Algorithms for controlling and tracking UAVs in indoor scenarios

Andrea Nisticò

Supervised by: Marco Baglietto, Fulvio Mastrogiovanni

DIBRIS - Department of Informatics, Bioengineering, Robotics, and Systems Engineering

University of Genova

In partial fulfillment of the requirements for the degree of

Robotics Engineering

September 18, 2015

Acknowledgements

Don't forget to acknowledge your supervisor!

To all the Master and PhD students of Robotics Engineering at the University of Genova.

Abstract

Quadcopter, also known as quadrotor, is a helicopter with four rotors. The rotors are directed upwards and they are placed in a symmetric formation with equal distance from the center of mass. The quadcopter is controlled by adjusting the angular velocities of the rotors which are driven by electric motors. An on board autopilot, namely PiXHawk with PX4 software, packs all the interfaces for sensors, motor controllers and radio antennas. It also manages the control, state estimation and signals processing. The goal of this master thesis is to develop algorithms for UAV trajectory planning and execution (as well as the related SW components), using the motion capture system as the source of the UAV position feedback. An IRIS quadrotor it is used form 3d Robotics. After an introduction to the components, the integration between them both hardware and software is presented. The model of the IRIS is briefly derived and explained as well as the control techniques involved in the open source autopilot. Moreover a software architecture is presented where a task based program is able to read a set of tasks from a list and let the robot execute them sending position set point through radio link. At the end an algorithm for tracking and landing on a mobile platform is explained and the overall results are presented.

Contents

List of Figures

Chapter 1

Introduction

Summary

This chapter will introduce the reader to the topic of this master thesis. It will present the problem statement and the followed work plan.

1.1 Overview

In the latest years, aerial robotics has grown so fast both in technology and popularity among the public and multirotor aircrafts such as quadcopters are one of the most promising fields. In fact those type of vehicles have become a standard platform for research in many laboratories , driven by totally different customer needs the research improved substantially. They have sufficient payload and flight endurance to support a number of indoor and outdoor applications, and the improvements of battery and other technology is rapidly increasing the scope for commercial opportunities. They are highly maneuverable and enable safe and low-cost experimentation in mapping, navigation, and control strategies for robots that move in three-dimensional space. Small quadrotors have been demonstrated for exploring and mapping 3-D environments; transporting, manipulating, and assembling objects; and acrobatic tricks such as juggling, balancing, and flips.

As stated before, the uses of this platform are many and limited only by the imagination of the designers. Figure 1.1 illustrates various application going from search and rescue to public defense, most of them in outdoor scenarios. Very interesting is to investigate the possible use of this technology in an indoor scene where the comforting signal of the GPS is not present and analyze every aspect of flight going from state estimation, control and stability to autonomy, path planning and high level automation.

Figure 1.1: Different applications of multirotors

This work investigates the integration between different heterogeneous entities as well as the possibility to design a safe, reliable and flexible architecture capable of letting the robot navigates safely in a closed environment. At the end a simple algorithm for landing on a mobile platform is presented with very interesting results demonstrating the potential of this setup.

1.1.1 Hosting laboratories

This thesis is carried out as a cooperation between the Laboratorium Lab (focus: autonomous robotics and ambient intelligence), the NOCC Lab (focus: control, identification and estimation) and the upcoming joint EMARO@DIBRIS Lab.

1.2 Problem statement

The goal of this master thesis to develop algorithms for UAV trajectory planning and execution (as well as the related SW components), using the motion capture system as the source of the UAV position feedback. Once achieved stability more complex aspect are investigated, a task based software architecture is presented and a simple technique for landing on a mobile platform is proposed.
The above stated, is formulated in to the following problem statement for the project:

It is possible, through position feedback given from a motion capture system, to design a set of algorithms and SW components enabling the IRIS to achieve stability and perform different tasks in an indoor scenario?

1.2.1 Work plan

The work got through several tasks covering all aspects, they can be distinctly divided in main blocks hence defining the work plan.

1. Analysis of state of the art approaches to UAV control, modeling, trajectory planning and execution
2. Technical analysis and documentation on the given tools (Autopilot, Motion Capture, IRIS Quadcopter)
3. Integration between mocap and IRIS
    - Integration and testing between mocap and onboard estimation modules
    - Integration and testing of onboard controller through set point control
4. Design of the high level software architecture
5. Testing the software, coding and debugging different kind of tasks
6. Design and testing an algorithm for landing on a mobile platform

Chapter 2

Related work

Summary

This chapter synthesizes the preliminary literature review explaining existing research, products end systems. It analyzes some robotics platforms and tools, in particular it presents an overview of existing autopilots boards and softwares. A suitable modeling technique for this kind of vehicles is described introducing additional non linearities and explaining when the model uses them.

Moreover it introduces a survey on various control techniques that are currently used highlighting pros an cons.
At the end some comments are given about software architecture trends which are used in this field.

2.1 Quadrotor platform

Flying objects have always exerted a great fascination on man encouraging all kinds of research and development. This intrigue for reaching the sky took all the form of imagination such as myths and legends. Many different and bizarre machines were theorized but very interesting were the precursors of the helicopter. Few visionaries, such as Leonardo da Vinci in the late 15th Century, foresaw an aerial application in the widely used windmill (14) coming up with the famous screw design as propeller. Even if it never worked, Leonardo's design was very innovative and this challenge has generated centuries of frustration and hundreds of dramatic attempts without being fade. Only in 1933 Juan de la Cierva gave us the autogiro. The first pratical rotocraft invented which led to what we usually call helicopter.

Few years after the first manned plane was invented, Dr. Cooper and Elmer Sperry designed the automatic gyroscopic stabilizer (2), which helps to keep an

aircraft flying straight and level. This was the kick of unmanned vehicle research which has its starting point back to WWI and WWII. Few years later, sophisticated machines are seen in battle fields such as the famous Predator by General Atomics, the Hammeread by Piaggio Aerospace or the Pioneer by Israel Aerospace Industries.

Figure 2.1: Predator Unmanned Aerial Vehicle

In the last ten years technology has taken a giant leap forward and the concept on Micro Aerial Vehicle (MAV) was born. Mavs are a class of UAVs with the feature of being relatively small and may be autonomous. Development is driven by commercial, research, government, and military purpose. The small craft allows remote observation of hazardous environments inaccessible to ground vehicles. MAVs have been built also for hobby purposes, such as aerial robotics contests and aerial photography.

The union of Micro Aerial technology and the helicopter concept brings the quad rotor platform. Quadcopter, also known as quadrotor, is a helicopter with four rotors. The rotors are directed upwards and they are placed in a symmetric formation with equal distance from the center of mass. The quadcopter is controlled by adjusting the angular velocities of the rotors which are spun by electric motors.

Designers came up with many projects and the market is now full of different vendors. The standard platform is the four propeller shape but the architecture can be eventually expanded, with advantages and drawback, up to eight propellers. Those are eventually called octo-copters and they are made to transport heavy load but with very high power consumption. In order to choose the appropriate platform one must examine different models and evaluate which is

Image	Description	Designer
	**Falcon 8**, used by professionals for mapping and inspections.	Produced by Ascending Technologies (9)
	**Phantom 3**, ready to use high quality commercial platform.	Produced by DJI (10).
	**AR.Drone**, ready to fly and cheap.	Produced by Parrot (5).
	**Flamewheel**, this kit must be assembled by the user. Good quality.	Produced by DJI (10) and assembled by Lorenz Meier (12) from PixHawk team.
	**IRIS**, almost ready to use. High quality and robust but relatively cheap.	Produced by 3D Robotics (3).

Table 2.1: Some copters designs

the best based on the needs. We can divide copters available on the market in professionals, semi-professional, and not-professional.

Professional copters are often used by qualified people in applications such as film making, inspection, mapping and surveillance. An example is the **Falcon 8** by Asctec, it is very reliable and has fantastic performances. The price however is very high, it is not suitable to fly indoor and the autopilot is not open source. Some Asctec copters are designed for research but the price is too high for our purposes. Semi-professional copters are somewhat a step below the professional segment. Those copter are the most used among amateurs and they are very reliable. The **Phantom 3** is a well known platform, the quality is superb and it comes pre-assembled. Unfortunately the autopilot is not open source and an

additional structure for IR markers must be added since the free surface is pretty tight. Another solution is to build from scratch the platform from a **Flamewheel** frame. In this case every piece is freely chosen, such as motors and autopilot, but the assembling process takes time and may lead to different issues. A very cheap solution is to use an **AR.Drone** by Parrot. It is absolutely ready to fly and the autopilot is not open source, even if it supports ROS integration. Thus we do not have the total control on the robot.

A quadrotor architecture is chosen for this thesis since they are more reliable, suitable for indoor application and more compact in dimensions. IRIS from 3Dr (3D Robotics) is the experimental platform used in this project, it is almost ready to fly and very solid. The autopilot is open source and the price is relatively low. Table 2.1 lists the platform illustrated in this section and a more detailed description of IRIS is given in 3.1.

2.2 Motion capture systems

Motion capture system, or mocap for short, is equipment specialized in the act of recording motion (28) usually through external sensors like cameras. Application of this technology are various and extended in different fields of science and not.

In film making and media industry, mocap systems are used to track the motion of actors in order too translate it in a virtual environment for editing. This simplifies the animation process and enable animators to capture the natural flow of human motion.

Biomechanics is a very promising field and mocap systems plays an important role; researchers and clinicians use motion data to study and observe human performance so they can improve treatment during rehabilitation as well as improve performance in sport applications.

There are many industrial applications areas. From complex 3D vibration applications, vessel tracking above and under water, aerodynamics tests, automotive development, interior design and control design . The possibility to use motion capture underwater presents completely new possibilities to researchers in many application areas such as ship design, fishing industry and naval research.

Mocaps are widely used in control systems where some feedback is necessary in order to generate some controlling signal. Robot performance analysis in particular is a field that widely uses mocap solutions.

This technology is based on cameras able to detect Infra Red (IR) light. Every camera has a LED ring on it which fires IR beams, those beams are hence reflected by passive markers placed in advance on the object we want to observe. The reflected beams are finally detected by cameras and the software is able to reconstruct the position of each marker and the pose of the object.

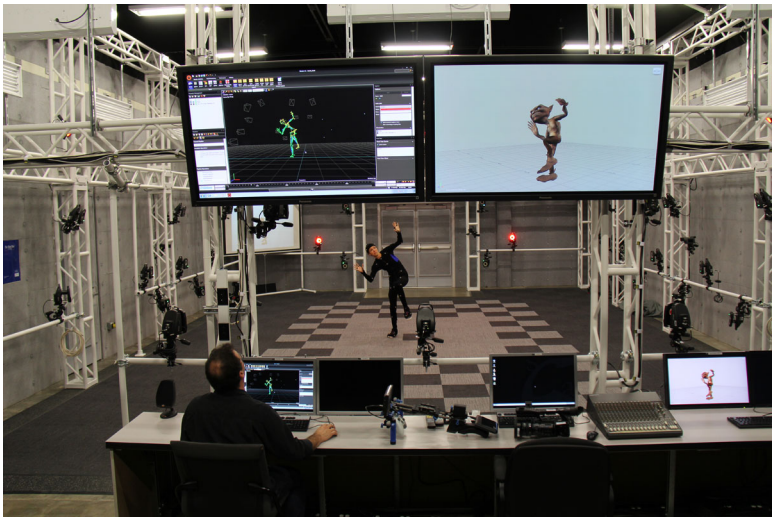


Figure 2.2: An actor in a motion capture stage, the human movement is copied in a 3d model

There are different vendors for this kind of equipment but the most famous are OptiTrack (19) and Vicon (33). They both offer high-end technology and quality solutions, the choice is just matter of taste and which proprietary software is more suited.

In the setup used in this project, the motion capture system consists in an OptiTrack Flex 13 with 8 cameras.

2.3 Current autopilots

The quadrotor architecture is a pretty old idea, it was not taken into account for long because people realized that in order to control four motors a decent amount of computing load and battery capacity are needed. However, the recent increase in electronic performances is why quadrotors are so famous right now.

The actual trend is to pack every component in a small amount of space, this idea led producers to design **autopilot boards** with related software.

The most famous board is ArduPilot Mega (8), a professional quality IMU autopilot that is based on the Arduino Mega platform. This autopilot can control fixed-wing aircraft, multi-rotor helicopters, as well as traditional helicopters. It is a full autopilot capable for autonomous stabilisation, way-point based navigation and two way telemetry with Xbee wireless modules. It is very simple to setup with their own utility hence no programming is required. The ArduPilot Firmware (6) is available on github and it is open source under GPLv3 licence. The 'Ardu' part in ArduPilot comes from the fact that originally this software was designed under

Arduino environment. Eventually the project outgrown the Arduino platform and they become independent from Arduino run time libraries even if they are still supported. The community is huge, the documentation is very good and recently ArduPilot was used also in important rescue missions becoming the high end solution in commercial and DIY products.

Till now APM seems to be the perfect choice from users stand point, but what about developers? Regarding this aspect some comments are necessary. When a new developer wants to contribute to a project the first thing he does is to check documentation and then experiment a bit with the code. From APM side, the documentation and support are superb (6). The big drawback is the technical organization of the code. Since it started from Arduino platform but it evolved quickly, many changes where made. Some files kept the original Arduino extension '*.pde*', some kept the famous *.ino* while some other became part of a c library. There is a bit of confusion that could slow down the project, plus the code style is not very modern even if reliable.

On the other hand, IRIS features a PixHawk board (see 3.1.1.1) which by default supports APM but PX4 autopilot can be also installed. PX4(23) is described in 3.1.1.2 and compared to APM has less features and less 'years of experience'. The community is still very active and the project it is growing very fast (25). It has every essential feature for an autopilot, just like APM it can be used on planes, helicopters, multicopters and rovers but the code is very organized an clear. The documentation is enough for a new developer , this why IRIS is running PX4 pilot in this project.

2.4 Modeling and control

The derivation of a dynamical model is the starting point for control design; quadrotor modeling was studied and explored extensively in the latest years with different complexity.

Newton-Euler equations are widely used to handle quadrotor modeling as they lead to fairly simple results like in (59) and in (62). A very common practice is to consider the robot as a rigid body, even if it it not true, and treat the aerodynamic propulsion separately. Since inputs of the system are the rates of each propeller, one can define some more realistic inputs calculating the relation of motors speed with force and torques generated (56). Differential inputs are hence used. In (57) this principle is explained very well, the robot is described with rigid body equations where three torques and one force is applied. Propellers thrusts are calculated as proportional to the square of each rotor angular speed ((39) and (57)).

One may also consider some aerodynamic and additional non linear effects

including them in the model and hence increasing accuracy. The simplest improvement could is to add friction with air as form of linear damping, called also drag. A more complex feature to model is called blade flapping (57). In translational flight, the advancing blade of a rotor sees a higher effective velocity relative to the air, while the retreating blade sees a lower effective velocity. This results in a difference in lift between the two rotors, causing the rotor blades to flap up and down once per revolution (51) since they are pretty flexible. The last effect is that the total thrust varies not only with the power input, but also with the relative wind velocity.

