regulators for pitch, roll and yaw should keep the plane stable while airborne.

Again, if the camera is strapped down, the plane should be kept levelled while cruising. If not, the camera searching for persons may miss certain areas. Surge velocity should also be kept constant such that algorithms can be developed to assure that captured images will overlap as little as possible.

Even though a levelled plane at a constant speed is preferable, better search algorithms can be achieved with a marginally stable flight. Indeed, if the craft were to roll back and forth systematically without interfering with other parts of the system, it would be able to scan a larger area of the ground in the same flight. This would have to be investigated further to see if it introduces any unwanted bi-effects.

6.4 Return

An important issue which has not been addressed yet is the return-to-origin case. Due to the limited range of any vehicle, some sort of estimator must be implemented to estimate remaining propulsion energy such that there is always enough energy to return to origin. When a limit is reached, the vehicle should abort way-point tracking and return to origin. Another approach instead of returning to origin would be to meet operators half-way to allow more time aloft.

6.5 Landing

Due to the nature of GNSS, a drone will not be able to land very precise by only following the received GPS signal. However, as the craft will be equipped with human-locating technology, it can use these tools to aim at the ground crew and orient itself. There could alternatively be run an algorithm to find faces or specific signs which the rescue corps can present, that is, if drones are camera equipped.

Since the drones should be relatively light with respect to their size and have a high structural integrity, they should be able to perform a typical *belly landing*. This is done by stopping the motor some meters before hitting the ground and letting the velocity be consumed by pushing elevators upwards. A great advantage of facing the elevators upwards is that it applies longitudinal dihedral in form of stability Kermode [2006]. If a drone is loaded heavy it may experience lateral instability.

It is common to use a gimbal mounted camera inside a dome underneath the drone. Landing a drone like this on the belly will most definitely cause damage if not the surrounding area is soft. On the other hand, if the drone could be landed inverted then the camera would be safe of damage. It does however require a special airframe and system. If the plane is able to land very precisely, a circular cushion or a catching net could also be used. Other equipment based techniques could be used, but this puts more responsibility in the crew and equipment.

With enough head wind any plane can be stationary in air. A plane with low stall speed and low drag require less head wind to be stationary and if position is right, the plane can merely be picked from the air. Even though a hobbyist can pick his plane from air, it is a non-trivial subject to program automatic routines for and would be an accomplishment in itself.

6.6 Faults

Without proper maintenance, faults will arise and its therefore wise to program emergency landing routines. This may be quite extensive to develop due to the variety of possible component faults and multiple routines must therefore be programmed for each faulty component. Of course, the craft should try to land itself without any additional damage and the landing routine may share procedure with

regular landing as explained in section 6.5. However, the ultimate goal is to prevent damage on property and injuries on humans and animals.

Initially, additional sensors are required to detect faults and especially on actuators, but other techniques can be used involving IMU sensors. By recording gyro and accelerometer data during regular actuator inputs, estimator functions for gyro and/or accelerometer data can be derived. This method can estimate the expected behaviour of the craft when actuated and if the error is to large, a possible fault is present. Still, this can be developed even simpler and other methods might be necessary to pursuit in order to detect faults.

6.7 Discussion

For the small drones which this report focuses on, hand lunch will be sufficient and avoids the set up of a launch ramp. Still, a ramp can be constructed quite small and handy.

The same goes with landing, the system should be able to land itself without any catching equipment. By restricting the system to simple launch and land actions, less hassle and knowledge is required by the rescue operators and the entire system simplifies. Training people for more complicated tasks can be expensive and time consuming. For a system which should always be operative and operative as fast as possible, it is wise to automate and simplify as much of the tasks as possible.

Chapter 7

System fusing

Until now, all the sections has presented each issue as a single topic. In this section, all the topics are fused together to form a complete working system which will be explained in detail.

The fusing introduces several issues and without proper planning, epiphanies are certain to occur. When the system are to be as cheap-as-possible, several compromises are made and it is important to make the right ones. For example, even though the AHRS works as a single program, it might not be able to run at the desired speed when other demanding code are to run at the same time. This directly impacts the number of interrupts and timers required. Hence, there must be enough interrupts and timers for each part of the program without having to introduce additional compromises such as the endless complexity/performance issue.

7.1 Overview

Micro controllers tend to have ”big” while loops. It may end in complex and chaotic code which is hard even for the authors to keep track off. For this reason it is important to design and simplify as much as possible. This program is laid out with a initiation routine and a main loop as illustrated in figure 7.1.

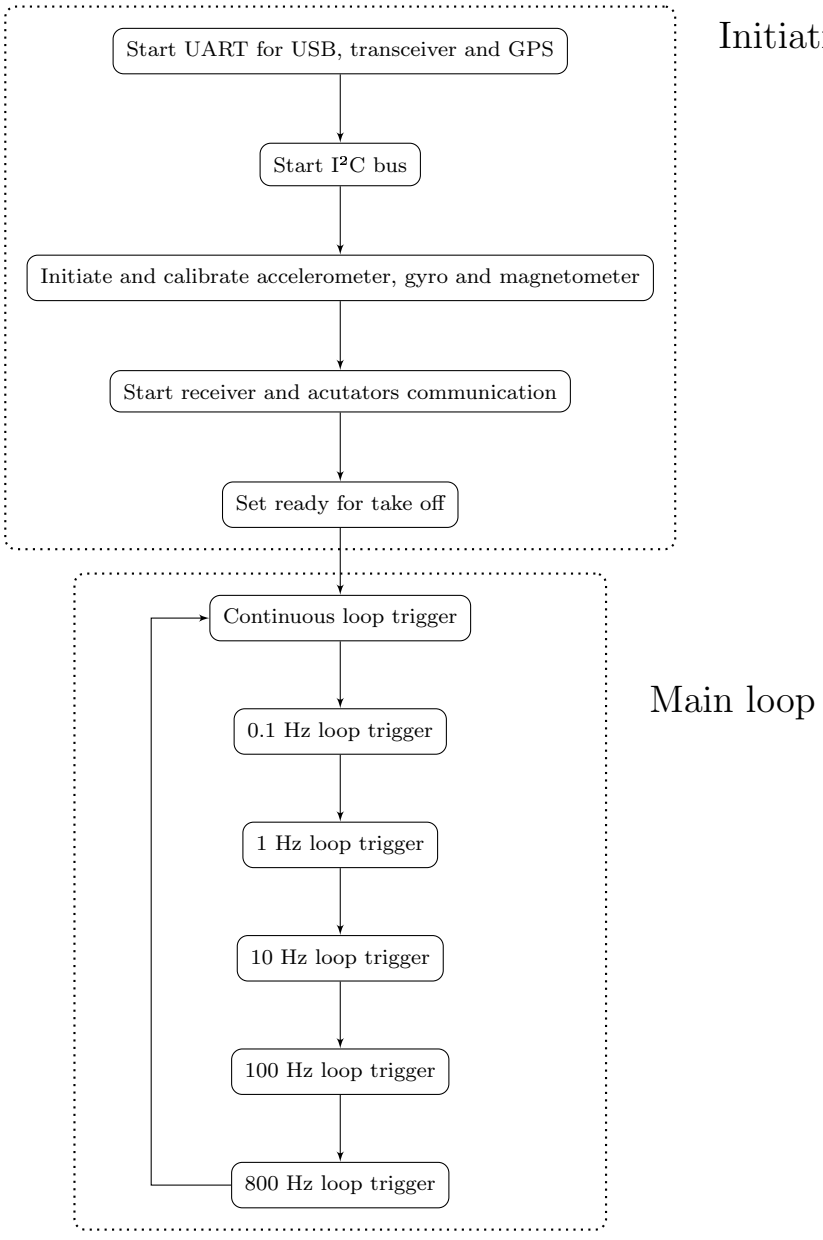


Figure 7.1: Main drone algorithm

7.2 Initiation routine

Besides including configurations, necessary classes and declaring object of those, the following sections explains the initiation routine illustrated in figure 7.1. These are only run once when micro controller boot.

7.2.1 UART

The following lines are in fact the first execution for the overall program. The baud rate settings from the configuration file is used througout listing 7.1 where the first serial initiation on line define UART speed with USB.

Listing 7.1: AUAVISAR.ino excerpt

```
1  Serial.begin(_SERIAL_BAUD);
2  #ifdef __MK20DX128__
3  while (!Serial) /// Some (!Atmel) chips require this hack to make serial work with
       the host
4  {
5    blink();
6  }
7  #endif
8  Serial.println("USB: serial started");
9
10
11 ///
12 /// CONNECT BASE STATION (WIRELESS TRANSCEIVER)
13 ///
14 Serial2.begin(_WIRELESS_BAUD); /// Serial2 means pin 9 and 10 on Teensy 3.0
15 Serial.println("WIRELESS: serial started");
16
17
18 ///
19 /// CONNECT GPS RECEIVER
20 ///
21 navigation.begin(_GPS_BAUD); /// Navigation handles GPS serial communication on
       Serial1 for pin 19 (rx) and 18 (tx) for Teensy 3.0
```

Line 2-6. checks if the micro controller is Teensy 3.0, if that is the case, a loop execute a blinking function which makes the single led on Teensy to blink. It will blink until a serial interface has been established. Both the serial communication and this blinking routine is present for debugging and should be disabled before any flight attempts.

Starting serial communication with the data link receiver is executed on line 14. and becomes ready instantly.

Since the Navigation class already has been explained it is easy to understand that the baud parameter parsed on line 21. is in fact the GPS baud rate.

7.2.2 Sensors

Similar as the derivation of AHRS in section 3 the following sensor initiation are performed here. I²C connectivity is achieved on the first line in listing 7.2.

Listing 7.2: AUAVISAR.ino excerpt

```
1  Wire.begin();
2  Serial.println("I2C: joined bus");
3
4  Serial.println("About to calibrate, do not move!");
5  delay(500);
6
7
8  /// Initialize accelerometer
9  acceleration.setHz( _HERTZ );
10 acceleration.setG( _MAX_G );
11 acceleration.initialize(9.81/269, 9.81/250, 9.81/260, 9.81/260, 9.81/253,
       9.81/264);
12 while (!acceleration.calibrate(true,true,false,_ACCELEROMETER_NOISE))
13 {
14   Serial.println("ACCELEROMETER: re-calibrating in 1 second.");
15   delay(1000);
16 }
17 Serial.println("ACCELEROMETER: calibrated");
18
19
20 /// Initialize gyroscope. Gyro is default in sleep mode and must be set active,
       this fails sometimes
21 while (!gyro.initialize((float) _HERTZ / 4000 )) /// 800 Hz / 2000 Degree/second *
       0.6 (tunable factor)
22 {
23   Serial.println("GYRO: init failed. Correct device address? Reattempt in 1 second
         .");
24   delay(1000);
25 }
26 while (!gyro.calibrate(true,true,true,_GYRO_NOISE))
27 {
28   Serial.println("GYRO: re-calibrating in 1 second.");
29   delay(1000);
30 }
31 Serial.println("GYRO: calibrated");
32
33
34 /// Initialize compass
35 compass.initialize();
36 /*while (!compass.calibrate(64, 1))
37 {
38   Serial.println("COMPASS: calibrate failed. Reattempt in 1 second.");
39   delay(1000);
40 }*/
41 compass.setGain(1);
42 compass.continuous();
43 Serial.println("COMPASS: ready");
```

Still considering listing 7.2, *Wire* is indeed the same class as in section 3. Acceleration initiation is started on line 9. and is calibrated on line 12. At this point it is important that the vehicle is perfectly still or the calibration will run until recorded values does not exceed the content of variable $\_ACCELEROMETER\_NOISE$.

Because the specific gyro is a bit tricky, line 21. is looped until the gyro has successfully been started. It is also calibrated on line 26. and must also be perfectly still to not overshoot $\_GYRO\_NOISE$.

The magnetometer is started on line 35. and properties are set on lines 41-42.

7.2.3 PID and connect

For extra tuning options, PIDs are applied for all four actuating axes including surge, roll, pitch and yaw. Initialization using the values from the configuration is applied as

Listing 7.3: AUAVISAR.ino excerpt

```
1  surge.set(_SURGE_P, _SURGE_I, _SURGE_D);
2  roll.set(_PITCH_P, _PITCH_I, _PITCH_D);
3  pitch.set(_ROLL_P, _ROLL_I, _ROLL_D);
4  yaw.set(_YAW_P, _YAW_I, _YAW_D);
```

Line 1. of

Listing 7.4: AUAVISAR.ino excerpt

```
1  receiver.attach(_PIN_THROTTLE, _PIN_AILERON, _PIN_ELEVATOR, _PIN_MODE);
2  #ifndef _DISABLE_ACTUATOR
3  actuate.attachLeft( _PIN_LEFT );
4  actuate.attachRight( _PIN_RIGHT );
5  actuate.attachMotor( _PIN_MOTOR );
6  #endif
```

uses the receiver class to connect a RC receiver to the given parameter pins. While lines 3-5. sets pin configuration for which pin the actuators are connected to. That is, unless $\_DISABLE\_ACTUATORS$ is not defined in the configuration file.

7.3 Main loop

With figure 7.1 in mind, it can be seen from listing 7.5 that the main loop code reflect its algorithm quite well. *micros*() from line 3. returns microseconds since boot and is stored to the variable *time*. Its important to remember that these variables and timers will overflow and that it is wise to implement a overflow catcher like line 6. does. Rest of the exhibited code consists of triggering each function at the right time. Each of these functions will now be derived in the following sections.

Listing 7.5: AUAVISAR.ino excerpt - loop()

```
1  void loop()
2  {
3    time = micros(); /// Store time in memory for this iteration. This will overflow in
         70 minutes.
4
5    /// Overflow compensation
6    if (pHz01 > time)
7    {
8      pHz01 = pHz1 = pHz10 = pHz100 = pHz800 = time;
9    }
10
11   /// laying-8 Hz
12   continuous();
13
14
15   /// 0.1 Hz
16   if ( (time - pHz01) >= 10000000)
17   {
18     hz01();
19     pHz01 = time;
20   }
21
22
23   /// 1 Hz
24   if ( (time - pHz1) >= 1000000)
25   {
26     hz1();
27     pHz1 = time;
28   }
29
30
31   /// 10 Hz
32   if ( (time - pHz10) >= 100000)
33   {
34     hz10();
35     pHz10 = time;
36   }
37
38
39   /// 100 Hz
40   if ( (time - pHz100) >= 10000)
41   {
42     hz100();
43     pHz100 = time;
44   }
45
46
47   /// 800 Hz
48   if ( (time - pHz800) >= 1250)
49   {
50     hz800();
51     pHz800 = time;
52   }
53 }
```

7.3.1 Continuous

Because the Navigation-class has already been explained in section 4.3, line 3. of

Listing 7.6: AUAVISAR.ino excerpt - continuous()

```
1 void continuous()
2 {
3   navigation.update(); /// Update current position using GPS
4   wireless(); /// Updaate data link
5 }
```

is easily understood. Line 4. trigger and handle data link communication with base station. Its function is expressed in listing 7.7 and checks and abort execution if there are no data available on lines 4-7.

Listing 7.7: AUAVISAR.ino excerpt - wireless()

```
1 void wireless()
2 {
3   /// There are no data available from the transceiver
4   if (!Serial2.available())
5   {
6     return;
7   }
8
9   /// Returns true whenever a new line is read
10  if (!buffer.until(Serial2.read(),'\n'))
11  {
12    return;
13  }
14
15  char * word1 = strtok(buffer.get()," ");
16
17  /// Clear waypoints
18  if (equal(word1,"WPC"))
19  {
20    navigation.clearWaypoints();
21    Serial.println("NAVIGATION: removing all way-points");
22  }
23
24  /// Verify identity
25  else if (equal(word1, _IDENTITY))
26  {
27    char * word2 = strtok(NULL," ");
28
29    /// New waypoint
30    if (equal(word2,"WPA"))
31    {
32      navigation.addWaypoint(atof(strtok(NULL, " ")), atof(strtok(NULL, " ")), atof(
          strtok(NULL, " ")));
33      Serial.println("NAVIGATION: new waypoint added");
34
35      /// If this is the first point then load it
36      if (navigation.numberOfWaypoints() == 1)
37      {
38        navigation.loadNextWaypoint();
39        Serial.println("NAVIGATION: loaded first way-point");
40      }
41    }
42  }
43 }
```

When data is available, the data is buffered using the $buffer$ object on line 10. which only returns true when a new line is read. When that happens, a complete string is ready to be parsed and the first word of that string is extracted on line 15. From this, a simple communication protocol was made and used beyond. It is quite modest with only a few syntaxes.

Line 18. checks if $word1$ is equal to "WPC" which means clearing all way-points which happens on line 20. If not, and the first word equals the drones identificator given in the configuration then the current drone knows that the message is sent to it. If this is the case, then word two is selected on line 27.