Those effects are significant in aggressive maneuvering which include sudden change in direction and high speed hence they will not be taken into account.

Controlling such system is a great challenge which arises many issues and proposal by scientific community, hence different controls have been applied to it. The quadrotor does not have complex mechanical control linkages due to the fact that it relies on variation in motor speed for vehicle control. However, these advantages come at a price. Controlling a quadrotor is not easy because of the coupled dynamics and its commonly under-actuated design configuration (35).

The most utilized controllers are:

- PID control
- Linear Quadratic Regulator
- Feedback Linearization
- Sliding Mode Control
- Backstepping Control
- Adaptive Control Algorithms

The **PID controller** has been applied to a broad range of applications. It is indeed the most use controller in industry. It is easily tuned even with empirical methods, it has good robustness and it is very easy to implement. It follows that PID it is applied also to quadrotors with fairly good results especially in velocity control, attitude control and hovering (57), (46). However PID has some drawbacks due to the non linear nature of the system. Thus a linearization around hovering point is necessary (41) limiting quatrotor performances. Nevertheless PID is implemented most of commercial quadrotors available on the market. Hobbyists and developers mostly use PID for their platform because it is very simple to tune and it adapts to many structures and geometries. PX4 developers

decided to adopt this kind of control in a cascaded fashion to improve adaptability respect to different quadrotor models.
The **LQR optimal control** algorithm operates a dynamic system by minimizing a suitable cost function. Boubdallah and co-researchers applied the LQR algorithm to a quadrotor and compared its performance to that of the PID controller (40). LQR is in fact outperforming PID because it is designed on the full non linear system but still it does an average job.

In the field of non linear control, the first technique is **Feedback Linearization**. It relies on putting a non linear system in a special form where it can be treated as linear. It was implemented on a quadrotor platform (34) with good results. Nevertheless, feedback linearization presents an unforgivable drawback. The exact model must be know and this is never the case, in particular among DIY projects.

Sliding mode control is another approach to non linear control. Sliding mode works by applying a discontinuous control signal to the system to command it to slide along a prescribed path thus achieving stability. The main advantages are the good tracking and fact that there is no need to approximate the model dynamics. The discontinuous control signal may cause chattering and power loss. Sliding mode control is often used beside **backstepping** (42) which is a recursive algorithm that breaks down the controller into steps and progressively stabilizes each subsystem. Its advantage is that the algorithm converges fast leading to less computational resources and it can handle disturbances well. The main limitation with the algorithm is its robustness is not good.

More advanced tools are often used such as **Adaptive algorithms**. Adaptive control algorithms are aimed at adapting to parameter changes in the system. The parameters are either uncertain or varying with time and in many cases they were able to control systems where linearization failed. They were proven to be efficient in quadrotor control (36) and the adaptive scheme could lead to many interesting results especially in presence of wind and during pick and place operations of small loads.

To conclude this section I would like to point out one important aspect of PID control and why it is the selected algorithm for this project. There is a main difference between the presented algorithms: some are model dependent and some others are not. Feedback linearization for example is a very high model-dependent algorithm, same for dynamic canceling. It means that the perfect dynamical model must be known, the parameters well estimated and the dynamics well modeled. As experience teach, this is almost never the case. The PID is not model dependent, the choice of the gains makes it perfectly customizable for many structures and different type of rotor crafts. This is the key aspect on why it is mainly used, especially among amateurs.

2.5 Software architectures and control stations

A ground control station (GCS) usually is a land or sea-based control center that provides the facilities for human control of unmanned vehicles in the air or in space. A GCS could be used to control unmanned aerial vehicles or rockets within or above the atmosphere. It contains every feature needed to interact with the monitored system such as diagnostics, receive data and send commands, display vehicle state and perform emergency procedures. GCSs can be mobile or not and have different operational ranges.

For aerial robotics usually the control station consists in a software running on a laptop or tablet and a communication module (radio, wi-fi , bluetooth). Let us have a brief look on which available solutions the market proposes and which is the choice for this project.

Many companies offer proprietary solutions such as **Airware Control Station** (4) but open source solutions are more reachable and expandable. ArduPilot supports **APM Planner** (7) which is the offspring of the well established and mostly used Mission Planner created by Michael Oborne. APM Planner is an open-source ground station application for MavLink(see 3.1) based autopilots including APM and PX4/Pixhawk that can be run on Windows, Mac OSX, and Linux. The most important features are:

- Point-and-click waypoint entry, using Google Maps
- Download mission log files and analyze them
- Configure APM settings for user airframe
- Interface with a PC flight simulator to create a full hardware-in-the-loop UAV simulator
- Calibrate on board sensors
- See vehicle status, parameters and sensors output

It is written using Qt libraries since they have good GUI support and is the first choice among hobbyists and professionals. It can be easily configured to support different vehicles like boats, rovers, helicopters, multicopters and planes.

APM Planner features are shared also by another popular GCS, especially among PX4 users, named **QGroundControl** (26). Additionally to APM PLanner, QGroundControl supports multiple vehicle instances at the same time which is essential in swarm robotics. Moreover it has real time sensor plotting, in flight way points manipulation and digital video transmission from on board cameras. QGroundControl is used in this thesis to calibrate IRIS sensors and to tune some parameters.

Those GCSs are cleary designed and optimized for outdoor missions, they have GPS support and displayed maps from Google Earth. Position feedback from another source such as a mocap system is not supported. What is lacking is a control station suitable for indoor application and experiments. During this work I designed a software architecture that takes the place of QGroundControl and it is aimed to be used with Optitrack cameras . The work of this thesis focuses more on the robot management than user experience and interface. It is written in Qt libraries because there could be a future integration with QGroundControl and because they offer many tools with a very good documentation. It sacrifices features such as Point-and-click waypoint entry and sensor calibration and it centralizes the robot performance and in particular robot autonomy. It is responsible of passing the robot position feedback from the mocap. The concept of task is presented as an action that IRIS can perform, the software scheduler takes a list of actions created by the user and let the robot execute them autonomously. This architecture is designed to be expanded as much as possible adding new tasks and modes.

The main goal is to go towards a behavioral architecture for quadrotors, inspired by an old work by Montgomery James (60), and when the actions are sufficient integrate a planner to replace the user during the task list creation. Autonomy is the key aspect for this software, the idea is to abandon the figure of the mission operator and switch to a more autonomous system where the human role is just to set a goal.

2.6 Landing on a moving platform

Since many years people studied a way to land aircrafts safely, important studies has been made especially in the field of aerospace exploiting and designing different techniques to land spacecrafts on other planets. Moreover it happens that the target where the robot wants to land is not still, but performing some kind of motion on the ground. Landing a jet on an air carrier is a perfect example of a moving landing site, the platform is moving and possibly oscillating with waves.

With the arrival of MAVs (Micro Aerial Vehicles) new challenges arise due to their dexterity and small sizes. Usually battery life is very limited and vehicles need to reach automatically recharge stations. A quadrotor may land on a moving car after its mission, on a robot in some cooperation task or on a floating boat to be recovered. All these procedures need a very high precision trajectory tracking and perception of the environment.

Small flying animals are capable of safe and accurate landings while relying only on proprioceptive sensors and visual information. Since this capability holds a

promise of landing safely with limited sensors and processing, it has served as inspiration for recent spacecraft landing studies(53). A very interesting research (37) demonstrated how bees, with very simple measurements, are able to land on many surfaces. Dedicated eyes calculate the *optical flow* of images meaning that they are able to track features and estimate the velocity of each feature in image plane. The results of this measurement is a vector field describing how a particular object (feature) in the image plane moves in time or from one frame to another in case of cameras. When one moves towards a surface with some pattern on, it is easy to experience an expansion towards the external edges of the image of the features (e.g. corners of a pattern). It is easy as well to feel this expansion faster as one is closer to the wall. Keeping this rate of expansion constant, bees are able to land easily. As closer they get to the surface, they measure a higher rate and in consequence they slow down to keep it constant. With this technique bees can safely land assuring that the approaching speed at touchdown is close to zero.

Another useful quantity that can be obtained from the optical flow is the Time To Contact, or TTC, defined also the ratio of the height and the vertical velocity. From this simple relation some descending control laws are derived (43).

Those methods are developed assuming to have an optical flow sensor or a camera on board and pointing downwards. Unfortunately the IRIS does not have those features and we can rely only on the mocap and the IMU feedback. In this project the position of the platform will be estimated by the motion capture after putting markers on it, this may mimic a localized vehicle sending its position to the quadrotor ensuring the landing maneuver.

The problem is separated in two different aspects, tracking and landing. In order to land tracking must be ensured and the research community proposed different methods. Some used non linear tracking (63) and others, assuming that the velocity of the platform is estimated, compensate the delay of the robot respect to the target with a velocity feed-forward (48). A PI controller is synthesized in (50) closing a velocity control loop while a non linear control law assures the descending task.

Velocity control is not yet implemented in the software architecture and the only reference that is sent is a position set point. Hence I decided to rely only on position control, which may limit the performance of the maneuver but it is simple to implement and it is shown to be a valid strategy.
The idea I followed is to remain as simple as possible avoiding non linear controllers. The experiment with bees, even if very interesting, do not take into account the tracking of a platform which must be integrated. Moreover it shows that bees use a linear law for descending: the vertical velocity decreases linearly with the height from the target. My algorithm is pretty simple but in some sense

similar. The height of the IRIS decreases linearly with the horizontal position error respect to the center of the platform. This assures that the tracking has an higher priority respect to the landing, which will never occur if the robot is too far away. At the end tracking is made by a PI controller, closed respect the horizontal position error, with an offset. Since we hypothesized that velocity both for the robot and the platform is not available, feedback on position is the best way to solve the task. There are some drawbacks on using a PI controller such as the overshoot when the platform changes direction. Assuming that the motion of the landing site is smooth, the controller is able to secure the landing.

Chapter 3

System description

Summary

This chapter introduces to the reader the current laboratory setup. The first section illustrates which kind of tools are used as well the specifications of each component. The second section instead will present the integration both hardware and software of each part of the system.

3.1 The setup

This section introduces and explains the tools, both hardware and software, that are used in the experiments.

We can identify five distinct parts:

- IRIS quadcopter from 3d robotics
- Motion Capture system from optitrack
- Linux machine
- Windows machine
- Flight arena

While IRIS, the mocap system and the flight arena deserve three separate sections, the two machines are described together since they play a central role in the integration part (see 3.2).

3.1.1 IRIS Quadcopter

The IRIS quadcopter is flying robot manufactured by 3D Robotics and commercially available as a kit or ready-to-fly. IRIS flight control system employs by default the open source software ArduPilot running on a Pixhawk autopilot board (based on the PX4 hardware developed by ETH).

Figure 3.1: The IRIS quad copter

The main distinct components of the IRIS quadcopter are the airframe (central body, arms, motors and legs), four 850 kV DC brushless motors controlled in open-loop through PWM, one 11.1 V LiPo battery and control electronics.
The use is mainly commercial, it is optimized for aerial shooting and provides a lot of features the user can play with. IRIS can be easily setup to follow any GPS-enabled Android device with OTG compatibility, this technology controls also the gimbal to keep the camera centered on the target capturing videos and actually becoming a free hand camera. Using the free DroidPlanner app, the user can plan flights by simply drawing a flight plan on any Android tablet or phone. This allows for hands free flight control with virtually unlimited waypoints even keeping IRIS pointing to the same location via a Region of Interest (ROI) waypoint throughout the entire flight (15).

This device is designed to fly outdoor, nevertheless it is possible with some software tuning to use the IRIS in indoor scenarios. The component which make

Figure 3.2: IRIS and control devices.

this possible is the actual brain of the robot: the PixHawk board with PX4 software.

3.1.1.1 PixHawk board

PIXHAWK (figure 3.3) is a high-performance autopilot-on-module suitable for fixed wing, multi rotors, helicopters, cars, boats and any other robotic platform that can move. It is targeted towards high-end research, amateur and industry needs and includes all hardware required for remote control (20), stabilization and navigation functions namely:

- an embedded processor (32bit STM32F427 Cortex M4 core with FPU)
- a fail-safe processor (32 bit STM32F103 failsafe co-processor)
- a set of motion sensors, including an Inertial Measurement Unit sensing accelerations and angular speeds of the aircraft, a barometer to compute relative altitude and vertical speed from changes in static air pressure, a magnetometer for compass and a GPS receiver.
- ESCs (Electronic Speed Controllers) that handle low-level motors control loops to maintain required thrust through open-loop PWM control.
- RC Radio to receive direct pilot commands from an RC hand-held controller (2.4 GHz radio link).
- telemetry module, a digital radio transceiver (433 MHz) receiving commands from the ground control station and sending telemetry data back to the ground via the *MAVLink* protocol.

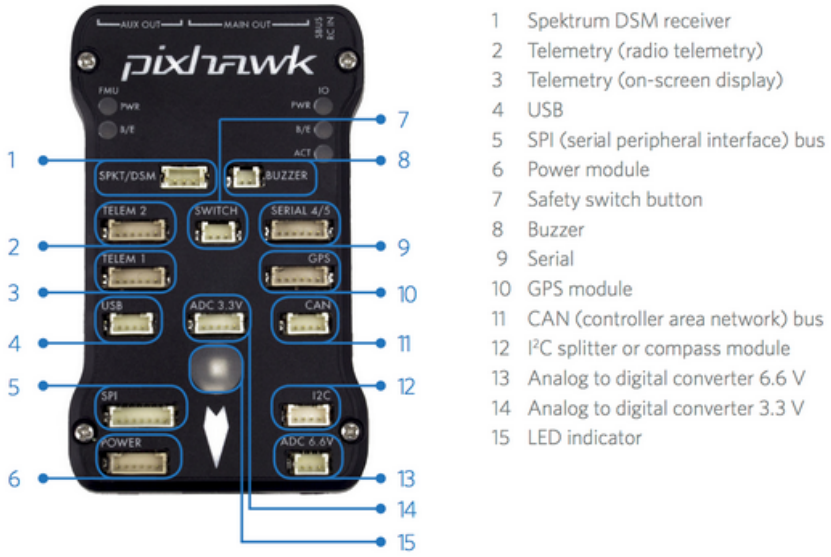


Figure 3.3: Pixhawk board and its connections.

It supports redundant technology both for power and processor, it is equipped with various interfaces such as CAN, SPI, I2C, UART and supports a micro USB for on board real time logging.