If "WPA" meaning way-point add is received as $word2$, then the subsequent words are numeric coordinates and handled as such. Thus, gets added as a way-point on line 32. Be aware that the function $atof()$ used on line 32. expects a number (float) and if not provided, the entire program may crash! When the first way-point is added it is automatically loaded as first way-point on line 38. and is only used if the drone is set to way-point mode.

The keywords (protocol) mentioned here is tightly connected with the software derived in section 8.

7.3.2 0.1 Hertz

This loop does nothing else than reporting the drone last known position. Its done by sending drone identifier, P (for position) and three numerical values for latitude, longitude and altitude. This is written serially to the transceiver as shown in listing 7.8.

Listing 7.8: AUAVISAR.ino excerpt - 01hz()

```
1  void hz01()
2  {
3    /// Send vehicle position to base station
4    Serial2.print(_IDENTITY);
5    Serial2.print(" P ");
6    Serial2.print(navigation.latitude(), 8);
7    Serial2.print(" ");
8    Serial2.print(navigation.longitude(), 8);
9    Serial2.print(" ");
10   Serial2.println(navigation.altitude(), 8);
11 }
```

The values sent here is later parsed and used in section 8.

7.3.3 1 Hertz

The code in listing 7.9 is added for security reasons during the prototype period such that a drone is never lost during test phases. The algorithm for this is illustrated in figure 7.2.

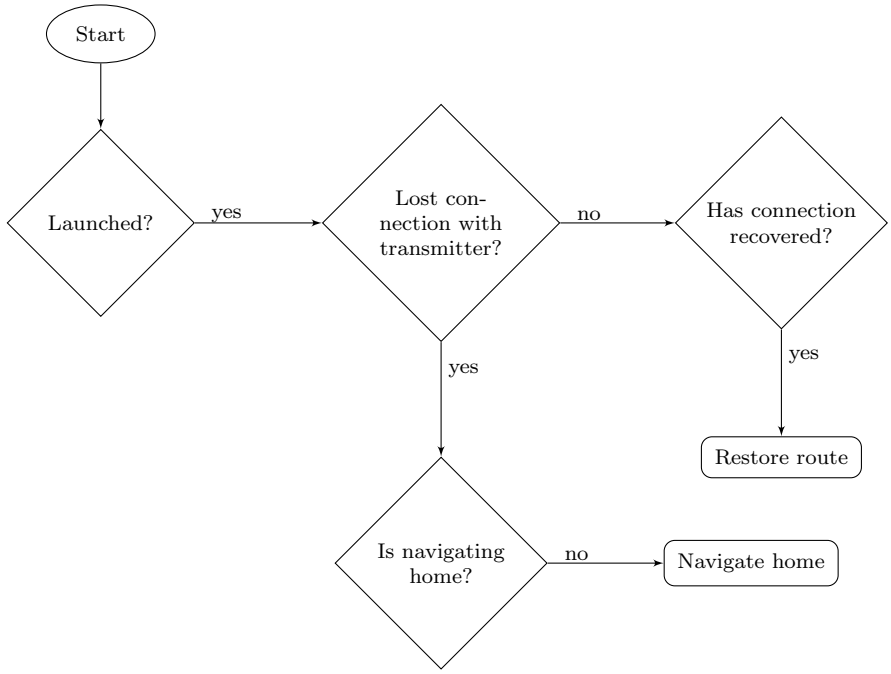


Figure 7.2: 1 Hz loop algorithm - check and handle loss of signal

Line 4-7. in listing 7.9 verifies if the vehicle has been launched or not. If it is still grounded it is not necessary to keep executing the rest. Variable $pHz1$ on line 10. is adopted from listing 7.5 and verifies that this is not the first iteration. If link has been lost and first iteration has passed, then the check on lines 15-19. decides if the vehicle should navigate home. Otherwise the link is recoved if it was lost. It is also set to resume way-point following on line 25.

Listing 7.9: AUAVISAR.ino excerpt - hz1()

```
1  void hz1()
2  {
3    /// Not launched, stop executing
4    if (!_LAUNCHED)
5    {
6      return;
7    }
8
9    /// If signal is lost, return home
10   if (pHz1 != 0 && receiver.lostLink( _PIN_THROTTLE ))
11   {
12     Serial.println("RECEIVER: no signal");
13
14     /// Has been ordered home, ignore future waypoints
15     if (!navigation.toHome())
16     {
17       navigation.home();
18       Serial.println("NAVIGATION: flying home");
19     }
20   }
21   /// If signal has been resumed, recover flight path
22   else if (receiver.recoverLink( _PIN_THROTTLE ))
23   {
24     Serial.println("RECEIVER: signal recovered");
25     navigation.loadNextWaypoint();
26   }
27 }
```

7.3.4 10 Hertz

The function which is triggered ten times per second has mainly the responsibility to check when a way-point is reached and to load next point. The overall algorithm is outlined in figure 7.3.

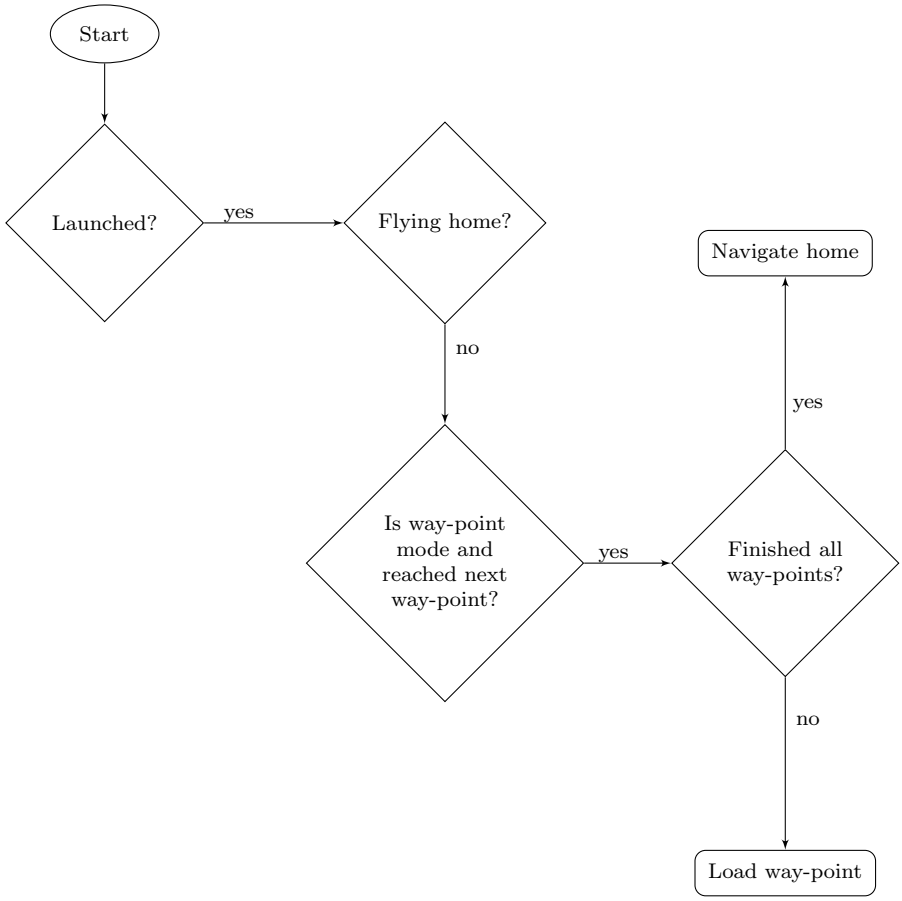


Figure 7.3: 10 Hz loop algorithm - check and handle way-points

As seen in listing 7.10, in the same way as the 1 Hz routine, unless the vehicle has been set as launched, further execution stops. The same applies if the vehicle is flying home as seen on lines 10-13. As a consequence, the vehicle will not accentually iterate to new way-points if it were to fly within one.

Listing 7.10: AUAVISAR.ino excerpt - hz10()

```
1  void hz10()
2  {
3    /// Not launched, stop executing
4    if (!_LAUNCHED)
5    {
6      return;
7    }
8
9    /// Has been ordered home, ignore future waypoints
10   if (navigation.toHome())
11   {
12     return;
13   }
14
15   /// WP-mode is set, check if next waypoint is ready
16   if (waypointMode() && navigation.next(_NAVIGATION_THRESHOLD) )
17   {
18     Serial.println("NAVIGATION: Waypoint reached");
19
20     if (navigation.loadNextWaypoint())
21     {
22       Serial.println("NAVIGATION: New waypoint loaded");
23     }
24     else
25     {
26       Serial.println("NAVIGATION: finished all waypoints, flying home");
27       navigation.home();
28     }
29   }
30 }
```

Further more, on line 16. it is checked if way-point mode is enabled and if a way-point is reached. When both these are true, an attempt to load next way-point takes place on line 20. If it fails, all way-points have been loaded and the vehicle is set to fly to origin. Notice that when set to fly home, the vehicle will take the shortest path and does not respect other vehicles or objects in its path.

Depending on GPS refresh rate and the threshold of Haversine, the explained function may have to run on a faster pace than 10 Hz.

7.3.5 100 Hertz

In this function, the result of switching between manual, aided or way-point mode is handled as illustrated in algorthihm 7.4.

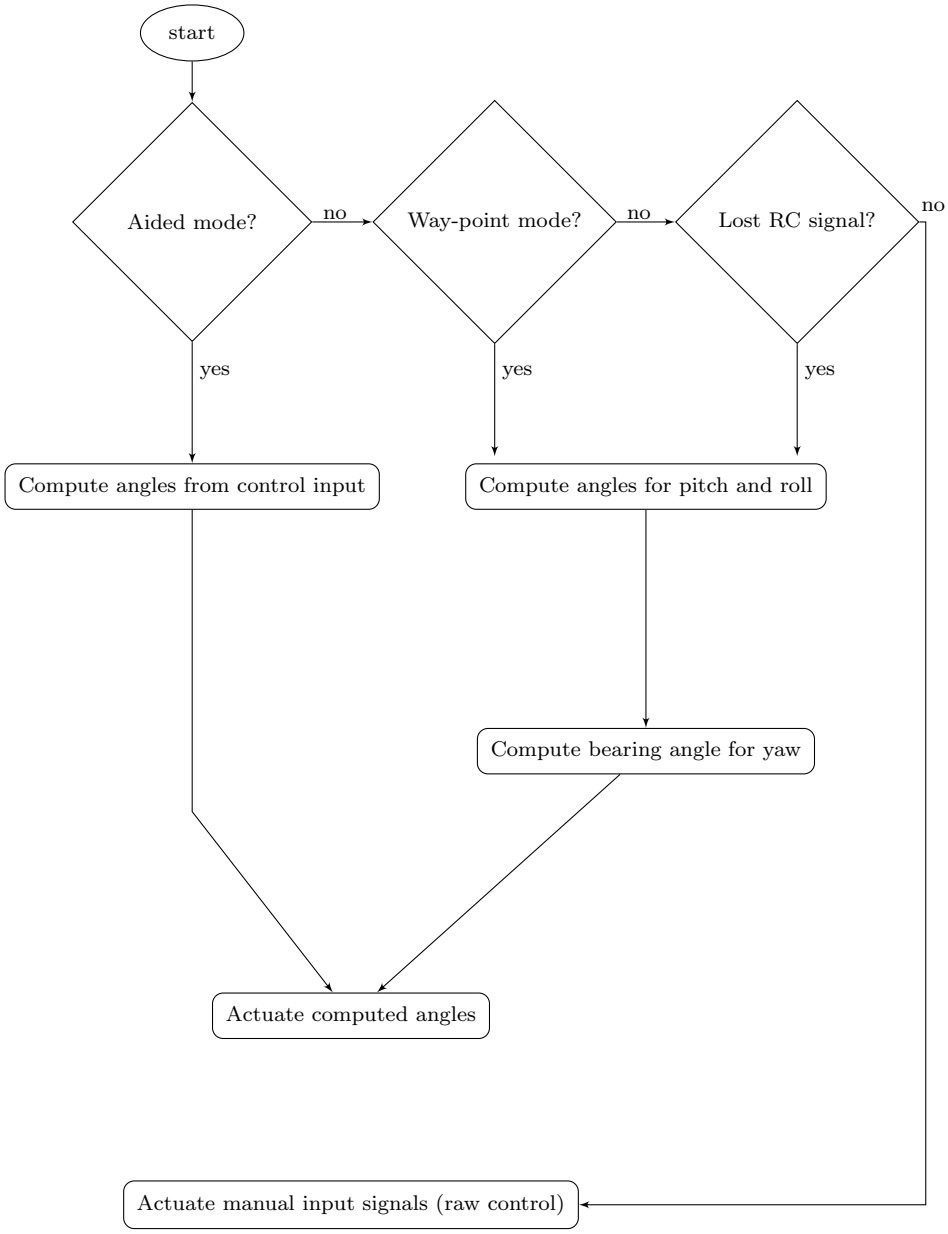


Figure 7.4: 100 Hz loop algorithm - actuate system

Line 4. of listing 7.11 determines if manual control should be used. If not, another check is performed on line 9. to separate between aided and way-point mode. Using roll and pitch angles from the AHRS, $\tilde{\phi}$ represented as *phi_e* and $\tilde{\theta}$ represented as *theta_e* are errors for roll and pitch, respectively. If drone position is known, bearing angle for $\tilde{\psi}$ represented as *psi_e* is computed on line 18. If not, line 23. set $\tilde{\psi}$ to turn left and to gain elevation on line 22. Hence, fly in circle upwards - circle climb.

Similar for aided mode, its error variables are set on lines 35-37., but uses control input from aileron and elevator as set points.

Lines 41-43. applies given constraints on actuating signals and are finally actuated on line 46.

For manual control, lines 53-55. simply parses raw PWM signals from receiver to the actuators. Lines beyond this are present for debugging purposes.

Listing 7.11: AUAVISAR.ino excerpt - hz100()

```
1  void hz100()
2  {
3    /// If either aided control OR wp-mode is selected OR the transmitter is out of
         range
4    if (aidedMode() || waypointMode() || receiver.lostLink( _PIN_THROTTLE ))
5    {
6      ///
7      /// WAYPOINT TRACKING
8      ///
9      if (waypointMode() || receiver.lostLink( _PIN_THROTTLE ))
10     {
11       /// Errors
12       phi_e = roll.stabilize( degrees(ahrs.phi())*-1, 0); /// phi_e = phi - phi_d
13       theta_e = pitch.stabilize(degrees(ahrs.theta())*-1, 0); /// theta_e = theta -
              theta_d
14
15       /// Must have enough satelites to determine own position and course
16       if (navigation.satellites() >= 4)
17       {
18         psi_e = yaw.stabilize(compass.heading(), navigation.course()); /// psi_e =
               psi - psi_d
19       }
20       else
21       {
22         theta_e -= 10; /// Make sure plane has enough pitch
23         psi_e = 20; /// Fly in circle (left) until enough satellites are available
24       }
25
26       /// Manual throttle control
27       actuate.writeMicrosecondsMotor( receiver.signal( _PIN_THROTTLE ) );
28     }
29     ///
30     /// AIDED CONTROL (CLOSED LOOP)
31     ///
```

```
32      else
33      {
34        /// errors
35        phi_e = roll.stabilize(degrees(ahrs.phi())*-1, receiver.angle( _PIN_AILERON )
              - 90);
36        theta_e = pitch.stabilize(degrees(ahrs.theta())*-1, receiver.angle(
              _PIN_ELEVATOR ) - 90);
37        psi_e = 0;
38      }
39
40      /// Limit actuating signals
41      phi_e = constrain(phi_e, - _MAX_ROLL_ANGLE, _MAX_ROLL_ANGLE);
42      theta_e = constrain(theta_e, _MAX_PITCH_ANGLE*-1, _MIN_PITCH_ANGLE*-1); ///
            theta is flipped
43      psi_e = constrain(psi_e, - _MAX_YAW_ANGLE, _MAX_YAW_ANGLE);
44
45      /// Actuate rudders according to model based on orientation errors
46      actuate.orientations(phi_e, theta_e, psi_e);
47    }
48    ///
49    /// MANUAL CONTROL OF FLYING WING (OPEN LOOP)
50    ///
51    else
52    {
53      actuate.writeMicrosecondsMotor( receiver.signal( _PIN_THROTTLE ) );
54      actuate.writeMicrosecondsLeft( receiver.signal( _PIN_AILERON ) + (1500-receiver.
            signal( _PIN_ELEVATOR )) );
55      actuate.writeMicrosecondsRight( receiver.signal( _PIN_AILERON ) - (1500-receiver
            .signal( _PIN_ELEVATOR )) );
56    }
57
58    #ifdef _ERRORS
59    Serial.print(phi_e);
60    Serial.print("\t");
61    Serial.print(theta_e);
62    Serial.print("\t");
63    Serial.print(psi_e);
64    Serial.println();
65    #endif
66
67    #ifdef _SIGNALS
68    Serial.print(receiver.signal( _PIN_THROTTLE ));
69    Serial.print("\t");
70    Serial.print(receiver.signal( _PIN_AILERON ) + (1500-receiver.signal(
          _PIN_ELEVATOR )) );
71    Serial.print("\t");
72    Serial.print(receiver.signal( _PIN_AILERON ) - (1500-receiver.signal(
          _PIN_ELEVATOR )) );
73    Serial.print("\t");
74    Serial.println(receiver.signal( _PIN_MODE ));
75    #endif
76  }
```

7.3.6 800 Hertz

This is the second fastest routine in this system and is dedicated to reading measurements from the inertial motion sensors and parse values through an orientation filter. As seen in figure 7.5, its algorithm is trivial.

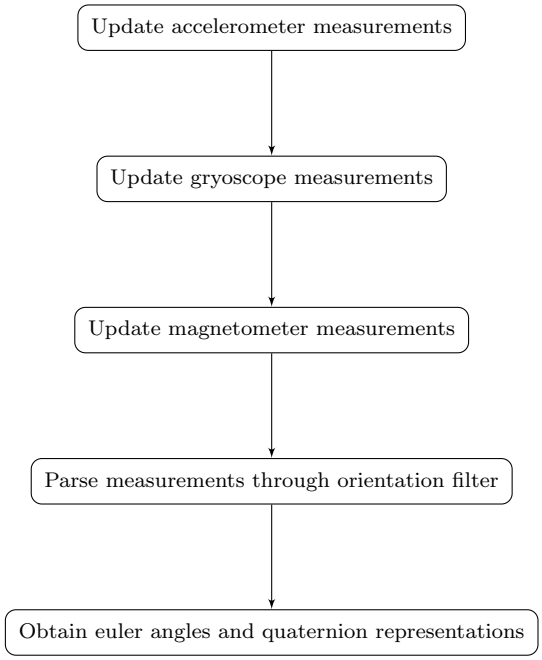


Figure 7.5: 800 Hz loop algorithm - update AHRS

As a consequence of the vast elaboration in 3, this section is almost superfluous. The first few lines of listing 7.12 measures IMU values. Because of the addressed issue of section 3.3.3 a low pass filter has been implemented on all the axes of the gyroscope. A threshold value along with the filter has been hard coded on lines 9-20.

Listing 7.12: AUAVISAR.ino excerpt - hz800()

```
1  void hz800()
2  {
3    /// Download sensors data
4    acceleration.measure();
5    gyro.measure();
6    compass.measure();
7
8    /// Altering gyro measurements due to noise! .. using low pass filter
9    if ( abs(gyro.x(0) - gyro.x(1)) > 800)
10   {
11     gyro.setAxisValue(0,0,gyro.x(1));
12   }
13   if ( abs(gyro.y(0) - gyro.y(1)) > 800)
14   {
15     gyro.setAxisValue(1,0,gyro.y(1));
16   }
17   if ( abs(gyro.z(0) - gyro.z(1)) > 800)
18   {
19     gyro.setAxisValue(2,0,gyro.z(1));
20   }
21
22
23   /// Filter data
24   //ahrs.MARG(acceleration.x(), acceleration.y(), acceleration.z(), radians(gyro.x())
         , radians(gyro.y()), radians(gyro.z()), compass.x(), compass.y(), compass.z())
         ;
25   ahrs.ARG(acceleration.x(), acceleration.y(), acceleration.z(), radians(gyro.x()),
         radians(gyro.y()), radians(gyro.z()));
26
27
28   /// Display Euler angles for visualisation
29   #ifdef _SIMULATOR
30   Serial.print(ahrs.phi());
31   Serial.print("\t");
32   Serial.print(ahrs.theta());
33   Serial.print("\t");
34   Serial.print(ahrs.psi());
35   Serial.println();
36   #endif
37
38   /// Display IMU data
39   #ifdef _IMU
40   Serial.print(acceleration.x());
41   Serial.print("\t");
42   Serial.print(acceleration.y());
43   Serial.print("\t");
44   Serial.print(acceleration.z());
45   Serial.print("\t");
46   Serial.print(gyro.x());
47   Serial.print("\t");
48   Serial.print(gyro.y());
49   Serial.print("\t");
50   Serial.print(gyro.z());
51   Serial.println();
52   #endif
53 }
```

Estimated orientation euler angles are accessible through *.phi*(), *.theta*() and *.psi*() and *.q*0(), *.q*1(), *.q*2() and *.q*3() for a quaternion representation.

7.4 Actuating

Since this was implemented for a flying wing with only two horizontal control surfaces, a model must be derived. By declaring left and right control surface as desired deflection δ_L and δ_R, respectively, the deflections can be set up as

$$\delta_L = \delta_{bias1} + \delta_{roll} + \delta_{yaw} - \delta_{theta} \tag{7.1}$$

$$\delta_R = \delta_{bias2} + \delta_{roll} + \delta_{yaw} + \delta_{theta} \tag{7.2}$$

where $\delta_{bias1,2}$ are trim variables, δ_{roll} desired roll angle, δ_{theta} desired pitch angle and δ_{yaw} is desired yaw angle. Consequently, the desired deflection angles may exceed possible deflection. The actual deflection angle must therefore be constrained and angle to be actuated becomes

$$\Delta_L = \begin{cases} \Delta_{Lmax} & \text{if } \delta_L > \Delta_{Lmax} \\ \Delta_{Lmin} & \text{if } \delta_L < \Delta_{Lmin} \\ \delta_L & \text{else} \end{cases} \tag{7.3}$$

$$\Delta_R = \begin{cases} \Delta_{Rmax} & \text{if } \delta_R > \Delta_{Rmax} \\ \Delta_{Rmin} & \text{if } \delta_R < \Delta_{Rmin} \\ \delta_R & \text{else} \end{cases} \tag{7.4}$$

where Δ_{Lmax} and Δ_{Rmax} is maximum allowed deflection of left and right control surface, respectively. Δ_{Lmin} and Δ_{Rmin} are minimum allowed deflection of left and right control surface, respectively.

This has been implemented in an actuating class which has been used in listing 7.4 and 7.11. Desired deflections angles are parsed to the function in listing 7.13 where three variables are set for δ_{roll}, δ_{theta} and δ_{yaw}. It also triggers the function in listing 7.14 which incorporates equations 7.1 and 7.3.

Listing 7.13: Actuator.cpp excerpt - orientation()

```
1 void Actuator::orientations(int phi, int theta, int psi)
2 {
3   this->_phi = phi;
4   this->_theta = theta;
5   this->_psi = psi;
6   update();
7 }
```

Considering listing 7.14 the model is implemented on lines 4-5. The restrictions

are applied on lines 7-10. to form the actual deflection angles before the being actuated on lines 13-14.

Listing 7.14: Actuator.cpp excerpt - update()

```
1  void Actuator::update()
2  {
3    /// phi and theta is (should be) between -90 and 90 deg
4    int angleLeft = 90 + this->_phi + this->_psi - this->_theta; /// 90 is in neutral
         position
5    int angleRight = 90 + this->_phi + this->_psi + this->_theta; /// 90 is in neutral
         position
6
7    if (angleLeft < 0) angleLeft = 0;
8    if (angleRight < 0) angleRight = 0;
9    if (angleLeft > 180) angleLeft = 180;
10   if (angleRight > 180) angleRight = 180;
11
12   /// Send desired movements to acutaors
13   left.write(angleLeft);
14   right.write(angleRight);
15   //motor.write(angleMotor); /// 0 is neutral position (off)
16 }
```

7.5 Discussion

This completes code review for a complete, affordable and autonomous (way-point following) drone. The system has been tested and all main components are working including the closed-loop dynamics.

The actuator code in listing 7.14 implicitly indicate that the actuating is degree based operating on an interval of 0-180. This interval is selected because the actuators physically allows horn turning of 0-180°. It does however not mean that the control surfaces have the same angular freedom. As a consequence, the system has poorer resolution than by using a microsecond based approach. With a few adjustments this could be improved for the presented class.

It has not been verified if the 800Hz loop is actually able be run at 800Hz because of other demanding code. Nevertheless, as the closed-loop is working it should not be far beyond. If it should be a problem, the sensor sampling can probably be ran at 400Hz or even 100Hz and still being able to stabilize itself.

Chapter 8

Search And Rescue Control Center

In section 1.4.10 a few requirements were stated for how a GUI should be layout. The focus on intuitive usage and functionality has been first priority. Price, compatibility, stability and performance has also been considered. Because the framework Qt Project supports embedded Linux and Windows, desktop Linux, Windows and OS X plus some other operative systems (OS) such as Android and Symbian, this was the preferred framework.

8.1 Set up

Before diving into the code, some explanation is appropriate. Qt is both a high level language and a GUI. It has classes (objects) which only work behind the scenes and other classes which are graphical objects. The latter is naturally controlled in the background such that graphical changes occur and can be perceived by the user.

The graphical is set up as both extensible markup language (XML) and Qt-objects. It is possible to set up the entire view without XML, but this must be hard coded and is not convenient. Therefore, the graphical program Qt-design has been used to place the elements at correct place. Some manual XML-edition was necessary to achieve desired positions. Luckily, the Qt-creator parses edited XML and automatically create objects in the user frame. One of the most essential objects placed in the designer for this application is the QWebFrame. This is a widget which lets

one embed web pages and provide layers for interacting with the content in the web page through javascript which is a huge advantage of Qt.

The overall program is set up as illustrated in figure 8.1 where *Clear*, *Upload*, *Quit* and some javascript are the actions the user can trigger.

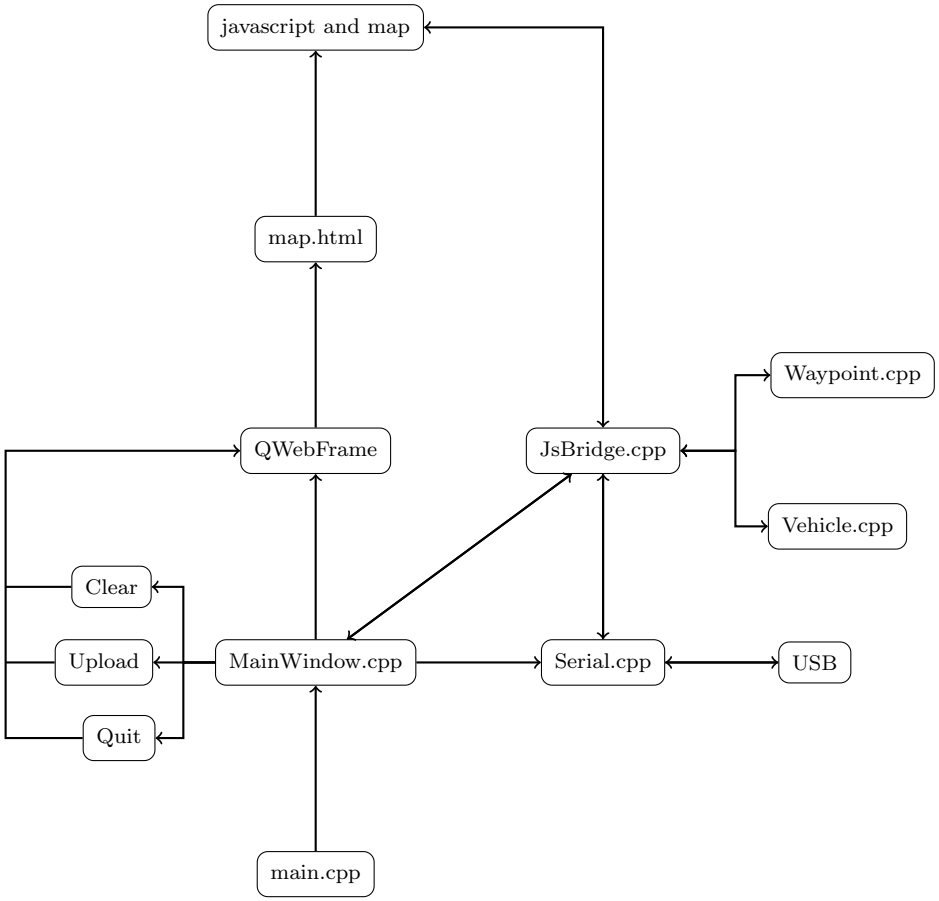


Figure 8.1: Internal program algorithm and connections

8.2 Map service

To get global map with geodetic coordinates coupled with flight photos is something that was not possible a few years a go. At least not for free with the capabilities of today. Due to many free map sources and programmer friendly API, realisation of the desired software is not far beyond. One provider which basically gives away its data is Google with their Maps-service. Due to extensive API documentation, reliable uptime and customization, this service was chosen as foundation for waypoint marking. A common problem with all these services is the requirement for a internet connection. When moving or zooming the map, new images must be downloaded. One can always store the images for later use, but may break the providers terms of usage. This is naturally solved by purchasing a license from the provider.

8.3 Web set up

To understand the following sections, knowledge of the languages HyperText Markup Language (HTML), Cascading Style Sheets (CSS) and JavaScript is advantageous.

8.3.1 Markup Language

Googles map services are written for web browsers. Instead of just routing the map service directly to the QWebFrame, there has been created a *map.html*-file which appends some customization to be explained here.

The following markup file define a typical webpage with the specified HTML, head and body-tags. Considering listing 8.1, the element on line 4. incorporates CSS which tells the parser that there should be no frames around the page for a large screen real estate.

Listing 8.1: map.html excerpt - setup

```
1  <!DOCTYPE html>
2  <html>
3    <head>
4      <style type="text/css">
5      html, body
6      {
7        width:100%;
8        height:100%;
9        padding:0;
10       margin:0;
11     }
12     </style>
13     <meta http-equiv="content-type" content="text/html; charset=UTF-8" >
14     <title>Search And Rescue Control Center</title>
15     <script type="text/javascript" src="http://maps.google.com/maps/api/js?v=3.1&
           sensor=false"></script>
16   </head>
17   <body onload="initMap('map', 63.3, 10.3);initPolyline();">
18     <div id="map"></div>
19     <script type="text/javascript" >
20     document.getElementById('map').style.width = (screen.width-23)+"px";
21     document.getElementById('map').style.height = (screen.height-95)+"px";
22     </script>
23   </body>
24 </html>
```

The *script*-element on line 15. downloads javascripts from Googles servers and load them. The map is loaded from the *onload* attribute of the *body*-element on line 17. when the body is finish parsed. The map loads inside the *div*-element on line 19. through its identifier *map*. The size for this element is set in the subsequent lines where the sizes are grabbed from the monitor resolution through *screen.width* and *screen.height*. To point to the *div*-element, the identifier is used once again. The element is reached through *document.getElementById*() and element properties are accessed and set from there.

8.3.2 JavaScript API

There are two routines which are called upon page load. The first is defined in listing 8.2 where the map options are defined in *mapOptions* and its configuration becomes self explanatory when using the service for a minute.

Listing 8.2: map.html excerpt - initMap()

```
1       function initMap(mapHolder, initLatitude, initLongitude)
2       {
3         var options = {
4           zoom: 7,
5           center: new google.maps.LatLng(initLatitude, initLongitude),
6           mapTypeId: google.maps.MapTypeId.TERRAIN,
7           draggableCursor: 'auto',
8           draggingCursor: 'move',
9           disableDoubleClickZoom: true,
10          streetViewControl: false
11        };
12        _map = new google.maps.Map(document.getElementById(mapHolder), options);
13        google.maps.event.addListener(_map, "click", mapLeftClick);
14      }
```

A new map object is created on line 12. and loaded to the given *mapHolder* which was known as *map* from earlier. Next line adds a function which triggers on mouse clicks and uses the callback function $mapLeftClick$.

The second is

Listing 8.3: map.html excerpt - initPolyLine()

```
1       function initPolyline()
2       {
3         var options = {
4           strokeColor: "#3355FF",
5           strokeOpacity: 0.8,
6           strokeWeight: 4
7         };
8
9         _polyLine = new google.maps.Polyline(options);
10        _polyLine.setMap(_map);
11      }
```

which load Googles polyline class and loads it to the map. It is used to draw lines on a map and is customizable through *polyOptions*. Line color, opacity and width have been customized where stroke color is set to blue.

Listing 8.4: map.html excerpt - mapLeftClick()

```
1     function mapLeftClick(event)
2     {
3       if (event.latLng)
4       {
5         var marker = createMarker(event.latLng);
6         _markers.push(marker);
7         _polyLine.getPath().push(event.latLng);
8         marker = null;
9       }
10      event = null;
11    }
```

The callback function in listing 8.4 gets latitude and longitude from $event.latLng['ib']$ and $event.latLng['jb']$, respectively. From lines 5-7. one can understand that a new marker is made and attached to a polyline. So when a user left click on the map, a new marker is created and will be displayed as the default icon. The icon can be changed which will be shown later. Both the marker and the line is stored in *markers* and *polyline* as lines 6-7. indicate. Actually, by pushing the coordinates to *polyLine*, a new line on the map between last created point and the current is created.