3.1.1.2 PX4 Autopilot

PX4 (23) is the software stack running on the autopilot board, it was a master thesis project at ETH pursued by Lorenz Meier and it became a big open source project on github (25) with contributors from all over the world. It works on NuttX (17) which is an open source real time OS and it uses uOrb (micro orb) as middleware. Different modules runs in parallel and they can be grouped in different sets:

- PX4 Flight Stack (estimation and control, cross-platform), the core modules dealing in this thesis
- PX4 Middleware (ORB, NuttX)
- PX4 ESC Firmware (for motor controllers)
- PX4 Bootloader (for STM32 boards)
- Operating System (NuttX or Linux/Mac OS)

3.1.1.3 MAVLink Protocol

MAVLink is a very lightweight, header-only message for micro air vehicles created and released under LGPL licence by Lorenz Meier in 2009. It can pack, with very little overhead, C-structs over serial channels with high effiency and send these packets to the ground control station (Fig. 3.4). It is extensively tested on the PX4, PIXHAWK, APM and Parrot AR.Drone platforms. It has the possibility to be used with at maximum 255 vehicles on the same control station, it runs on multiple microcontrollers and OS such as ARM7, ATMega, dsPic, STM32 or Windows, Linux, MacOS and iOS with only 8 byte of overhead. The intense testing of this protocol made it the most used communication protocol in aerial robotics due to his effectiveness. Mavlink is used to send message over radio link in this project.

Figure 3.4: Ground Control Station used to visualize robot parameters, plan missions or calibrate sensors.

Supported data types MavLink supports fixed-size integer data types, IEEE 754 single precision floating point numbers, arrays of these data types and the special MavLink version field, which is added automatically by the protocol. Table 3.1 contains a list of every data type that can be transported by MavLink packets (16).

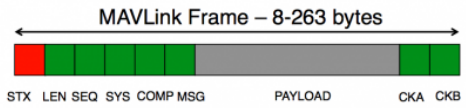
This protocol was designed towards two properties: transmission speed and safety. It allows to check the message content, it also allows to detect lost messages but still only needs six bytes overhead for each packet as illustrated in figure 3.5. Regarding the packets size we have that:

Mavlink supported Data types

uint8_t	Unsigned 8 bit
int8_t	Signed 8 bit
uint16_t	Unsigned 16 bit
int16_t	Signed 16 bit
uint32_t	Unsigned 32 bit
int32_t	Signed 32 bit
uint64_t	Unsigned 64 bit
uint64_t	Signed 64 bit
char	Characters or strings
float	IEEE 754 single precision floating point number
double	IEEE 754 double precision floating point number
uint8_t_mavlink_version]	Unsigned 8 bit field automatically filled on sending with the current MAVLink version - it cannot be written, just read from the packet like a normal uint8_t field

Table 3.1: Mavlink supported data types.

- The minimum packet length is 8 bytes for acknowledge packets without payload
- The maximum packet length is 263 bytes for full payload



Byte Index	Content	Value	Explanation
0	Packet start sign	v1.0: 0xFE (v0.9: 0x55)	Indicates the start of a new packet.
1	Payload length	0 - 255	Indicates length of the following payload.
2	Packet sequence	0 - 255	Each component counts up his send sequence. Allows to detect packet loss
3	System ID	1 - 255	ID of the SENDING system. Allows to differentiate different MAVs on the same network.
4	Component ID	0 - 255	ID of the SENDING component. Allows to differentiate different components of the same system, e.g. the IMU and the autopilot.
5	Message ID	0 - 255	ID of the message - the id defines what the payload "means" and how it should be correctly decoded.
6 to (n+6)	Data	(0 - 255) bytes	Data of the message, depends on the message id.
(n+7) to (n+8)	Checksum (low byte, high byte)	ITU X.25/SAE AS-4 hash, **excluding packet start sign, so bytes 1..(n+6)** Note: The checksum also includes MAVLINK_CRC_EXTRA (Number computed from message fields. Protects the packet from decoding a different version of the same packet but with different variables).	

Figure 3.5: MAVLink package anatomy.

3.1.2 Motion capture system

The localization of the robot in the 3D space relies on the OptiTrack FLEX 13 motion capture syestem, composed by 8 infrared cameras (Fig. 3.6) mounted on the roof of the laboratory in a squared fashion around the flight perimeter. It is the mid level product in OptitTrack catalog (1000 dollars per camera (19)) capable of tracking multiple objects in a medium volume space of about 4 X 4 X 4 meters.

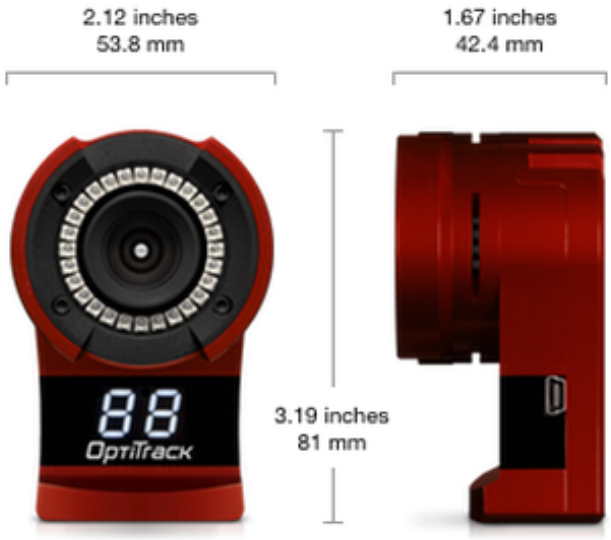


Figure 3.6: Flex 13 camera and dimensions.

3.1.2.1 Image sensor specifications and performances

- Latency: 8.3 ms.
- Frame Rate: 30-120 FPS (adjustable).
- Imager Resolution: 1280 X 1024 (1.3 Megapixels).

This camera features a status LED and a 2 digits numerical screen for diagnostics, mounting supports, an image sensor and a lens. A 28 LED ring around the lens ensures the required IR (Infra Red) illumination both in stroboscopic and continuous modes.

Object are defined by attaching to them at least 3 **passive infrared markers** (Figure 3.7) able to reflect infrared light, solving rigid bodies and ensure tracking. The number of markers that the OptiTrack is capable of tracking depends on the

size of the markers and the distance they are from the camera. This number may change from model to model but is about 100 in our case, more than enough for this project.

The 3D location of markers can be resolved with millimeter accuracy and resolution depending on capture volume size and camera configuration. Increasing the number of cameras can help improve the tracking performance if needed and in this case 0.3 mm of error for each marker is achieved.

Figure 3.7: Passive IR marker able to reflect IR radiation

3.1.2.2 Software, cameras layout and requirements

The system uses the proprietary software Motive (19), an Optical motion capture software which incorporates many features such as rigid body solving and marker tracking. Motive is able to manage every camera parameter such as exposure, brightness and gain as well as the overall calibration. Two USB hubs divide the eight cameras in two groups collecting four USB cables each coming from each sensor. Both hubs are connected through USB cables to the Windows machine where Motive is installed, moreover a sync cable responsible for the synchronization runs from one hub to the other. Figure 3.8 explains the connections of a single hub.

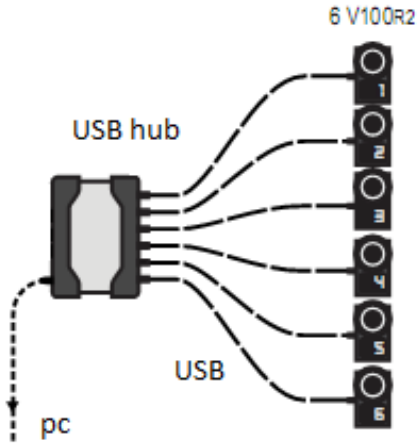


Figure 3.8: Example of a single hub attached to 6 cameras, sync cable between the two hubs is not shown.

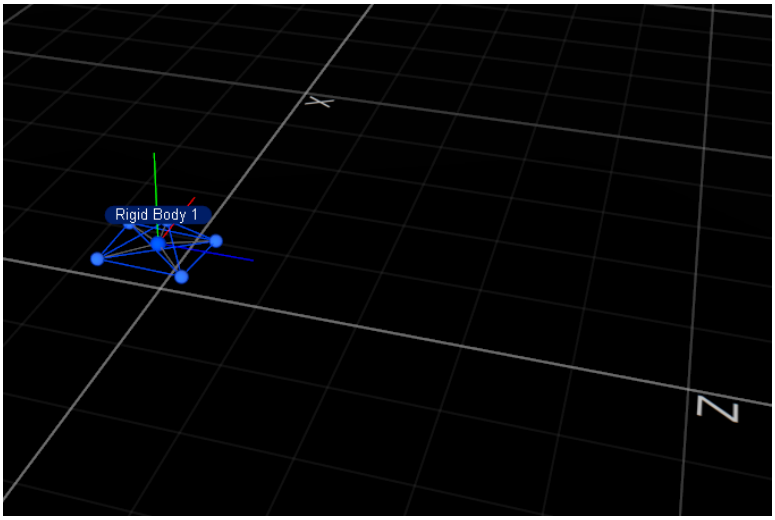


Figure 3.9: IRIS Rigid body created with Motive software.

The reader must note that in order to have good tracking performances, the setup needs to assure some properties during the experiments:

- At least 3 markers must be present on the object.
- The distance between each marker must be constant during the tracking period (Rigid body assumption).

- Cameras must see at least 3 markers of the tracked rigid body.
- Occlusions are not taken into account by Motive (19).
- Rigid bodies with symmetric shapes induce big uncertainties in attitude estimation.

I solved this problems mainly in two ways. Pointing each camera to the center of the flight space approximately at 0.5 meters from the ground gives the best coverage and few unseen areas at the margins.

Regarding the robot, with 5 markers placed in a 3D asymmetrical configuration on the body frame as in figure 3.10,is assured that at least 3 markers are always visible and there are no singularity regarding attitude.

Figure 3.10: Markers placed on IRIS in asymmetrical 3D positions.

3.1.3 Flight Arena

The flight arena is the stage of the setup. Since the laboratory where this thesis has taken place is new, I was involved in the relocation of the equipment from the old to the new lab. Next I attached every camera to the supports previously drilled on the roof and after I took care of the cables management. Since the maximum allowed length of the camera cables is 5 meters, I managed to place the hubs on the top at two opposite edges of the square (the ones intersecting x

axis). By that, the available length is sufficient to connect three cameras of one edge and one from the adjacent edge to on hub and the rest to the other. The sync and the two hubs cables are running on the roof and crossing the room while the camera cables are displaced on the perimeter. Finally I oriented each camera towards the center of the room in order to have the highest coverage possible and calibrated the system. The flying space measures approximately 3 x 3 meters and

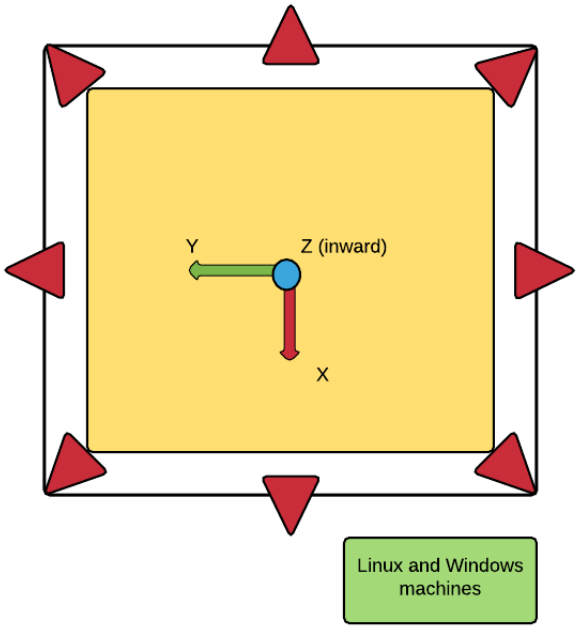


Figure 3.11: A scheme of the flight arena. In red there are the cameras, the yellow part is the flying space while the outer square is the protective net.

2 meters high defined by the capture volume of the mocap. Outside there is a security perimeter enclosing cameras made by a nylon net of about 4 x 4 meters till the roof. The origin of the position measures is at the center of the room placed on the floor with the z axis going downwards as shown in figure 3.11. A panoramic view of the arena, taken from the top left corner, is shown in figure 3.12.

3.2 Overall Integration

This section explains to the reader how every part of the setup is interfacing with each other, the flow of data packets from OptiTrack to IRIS and the operation done to the information at each step.

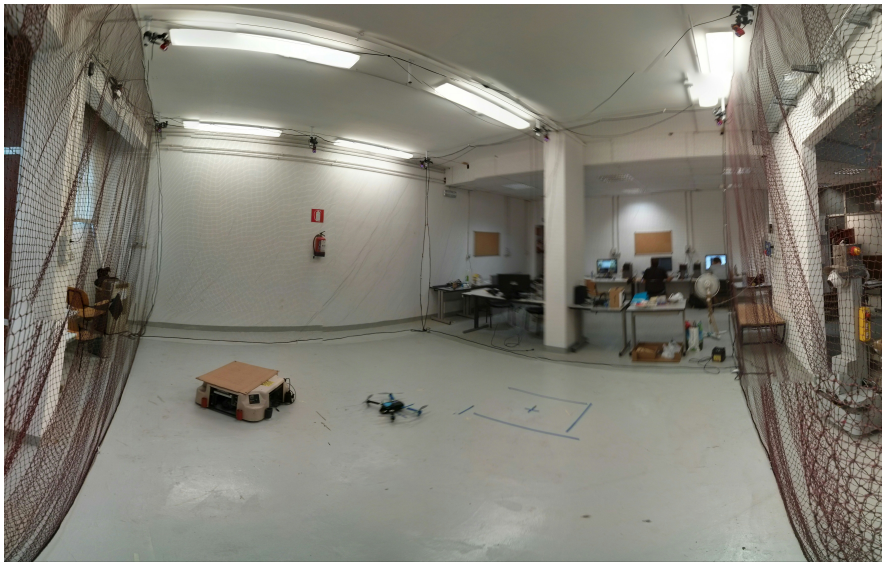


Figure 3.12: Panoramic view of the flying space.

3.2.1 Hardware interfaces

Each camera is equipped with a 5 meters USB cable and, as stated in 3.1.2, four cables are connected to one hub and four to the second hub. As outputs, the hubs are equipped with a 5 meter USB cable that can be expanded to 10 meters, those two connection are inputs for the Windows machine.

Windows machine specs

- OS: Windows 7
- Processor: Intel i7 3.60 GHz
- RAM: 32 Gb
- Video Card: NVIDIA GTX 970
- Software tools: Motive

This computer is directly connected to a Linux machine through Ethernet cable (RJ-45 connectors).

Linux machine specs

- OS: Ubuntu 14.04 LTS
- Processor: Intel i7 2.40 GHz

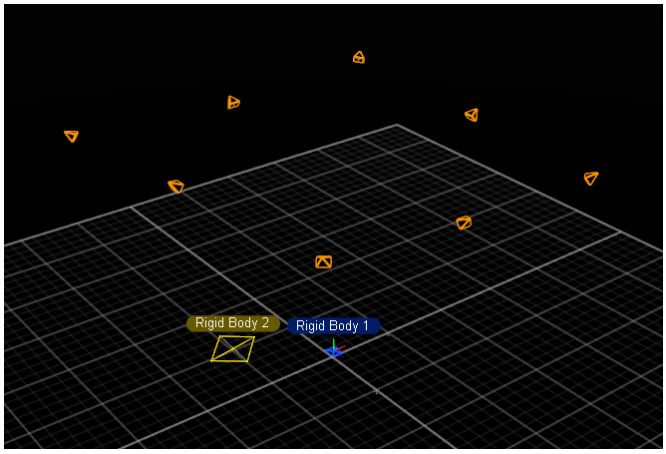


Figure 3.13: Flight arena seen by Motive with a couple of rigid bodies and cameras.