The marker function is elaborated in listing 8.5 and is called from $mapLeftClick()$.

Listing 8.5: map.html excerpt - createMarker()

```
1   function createMarker(point)
2   {
3     var marker = new google.maps.Marker({
4       position: point,
5       map: _map,
6       draggable: true
7     });
8
9
10    /// Move polyline to the new position upon drag
11    google.maps.event.addListener(marker, "drag", function()
12    {
13      for (var m = 0; m < _markers.length; m++)
14      {
15        if (_markers[m] == marker)
16        {
17          _polyLine.getPath().setAt(m, marker.getPosition());
18          break;
19        }
20      }
21    });
22
23
24    /// Delete point on click
25    google.maps.event.addListener(marker, "click", function(event)
26    {
27      /// Look for specific point
28      for (var m = 0; m < _markers.length; m++)
29      {
30        if (_markers[m] == marker)
31        {
32          marker.setMap(null);
33          _markers.splice(m, 1);
34          _polyLine.getPath().removeAt(m);
35          break;
36        }
37      }
38    });
39
40    return marker;
41  }
```

Listing 8.5 contains the last function to be explained here. Line 3. defines a new marker object at coordinates *point* and enable draggable properties. The successor $addListner()$ on line 11. adds an event to the marker object which was just created. This means that the code in the event is not executed in this scope, but when a marker is dragged. By dragged its meant when cursor is clicked when above marker and then moved. Hence, when dragged, all markers in the map is examined until the specific marker is found. When this specific marker is found, its polyline is updated to point in a new direction.

Another trigger is added for the marker upon regular clicks on line 25. A single click results in removing of the marker and its associated polyline. The marker

is also removed from the array *markers* which exists for this very reason, to handle markers after creation.

The explained code until now does not include any interfacing with program code in Qt. In fact, all the presented code is web browser compatible and can be tried or debugged there.

8.4 Qt

To understand the following sections, knowledge of the languages XML, Qt and C++ is advantageous.

8.4.1 Design

Qt uses user interface (UI) files (.ui) to define the GUI. These are XML-based files and a simplified design-file for this program has been summarized in listing 8.6.

Listing 8.6: mainwindow.ui simplified

```
1  <ui version="4.0">
2   <class>MainWindow</class>
3   <widget class="QMainWindow" name="MainWindow">
4    <widget class="QWidget" name="centralWidget">
5     <layout class="QVBoxLayout" name="verticalLayout">
6      <item>
7       <widget class="QWebView" name="webView"/>
8      </item>
9      <item>
10      <layout class="QHBoxLayout" name="horisontalLayout">
11       <item>
12        <widget class="QPushButton" name="pushButton_1"/>
13       </item>
14       <item>
15        <widget class="QPushButton" name="pushButton_2"/>
16       </item>
17       <item>
18        <widget class="QPushButton" name="pushButton_3"/>
19       </item>
20      </layout>
21     </item>
22    </layout>
23   </widget>
24   <widget class="QStatusBar" name="statusBar">
25    <widget class="QLabel" name="statusMessage"/>
26   </widget>
27  </widget>
28 </ui>
```

The first widget which encapsulate all the other is $QMainWindow$ on line 3. The first important tag after that is $QVBoxLayout$ on line 5. This is enables widgets below itself to stack vertical in the window and follow the window size

whenever resized. If it was not present, $QWebView$ on line 7. would have a fixed size no matter how large the program window was.

Since $QVBoxLayout$ conquerors most of the screen, its understandable that $QStatusBar$ on line 24. has a small space. The latter has a $QLabel$ child which provide view of regular text. Thus, custom text can be written in $QStatusBar$ which is located at the bottom of the screen.

Similar to $QVBoxLayout$, $QHBoxLayout$ set up elements, but horizontally. Since this element is declared in an *item*-tag beneath $QVBoxLayout$, the elements in the horizontal layout will not get in line with the vertical elements. Its children, are button elements for clearing the map, uploading way-points and quitting the program. Figure 8.2 illustrate how the explained design looks like. The title bar in top is a OS specific window decorator and can be omitted for more screen reel estate.

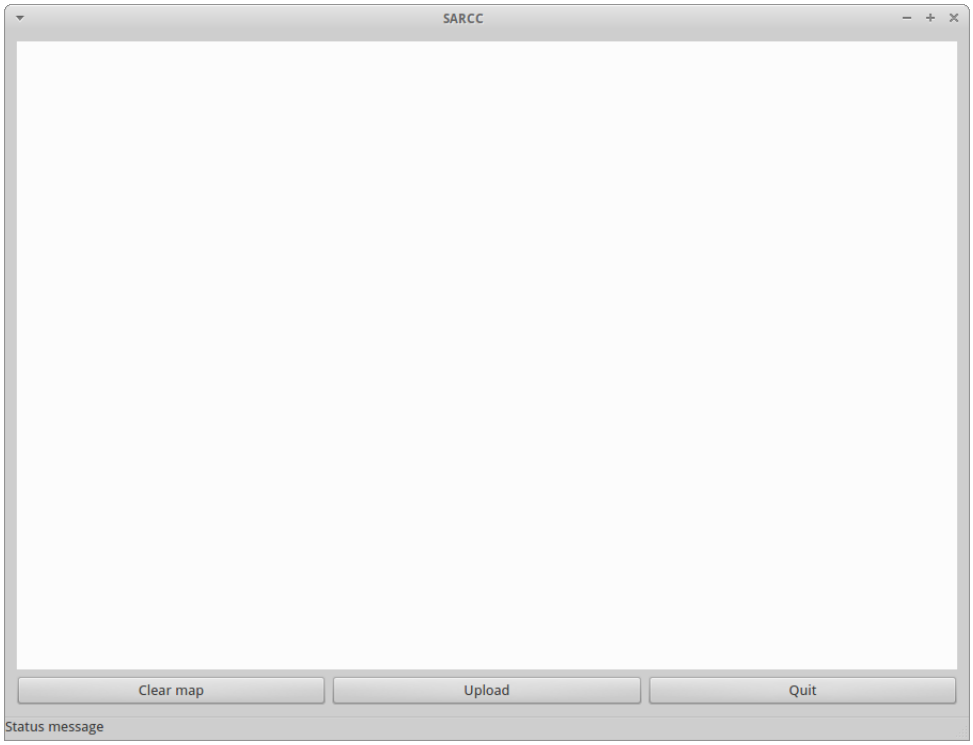


Figure 8.2: Control Center empty design

8.4.2 Main

C++ require a *main*() routine when building an application. Thus, a main.cpp file is present in this build. This file only loads up the desired Qt application and can be written as listing 8.7.

Listing 8.7: main.cpp

```
#include <QApplication>
#include "mainwindow.hpp"

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    MainWindow w;
    w.show();

    return a.exec();
}
```

8.4.3 Main window class

Listing 8.8: mainwindow.cpp excerpt - constructor

```
MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    _ui(new Ui::MainWindow)
{
    /// setup objects
    _ui->setupUi(this);
    _jsBridge = new JsBridge(this);
    _serial = new Serial(this, _jsBridge);

    /// remove scrollbars
    _ui->webView->page()->mainFrame()->setScrollBarPolicy(Qt::Horizontal, Qt::
        ScrollBarAlwaysOff);
    _ui->webView->page()->mainFrame()->setScrollBarPolicy(Qt::Vertical, Qt::
        ScrollBarAlwaysOff);

    /// Load webpage contents
    loadHtmlFromFile(QDir::currentPath()+"/map.html");
    setStatus("Loading map..");

    /// Declare object "qt" in javascript
    _ui->webView->page()->mainFrame()->addToJavaScriptWindowObject("qt", _jsBridge);

    /// connect to serial device
    if (_serial->connect("ttyUSB0",9600,static_cast<QSerialPort::DataBits>(8),
        static_cast<QSerialPort::Parity>(0),static_cast<QSerialPort::StopBits>(1),
        static_cast<QSerialPort::FlowControl>(0)))
    {
       setStatus("Connected to " + _serial->port());
    }
    else
    {
       setStatus("Unable to connect to ttyUSB0: "+_serial->error());
    }
}
```

The constructor in listing 8.8 is initialised from the previous *main*() routine and initiate additional routines such as initiate UART connection and JavaScript bridge connection. The latter is a communication handler between javascript and Qt and will be elaborated in section 8.4.5. QWebView automatically adds scrollbars, but since the map can be dragged around with the cursor they are removed on lines 11-12. Line 15. loads the content from *map.html* into the QWebFrame. An appropriate message is sent to the status bar from the line below.

To communicate with Qt, the JavaScript bridge object is made available in *map.html* through the virtual object "qt" on line 19. By using this "qt" object, all the public slots in *JsBridge.cpp* are available within *map.html*.

Lines 22-29. connects to given serial port at given baud rate. A message is set in the status bar based upon connection condition.

To load the file to QWebFrame, the routine in listing 8.9 defines a QFile object on line 2. where the file properties are checked on line 3. such that no faults should arise when trying to read it. Line 6. reads the file contents and set the QWebContent on the subsequent line.

Listing 8.9: mainwindow.cpp excerpt - loadHtmlFromFile()

```
1 void MainWindow::loadHtmlFromFile(QString _file)
2 {
3     QFile file(_file);
4     if (file.open(QIODevice::ReadOnly | QIODevice::Text))
5     {
6         QString html = QTextStream(&file).readAll();
7         _ui->webView->setHtml(html);
8     }
9 }
```

Changing status message in status bar is done with listing 8.10.

Listing 8.10: mainwindow.cpp excerpt - setStatus()

```
1 void MainWindow::setStatus(QString message)
2 {
3     _ui->statusMessage->setText(message);
4 }
```

To evaluate JavaScript in the QWebFrame, the routine in listing 8.11 have been declared public.

Listing 8.11: mainwindow.cpp excerpt - evaluateJs()

```
1 void MainWindow::evaluateJs(QString js)
2 {
3     _ui->webView->page()->mainFrame()->evaluateJavaScript(js);
4 }
```

The end user may interfere with the map, but also three additional buttons. To clear applied actions in the map, the function in listing 8.12 is triggered when the ”Clear”-button declared in section 8.4.1 is pressed.

Listing 8.12: mainwindow.cpp excerpt - clear-button

```
1 void MainWindow::on_pushButton_1_clicked()
2 {
3     setStatus("Map cleared");
4     _jsBridge->clear();
5     loadHtmlFromFile(QDir::currentPath()+"/map.html");
6     _ui->webView->page()->mainFrame()->addToJavaScriptWindowObject("qt", _jsBridge);
7 }
```

The function for the ”Upload”-button in listing 8.13 is slightly more advanced. The *_serial* action on line 4. sends a code to all connected drones with instructions to clear all current way-points such that the new ones will override all existing. This corresponds with the parser in listing 7.7. On line 6. a list to hold JavaScript is created and are filled with code which iterate through all the marked way-points in the map. *waypoint*() triggering is added on line 8. and is finally told to be uploaded on line 10. The code in the quotes are not executed at this point and is merely stored as items in the variable js. Line 12. sends the combined code to the QWebFrame where its eventually evaluated.

Listing 8.13: mainwindow.cpp excerpt - upload-button

```
1  void MainWindow::on_pushButton_2_clicked()
2  {
3      setStatus("Uploading to drones");
4      _serial->write("WPC\n"); /// WPC = WayPoint Clear = Delete all waypoints
5
6      QStringList js;
7      js << "for (var i=0;i < _markers.length;i++) {";
8      js << "qt.waypoint(_markers[i].getPosition().lat(), _markers[i].getPosition().
           lng(), 0.0);";
9      js << "}";
10     js << "qt.upload();";
11
12     evaluateJs(js.join("\n"));
13 }
```

The final function to discuss is the ”Quit”-button and since it contains just ”exit(0);” its left out of any listing.

8.4.4 Serial class

To communicate with the transceiver over USB a serial communication must be established. With QSerialPort at base, a new serial class was written to handle connection and common actions. Its constructor from listing 8.14 set up a new QSerialPort object on line 5. and uses its pointer to connect an interrupt on line 8. As a result, *readData*() in listing 8.17 is triggered whenever new data is ready. With figure 8.1 in mind, notice that *connect*() used on line 8. is a function of MainWindow.

Listing 8.14: Serial.cpp excerpt - constructor

```
1 Serial::Serial(QObject *parent, JsBridge *jsb) : QObject(parent)
2 {
3     _parent = parent;
4     _jsBridge = jsb;
5     _serial = new QSerialPort(parent);
6
7     /// When data is ready, QSerialPort::readyRead() returns true and Serial::
          readData() is called
8     _parent->connect(_serial, SIGNAL(readyRead()), this, SLOT(readData()));
9 }
```

After a serial object is created the connect function in listing 8.15 can be used to connect to given port using various parameters. From lines 4-11. an automatic connection routine exists in case the serial port is unknown. The port is set on line 18. and various UART settings are set throughout the function.

Listing 8.15: Serial.cpp excerpt - connect()

```
1 bool Serial::connect(QString port, qint64 baudRate, QSerialPort::DataBits dataBits,
      QSerialPort::Parity parity, QSerialPort::StopBits stopBits, QSerialPort::
      FlowControl flowControl)
2 {
3     /// find port automatically
4     if (port == "")
5     {
6         foreach (const QSerialPortInfo &info, QSerialPortInfo::availablePorts())
7         {
8             port = info.portName();
9             break;
10        }
11    }
12
13    if (port == "")
14    {
15        return false;
16    }
17
18    _serial->setPortName(port);
19    if (!_serial->open(QIODevice::ReadWrite))
20    {
21        return false;
22    }
23    if (!_serial->setBaudRate( baudRate ))
24    {
```

```
25          return false;
26      }
27      if (!_serial->setDataBits( dataBits ))
28      {
29          return false;
30      }
31      if (!_serial->setParity( parity ))
32      {
33          return false;
34      }
35      if (!_serial->setStopBits( stopBits ))
36      {
37          return false;
38      }
39      if (!_serial->setFlowControl( flowControl ))
40      {
41          return false;
42      }
43
44      _port = port;
45      return true;
46  }
```

Listing 8.16 is buffering data in the same way as explained in section 5.2.2, but with a more high level approach. All incoming data is stored in the variable $\_buffer$ and line 5. splits its contents using the given delimiter (newline). If $\_buffer$ contains several delimiters, the array $list$ size becomes arbitrarily. Because of this, the lines 8-11. adds all of it contents until the last index of $list$ to the string $returnValue$.

Listing 8.16: Serial.cpp excerpt - bufferUntil()

```
1  QString Serial::bufferUntil(QString data, QString delimiter)
2  {
3      _buffer.append(data);
4
5      QStringList list = _buffer.split(delimiter);
6      QString returnValue = "";
7
8      for (int i=0; i < list.length()-1; i++)
9      {
10         returnValue.append(list.at(i));
11     }
12
13     _buffer = list.at(list.length()-1);
14
15     return returnValue;
16 }
```

The buffer routine is triggered by listing 8.17 which reads all incoming data on line 3. and buffers it on line 4. If valid data is present then it is sent to the JavaScript bridge for further handle.

Listing 8.17: Serial.cpp excerpt - readData()

```
1  void Serial::readData()
2  {
3      QString raw = QString(_serial->readAll());
4      QString data = bufferUntil(raw, QString("\n"));
5
6      if (data == "")
7      {
8          return;
9      }
10
11     /// send data to parent function
12     _jsBridge->serialDataHandle(data);
13 }
```

Because its desirable that data arrives chronological from sending point the writing function in listing 8.18 incorporates waiting on line 14. An arbitrarily $bufferSize$ is used to split the *data* into chunks. A chunk is sent at line 11. and then pausing is applied for for each $bufferSize$. Lines 7-8. converts the string into bytes. $data.mid()$ returns a string that contains maximum $bufferSize$ characters of this string, starting at i.

Listing 8.18: Serial.cpp excerpt - write()

```
1  void Serial::write(QString data, int bufferSize, long milliseconds)
2  {
3      /// Send bytes then wait milliseconds
4      for (int i=0; i < data.length(); i += bufferSize)
5      {
6          /// Split into chunks
7          QByteArray qba;
8          qba.append(data.mid(i,bufferSize));
9
10         /// Send chunks
11         _serial->write(qba);
12
13         /// Wait
14         Misc::usleep(milliseconds);
15     }
16 }
```

The waiting used in listing 8.18 is part of a *Misc*-class presented in listing 8.19.