- RAM: 8Gb
- Video Card: NVIDIA GeForce GTX 760M
- Software tools: Qt C++ Libraries, Px4 build environment (24)

At the end, via USB , a telemetry module (Fig. 3.14) is attached to the linux machine. This radio link allows the control station to communicate with the robot

Figure 3.14: Telemetry module from 3d Robotics used to recieve and transmit MAVLink Packages with IRIS.

wirelessly with acceptable performance. It is mainly used for acquiring telemetry and robot status, transmit new mission goals, check robot parameters such as

controller gains and calibrate onboard sensors.

Its main features are (3):

- 433 MHz (for Europe).
- micro usb port: it can be used also from tablets.
- UART Interface.
- 2-way full-duplex communication.
- MAVLink protocol framing.

Moreover, IRIS features a standard RC transmitter, shown in figure 3.2, from which the user can operate the robot in three main modes:

1. Fully manual: radio sticks control directly velocity of the propellers.
2. Semi-auto: height is stabilized, left stick control height position set point while attitude is manual.
3. Fully-auto: horizontal position is stabilized through position feedback, right stick controls x and y set points.

3.2.2 Software interfaces

Motive has the feature to transmit the pose of every tracked rigid bodies through a multicast IP. By this option, the raw pose of the robot is transmitted to the Linux machine. The software architecture running on Linux reads data coming directly from Motive through a *Receiver component*. Without the use of an *adapter component*, since the program is specific for this application, data from Motive is processed as explained in section 3.2.2.1 at the moment it arrives. In the mean while a position set point is generated inside the architecture, both pose estimation and position set point are packed using MavLink protocol. At the end data is sent through a socket interface an the telemetry module takes care of transmitting everything to IRIS with a rate of approximately 10Hz.

Figure 3.15 sums up in a scheme the relation between each part.

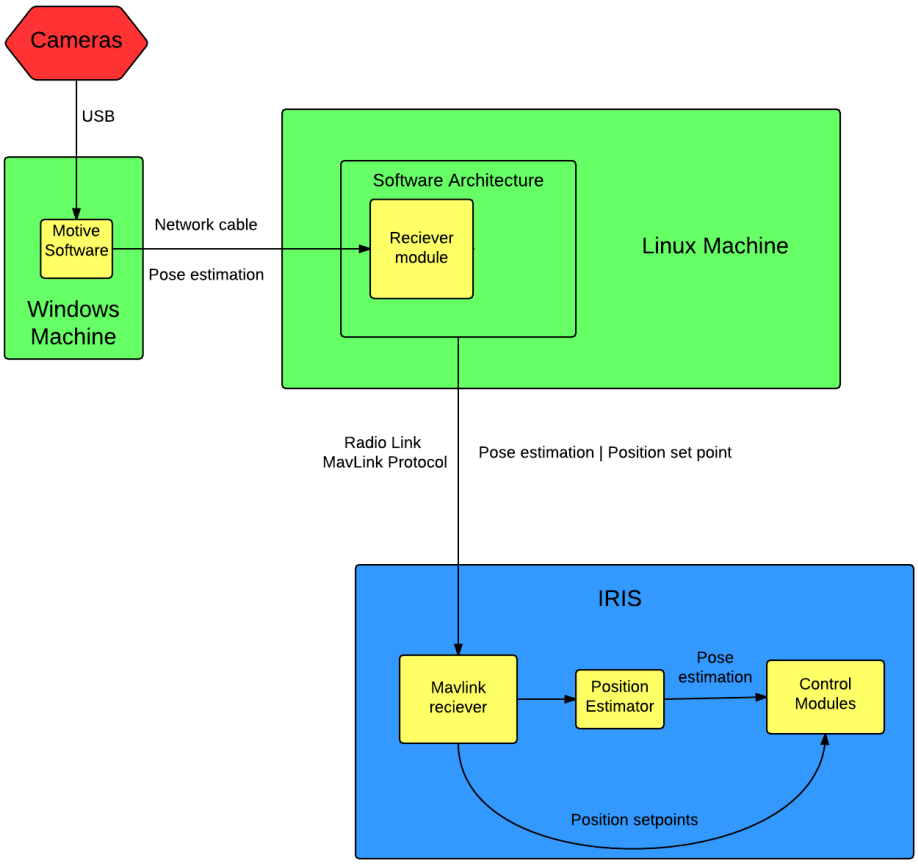


Figure 3.15: Scheme of the setup with connections between parts

3.2.2.1 Adapting reference frames

Motive transmits a 6 degree of freedom pose (position and orientation)respect to a fixed reference frame $\Re^m$ in the form:

$$Pose^m = \begin{bmatrix} P^m \\ Q^m \end{bmatrix} \tag{3.1}$$

where $P^m = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$ is the position vector and $Q^m = \begin{bmatrix} w \\ x \\ y \\ z \end{bmatrix}$ is the rotation quaternion, everything expressed in Motive reference frame $\Re^m$.

Now let us define a second reference frame, in this case is a North-East-Down

frame (29) mainly used in aeronautics and aerial robotics, namely $\Re^E$. The peculiarity of this reference is that:

- x axis is aligned with North.
- y axis is aligned with East.
- z axis goes down towards the earth.

The only constraint of $\Re^{\mathrm{m}}$ is that y is vertical and x parallel to the ground, hence x and y can be set freely at the moment of cameras calibration Figure 3.16 shows both reference.

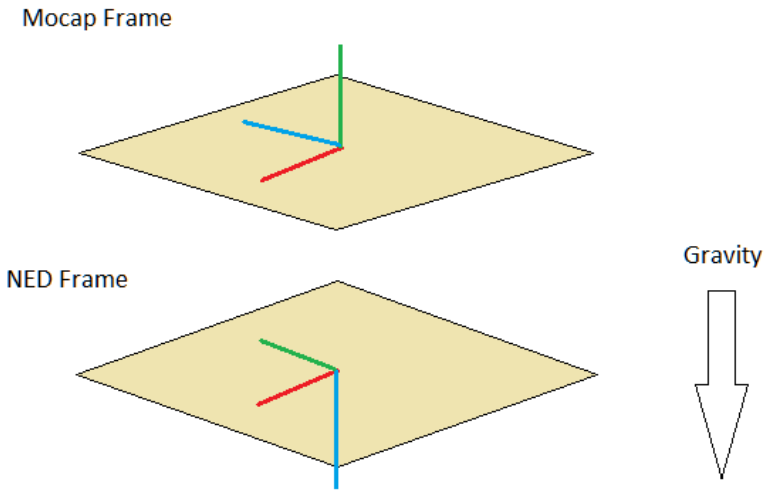


Figure 3.16: NED and Mocap frames depicted. X axis in red, y axis in green and z axis in blue

It is easily shown that the relation between frames is a 90 degree rotation of $\Re^{\mathrm{m}}$ over x. Let $R_x(\theta)$ be the rotation matrix on x, P^b arbitrary 3-elements vector column respect to a general base frame b, then the following is valid:

$$(P^{\mathrm{m}})^{\mathrm{T}} R_x(\theta) = P^{\mathrm{E}} \tag{3.2}$$

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) \\ 0 & \sin(\theta) & \cos(\theta) \end{bmatrix} \tag{3.3}$$

Putting $\theta = \pi/2$ inside 3.3,then 3.2 become:

$$\begin{bmatrix} x & y & z \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} x \\ z \\ -y \end{bmatrix} \tag{3.4}$$

and

$$P^{\mathrm{E}} = \begin{bmatrix} x \\ z \\ -y \end{bmatrix} \tag{3.5}$$

where $[x, y, z]$ are the coordinates of P^m in $\Re^m$.

As regards the rotational part, some comments must be done. Since the applied rotation applied does not influence the estimated attitude, we can safely state that:

$$Q^{\mathrm{E}} = \begin{bmatrix} w \\ x \\ z \\ -y \end{bmatrix} \tag{3.6}$$

where $[w, x, y, z]$ are the elements of Q^m.

Equations 3.5 and 3.6 describe the relation between $\Re^m$ and $\Re^E$ in a simple but effective way in fact just by changing the order of the elements of the pose coming out Motive, a pose that IRIS can understand is generated.

Chapter 4

Modeling IRIS

Summary

This part concerns the modeling of the IRIS quad rotor. A model of the quadrotor is necessary to develop a controller and understand better its dynamics. The first part presents some generalities including the reference frames used in order to define the world and the robot. It also contains a list of assumption done to simplify the model. After this are explained the physical principles involved and the actual derivation of the model.

4.1 Generalities

This sections explains general concepts necessary to proceed with modeling. It starts by defining reference frames, then it describes how maneuvering is done and at the end i presents general assumption under which the model is derived.

4.1.1 Reference frames

First element is a world base frame, in section 3.2.2.1 is defined $\Re^E$ which is a North East Down frame with its own features. In our case, the world frame is decided freely after calibration. It is the frame respect to which cameras give the pose estimate. Hence $\Re^E$ still keeps its name being world fixed frame but it has no constraint of being oriented North-East-Down. The only constraint for $\Re^E$ is that z **axis is directed downwards along gravity while** x, y **axis are parallel to ground.**

Next frame is the body frame, namely $\Re^B$. It is attached to the center of mass of the robot with the x axis pointing to the front and z axis downwards, y axis

is generated accordingly with the right hand rule (see fig 4.2+).

The relation between $\Re^B$ and $\Re^E$ is represented by a transformation matrix, namely ${}^E T_B$ which defines the transformation of $\Re^B$ respect to $\Re^E$ as base frame.

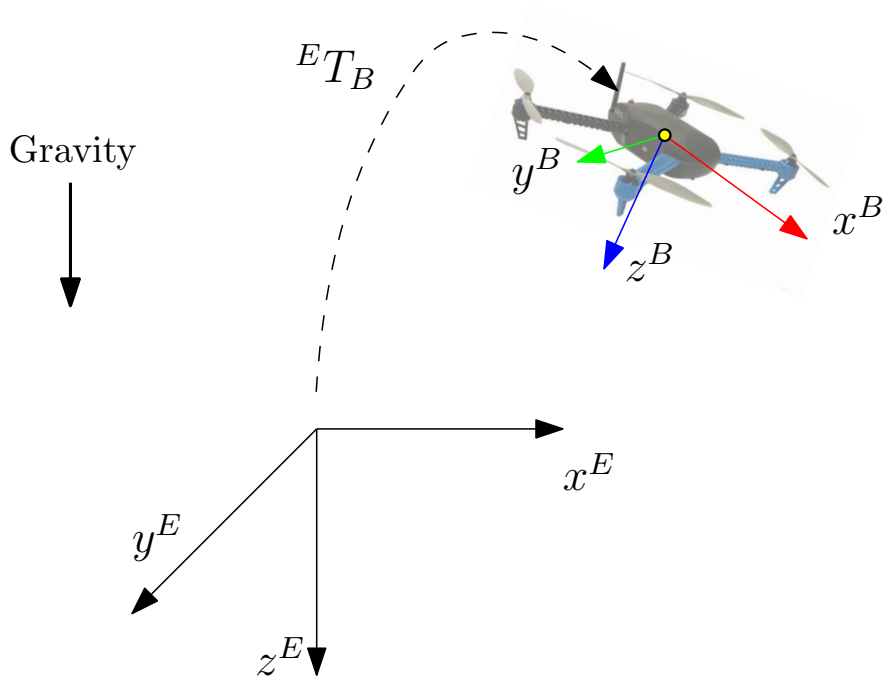


Figure 4.1: Earth-fixed reference frame, body-fixed reference frame and the transformation between them.

4.1.2 General assumptions

The analysis of this system is pretty complex. In order to simplify the derivation of the model some assumption are made and some non linearities are disregarded.

When a rotor translates horizontally through the air, the advancing blade has a higher absolute tip velocity and will generate more lift than the retreating blade. The mismatch in lift generates an overall moment on the rotor disk in the direction of the apparent wind causing the blade to flap as shown in figure 4.2 and the generated thrust is inclined respect to the rotor axis. This effect, called *Blade Flapping* is not included in the derived model.

A second disregarded non linear effect is the *Total Thrust Variation in Translational Flight* (52). The analysis of this phenomenon is quite complex and it is described by a qualitative explanation. When a rotor translates in air, it suffers

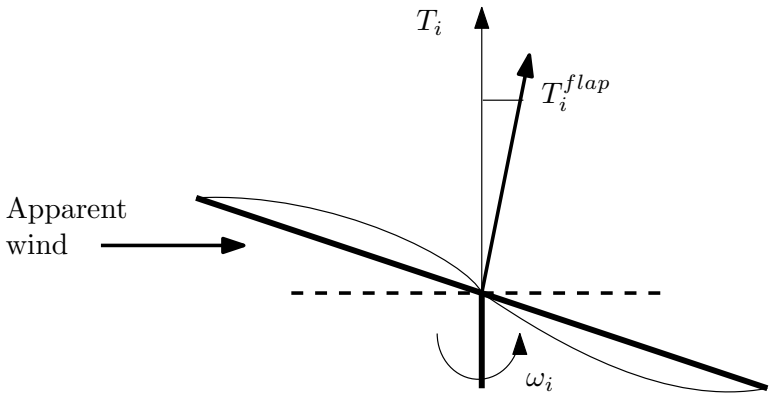


Figure 4.2: Blade flapping effect. T_i is the ideal vertical thrust while T_i^{flap} is the real thrust vector

form apparent wind and an increase of air flux through the blades. This leads to an increase of lift and thrust by the rotor.

Moreover, viscous friction experienced by the robot when it moves is negligible at small velocities. The robot can be also considered as a rigid body since there are no significant strains and forces applied which may deform the structure.

To summarize, the following assumption are made:

- Blade Flapping is not considered
- Thrust Variation in Translational Flight is disregarded
- Viscous friction (drag) is neglected
- The robot is considered as a rigid body
- Motor dynamics are managed internally by the board and not included in the model

4.2 Transformation matrices

This introduces some mathematical tools necessary to develop a full dynamical model for the quadrotor system. Moreover I will present here the conventions used for representing angles and rotations and the derivation of the matrices involved.

In section 4.1.1 I introduced ${}^{E}T_{B}$ as the transformation matrix of $\Re^{B}$ with respect to $\Re^{E}$. In particular ${}^{E}T_{B}$ is a 4x4 matrix and has a fixed structure (55):

$$ {}^{E}T_{B} = \begin{bmatrix} {}^{E}R_{B} & {}^{E}P_{B} \\ O_{3}^{T} & 1 \end{bmatrix} \tag{4.1} $$

in this definition we can see three different terms. ${}^{E}R_{B}$ is the rotation matrix of frame $\Re^{B}$ with respect to $\Re^{B}$; ${}^{E}P_{B} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$ is the position of the center of mass of the robot respect to the Earth frame and $O_3 = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$.