Listing 8.19: Misc.cpp

```
1  #include "Misc.hpp"
2  #include <QTime>
3  #include <QCoreApplication>
4  #include <QEventLoop>
5
6
7  void Misc::usleep(long milliseconds)
8  {
9      QTime t;
10     t.start();
11     while(t.elapsed() < milliseconds)
12     {
13        QCoreApplication::processEvents(QEventLoop::AllEvents, milliseconds-t.elapsed
               ());
14     }
15 }
```

The advantage of using this method for sleeping is that it only pauses the program at place of trigger and not the entire application. In other words, the UI is usable even though minor parts of the program is sleeping.

8.4.5 JavaScript Bridge class

From figure 8.1 its seen that this class enables a link between JavaScript and Qt. It also handles both vehicles and way-points. This is because both vehicles and way-points are graphed as such. Their classes are therefore declared in this class constructor shown in listing 8.20.

Listing 8.20: JsBridge.cpp excerpt - constructor

```
1 JsBridge::JsBridge(QObject *parent) : QObject(parent)
2 {
3     _IDENTITY_PREFIX = "F";
4     _parent = parent;
5     _vehicle = new Vehicle(this);
6     _waypoint = new Waypoint(this);
7 }
```

The identifier on line 3. is used to match the identification name programmed in the drones. In the serial data handler routine in listing 8.21 the identifier is put to use were it can be seen that it plays an essential role. For this reason, it may never be empty.

Listing 8.21: JsBridge.cpp excerpt - serialDataHandle()

```
1  void JsBridge::serialDataHandle(QString data)
2  {
3      /// Look for new vehicle positions
4      QRegExp rx("^"+_IDENTITY_PREFIX+"([^ ]*) P ([^ ]*) ([^ ]*) (.*)$");
5      if (rx.indexIn(data) != -1)
6      {
7          QStringList list = rx.capturedTexts();
8          QString id = _IDENTITY_PREFIX+list.at(1);
9          float latitude = list.at(2).toFloat();
10         float longitude = list.at(3).toFloat();
11         float altitude = list.at(4).toFloat();
12
13         /// Add vehicle if not existing
14         if (!_vehicle->exists(id))
15         {
16             _vehicle->add(id, latitude, longitude, altitude);
17         }
18         else
19         {
20             _vehicle->move(id, latitude, longitude, altitude);
21         }
22     }
23 }
```

Line 4. uses regular expressions designed to match ”identifier P latitude_number longitude_number altitude_number” where all the quoted values are shifting beside P. The single P indicate position update. Line 5. ensures that a valid match must occur before the vehicle will appear in the program. The subsequent lines extract the desired values from the string and converts them to proper format. Line 14. checks whether the vehicle is already registered and either add vehicle to map or update the existing vehicle to a new position.

Now, from listing 8.13 one can recall that a way-point function was triggered when the ”Upload”-button was clicked. That very function is located in listing 8.22 and parses the coordinates further to the way-point class.

Listing 8.22: JsBridge.cpp excerpt - waypoint()

```
1  void JsBridge::waypoint(double latitude, double longitude, double altitude)
2  {
3      _waypoint->add(latitude,longitude,altitude);
4  }
```

As a consequence of the same action, the function in listing 8.23 utilizes the stored coordinates on lines 19-26.

Listing 8.23: JsBridge.cpp excerpt - upload()

```
1  void JsBridge::upload()
2  {
3      MainWindow * parent = qobject_cast<MainWindow*>(_parent);
4      if (_vehicle->list().length() == 0)
5      {
6          parent->setStatus("No drone is present");
7      }
8      else
9      {
10         MainWindow * parent = qobject_cast<MainWindow*>(_parent);
11         QStringList vehicles = _vehicle->list();
12         QList<Waypoint::geodetic> waypoints = _waypoint->coordinates();
13         QString data;
14
15         /// Loop through all connected vehicles..
16         for (int i=0; i < vehicles.length(); i++)
17         {
18             /// ..and send all waypoints
19             for (int j=0; j < waypoints.length(); j++)
20             {
21                 /// Format serial data: ID WPA LAT LON ALT\n
22                 data = vehicles.at(i) + " WPA "+QString::number(waypoints.at(j).
                       latitude,'f',8)+" "+QString::number(waypoints.at(j).longitude,'f'
                       ,8)+" "+QString::number(waypoints.at(j).altitude,'f',8)+"\n"; ///
                        WPA = WayPoint Add = create new way point
23
24                 /// Write data to serial. Send 8 byte each 100ms
25                 parent->_serial->write(data, 8, 100);
26             }
27         }
28
29         /// Remove all waypoints till next time
30         _waypoint->clear();
31
32         parent->setStatus("Upload complete");
33     }
34 }
```

Line 3. define *parent* as a $MainWindow$ object such that public functions are reachable in this routine. Line 4. checks whether drones are connected or not, and display a message according to this. All registered vehicles are fetched on line 11. and way-points on line 12. These are undergone to get vehicle identifier and coordinates such that a string can be formed on line 22. The string is finally sent to each connected vehicle at line 25. Actually, the serial writing does not separate which vehicle it should sent to. All vehicles get the same messages and must to check them self if the message is to them.

8.4.6 Vehicle class

To be in control of all connected drones, the vehicle class stores their identifiers. The class have also JavaScript code which is sent to QWebFrame to display a drone and functions to move the appended icon when new position coordinates are received. When coordinates of a new drone is received listing 8.24 is called and identifier is stored on line 7.

Listing 8.24: Vehicle.cpp excerpt - add()

```
1  void Vehicle::add(QString id, double latitude, double longitude, double altitude)
2  {
3      if (id.length() == 0)
4      {
5          return;
6      }
7      _vehicles.push_back(id);
8
9      QStringList js;
10     js << "var position = new google.maps.LatLng(" + QString::number(latitude, 'f',
           8) + ", " + QString::number(longitude, 'f', 8) + ");";
11     js << "var image = 'file://"+QDir::currentPath()+"/icon-airplane.png';";
12     js << "var title = '" + id + "';";
13     js << "_info['"+id+"'] = '"+formatPopup(id, latitude, longitude)+"';";
14     js << "var marker = new google.maps.Marker({id: '"+id+"', position: position,
           map: _map, icon: image, title: title });";
15     js << "_vehicles.push(marker);";
16     js << "var infowindow = new google.maps.InfoWindow();";
17     js << "google.maps.event.addListener(marker,'click',function() { infowindow.
           setContent(_info['"+id+"']);infowindow.open(_map, marker); });";
18
19     /// send data to parent function
20     JsBridge * parent = qobject_cast<JsBridge*>(_parent);
21     parent->evaluateJs(js.join("\n"));
22
23     parent->setStatus("New drone connected");
24 }
```

The JavaScript on lines 10-17. is a bit chaotic and some explanation is on its place. Line 10. sets a new position for where something is going to be located in the map. $QString :: number()$ is used to give the coordinates eight decimals. Line 11. sets the path of an vehicle icon to *image* such that vehicle and marker icons are easily separated. Line 12. sets popup title to identifier and line 13. adds the popup information which is displayed when user clicks on the drone icon. Line 14. sets the new marker with the given settings. Since this is a vehicle, the marker is stored in *_vehicle* and not *_markers*. Lines 16-17. adds triggers for when the icon is clicked to open the popup and the final line sends the JavaScript. The popup function from line 13. has been defined in listing 8.25.

Listing 8.25: Vehicle.cpp excerpt - formatPopup()

```
1 QString Vehicle::formatPopup(QString id, double latitude, double longitude)
2 {
3     QStringList code;
4     code << "<strong>"+id+"</strong><br />Latitude:    "+QString::number(
          latitude,'f')+"<br />";
5     code << "Longitude: "+QString::number(longitude,'f')+"<br /><br />";
6     code << "<a href=\"javascript:void(0);\" onclick=\"qt.returnHome(\\'"+id+"\\')
          ;\">Return home</a>";
7
8     return code.join("");
9 }
```

Still considering listing 8.25, line 4. sets the identifier as a header. Coordinates are displayed below and clickable triggers for direct actions for the selected drone is below that. Its apperance can be seen in figure 8.3.

After a drone has been added to the map it must also be able to be moved as it is seldom stationary. This is done using listing 8.26.

Listing 8.26: Vehicle.cpp excerpt - move()

```
1 void Vehicle::move(QString id, double latitude, double longitude, double altitude)
2 {
3     QStringList js;
4     js << "var position = new google.maps.LatLng(" + QString::number(latitude, 'f',
          8) + ", " + QString::number(longitude, 'f', 8) + ");";
5     js << "for (var m = 0; m < vehicles.length; m++) { if (vehicles[m].id == '" + id
           + "') { vehicles[m].setPosition(position); } }";
6     js << "_info['"+id+"'] = '"+formatPopup(id, latitude, longitude)+"';";
7
8     /// send data to parent function
9     JsBridge * parent = qobject_cast<JsBridge*>(_parent);
10    parent->evaluateJs(js.join("\n"));
11 }
```

To be able to move a vehicle its identifier *id* must be known. Line 4. sets the new position and since only the identifier is known and not the object, the *_vehicle* array must therefore be undergone until the object is found. When found, the new position can be set on line 5. New popup info is updated on line 6.

To avoid double appending, the function in listing 8.27 will iterate through all vehicles and return true if given identifier already exists. This function is put to use in listing 8.21.

Listing 8.27: Vehicle.cpp excerpt - exists()

```
bool Vehicle::exists(QString id)
{
    for (int i=0; i < _vehicles.length(); i++ )
    {
        if (_vehicles.at(i) == id)
        {
            return true;
        }
    }

    return false;
}
```

8.4.7 Way-point class

This class holds the responsibility to hold geodetic coordinates of marked way-points. The coordinates are defined as a struct in way-point header and an excerpt is in listing 8.28.

Listing 8.28: Waypoint.hpp excerpt - struct

```
    public:
        typedef struct geodetic
        {
            double latitude;
            double longitude;
            double altitude;
        } Point;
```

It is explicit used in listing 8.29 when a new coordinate point is being created.

Listing 8.29: Waypoint.cpp excerpt - newPoint()

```
Waypoint::Point newPoint(double latitude, double longitude, double altitude)
{
    Waypoint::geodetic c;
    c.latitude = latitude;
    c.longitude = longitude;
    c.altitude = altitude;
    return c;
}
```

All the way-points are stored in a vector list and to append a single point; listing 8.30 has been created using the previous function in doing so.

Listing 8.30: Waypoint.cpp excerpt - add()

```
void Waypoint::add(double latitude, double longitude, double altitude)
{
   _coordinates.push_back(newPoint(latitude, longitude, altitude));
}
```

Since the list is private the function in listing 8.31 has been included to access the points outside the class and used in listing 8.23.

Listing 8.31: Waypoint.cpp excerpt - add()

```
1 void Waypoint::add(double latitude, double longitude, double altitude)
2 {
3    _coordinates.push_back(newPoint(latitude, longitude, altitude));
4 }
```

8.5 Result

The finished program is illustrated in figure 8.3. When started, it connects to the given serial device and prints status message in status bar. At the same time, the map is loaded and becomes ready for way-point marking. Figure 8.3 shows a marked path which was made by clicking the desired points in the desired order. The way-points can be removed by either clicking ”Clear” or clicking on each marker. When dragging a marker to a new position, the attached polylines will stick to it. To upload the marked way-points, one click on the ”Upload”-button is enough

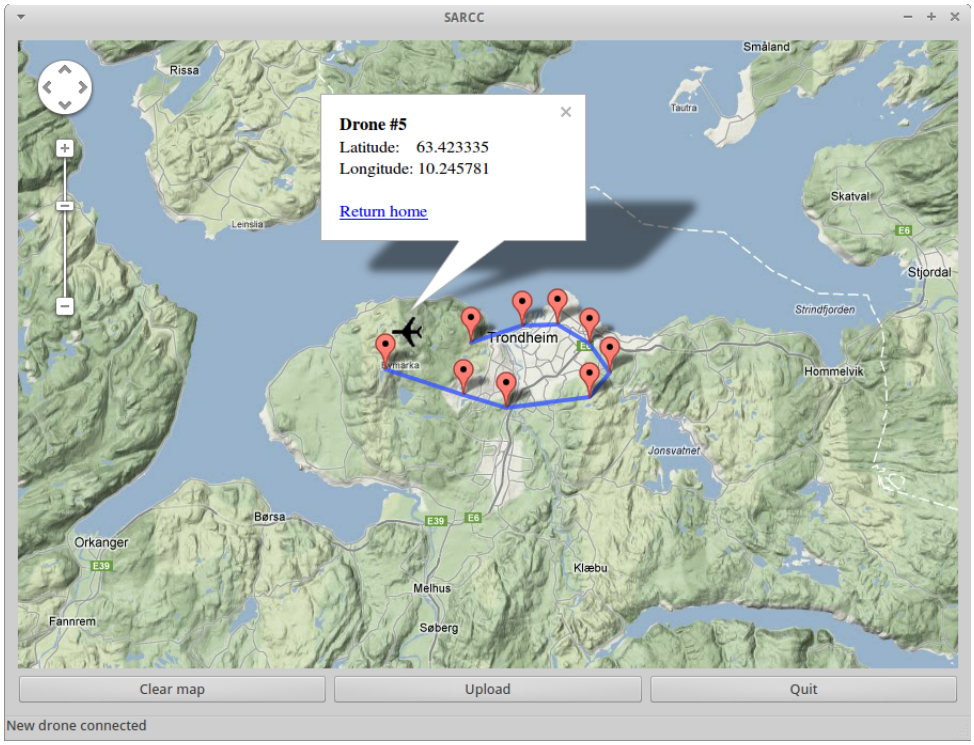


Figure 8.3: Control Center

and the points are distributed to all connected drones. The aeroplane icon in the

map is a connected drone. By clicking it, a window appears with drone identifier, coordinates and clickable commandos.

8.6 Discussion

For the end user, the program is very user friendly, simple and quick. As already mentioned, the framework is very customizable and free. There is however one drawback and one improvement which should be corrected in the future. The main drawback is the requirement of a valid internet connection. This can be omitted by downloading map data, but breaks Googles terms of usage. As a consequence a license must be purchased. If this is not desired, another map service must be used in its place.

The path planning is not very smart at this point. Since all way-points will be programmed equally to all connected drones, they will scan the same area. Therefore, a future version should program different trajectories for each drone.

To complicate the problem even more, a smart search path algorithm should be written. This could e.g. respect likely walked paths, impossible paths and more carefully searching at a specific coordinate. E.g. where the missing person was last located.

The hardware which should run the proposed software has not been discussed yet. Due to simplicity, intuitively and popularity, a robust, but lightweight touch screen enabled device could be used as station for interaction and is commercial available.

It should be mention that the way-points programmed with this software is strictly based on the geodetic coordinates given through the framework. It would be wise to verify its accuracy and if necessary, add biases to latitude and longitude to get correct coordinates.

Chapter 9

Recognize humans

Separate humans from the environment is a non-trivial problem and having to do it from above is probably even harder. Therefore, the detection of any mammal in the search region will be assumed adequate. However, by using multiple assumptions and solutions, the certainty of human findings can be increased. This section will therefore explain different techniques for distinguishing humans from the given environment.

9.1 Thermography

A thermal camera detects radiation in the infrared range of the electromagnetic spectrum and produce thermograms. Infrared radiation is emitted by all objects which means that it does not need any illumination to be perceived. The average radiation frequency increases with temperature. Therefore, warm-blooded mammals become easily visible against its surrounding when viewed through a thermal imaging camera. The same technique can be used to search for hot engine parts or bonfire which would indicate that humans are nearby.

If such a camera can be created sufficiently small, it can be attached on a supposed UAV. An onboard computer can analyse the thermograms and verify if a mammal is present.

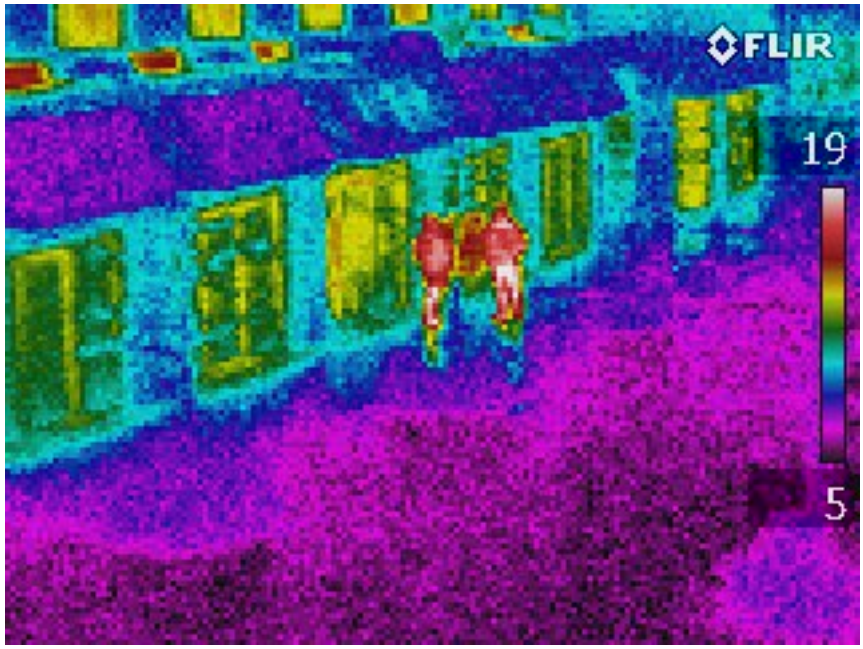


Figure 9.1: Typical thermogram including two humans

9.2 Pure vision

Using computer vision for shape or texture detection to recognize humans is the way mammals interpret vision and implementing the same computer wise is too complex to be considered in this text. Image analysis often require a large amount of computer power if rapidity is required. This can of course be omitted if good algorithms are used. By using à priori information such as skin color, cloth color and similar, computational time can be decreased.