Rotation matrix and angles

Every rotation in 3D space can be defined by 3 successive rotations about 3 principal axis; we can define separately each rotation matrix about each axis and the multiply them.

Each Right-Hand rotation about a particular axis is defined by a positive angle (38) from earth to body frame and the following definitions are true:

$$ R_{x}(\phi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & \sin(\theta) \\ 0 & -\sin(\theta) & \cos(\theta) \end{bmatrix} \tag{4.2} $$

$$ R_{y}(\theta) = \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{bmatrix} \tag{4.3} $$

$$ R_{z}(\psi) = \begin{bmatrix} \cos(\psi) & -\sin(\psi) & 0 \\ \sin(\psi) & \cos(\psi) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{4.4} $$

The general rotation matrix can be obtained by multiplying elemantary rotations, but please note that the order is important.

The most common sequence associated with the name *Euler angles* is (z, x, z) or ${}^{E}R_{B} = R_{z}(\phi)R_{x}(\theta)R_{z}(\psi)$.

There is however a more suitable sequence with fits our case. The angles associated with the sequence (x, y, z) are sometimes called *Cardan angles* or *Tait-Bryan*

angles. Commonly used in aerospace where ϕ, θ, and ψ are known respectively as **roll**, **pitch**, and **yaw**. These angles describe a vehicle whose forward direction is along the positive body-fixed x-axis and the body-fixed z axis downward, like in the case of the quadrotor (see 4.1.1). In such configuration, the home position $\begin{bmatrix}\phi\\ \theta\\ \psi\end{bmatrix} = \begin{bmatrix}0\\ 0\\ 0\end{bmatrix}$, is flat and level pointing forward along the world x axis. The non intuitive downward pointing z axis is chosen in order to make a positive change in θ correspond to pitching upward (45).

That said, the multiplication order used is (x, y, z). Developing the calculations, the general rotation matrix becomes:

$$ {}^{E}R_{B} = R_x(\phi)R_y(\theta)R_z(\psi) = \begin{bmatrix} c_\theta c_\psi & s_\phi s_\theta c_\psi - c_\phi s_\psi & c_\phi s_\theta c_\psi + s_\phi c_\psi \\ c_\theta s_\psi & s_\phi s_\theta s_\psi + c_\phi c_\psi & c_\phi s_\theta s_\psi - s_\phi c_\psi \\ -s_\theta & c_\theta s_\phi & c_\theta c_\phi \end{bmatrix} \tag{4.5} $$

where $c = cos$ and $s = sin$ for simplicity, Moreover, the following limits are true by definition:

- $-\pi < \phi < \pi$
- $-\pi/2 < \theta < \pi/2$
- $-\pi < \psi < \pi$

while the following shows the transformation from the derivative of the Euler angles to the angular velocity in the body frame(48):

$$ W = \begin{bmatrix} 1 & 0 & -s_\theta \\ 0 & c_\phi & c_\theta s_\phi \\ 0 & -s_\phi & c_\theta c_\phi \end{bmatrix} \tag{4.6} $$

Please also note that ${}^{E}R_{B}$ is an orthonormal 3x3 matrix so its inverse it is equal to its transpose (55), hence $({}^{E}R_{B})^{-1} = ({}^{E}R_{B})^{T} = {}^{B}R_{E}$.

4.3 Propulsion and controls

The four motors installed are responsible of the propulsion of the quadcopter. Each rotor-motor couple rotates with an angular velocity ω_i and generates: an upward lifting force f_i parallel to the body z axis z^B and a reaction torque τ_r. This quantities approximated by a linear model where:

$$ f_i = k\omega_i^2 \tag{4.7} $$

$$\tau_i = k_d \omega_i^2 \tag{4.8}$$

k and k_d are positive constants depending on atmospheric conditions and blade geometry. This constants can be identified through dynamical tests.

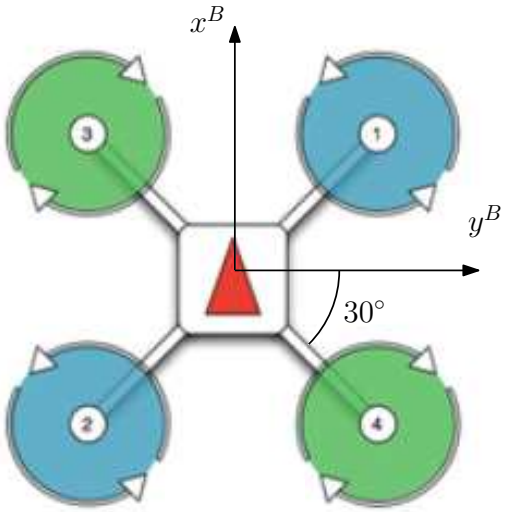


Figure 4.3: Motor labeling and spinning

The total thrust T acting on the robot's center of mass, parallel to z^B and directed upwards is:

$$\boldsymbol{T^B} = \begin{bmatrix} 0 \\ 0 \\ \sum_{i=1}^{4} f_i \end{bmatrix} \tag{4.9}$$

while the total moments

$$\tau_\phi = \sum_{i=1}^{4} d_1 \sin(\alpha_i) f_i \tag{4.10}$$

$$\tau_\theta = -\sum_{i=1}^{4} d_i \cos(\alpha_i) f_i \tag{4.11}$$

$$\tau_\psi = \sum_{i=1}^{4} \sigma_i \cos(\alpha_i) \tau_i \tag{4.12}$$

where f_i and k are defined in 4.7; τ_i and k_d in 4.8; d_i is the distance from the center of the rotor to the center of mass; σ_i is positive if ω_i is positive and negative

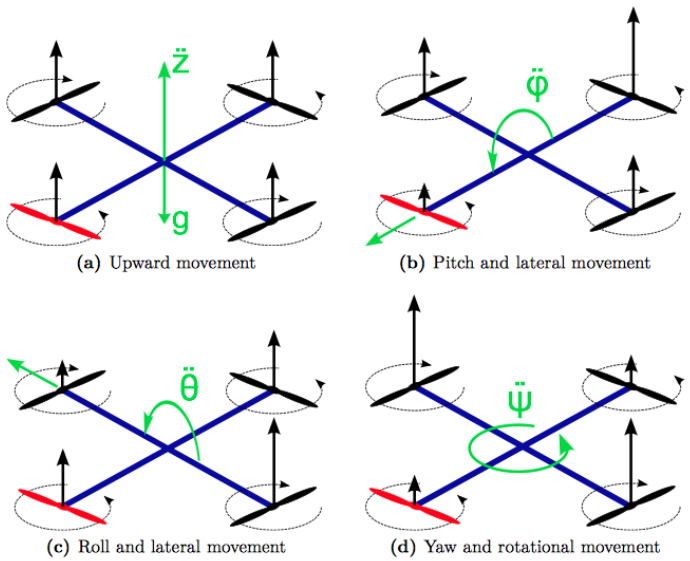
otherwise; α_i is the angle between the vector going from the center of mass to the *i-th* motor and y^B (30° for IRIS).
By rearranging 4.10, 4.11 and 4.12 in matrix form:

$$\begin{bmatrix} T^B \\ \tau_\phi \\ \tau_\theta \\ \tau_\psi \end{bmatrix} = \begin{bmatrix} k & k & k & k \\ kd_i \sin(30) & -kd_i \sin(30)k & -d_i \sin(30) & d_i \sin(30) \\ -kd_i \cos(30) & -kd_i \cos(30)k & d_i \cos(30) & d_i \cos(30) \\ k_d & -k_d & -k_d & k_d \end{bmatrix} \begin{bmatrix} \omega_1^2 \\ \omega_2^2 \\ \omega_3^2 \\ \omega_4^2 \end{bmatrix} \tag{4.13}$$

$$\begin{bmatrix} T^B \\ \tau_\phi \\ \tau_\theta \\ \tau_\psi \end{bmatrix} = H \begin{bmatrix} \omega_1^2 \\ \omega_2^2 \\ \omega_3^2 \\ \omega_4^2 \end{bmatrix} \tag{4.14}$$

where in this case T^B is the value of the force directed through z^B, τ_ϕ is the torque along x^B, τ_θ is the torque along y^B and finally τ_ψ is the torque along z^B. The force T^B is responsible of the translation or the body frame while the three torques generate rotations in each principal axis.

Figure 4.3 shows how motors are labeled in IRIS and the respective direction of rotation while 4.4 is a diagram with the applied forces on the rigid body.



(a) Upward movement (b) Pitch and lateral movement

(c) Roll and lateral movement (d) Yaw and rotational movement

Figure 4.4: Dynamics of the copter. Assume that x axis points towards the red propeller for simplicity, black arrows are the generated forces f_i from aerodynamics. Green arrows represent forces and torques generated combining the rotational speed of the propellers.

4.4 Complete model

Equation 4.13 shows the relation between the squared velocity of each rotor and the generated force and torques . Hence the problem can be split in two parts, aerodynamics and equations of motion. Therefore, **the model is defined as a rigid body on which one external force T^B and three external torques $[\tau_\phi; \tau_\theta; \tau_\psi]$ are applied.** Those external perturbations are calculated in 4.3 and they correspond to the aerodynamic propulsion of the blades.

4.4.1 Rigid body equations

In order to develop the full dynamical model, the control signals must be introduced. Let us define U as a 4-element control vector, then the following is true by definition:

$$\mathbf{U} = \begin{bmatrix} T^B \\ \tau_\phi \\ \tau_\theta \\ \tau_\psi \end{bmatrix} \tag{4.15}$$

Let us also approximate the inertia of the blades equal to zero for simplicity as in (62). We can define the following quantities:

- $\boldsymbol{r} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$ position of the robot in Earth frame.
- $\boldsymbol{\eta} = \begin{bmatrix} \phi \\ \theta \\ \psi \end{bmatrix}$ attitude of the quadcopter respect to earth frame (roll, pitch, yaw) presented in 4.2.
- $\boldsymbol{\Omega} = \begin{bmatrix} p \\ q \\ r \end{bmatrix}$ body angular speed respect to each axis in body frame.
- ${}^E R_B = R(\eta)$ rotation matrix encoding body attitude defined in 4.2 such that $\boldsymbol{x^E} = R(\eta)\boldsymbol{x^B}$.
- $W = W(\eta)$ Euler angle rate matrix defined in 4.2 relating $\boldsymbol{\Omega}$ with $\dot{\boldsymbol{\eta}}$

The dynamics of the quadcopter can be described by the use of Newton-Euler cardinal equation for motion of a general 6 DOF rigid body suffering from an external force and torque. The four equations for the undergoing motion are defined as the following:

$$\dot{\boldsymbol{r}} = \boldsymbol{v} \tag{4.16}$$
$$m\dot{\boldsymbol{v}} = \boldsymbol{F} \tag{4.17}$$
$$\dot{R} = R\Omega_{\times} \tag{4.18}$$
$$I\dot{\boldsymbol{\Omega}} = -\boldsymbol{\Omega} \times I\boldsymbol{\Omega} + \boldsymbol{\tau} \tag{4.19}$$

where the vector $\boldsymbol{F}$ is the resultant of all the external forces applied on the rigid body's center of mass and $\boldsymbol{\tau}$ is the total torque acting on the body. $\Omega_{\times}$ is a 3x3 skew symmetric matrix such that $\Omega_{\times}\boldsymbol{v} = \boldsymbol{\Omega} \times \boldsymbol{v}$ for the vector cross product $\times$ and any vector $\boldsymbol{v}$ (57). The matrix $I = \begin{bmatrix} I_{xx} & 0 & 0 \\ 0 & I_{yy} & 0 \\ 0 & 0 & I_{zz} \end{bmatrix}$ is a diagonal inertia matrix due to the symmetries of the body about principal axis while m is the total mass of the robot.

Equations for translation Since viscosity in air is not considered, the total forces acting on the robot are the gravity $m\boldsymbol{g}$ and the thrust generated by the rotors $\boldsymbol{T^B}$ calculated in equation 4.9.

Let us define $U_1 = T^B$ meaning the first element of the control vector U introduced in 4.15 where T^B is the module of the thrust directed along z^B (see equation 4.13). As consequence (4.17) can be written as follows:

$$\dot{\boldsymbol{v}} = \frac{1}{m}\left(\begin{bmatrix} 0 \\ 0 \\ mg \end{bmatrix}^E + R \begin{bmatrix} 0 \\ 0 \\ U_1 \end{bmatrix}^B \right) \tag{4.20}$$

Note: the first column vector in equation 4.20 is the gravity force and is directed along z^E on its positive direction (downwards). The second column vector is the thrust in body frame which is along z^B pointing to its positive direction. The multiplication with the rotation matrix R rotates the thrust vector expressing in in earth frame since $\boldsymbol{v}$ must be expressed in earth frame.

Equations for rotation The total torques acting on the robot are those originated by the commanding maneuvers calculated in equation 4.13 and the gyroscopical effect induced by the rotating blades.

The gyroscopic torques can me neglected because the mass of the blades is very low, then the total torque applied on the rigid body is the one originated

by the lifting forces and we have that $\boldsymbol{\tau} = \begin{bmatrix} \tau_\phi \\ \tau_\theta \\ \tau_\psi \end{bmatrix}$. Those values are namely the second, the third and the forth elements of the control vector U hence equation 4.19 becomes:

$$\dot{\boldsymbol{\Omega}} = I^{-1}(\begin{bmatrix} U_2 \\ U_2 \\ U_4 \end{bmatrix} - \boldsymbol{\Omega} \times I\boldsymbol{\Omega}) \tag{4.21}$$

where it assumes this simple form due to the assumption of null gyroscopic effects.

Moreover, equation 4.18 can be substituted by the explicit relation between angle rates in earth and in body frame through the matrix W (54). Thus, equation 4.18 is replaced by:

$$\dot{\boldsymbol{\eta}} = W\boldsymbol{\Omega} \tag{4.22}$$

where, for recalling, $\boldsymbol{\Omega}$ is the angular velocity vector in body frame.

Full non linear model

Expanding equations 4.16, 4.20, 4.22 and 4.21 we obtain the full mathematical model represented by the following differential equations:

$$\begin{cases} \dot{x} = v_x \\ \dot{y} = v_y \\ \dot{z} = v_z \\ \dot{v_x} = -\frac{U_1}{m}(\cos(\phi)\sin(\theta)\cos(\psi) + \sin(\phi)\sin(\psi) \\ \dot{v_y} = -\frac{U_1}{m}(\cos(\phi)\sin(\theta)\sin(\psi) + \sin(\phi)\cos(\psi) \\ \dot{v_z} = g - \frac{U_1}{m}(\cos(\phi)\cos(\theta) \\ \dot{\phi} = p - r\sin(\theta) \\ \dot{\theta} = q\cos(\phi) + r\cos(\theta)\sin(\phi) \\ \dot{\psi} = r\cos(\phi)\cos(theta) - q\sin(\phi) \\ \dot{p} = \frac{1}{I_{xx}}(U_2 + qr(I_{yy} - I_{zz})) \\ \dot{q} = \frac{1}{I_{yy}}(U_3 + pr(I_{zz} - I_{xx})) \\ \dot{r} = \frac{1}{I_{zz}}(U_4 + pq(I_{xx} - I_{yy})) \end{cases} \tag{4.23}$$

I want to stress that this model does not take into account factors such as aerodynamic drag, ground effect, blade-flapping, gyroscopic effects or advanced aerodynamics phenomenons. Even with this approximations, this model is the most used in the research because it assures good precision.