Another advantage to exploit is the use of live imaging, if the person is conscious, he or she will probably wave at a passing drone when nearby. Algorithms detecting a typical wave-motion can be derived and may be easier to develop than for static human shapes.

If the missing person is wearing clothing, in terms of colors, which stands out from the rest of the environment, a camera equipped UAV can utilize this property to assume human localization. In general, this is a poor solution, but in cases where people are wearing unnatural coloured clothing this can be adequate. At sea, an

orange life jacket or life rafts will stand out.

When it comes to required computational power it is easy to forget the tremendous cooling options available on a plane. By equipping each processor with a special heat sink which leads heat out the wings, a lot of cooling is achieved and more computational power can be provided. Nevertheless, requiring a lot of cooling indicate that a lot of energy is wasted and energy is limited on a drone.

9.3 Human speaking frequency

When a mammal is physically looking for another of its kind, the natural thing thing to do is call on them and the other way around if a mammal wants to be found. Its therefore reasonable to assume that a conscious missing person will yell towards a drone when realizing that it is human made. This phenomenon can be exploited by using microphones and listening for frequencies in human voice. According to [Ronald J. Baken, 2000] voice frequency is between 85-180Hz for males and 165-255Hz for females. Along with low- and high-pass filters, a low level, low resource and cheap sensor could be created.

9.4 Tracking lights

When aiming a camera at a illuminating source, the lens gets blinded and the light source become bright in the resulting image. Thus, this can be used as an localization of a human and applies specially at sea were life jackets usually have lights.

Surely, places were several light sources are present can interfere and the method may not be reliable.

9.5 Single board computer image analysis proves viable

Due to the conclusion of Koteich and Rennæs [2011] an uncertainty of image analysis on single board computers was raised. Furthermore, with thermograms in mind, a simple experiment was carried out to partially answer this question. Even though this experiment is focused on thermograms, the same technique can be used to find unnatural colors in an Scandinavian environment where a missing person

are wearing clothing with unnatural colors. Of course, a better way could be used, but would most likely require more computational power.

9.5.1 Set up

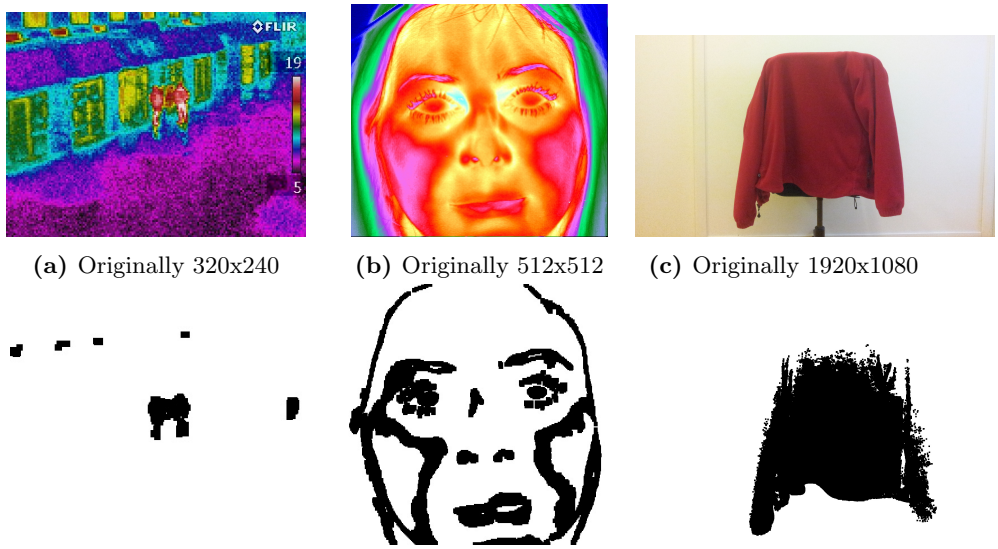
The single board computer Raspberry Pi [RaspberryPi.org] was set up using a custom bash script which can be found using appendix E and used for the experiment.

Figure 9.2: Webcam connected to Raspberry Pi

This board is not ideal as the central processing unit (CPU) is slow and even though the graphical processing unit (GPU) is powerful for it size, it is unavailable (at this point) due to proprietary firmware. On the other hand, this proves the results more imposing.

9.5.2 Comparison

Together with the Debian based distribution; Rasbian [raspbian.org], OpenCV [OpenCV.org] and a small code in appendix E, two thermograms and a HD image was analysed and timed. A simple comparison can be found in figure 9.3 where input images are on top and altered images at bottom. Keep in mind that these images has been resized for this report and that their original size was used when analysed. Resulting in notable time differences.



(a) Originally 320x240 **(b)** Originally 512x512 **(c)** Originally 1920x1080

(d) 0.021s **(e)** 0.060s **(f)** 0.460s

Figure 9.3: Time comparison of image analysing when finding red colors

Thermogram in figure 9.3a has four roof-windows leaking heat, two persons in the middle and a color bar at the right. All these are marked and by focusing on the largest area the persons position is known. The face in figure 9.3b appears to have only some red areas and not a complete red circle as one might expect. This has probably to do with the fact that the face is captured at close distance with good resolution. From a distance the head would illuminate enough heat to appear

red like in figure 9.3a.

Finally, considering figure 9.3c it becomes known that a small computer can alter full HD images at half a second and raises the question if its sufficient for a fast travelling drone.

9.5.3 Fast enough

Assuming that the velocity $v = 72km/h = 20m/s$, flying at the height $h = 40m$ above the ground and that a thermal imaging camera aimed directly at ground with viewing angles; vertical $\alpha = 35°$ and horizontal $\beta = 45°$ as illustrated in figure 9.4. The vertical height of the captured image in meters L can be calculated as

$$L = 2 \cdot \tan(\alpha) \cdot h = 2 \cdot \tan(35°) \cdot 40m \approx 56m \tag{9.1}$$

and the image width W as

$$W = 2 \cdot \tan(\beta) \cdot h = 2 \cdot \tan(45°) \cdot 40m \approx 80m \tag{9.2}$$

Considering the vertical image length L and velocity v, it is clear that one covered area will be overlapped by nearly $2/3$ if the capture rate is set at 1 frame per second (FPS). With a computation time of $0.5s$ the image can be analysed several times before a completely new one is available!

9.5.4 Resolution

Finally, at a given height h, the resolution plays an essential role. Using meters is not sufficient because unless a human is laying flat and radiating heat from the entire body, the resolution may be too poor. Therefore, a quick pixel-meter ratio must be derived.

By assuming that the thermal imaging camera has the resolution 640x480. The

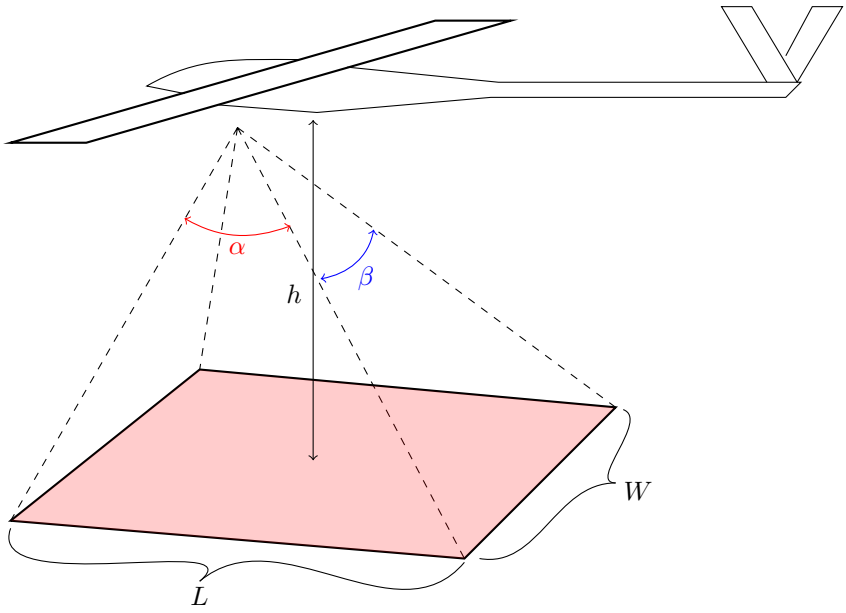


Figure 9.4: Drone camera capturing area

ratio can be calculated as

$$W_{ratio} = \frac{640px}{80m} = 8px/m = 12.5cm/px \tag{9.3}$$

$$L_{ratio} = \frac{480px}{56m} \approx 8.57px/m \approx 11.7cm/px \tag{9.4}$$

which is both sufficient for detection of a human face or projected area.

9.5.5 Discussion

To conclude, the use of a thermal imaging camera coupled with a small computer would be able to find mammals within a reasonable time. Given that the thermal imaging camera has the resolution $640x480$ and create precise pictures. This may be too generous, but if the vehicle were to fly closer to ground, the resolution can be poorer without loosing accuracy.

Through whole the experiment, it is assumed that a thermal imaging camera is small enough to be fitted on a small drone.

9.6 Searching from air

To take it a step further, it was decided to test real image analysis aloft using affordable parts to see if it was able to perform as expected. There was also a desire to check how accurate the reported drone position was with the persons actual position during transit.

9.6.1 Test rig

The rig in figure 9.5 features a Raspberry Pi connected to a HD web camera, a HD reference camera, a micro controller with an Atmel 328P processor, a FGPMMPA GPS receiver at 1 Hz, XBee 2.4 GHz transceiver and an unmanned platform.

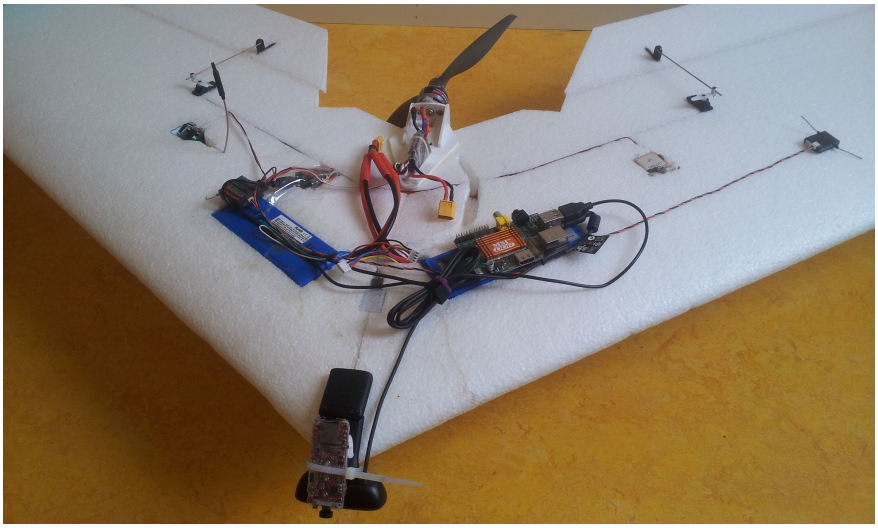


Figure 9.5: Test rig

A live version of the program used in section 9.5 can be found using appendix E and was written to communicate with micro controller which also has its own program found using appendix E. If the listed sensors are removed from figure 1.2 the remaining pieces illustrate how this system was connected. The algorithm for the overall program is shown in figure 9.6.

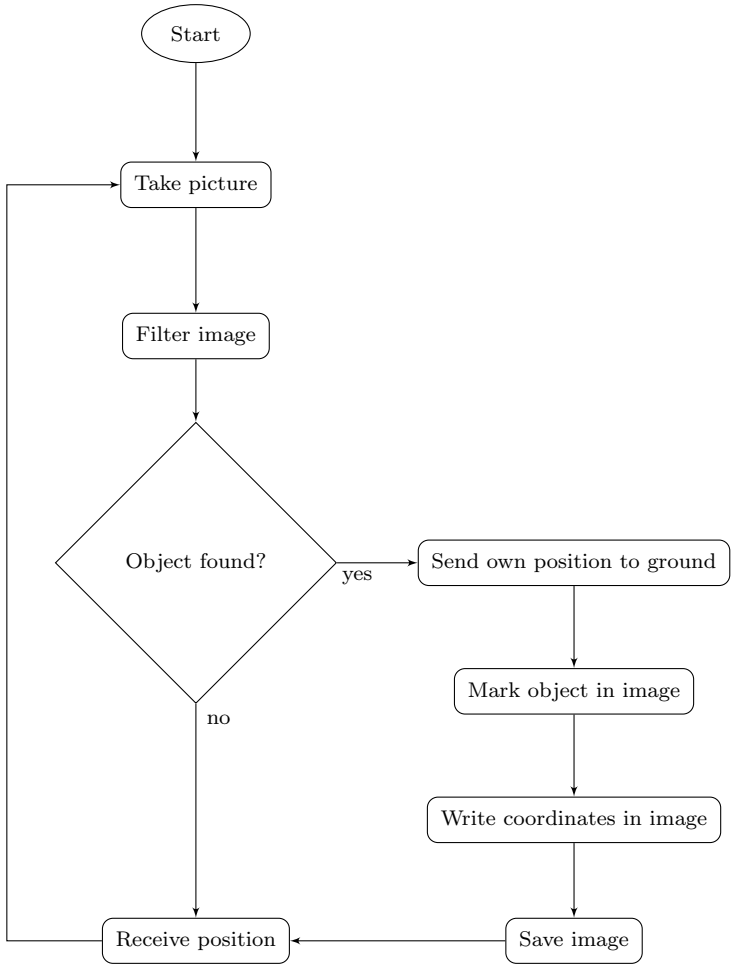


Figure 9.6: Overall searching algorithm

From figure 9.6 it is understood that the searching is executed continuously where the algorithm speed is mostly limited by the image filtering.

9.6.2 At the field

A laptop was used as ground station with a transceiver connected over a USB interface. This link was used to receive drone coordinates at 0.1 Hz intervals and drone coordinates when an assumed human was found. Disregard of the system derived in section 7, this system was piloted manually using a common RC system for best accuracy.

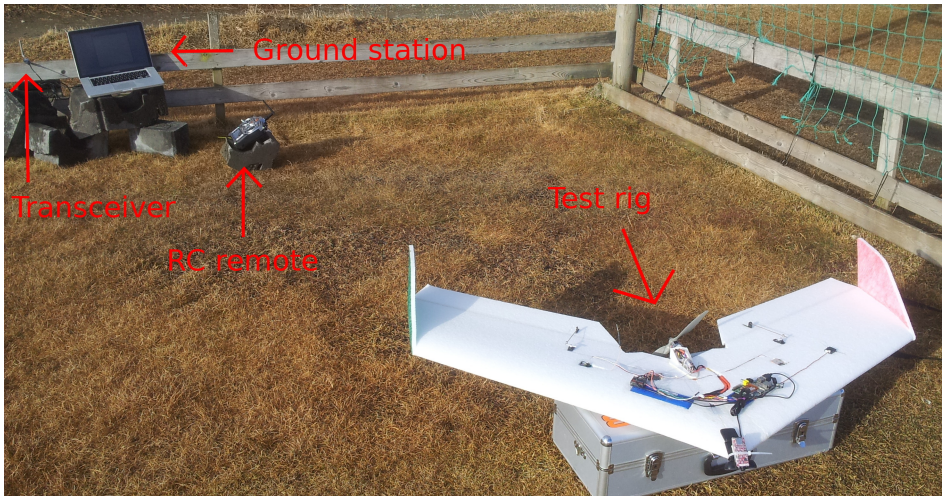


Figure 9.7: Setup at the field

With ten GPS satellites connected, the craft was laid down at a fixed position where the emulated missing person was standing throughout the experiment. At this position, all recorded positions from the GPS receiver was found to be 63.3193° latitude, 10.2726° longitude and 3.0 meters above assumed ground. These three numbers and those listed in table 9.1 is written with the estimated accuracy that the GPS receiver provide.