Simplified model

Further simplifications can be applied, let us consider the quadrotor in the flat position and now assume that the variation of roll and pitch angles are reasonably small. This is a logical assumption since the quadcopter is designed to fly around such configuration; we can assume that $\phi \approx 0$ and $\theta \approx 0$ and the matrix W becomes an identity matrix as consequence. Therefore under those conditions we have that $\dot{\boldsymbol{\eta}} = \boldsymbol{\Omega}$ and the model is:

$$
\begin{cases}
\dot{x} = v_x \\
\dot{y} = v_y \\
\dot{z} = v_z \\
\dot{v_x} = -\dfrac{U_1}{m}(\cos(\phi)\sin(\theta)\cos(\psi) + \sin(\phi)\sin(\psi) \\
\dot{v_y} = -\dfrac{U_1}{m}(\cos(\phi)\sin(\theta)\sin(\psi) + \sin(\phi)\cos(\psi) \\
\dot{v_z} = g - \dfrac{U_1}{m}(\cos(\phi)\cos(\theta) \\
\dot{\phi} = p \\
\dot{\theta} = q \\
\dot{\psi} = r \\
\dot{p} = \dfrac{U_2}{I_{xx}} \\
\dot{q} = \dfrac{U_3}{I_{yy}} \\
\dot{r} = \dfrac{U_4}{I_{zz}}
\end{cases}
\tag{4.24}
$$

Please note that only the rotational dynamics are simplified. This model, even with the various assumptions and the linearization, is often used to try simple control algorithm and design linear controllers. In our case it is useful since this project do not include the study on aggressive maneuvering or high velocity motions, the small angle variation for roll and pitch is a valid assumption in fact thi model is used to design controllers for hovering.

m	total mass of the robot	1.308 Kg
$I_x x$	inertia for the x axis	0.0018 Kg m^2
$I_y y$	inertia for the y axis	0.0012 Kg m^2
$I_z z$	inertia for the z axis	0.0027 Kg m^2
k	thrust coefficient	0.1
k_d	drag coefficient	0.1

Table 4.1: IRIS parameters. The moment of inertia are taken from the autopilot parameters file while the aerodynamic coefficients are taken from the blade technical sheet

Chapter 5

Control and state estimation

Summary

Due to the nature of the dynamics of the quadrotor, several control algorithms have been applied to it. As to be expected, each control scheme has its advantages and disadvantages. This chapter presents the techniques that are used to estimate the system's states and to stabilize IRIS which are currently implemented in the PX4 Firmware.

After a quick overview of the estimator modules, the controller architecture is presented. Moreover, this chapter explains in details how the on board autopilot interfaces with the software architecture and which modules are involved.

5.1 Introduction to the PX4 Flight Stack

The PX4 Flight Stack denotes the list of all the applications running on board the PixHawk. Those modules provide the services and methods which are necessary to manage the radio communications, inter process message pass-through, data logging, state estimation, control, high level states and low level communication with motors and sensors.

5.1.1 Message pass-through

The core on board applications are started at system startup, others can be started via the NuttShell or forced to startup by inserting them in the start boot file. Every application runs independently with its own frequency; the interfaces between processes are managed by *uOrb* middleware (Micro Orb) which, with the use of topics, guarantees the message pass-through for data packets over named buses. Those topics encode structs and they are pre defined. In PX4, a

topic (often called node) contains only one message type, e.g. the *vehicle attitude* topic transports a message containing the attitude struct (roll, pitch and yaw estimates).

Nodes can publish a message on a bus/topic or subscribe to a bus/topic. They are not aware of who they are communicating with. There can be multiple publishers and multiple subscribers to a topic (Figure 5.1). This design pattern prevents locking issues and is very common in robotics. **To make this efficient, there is always only one message on the bus and no queue is kept** (32). The total list of *uOrb* topics can be found in the uOrb folder of the PX4 Firmware since the online documentation is not updated.

The external communication (through radio link) is managed by mavlink, previously presented in section 3.1.1.3, however Mav packets are translated in uOrb topics internally.

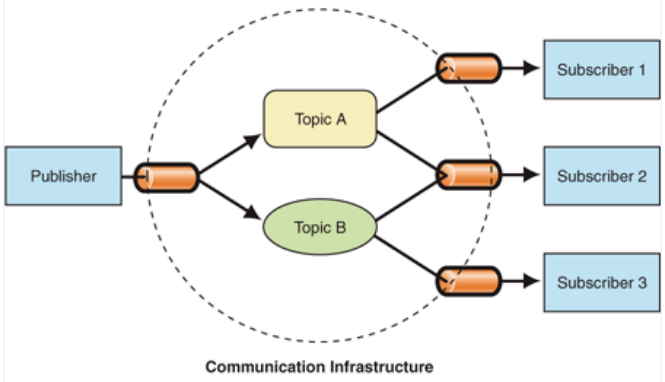


Figure 5.1: Publish/Subscribe design pattern.

5.1.2 Onboard nodes

The most relevant apps that are started at boot can be divided in groups.

System applications Those kind of nodes manage the internal and external process communications , logging and testing. They provide the basic services on which the other modules rely:

- mavlink - is dedicated to pack and unpack mavlink messages.
- sdlog2 - takes relevant topics and creates a log file on sd card at every flight.

- test - mainly used for troubleshooting.
- uOrb - inter-process communication middleware .

Drivers As one may imagine, those nodes represent the layer between hardware and software. They manage the communication with sensors, ports and buses:

- esc_calib - calibration of the electronic speed controllers for motors.
- fmu - manages the board input and output pins.
- GPS - GPS reciever driver
- pwm - command the pwm to be sent to motor controllers
- sensor - communication with various sensors (inertial, baro)

Attitude and position estimators Those modules are responsible for position and attitude estimation, they are part of the core of the flight stack:

- position_estimator_inav - estimates position with inertial sensor, gps and mocap measuraments.
- att_estimator_ekf - estimates attitude using inertial sensors.

Multirotor Attitude and Position Controllers Those are the key modules regarding this thesis. They implements the algorithms for position and attitude control:

- mc_pos_control - position controller
- mc_att_control - attitude controller

Flight safety and navigation Those are the key modules regarding this thesis. They implements the algorithms for position and attitude control:

- commander - internal state machine which determine system states for safety (flying, idle , emergency , on ground)
- navigator - highest level of abstraction, it implements mission following and failsafe

5.1.3 State machine overview

The system relies on a state machine in order to manage what it can and cannot do at a particular moment. The main task of this module is to assure safety from an high level point of view. Other safety procedures are implemented also on a lower level on hardware.

The main states are *ARMED* , *STAND-BY* , *INIT* , *ERROR* and *UNLOCKED*. At the moment the user turns on the robot, the state machine is in *INIT* state. Here all the procedure for sensor initialization and checklists are done. The next state then becomes *STAND-BY* where IRIS waits for orders. By manually pressing a safety button, the state changes to UNLOCKED and back to STAND-BY by pressing again the button. In UNLOCKED state, the motor are electrically connected to the power source thus they can be armed. With a button on the remote control, the motors are armed and start spinning with idle velocity and the robot can fly. From any state, an error signal may arrive changing the actual configuration to ERROR. Here the motors are turned off after the automatic landing procedure and the robot must be rebooted. Those concept are depicted in figure 5.2.

Figure 5.2: IRIS state machine

5.2 Estimation modules

Since I am using an older version of PX4 because it is stable and well tested, the estimation occurs in two different modules. The first module, called *position_estimator_inav* , is used to estimate the position of the quadcopter while the second, named *att_estimator_ekf* , estimates the attitude. The choice of dividing in two different processes the estimation phase is mainly because attitude has an higher dynamics than position, this gives the possibility to run the attitude estimator at a frequency higher than the position estimator thus having better results.

5.2.1 Position estimator

The position estimator is based on inertial model and optimized for multirotors. It reads multiple sensors and estimates 3D position and velocity, in local and global frames. It is a fixed gain estimator and it relies on the following sources with the respective gains:

Sensor	**ID**	**Gain**
Accelerometer (for altitude)	Used as an absolute measure	-
Barometer (gives absolute altitude)	INAV_W_Z_BARO	0.0001
Accelerometer (for x and y)	Used as an absolute measure	-
GPS altitude	INAV_W_Z_GPS_P	0.005
GPS Position	INAV_W_XY_GPS_P	1
GPS Velocity	INAV_W_XY_GPS_V	2
GPS climb rate	INAV_W_Z_GPS_V	2
Mocap estimate (altitude)	INAV_W_Z_VIS_P	5
Mocap estimate (altitude)	INAV_W_XY_VIS_P	7

Table 5.1: Correction gains and available measures

Note: the accelerometers are used as an absolute measure, thus treated as the input of the system used for prediction (equation 5.1). Moreover I changed the gain *INAV_W_Z_BARO* from 50 to 0.0001 because the sensor gives the absolute altitude above sea level while the mocap gives the height respect to the earth frame (the floor of the room). This causes a mismatch in the two measures leading to false prediction with a constant offset, hence the gain for the barometer is set very low.

Working principle of position estimator

The algorithm for position estimation is divided in two phases: prediction and update. The model used for **prediction** is the following:

$$\begin{aligned} \boldsymbol{x_k}(\boldsymbol{x_{k-1}}, dt, \boldsymbol{a}) &= \boldsymbol{x_{k-1}} + \boldsymbol{v_k} dt + \frac{1}{2}\boldsymbol{a})dt^2 \\ \boldsymbol{v_k}(\boldsymbol{v_{k-1}}, \boldsymbol{a}, dt) &= \boldsymbol{v_{k-1}} + \boldsymbol{a}dt \end{aligned} \tag{5.1}$$

where the vector $\boldsymbol{x_k}$ encodes the position in earth frame, the vector a represent the acceleration in each axis given by the accelerometer, v_k is the vector of velocities and dt is the time different between the steo k and $k-1$.

In compact form the we can write the system as the following:

$$\begin{aligned} \boldsymbol{F_k}(\boldsymbol{x_{k-1}}, dt, \boldsymbol{v_{k-1}}, \boldsymbol{a}) &= \begin{bmatrix} \boldsymbol{x_k} \\ \boldsymbol{v_k} \end{bmatrix} \\ \boldsymbol{y_k} &= H\boldsymbol{F_k} \end{aligned} \tag{5.2}$$

where $\boldsymbol{y_k}$ is the output (from sensors) and the H matrix relates the state with the measurements.

Equation 5.2 is usually called **prediction**. It means that with the last available estimate of $[\boldsymbol{x_{k-1}}, \boldsymbol{v_{k-1}}]$ and the actual value of the acceleration $\boldsymbol{a}$ (input of the model) given by the accelerometers, we can predict what could be the next value for $\boldsymbol{F_k}$ and by consequence the output vector $\boldsymbol{y_k}$.

The last item is the **correction** or the calculation of the actual estimated value for the state. The correction equation takes the following form:

$$\boldsymbol{F_k^{est}} = \boldsymbol{F_k^p}(\boldsymbol{x_{k-1}}, dt, \boldsymbol{v_{k-1}}, \boldsymbol{a}) + L(\boldsymbol{y_k} - \boldsymbol{y_k^p}) \tag{5.3}$$

meaning that the estimated (corrected) state $\boldsymbol{F_k^{est}}$ is equal to the **prediction** calculated in equation 5.2 (the superscript p stands for prediction) plus the **innovation** $L(\boldsymbol{y_k} - \boldsymbol{y_k^p})$. The innovation term is composed by L which is a diagonal matrix with the values of the correction gains listed in table 5.1 for each sensor and the difference of the **actual measure read by the sensor**, namely $\boldsymbol{y_k}$, with the predicted output $\boldsymbol{y_k^p}$ calculated in 5.2.

At the end the results are published on *vehicle_local_position* topic.

5.2.2 Attitude estimator

Attitude estimation is a bit more complex being a faster dynamics and accurate results are needed. This module relies on an extended Kalman filter, which is the

non linear version of the very famous linear quadratic estimator. The model is of the form:

$$\begin{aligned} \boldsymbol{x_{k+1}} &= \boldsymbol{f}(\boldsymbol{x_k}, \boldsymbol{u_k}) + \boldsymbol{\xi_k} \\ \boldsymbol{y_k} &= \boldsymbol{g}(\boldsymbol{x_k}) + \boldsymbol{\eta_k} \end{aligned} \tag{5.4}$$

where $[\boldsymbol{\xi_k} \sim N(0, Q), \boldsymbol{\eta_k} \sim N(0, R)]$ are gaussian noises on the system and on the measurements with zero mean and [Q,R] covariances. The available sensors used are:

- magnetometer - for calculating magnetic field on the three robot axis (in robot frame).
- gyroscope - for the calculation of angular rates in body frame (p,q,r).
- accelerometer - for retrieving the value of the acceleration (e.g estimation of the vertical through gravity).

The model used is the rotational part of the system 4.23 (last six equations) with the angular accelerations plus the three value of the magnetic field given by the magnetometer. A well detailed description of the algorithm for this particular case is given in (11).

Yaw estimation The lab environment presented a couple of issues regarding the measure of the magnetic field. Being indoor, with electronic equipments and with electrical cables passing under the floor, the magnetic field is not constant along the room volume. This causes bad estimates for the yaw. Being yaw dynamics pretty slow with respect to roll and pitch, we can estimate it off board with a simple trick.

Let $\boldsymbol{mag}^B = \begin{bmatrix} mag_x \\ mag_y \\ mag_z \end{bmatrix}$ the magnetic field vector in body frame given by the magnetometer and fed to the attitude estimator. Since the earth magnetic field is inclined by almost 30 degrees downwards and pointing north, we can fake it and decide ourself where the north is. Hence let $\boldsymbol{mag}^{\boldsymbol{E}} = \begin{bmatrix} 1 \\ 0 \\ 0.4 \end{bmatrix}$ meaning that x^E becomes the north. The attitude estimator accepts the magnetometer output in body frame hence the "fake" value we provide, by hacking in the module source file, becomes:

$$\boldsymbol{mag}^{\boldsymbol{B}} = {}^B R_E \boldsymbol{mag}^{\boldsymbol{E}} \tag{5.5}$$

where ${}^B R_E$ is the transpose of R which depends on attitude angles. Roll and Pitch are estimated on board but the yaw is obtained by the mocap measure,

thus we can construct the rotation matrix and calculate the rotated magnetic field in body frame. This approach works, is simple and gives us the possibility to choose where the north(yaw = 0) is without having the problem of alignment of x^E with the North-South line.

5.3 Controller architecture

As stated in Chapter 2, those modules implement PIDs controller. The architecture is in the form of what is called a cascaded structure, a very common practice in control design for flying vehicles.

The basic idea is to break up the dynamics of the quadrotor and face the problem piece by piece (58). Thus, the design is divided in four sub controllers placed in a cascaded fashion one after the other. Each controller generates the input for the next one and takes the output of the previous. From the highest to the lowest level we have: *position control*, *velocity control*, *attitude control* and *motor control*. In the context of this thesis, dynamics of motor control are not analyzed but just a brief overview is given.

The **position controller** calculates the velocities set points in the three directions which are fed to the velocity controller. The task of the **velocity controller** is to track those set points by generating attitude reference signals and total thrust. The desired angular pose is tracked by the **attitude controller** which generates the desired torques as input of the **mixer**. The role of the mixer is to calculate each rotor angular speed in order to track the desired signal. This is simply done by inverting equation 4.14 and expressing the angular speed vector in function of the input vector U obtaining:

$$H^{-1}\begin{bmatrix} T^B \\ \tau_\phi \\ \tau_\theta \\ \tau_\psi \end{bmatrix} = \begin{bmatrix} \omega_1^2 \\ \omega_2^2 \\ \omega_3^2 \\ \omega_4^2 \end{bmatrix} \tag{5.6}$$

where the left hand side vector is the output of the attitude (for torques) and position (for thrust) controllers while the right hand side vector is the square of the desired rotor angular speeds. The matrix H is the mixer matrix, it is specific for the robot configurations and it is usually given as a parameter file from the quadcopter designer. At the end of the chain there is the motor controller which, through PWM, assures the convergence to the desired spinning velocity.

Note that those controllers run at different rates, from the position controller which has the lowest rate to the attitude controller which have the highest. Moreover the following assumption is made: **in the chain, one controller converges**

faster than its previous one. This is necessary otherwise the cascade will not work properly. Those concepts are represented in figure 5.3.

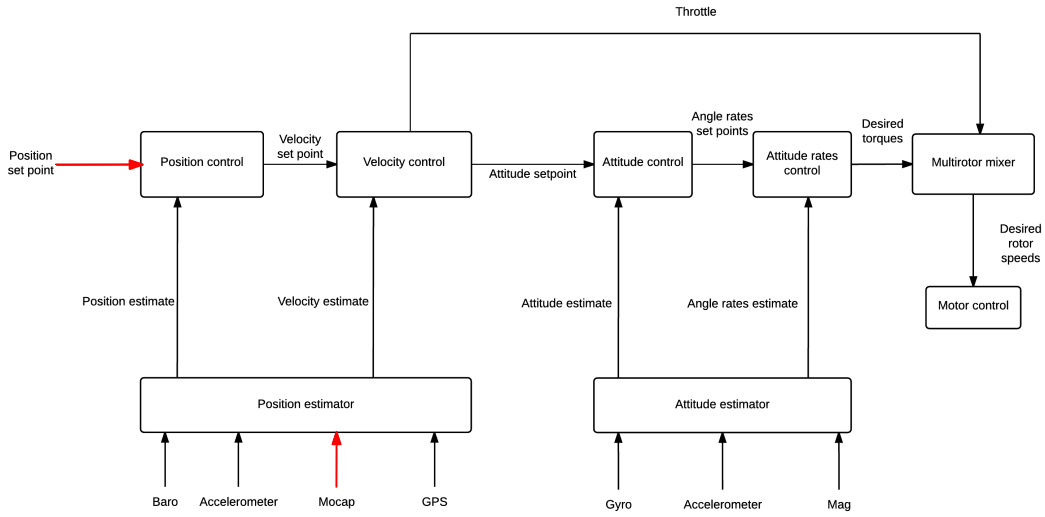


Figure 5.3: Controller overall structure. In red external signals

For the IRIS, three main modes are available:

- manual - the signals from the radio command are fed directly into the mixer
- altitude stabilized - throttle signal is calculated by the controller in order to track an altitude set point given by the user with the radio command. Attitude is still manual.
- position stabilized - the full position set point, **given by the software architecture**, is tracked by the controller chain.

Note: in position stabilized usually the set point is given by the radio command, I changed the autopilot in order to bypass the radio and send set points with the software architecture. Altitude stabilized is used for emergency since one can easily switch mode with a button on the radio.

5.3.1 Position controller

The position control is straightforward. First the position error is calculated with the feedback from the inertial estimator

$$\boldsymbol{e_p} = \boldsymbol{pos_{sp}} - \boldsymbol{pos} \tag{5.7}$$

and then the desired velocity vector is generated with a proportional control law having:

$$\boldsymbol{vel_{sp}} = K_p \boldsymbol{e_p} \tag{5.8}$$

where $K_p = 1$ is the proportional gain. $\boldsymbol{pos}$, $\boldsymbol{pos_{sp}}$ and $\boldsymbol{v_{sp}}$ are 3-elements vectors and they represent respectively the actual robot position in earth frame, the position target set by the software architecture and the generated velocity setpoint.

5.3.2 Velocity controller

Velocity control occurs in two stages. First a desired thrust vector is calculated with a PID control law and then the desired attitude is generated from the thrust vector. The velocity error is defined as:

$$\boldsymbol{e_v} = \boldsymbol{vel_{sp}} - \boldsymbol{vel} \tag{5.9}$$

and the desired thrust vector in earth frame

$$\boldsymbol{th_{sp}} = K_p^{vel} \boldsymbol{e_v} + K_d^{vel} \dot{\boldsymbol{e}}_{\boldsymbol{v}} + K_i^{vel} \int \boldsymbol{e_v} \tag{5.10}$$

with $[K_p^{vel}, K_d^{vel}, K_i^{vel}] = [0, 0, 0]$ and gains are diagonal matrices with the values for x y and z in order to be able to set different gains for different dynamics. At this point we can calculate attitude set points, in the form of rotation matrix to avoid singularities, from thrust vector. The desired z axis of the robot is

$$\boldsymbol{z_{des}^B} = -\frac{\boldsymbol{th_{sp}}}{\|\boldsymbol{th_{sp}}\|} \tag{5.11}$$

since the thrust vector points up and z down. Next, from the yaw value ψ given by the attitude estimator, we calculate the intermediate vector for axis y in XY plane

$$\boldsymbol{y_c} = [-sin(\psi), cos(\psi), 0]^T \tag{5.12}$$

and then with the cross product the desired robot x axis is calculated

$$\boldsymbol{x_{des}^B} = \boldsymbol{y_c} \times \boldsymbol{z_{des}^B} \tag{5.13}$$

As consequence the desired robot y axis is

$$\boldsymbol{y_{des}^B} = \boldsymbol{z_{des}^B} \times \boldsymbol{x_{des}^B} \tag{5.14}$$

The desired rotation matrix, ready to be sent to the attitude controller is

$$\boldsymbol{R_{des}} = \begin{bmatrix} \boldsymbol{x_{des}^B} & \boldsymbol{y_{des}^B} & \boldsymbol{z_{des}^B} \end{bmatrix} \tag{5.15}$$

where $[\boldsymbol{x_{des}^B}, \boldsymbol{y_{des}^B}, \boldsymbol{z_{des}^B}]$ are 3-dimensional column vectors expressed in earth frame denoting the desired axis of the body.
Finally the total thrust U_1 is calculated, the first element of the input vector which is directly sent to the mixer is

$$U_1 = \|\boldsymbol{th_{sp}}\| \tag{5.16}$$

5.3.3 Attitude controller

In order to calculate the last three element of the input vector $\boldsymbol{U}$, the attitude controller takes place. First thing to do is to define the rotation error between the desired rotation matrix and the actual one. Let $S = \begin{bmatrix} 0 & -c & b \\ c & 0 & -a \\ -b & a & 0 \end{bmatrix}$ be a general skew-symmetric matrix, the following is true

$$S^\wedge = \begin{bmatrix} a \\ b \\ c \end{bmatrix} \tag{5.17}$$

and the operator $\wedge$ is called vee map. Then we can define the error vector as the following:

$$\boldsymbol{e_r} = \frac{1}{2}(R_{des}^T R - R^T R_{des})^\wedge \tag{5.18}$$

Next, similarly to the position controller, we introduce the desired angular rater in body frame as

$$\boldsymbol{\Omega_{des}^B} = K_p^{rot} \boldsymbol{e_r} \tag{5.19}$$

and right after the angular rate error is

$$\boldsymbol{e_\omega} = \Omega_{des}^B - \Omega^B \tag{5.20}$$

with Ω^B and R given by the attitude estimator. Finally, through PIDs controller, the command torques are generated and:

$$\begin{bmatrix} U_2 \\ U_3 \\ U_4 \end{bmatrix} = K_p^\omega \boldsymbol{e_\omega} + K_d^\omega \dot{\boldsymbol{e}}_{\boldsymbol{\omega}} + K_i^\omega \int \boldsymbol{e_\omega} \tag{5.21}$$

where the gains are diagonal matrices with the value of the gain of eac dynamics (roll, pitch and yaw rates).

5.4 Results and validation

Some preliminary tests were done in order to evaluate the performances of the controller and the estimator modules. The results are presented in plots taken from the on board logging module and then visualized with FlightPlot , an open source tool available on PixHawk website (13).

5.4.1 Estimator validation

The validation of the estimation is the first test made before flying. The robot was turned on and moved around the room by hands with the motors unarmed for safety. The on board estimation of the attitude is then compared with the vision estimate from the mocap.

Roll estimation The estimation for the roll is very good. The vision signal is delayed respect to the on board estimation due to the radio link delay. In this particular experiment, at second 15, the robot orientation was lost for a moment by the cameras probably because of an occlusion. At that moment a spike appears and it is visible on the plot in figure 5.4.

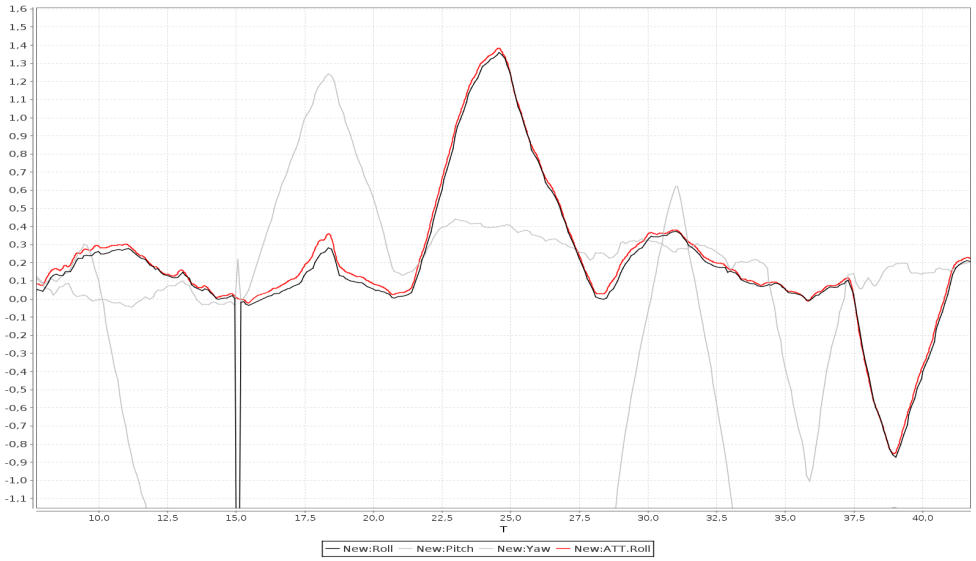


Figure 5.4: Estimation for the roll angle. In black the mocap value while in red the estimated value. Roll is measured in radiants.

Pitch estimation 5.5 As for roll, the estimation of the pitch il consistent. Similar spike appears after 15 seconds (figure 5.5).

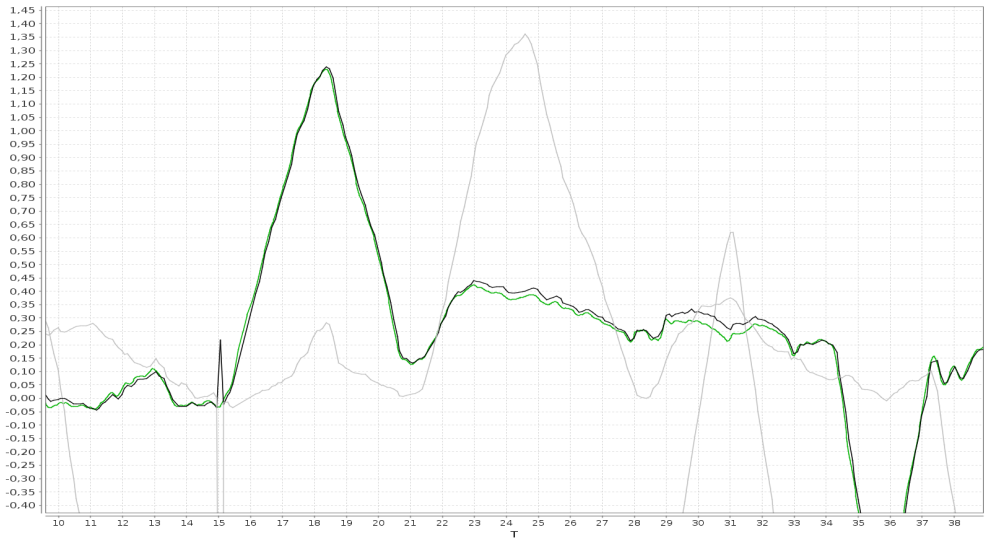


Figure 5.5: Estimation for the pitch angle. In black the mocap value while in green the estimated value. Pitch is measured in radiants.

Yaw estimation The yaw estimation is nearly perfect (see figure 5.6). It does not present the same delay as for roll and pitch because it relies mostly on the mocap feedback. After 43 seconds a jump appears because angle abruptly changed from $-\pi$ to π. However this does not affect the robot dynamics because internal computations are done with rotation matrix representation(see section 5.3.3).