9.6.3 Flight

There was executed a before-experiment-test to verify the system functionality and most interestingly the communication link and camera settings. On this first flight, the camera was faced forward to get a large glimpse of the desired object. The test went fine and everything worked as expected. For instance, the figure 9.8 shows contour marking of the interesting object in green color. Know that all these image manipulations were performed aloft using only computational power inside the vehicle!

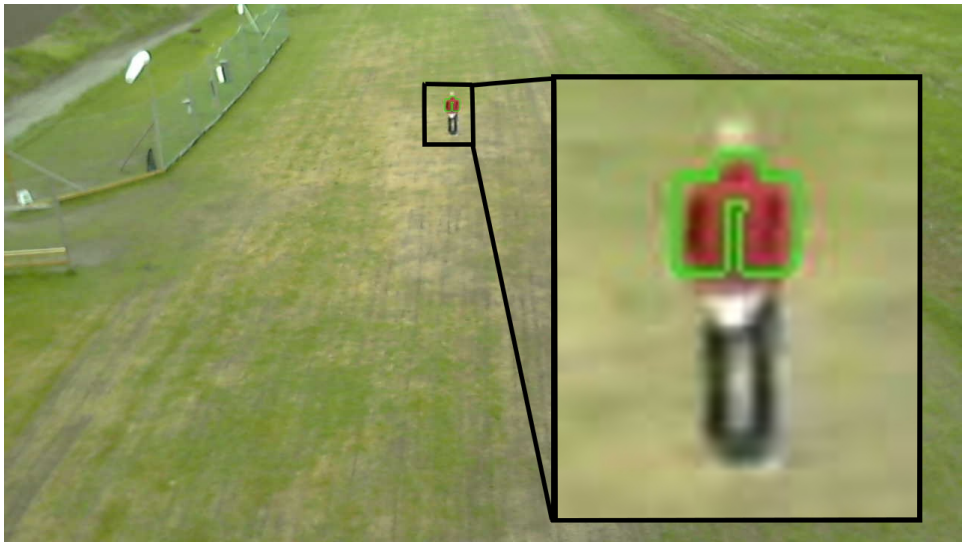


Figure 9.8: Picture taking and content marking is automated

By facing the camera downwards the person will be harder to spot as a persons projected area is usually smaller than their torso. There were now performed 27 flybys were the goal was to fly above the person and hopefully be caught on camera, analysed and reported. Figure 9.9 illustrate one of many captured pictures and has been downscaled and zoomed for this report.

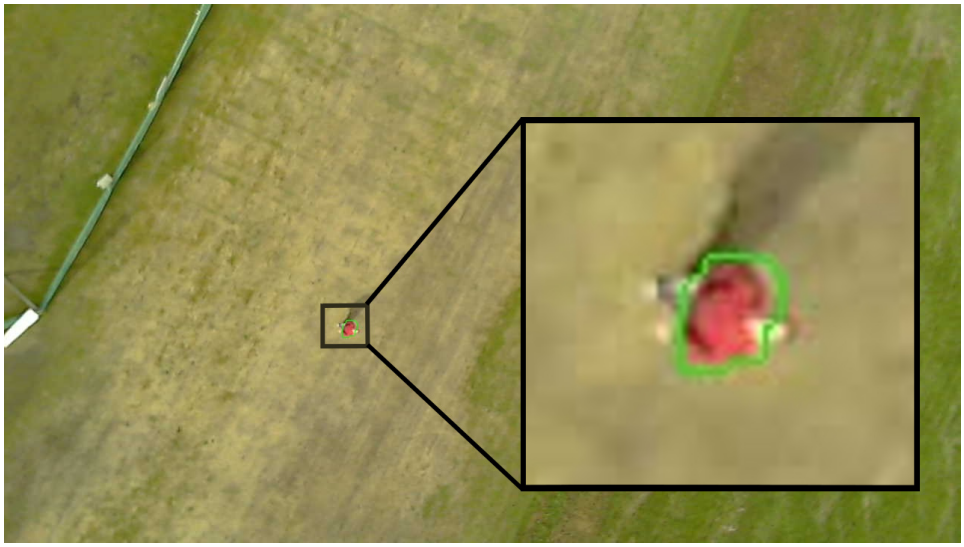


Figure 9.9: Person found below a transient drone

Additional images are found on the attached CD with NMEA codes as file name indicating their capturing position.

9.6.4 Result

The rig reported numbers of connected satellites to be between 9 and 11 at all times. This indicate that the GPS receiver should be able to estimate its position quite accurate. The recorded coordinates for each successful find has been listed in table 9.1 and the distance between each projected coordinate and where the person were standing has been calculated using equation (4.10). In figure 9.10 values of table 9.1 is presented as a histogram.

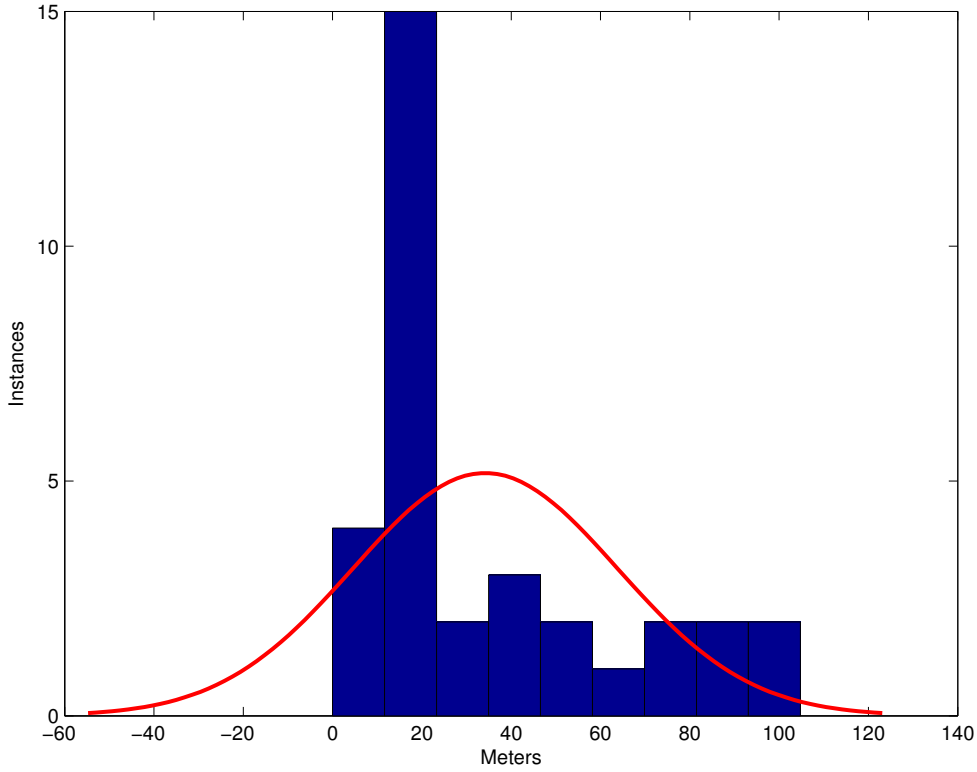


Figure 9.10: Normalized histogram of estimated distances to person

From the significant pedestal of figure 9.10 it can be concluded that a person is likely to be within a radius of 20 meters from the reported position.

Timestamp	Latitude	Longitude	Altitude	Distance
08:50:24	63.3190°	10.2736°	20.5 m	60 m
08:50:41	63.3194°	10.2727°	22.6 m	12.2 m
08:51:18	63.3193°	10.2729°	12.2 m	15 m
08:51:34	63.3194°	10.2725°	14.1 m	12.2 m
08:51:35	63.3194°	10.2724°	15.4 m	14.9 m
09:02:51	63.3195°	10.2726°	0.1 m	22.2 m
09:02:52	63.3194°	10.2725°	0.2 m	12.2 m
09:03:35	63.3190°	10.2739°	12.1 m	73 m
09:04:23	63.3192°	10.2729°	17.9 m	18.7 m
09:04:24	63.3192°	10.2728°	17.1 m	14.9 m
09:04:35	63.3194°	10.2724°	4.2 m	14.9 m
09:04:36	63.3193°	10.2723°	4.7 m	15 m
09:05:26	63.3191°	10.2733°	42.7 m	41.4 m
09:05:27	63.3191°	10.2732°	43.6 m	37.3 m
09:05:35	63.3193°	10.2733°	41.7 m	34.9 m
09:05:41	63.3194°	10.2729°	38.6 m	18.7 m
09:05:43	63.3195°	10.2728°	39.4 m	24.4 m
09:06:39	63.3193°	10.2724°	21.7 m	10 m
09:19:27	63.3191°	10.2726°	13.1 m	22.2 m
09:20:48	63.3190°	10.2745°	30.5 m	100.6 m
09:20:49	63.3189°	10.2745°	31.5 m	104.8 m
09:20:55	63.3188°	10.2740°	36.9 m	89.3 m
09:21:41	63.3189°	10.2740°	32.7 m	82.9 m
09:21:42	63.3189°	10.2739°	33.2 m	78.7 m
09:21:54	63.3192°	10.2727°	24.2 m	12.2 m
09:22:41	63.3190°	10.2735°	48.4 m	56 m
09:22:42	63.3191°	10.2735°	47.5 m	50.1 m
09:22:56	63.3193°	10.2727°	30.8 m	5 m
09:22:59	63.3193°	10.2724°	30.5 m	10 m
09:23:37	63.3191°	10.2729°	34.8 m	26.8 m
09:23:38	63.3192°	10.2729°	34.5 m	18.7 m
09:23:39	63.3192°	10.2729°	33.4 m	18.7 m
09:23:48	63.3193°	10.2726°	24.5 m	0 m

Table 9.1: Recorded geodetic coordinates and differences

Remember that the reported coordinates in table 9.1 are in fact coordinates of the plane and not the person. By knowing exact ground altitude and camera angle the persons coordinates can probably be estimated to a sufficient position.

9.6.5 Discussion

Because the GPS was only able to run at 1 Hz the estimated distance to the person could probably have been reduced if quicker position estimates were provided. It should also be mentioned that the weather the day the experiment took place was shifting from sunny, cloudy and to be raining. A shifting weather will complicate the position estimation which again can throw the distance estimation off even more. On the other hand, weather can not be expected to be perfect in SAR operations and the presented results may therefore be quite valid.

Notice that even though a cheap camera was in a fixed mount exposed to noise and vibrations the system was able to detect a person without false positives. Of the 27 flybys there was reported 33 successful finds which indicate that overlap was present several times which is a good thing in this case. If a fully complete SARD is able to report missing persons within a radius of 20 meter then the person is very lucky! Distributing a SAR team to search and call within that area the person will probably be found quick. Also, sending a manned helicopter to the same area will most likely find the person instantly.

9.7 Discussion

It is natural to cover skin to loose less heat when cold. Even though this may keep higher temperatures to be caught on camera, heat is released and can be detected. The person is however most likely to be seeking shelter and may be very hard to locate. On the other hand, this probably means that the person is likely to survive longer in the cold. To locate hidden persons it may also prove advantageous to fly in both directions to get insight from several directions. Nonetheless, a passing drone will likely grab a conscious persons attention such that the human is facing the drone. Unlike a petrol driven engine, an electric powered drone is likely to harder to hear in bad weather. Still, a thermal imaging camera would be a very powerful tool for finding mammals in AUAVISAR. Nevertheless, with this conclusion, it is assumed that the camera can be produced small, power efficient and sufficiently accurate to be placed on a small drone and perform these actions. Both [Infrared] and [Flir] sells camera cores suitable for an AUAVISAR and a quick look on the vendors websites reveals that these cameras could easily be the most expensive part on an SARD.

Due to these price concerns and assumptions, its a good idea to also enhance camera vision techniques beyond the scope of this report. An issue which arises with this method is sensing abilities at night. By using a night camera, contours can be registered, but colors are not available. If the mammal is viewing directly towards the camera, the eyes will reflect portions of the illuminated infrared light and two easily identifiable circular points should be present in the snapshot.

Chapter 10

Overall discussion

Throughout the report there has been very little focus on environmental disturbances such as side winds which is a common problem for all fixed-wing aircrafts. With a prototype underactuated in yaw, side winds will force the craft off its course and the guidance system must therefore apply roll for the plane to keep its course. This will certainly be problematic for a fixed camera solution unless equipped with a wide lens. However, by having à priori information about the wind direction the trajectory programming may be set up such that each craft is flying towards or way from the wind to minimize side-slip. Still, any flying wing is applicable for side rudders to counteract side-slip.

Future adopters (specially cybernetics) may argue why a MATLAB/Simulink approach has not been used. The very reason for this is affordability, power performance and reproduction. Even though [Skøien, 2011] points out the downsides of using micro controllers only the reasons regarding multithread and priority levels remains, but has been sacrificed for the mentioned reasons along with small power consumption.

Even though this project has been focusing on drones and their ability to replace full scale helicopters in SAR, there are no intention to so nor able to do so. An UAV can ”never” replace a manned helicopter when it comes to actually rescuing persons and picking them up from ground. Therefore, the main goal remains to augment SAR and not replace them.

Besides sending up several drones to search an area, there can be constructed a

second system where each rescue worker are stalked by a drone. With autonomous flying and cameras, the rescue worker can get a better aerial view from his standpoint through a small hand held device which is connected to the drone. If a detour lays ahead, the worker can command the drone to the unavailable area giving the worker sufficient overview such that the detour is avoided. A VTOL based frame would probably be best for this, but would most likely suffer from small operational time. The same system may also be used to deliver small objects to persons whom are stuck before being rescued. Delivery of food, water and bandages to these persons may be the difference between life and death in some situations.

Chapter 11

Further work

Besides fusing all required components and thoroughly testing a single drone. The following bullets points are advises for future research areas for SARDs. To solve these sufficiently the overall goal must be put to mind.

- Replace the camera used in section 9 with an actual thermal imaging camera and thoroughly test that the system is able to find a person from a passing drone.
- Append altitude and velocity sensor and verify that the drone is able to keep its desired ground distance and velocity evenly. For precise velocity measurements it is common to use Pitot tubes and differential pressure sensors. This will work on for an UAV, but bear in mind that the tube is vulnerable and other methods should be considered for a SARD. With an altitude system present, an auto landing procedure can be implemented.
- If one know the area for where the missing persons should be within. Create an algorithm to automatically program way-points for N-number of drones for an optimal path planning. If a recipe for a complete single SARD is available, then put together several drones and test the path system.
- Set up a complete launch routine for the operators where as much as possible is automated. This may include the use of inertial sensors to detect launch and to actuate actuators to get aloft. Elevation approaches should also be investigated. E.g. circle climb.

Chapter 12

Conclusion

The need of using drones in search and rescue (SAR) operations in modern society has been stated. For the SAR personnels convenience, it is desirable that the vehicle should be as simple as possible to avoid having a specialist standby at each operation and secondly; act as much on its own as possible, hence autonomous. The advantages of realising and distributing such a system will assist the majority. The underlying technology can be used in several applications and it is therefore in NTNUs interest to research ways of doing this.

The generality of the derived components such as the attitude and heading reference system (AHRS) will certainly be useful for the university in the years ahead. Being able to use an affordable AHRS with available source code in stead of a commercial product has lowered the threshold for realising applications requiring orientation knowledge.

Starting out with a small budget to assemble an affordable, robust and unmanned aerial vehicle for SAR operations proved difficult and time consuming. However, an airframe designed for the tough handling one would expect of SAR personnel has been suggested, tested thoroughly and proven worthy.

With an on-board emulation of a thermal imaging camera the project has proven that an automated system can locate the presence of a human being below a passing drone with an accuracy of 20 meter radius. The generality of this small and affordable vision system can be used for various tasks beyond SAR.

Because all the software of the major system components has been developed during this thesis and non-free software has been avoided the affordability has been respected. Surely, with this affordable approach, SAR departments has one less reason to refuse to utilize such a system and the advantages this would contribute. Hence, with this report in hand, future adopters should not have any unexpected expenses when copying the system to connect several drones into a complete working system. Ready to be distributed to SAR personnel around the country helping keeping you, me and our children alive.

Bibliography

Ira H. Abbott and Albert E. Von Doenhoff. *Theory of wing sections*. 1959.

Aerovision. Fulmar, 2012. http://www.aerovision-uav.com/ (2012-09-27).

an.no. Dobbelt så mange går seg vill i fjellet enn for ti år siden. Sep 2012. http://www.an.no/nyheter/article6142830.ece (2012-09-05).

ARPRO. Expanded polypropylene. http://epp.com (2012-03-27).

Edgar Bjørntvedt. Instrumentering av autonomt ubemannet fly: Cyberswan. Master's thesis, 2007.

Mark Cummins. *Probabilistic Localization and Mapping in Appearance Space*. PhD thesis, 2009.

James D. DeLaurier. An ornithopter wing design. Technical report, 1994.

Department of Economic and Social Affairs. World population prospects. Technical report, 2012. http://esa.un.org/wpp/unpp/panel_population.htm (2012-09-06).

Dryden Flight Research Center. Droid, 2012. http://www.nasa.gov/centers/dryden/Features/auto-gcas.html (2012-09-27).

Mikael Kristian Eriksen. Ground station and hardware peripherals for fixed-wing uav: Cyberswan. Master's thesis, 2007.

Festo. Bird flight deciphered, 2011. http://www.festo.com/cms/en_corp/11369_11378.htm (2012-09-23).

Flir. flir.com. http://www.flir.com/cvs/cores/index.cfm (2013-04-25).

Forsvarsdepartementet. Forskrift om fotografering mv fra luften. http://www.lovdata.no/for/sf/fo/to-19970106-0003-0.html (2012-04-02).

Thor Inge Fossen. *Handbook of Marine Craft Hydrodynamics and Motion Control.* Wiley, 2011.

John Haugan. *Formler og tabeller.* 1992.

E.L. Houghton and P. W. Carpenter. *Aerodynamics for engineering students.* 2003.

Hovedredningssentralen. Statistikk for hovedredningssentralene (samlet). Technical report, 2012.

Jon Bernhard Høstmark. Design of instrumentation for autonomous fixed wing uav. Technical report, 2006.

Jon Bernhard Høstmark. Modelling simulation and control of fixed-wing uav: Cyberswan. Master's thesis, 2007.

P&R Infrared. pr-infrared.com. http://www.pr-infrared.com/shop/category/camera-cores/ (2013-04-25).

A. C. Kermode. *Mechanics of Flight.* 2006.

Mohammad Ali Koteich and Karsten Rennæs. Overall technical uav solution. Technical report, 2011.

Michael A. Goodrich Lanny Lin, Michael Roscheck and Bryan S. Morse. *Supporting Wilderness Search and Rescue with Integrated Intelligence.* PhD thesis, 2010.

Luftfartstilsynet. Bruk av ubemannede luftfartøy i norge. http://www.luftfartstilsynet.no/regelverk/aic-n/article1021.ece (2013-04-02).

Sebastian O.H. Madgwick. *An efficient orientation filter for inertial and inertial/-magnetic sensor arrays.* PhD thesis, 2010.

National Aeronautics And Space Administration. How lift is generated. http://www.grc.nasa.gov/WWW/k-12/airplane/lift1.html (2013-06-05).

Michael C. Y. Niu. *Airframe Structural Design.* 1988.

OpenCV.org. Opensource computer vision. http://opencv.org/ (2012-10-01).

PC-Aero. Pc-aero. http://pc-aero.de/ (2013-04-29).

Post- og teletilsynet. Forskrift om radioamatørlisens, a. http://www.lovdata.no/cgi-wift/ldles?doc=/sf/sf/sf-20091105-1340.html (2012-04-02).

Post- og teletilsynet. Generelle tillatelser til bruk av frekvenser, b. http://www.lovdata.no/ltavd1/filer/sf-20120119-0077.html (2012-04-02).

Qt Project. Cross-platform application and ui framework, 2013. http://qt-project.org/ (2013-03-08).

RaspberryPi.org. An arm gnu/linux box. http://www.raspberrypi.org/ (2012-10-01).

raspbian.org. Debian optimized version for raspberry pi. http://www.raspbian.org/ (2012-10-01).

Casey Reas and Benjamin Fry. Processing, 2001. http://processing.org (2013-03-11).

Ph.D. Ronald J. Baken, Robert F. Orlikoff. *Clinical Measurement of Speech Voice 2e*. Cengage Learning, 2000.

Daniel P. Shepard, Jahshan A. Bhatti, and Todd E. Humphreys. Evaluation of smart grid and civilian uav vulnerability to gps spoofing attacks. 2012. http://radionavlab.ae.utexas.edu/publications/evaluation-of-smart-grid-and-civilian-uav-vulnerability-to-gps-spoofing-attacks (2012-10-23).

Kristoffer Rist Skøien. A modular software and hardware framework with application to unmanned autonomous systems. Master's thesis, 2011.

Kristoffer Rist Skøien and Håvard Vermeer. General platform for unmanned autonomous systems. Technical report, 2010.

Paul Stoffregen. Teensy 3.0, 2012. http://www.pjrc.com/store/teensy3.html (2013-03-11).

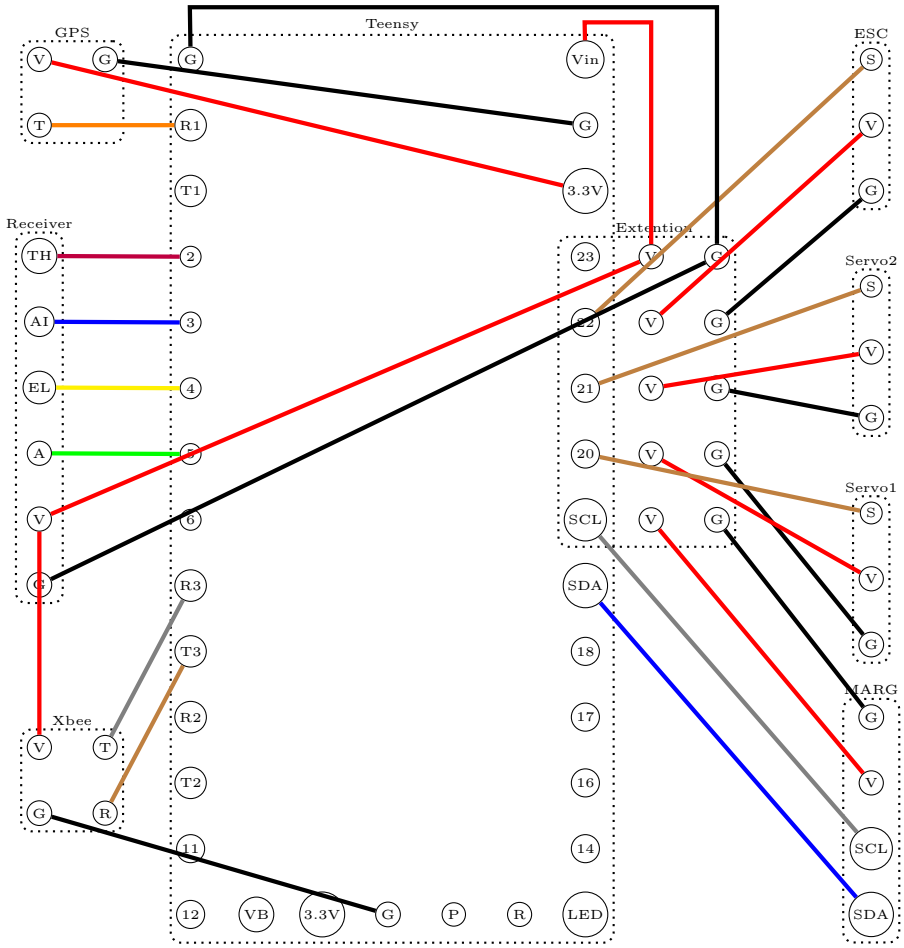
Muhammad Haris Afzal Val´rie Renaudin and G´rard Lachapelle. Complete triaxis magnetometer calibration in the magnetic domain. 2010.

G.K. Egan W.Y. Kong and T. Cornall. *Feature Based Navigation for UAVs*. PhD thesis, 2006.

Wujun Fu Zhan Lin, Zengcai Liu. Lithium polysulfidophosphates: A family of lithium-conducting sulfur-rich compounds for lithium-sulfur batteries. June 2013.

Appendix A

Complete wiring scheme



Appendix B

System Components

Component	**Quantity**	**Weight**	**Sum weight**	**Price ex. VAT**
Plane	1	390	390	$97
Battery	2	368	736	$52
Motor	1	66	66	$14.5
Gold connectors	1	9	9	$1.5
Propeller and spinner	1	25	25	$4
ESC	1	30	30	$7
Servo	2	16	32	$25
Microcontroller	1	8	8	$26
GPS receiver	1	10	10	$25
Datalink	2	3	6	$70
10DOF sensor board	1	10	10	$19
Embedded computer	1	45	45	$35
Web camera	1	91	91	$30
Misc wires	10	5	50	-
Totals			**1498 gram**	**$406**

Appendix C

How To Burn Code

There are at least two ways of burning code to the Teensy board present in the prototype drone, using the Arduino IDE environment or makefiles. The latter is included on the CD and is explained further in the second list below.

Using Arduino IDE

1. Connect USB.
2. Load Arduino IDE to your computer from the instructions provided by the Teensy developers (see bibliography).
3. Locate the desired program from CD.
4. Load the .ino file.
5. Select correct processor family and port from the menu.
6. Upload.

Using makefiles

1. Connect USB.
2. Navigate to the makefiles on the CD.
3. Edit the *main.cpp* to include your .ino file (program)
4. Run *make* from commando line interface in the same directory.
5. The files present in the folder will compile and upload the program.

Appendix D

How To SARCC

To get Search And Rescue Control Center up and running it must be configured with *qmake* and compiled with *make*.

Unless you have QtSerialPort present on your system it will have to be downloaded and installed. To install QtSerialPort do the following

1. Go to ”http://qt-project.org/wiki/QtSerialPort” and download latest version
2. Run ”git clone git://gitorious.org/qt/qtserialport.git”
3. Browse to the downloaded directory
4. Run ”qmake qtserialport.pro”
5. Run ”make”
6. Finally run ”sudo make install”

To get SARCC up and running do the following

1. Edit ”SARCC.pro” to update qtserialport path to match your own
2. Run ”qmake SARCC.pro”
3. Run ”make”
4. If no errors were present; start the application with ”./SARCC”

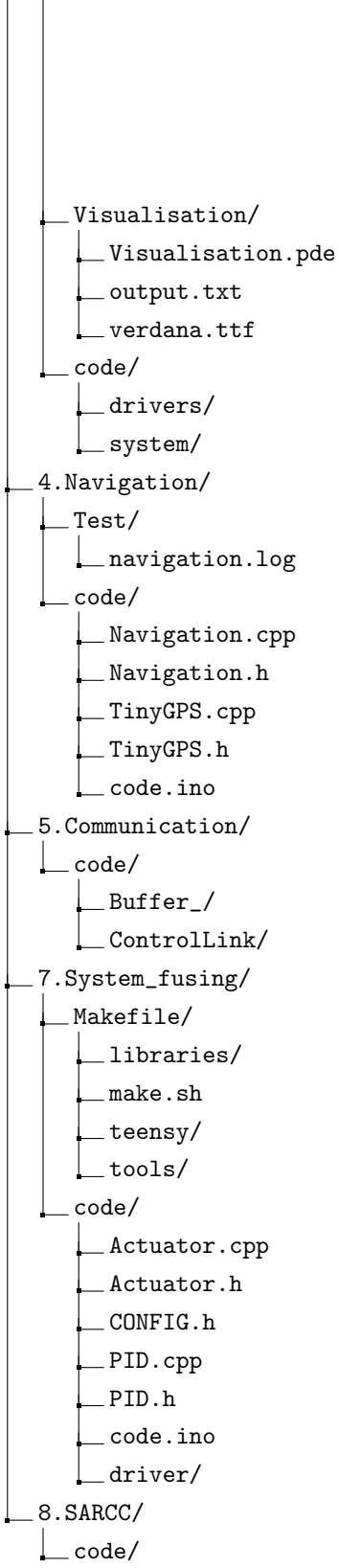
To further develop the software it is advised to use the development IDE Qt-Creator. It is available on all major platforms and when installed do the following

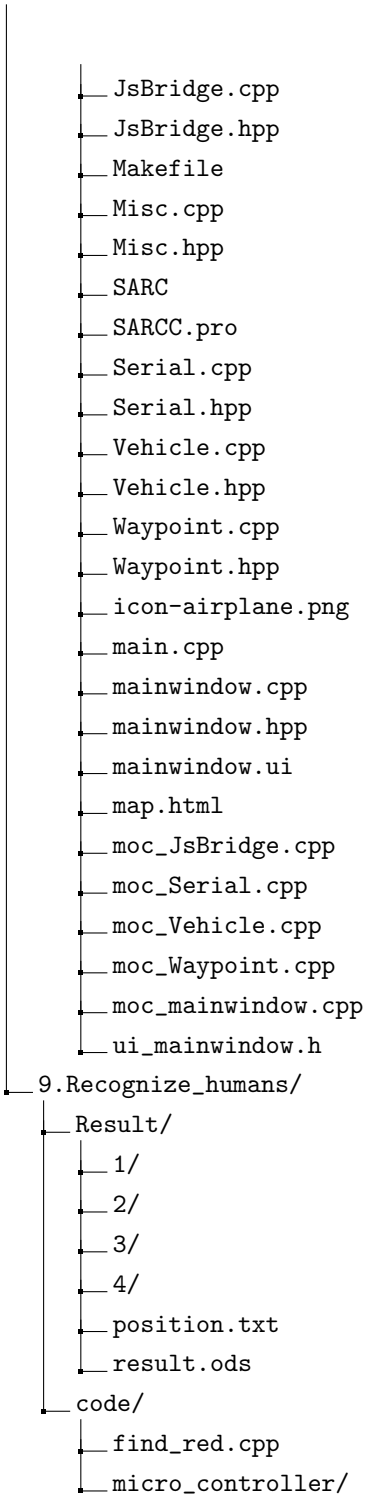
1. Start Qt-Creator
2. Choose to open file or project
3. Navigate to the file ”SARCC.pro”
4. Do not load old user settings if asked
5. Select proper Qt build options from the next dialog then click ”Configure project”

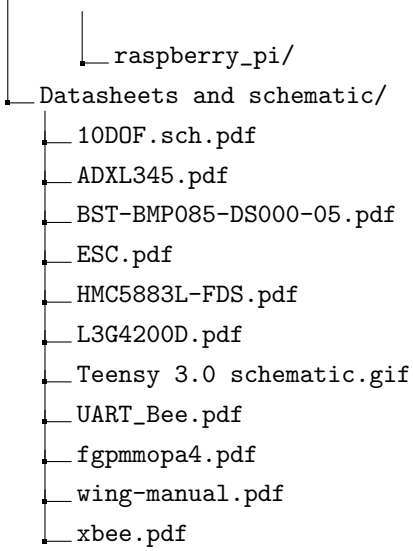
Appendix E

CD Folder Structure

```
CD
└── 3.AHRS/
    ├── AHRS.mp4
    └── Plots/
        ├── phi_0-10s.eps
        ├── phi_0-51s.eps
        ├── phi_10-20s.eps
        ├── phi_20-30s.eps
        ├── phi_30-40s.eps
        ├── phi_40-51s.eps
        ├── psi_0-10s.eps
        ├── psi_0-51s.eps
        ├── psi_10-20s.eps
        ├── psi_20-30s.eps
        ├── psi_30-40s.eps
        ├── psi_40-51s.eps
        ├── theta_0-10s.eps
        ├── theta_0-51s.eps
        ├── theta_10-20s.eps
        ├── theta_20-30s.eps
        ├── theta_30-40s.eps
        └── theta_40-51s.eps
```


Visualisation/
Visualisation.pde
output.txt
verdana.ttf
code/
drivers/
system/
4.Navigation/
Test/
navigation.log
code/
Navigation.cpp
Navigation.h
TinyGPS.cpp
TinyGPS.h
code.ino
5.Communication/
code/
Buffer_/
ControlLink/
7.System_fusing/
Makefile/
libraries/
make.sh
teensy/
tools/
code/
Actuator.cpp
Actuator.h
CONFIG.h
PID.cpp
PID.h
code.ino
driver/
8.SARCC/
code/


JsBridge.cpp
JsBridge.hpp
Makefile
Misc.cpp
Misc.hpp
SARC
SARCC.pro
Serial.cpp
Serial.hpp
Vehicle.cpp
Vehicle.hpp
Waypoint.cpp
Waypoint.hpp
icon-airplane.png
main.cpp
mainwindow.cpp
mainwindow.hpp
mainwindow.ui
map.html
moc_JsBridge.cpp
moc_Serial.cpp
moc_Vehicle.cpp
moc_Waypoint.cpp
moc_mainwindow.cpp
ui_mainwindow.h
9.Recognize_humans/
Result/
1/
2/
3/
4/
position.txt
result.ods
code/
find_red.cpp
micro_controller/


raspberry_pi/
Datasheets and schematic/
10DOF.sch.pdf
ADXL345.pdf
BST-BMP085-DS000-05.pdf
ESC.pdf
HMC5883L-FDS.pdf
L3G4200D.pdf
Teensy 3.0 schematic.gif
UART_Bee.pdf
fgpmmopa4.pdf
wing-manual.pdf
xbee.pdf