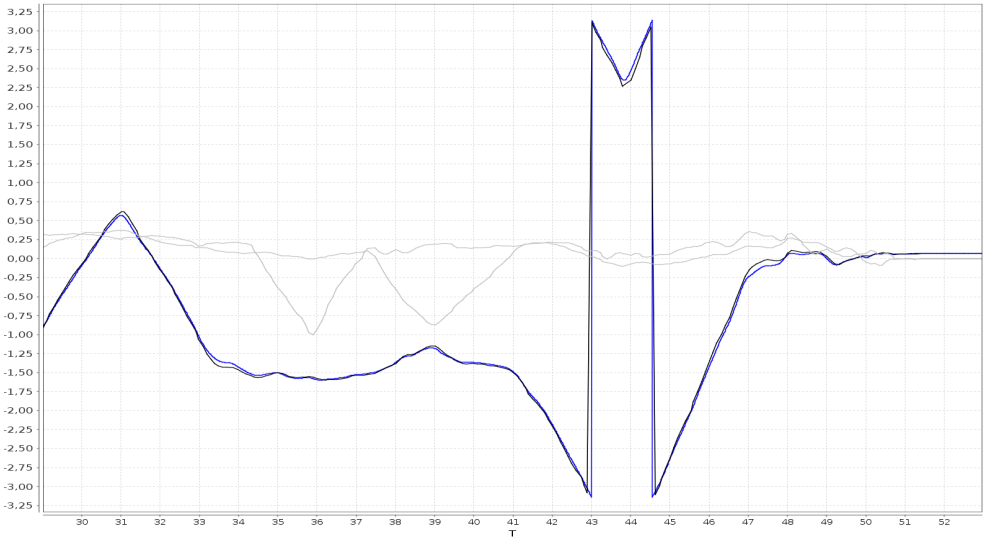


Figure 5.6: Estimation for the yaw angle. In black the mocap value while in blue the estimated value. Yaw is measured in radiants.

Position estimation results are not shown because they are optimal and the two curves are perfectly superimposed.

5.4.2 Controller results

The preliminary test, in order to evaluate the controller performances, is to send to the robot position set points in a square wave fashion. In other words, the robot executes a square trajectory with the vertices at: [0.7, -0.7] ; [0.7, 0.7] ; [-0.7, 0.7] ; [0.7, 0.7].

The plots for the position and the yaw are presented since they are the most relevant.

X and Y position The performance over the horizontal plane is decent. Oscillations are mainly induced by the wind generated by the rotors, bouncing on the walls and back to the robot. See Figure 5.7, 5.8.

Figure 5.7: Convergence on x. The red plot is the robot position while the black is the set point. Position is measured in meters.

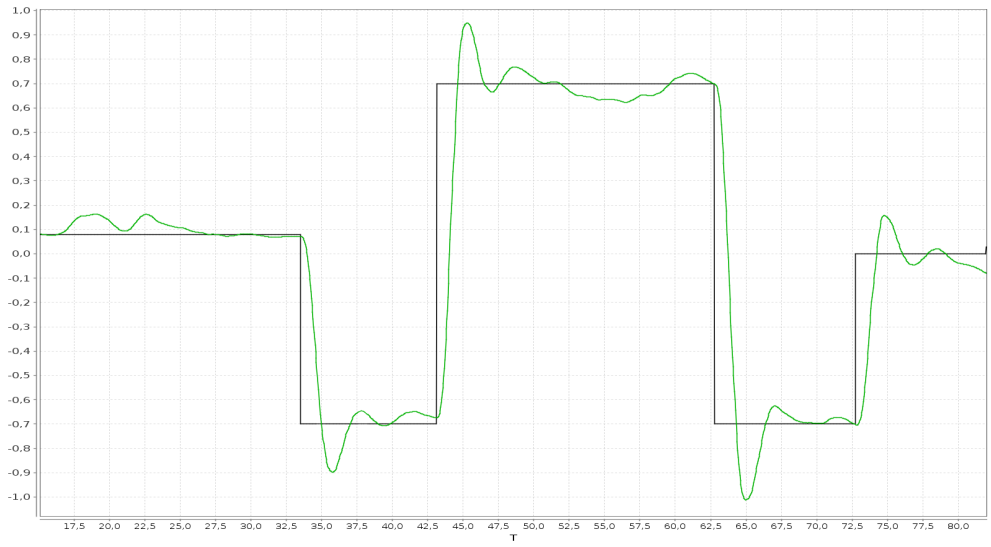


Figure 5.8: Convergence on y. The green plot is the robot position while the black is the set point. Position is measured in meters.

The maximum error is about 10 centimeter in hovering. The overshoot recalls a second order system response

Z position The z converges more slowly when the robot starts (see figure 5.9), this is because the integrator in the desired thrust vector needs to be saturated. By putting as initial condition of the integrator the value of the gravity force one can avoid this problem, which appears only at the beginning during flight. After 70 seconds the landing maneuver starts.

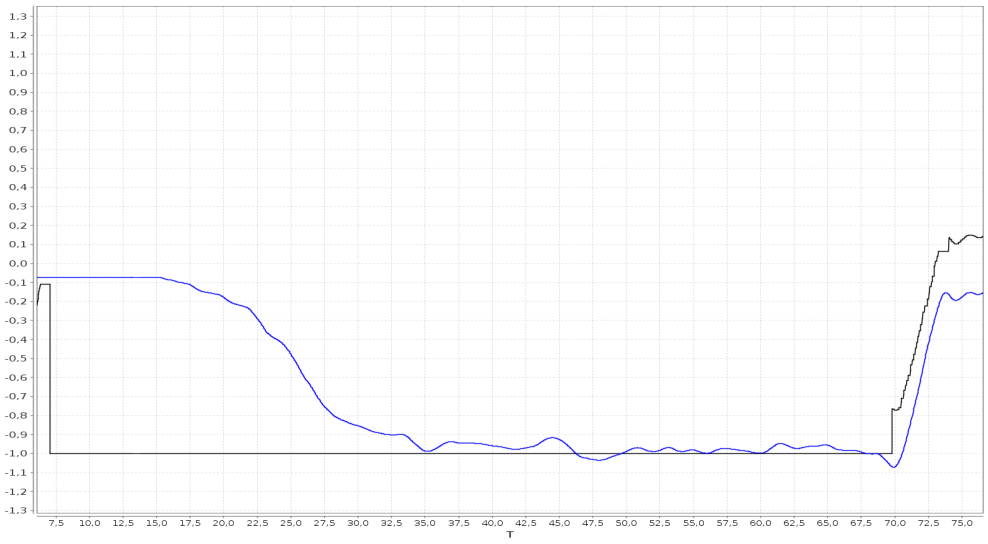


Figure 5.9: Convergence on z. The blue plot is the robot position while the black is the set point. Position is measured in meters. Note that with negative z the robot goes actually up.

Yaw convergence The yaw is commanded by sending yaw set points in a ramp fashion. The ramp appears segmented to the robot since the sending frequency is much lower than the processing rate on the board. In this particular experiment, the robot is asked to look at a specific point. The software architecture send to the robot a desired point (x,y) on the horizontal plane, the robot calculates the yaw needed to face at that particular location and then the controller tracks it. Two different points were chosen, that is why we have two ramps (see figure 5.10).

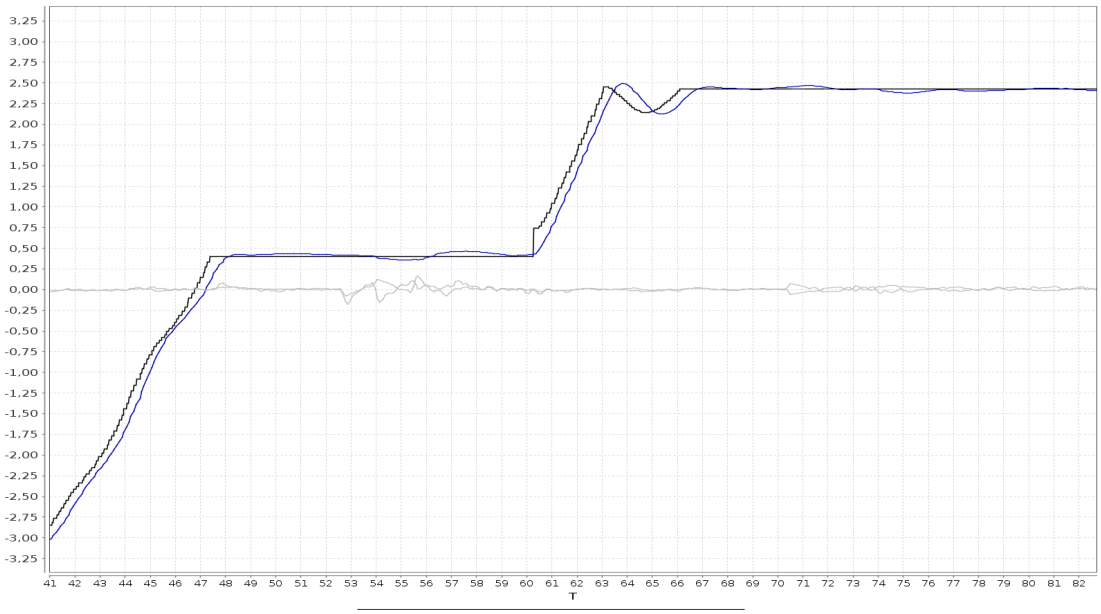


Figure 5.10: Yaw set point tracking, the blue is the real value while the black is the set point

Hovering with disturbances The last experiment consists in asking the robot to hover (stay still) on a point and perturbing it by pushing and pulling with a stick. Thus an external disturbing force, which is represented by arrows in figure figure 5.11, is applied.

Figure 5.11: Static hover with external perturbations. Red plot is x, green is is y and blue is z. The horizontal set point is at (0 , 0) and the height is variable. Light blue represent the set point on z.

Chapter 6

Software architecture

Summary

This Chapter introduces the reader to the software architecture running off board on Linux. The introduction presents a general overview on software architectures, expressed the needs of such a system and in particular it describes the environment and software tools involved. Next section presents the design pattern according to which the software is organized explaining its advantages and limitations. Then the software components are described with more detail one by one and the results of the experiments are reported.

6.1 Introduction to the proposed architecture

Architecture is usually intended as the process or product of planning, designing and constructing entities. Usually those entities refer to buildings and structure but the concept can be extended to vehicles, electrical and electronics components or softwares. The architect decides where to locate different elements such as walls, doors columns and windows and connect them in harmony with structural consistency. In the same way the software engineer connect, design and locate different software components. A component could be a program implementing an algorithm, some conversion or a graphical user interface. The basic idea of architecture definition is to design software structure and object interaction before the detailed design phase. Although serious architecture definition is being suggested for large projects only, arguably any software construction or implementation work must be preceded by an architectural design (31).

6.1.1 Motivations

At this point one may ask: why do we need to define an architecture? The answer is pretty simple: it makes things easier,more clear and simpler. A software architecture is an abstract view of a software system distinct from the details of implementation, algorithms, and data representation. Thus it gives an organizational map we may follow during the design flow. A well written software architecture should:

- Provide flexibility and adaptability.
- Allow for interoperability with other softwares and elements in general.
- Provide control on the system.
- Reduce maintenance time and cost.
- Help developers improving the software.

Reusability is a key aspect in design of this kind of systems. One software may be used for a different application changing only few parameters or modules. Each module should be self contained and work as a *black box*, meaning that once the input and output are defined, the actual implementation has no importance. Standardization clearly takes an important role, the way components communicate for example must be known by the developer. If different modules speak the same *language* or **protocol** (e.g. the MavLink standard) in engineering terms, it is simpler to interface them. Moreover, a self contained module is more easy to maintain and expand because developers can focus on that specific aspect without knowing what is happening outside. In that way specialist in different fields can cooperate designing each own part. The role of the software engineer is on one hand to design software components, and on the other to integrate them with modules written by others. Finally, it seems trivial to point it out, but the architecture must work respecting the specifications and providing the needed control on the system.

Note: This software was designed ad-hoc for indoor flight because there were not any other alternatives. Every control station is specialized for outdoor flight which is not our case. Moreover this software aims to become a research platform for controlled environments (e.g. indoor flight) for anyone who wants to contribute.

6.1.2 Programming environment and tools

Every job has its own tools. In order to implement what we theorized in the introduction of this Chapter we need to rely on software tools. There are many different frameworks which helps developers in implementing their own ideas.

The state of the art and widely used framework in robotics is ROS or Robotic Operative System (30). ROS is a publish/subscrbe middleware meaning that it packs function classes and features which provide inter process communication. It supports most of the libraries used in robotics for path planning, computer vision, control and so on. The main feature is that ROS is very easy to use and let the user create different parallel processes (or nodes) without focusing on low level aspects. As consequence of that, the designer can concentrate on the actual problem he is working on and leave lower level managing such as shared variables, timing or buffers to ROS. This feature increase exponentially the productivity while writing a program. Moreover, ROS is becoming a standard in research and also industry. That means that many packages are available online that one can use, the community and the documentation are superb and it is open source. This framework has all the features needed for a good base of a software architecture.

However there are three main reason which made me discard ROS as a choice. First of all, as stated in Chapter 2, most of the control stations are written with Qt libraries. Since one may thing to include this architecture in one of them in the future, could be an idea to go in the same direction. The second one is that the very first module of this architecture was written by a PhD student, Tommaso Falchi Delitalia, using the Qt framework. It was nice to have a base starting point and expand from that. The last reason, but not the least, is that ROS gave me important delay problems when I tried to integrate mavlink in it. Mavlink is not fully supported but some packages, in development, are out there and they simplify the design such as the acquisition of data from Motive (1).

Hence the used tools are **Qt libraries** (27) while the chosen programming language is C++, widely used and a standard in robotics. Qt is a powerful framework that let the user create user interfaces with good performance. It packs a set of classes, functions and libraries for almost any kind of needs. Moreover is portable on different platforms such as tablets, smartphones and the most important operative systems. for this reason it is used to implement control station, applications like navigators and vehicles control panels.

The functions and classes used for this project are debugging functions to print logs, multi-threading classes to implement parallel modules, sockets interfaces classes and system functions to manage lower level services.By far Qt seems perfect, it provides a very easy access to many features, the documentation is very clear and the learning time is pretty low. The very big disadvantage is the lacking of inter process communication support. The pub/sub design is perfect

for robotics application but Qt does not provide any help on that, at least for now. Thus the drawbacks of using this kind of framework is that we need to manage inter process message pass-through in some way. This is done and explained in section 6.3. A porting on ROS could be interesting in the future, after solving delay related issues, for research interests.

6.2 Design patterns

In the course of history the concept of *style* was born and developed. An architectural style is characterized by the features that make a building or other structure notable and historically identifiable. The style identifies a common trend, elements or rules that are used in a particular period or by a group of people during the design flow. The same concept evolved to various disciplines; in computer science, the architecture "style" is often called **Design Pattern**. The analogy with building design is strong, the formal definition of design pattern is the following:

Definition of Design Pattern : *a design pattern is a general reusable solution to a commonly occurring problem within a given context in software design.*

The pattern is a description or template for how to solve a problem that can be used in many different situations, usually for groups of problems. Similarly to the architecural style, the software pattern is defined by common structures, participants, communications, protocols and standards. Design patterns can speed up the development process by providing tested, proven paradigms. Thus the software architecture is defined by its style, namely design pattern, which is chosen among the available ones. First step to follow during the prototyping of the software is to have clear in mind which kind of problem we are facing and which are the constraints of the system.

At this point the reader should ask how is the software interfacing with the robot and which kind of control does it have on the system. As stated in Chapter 5, the external signals that are input of the on board flight stack are the 4-D position value from the mocap and the 4-D position set point. By 4-D we mean a vector composed with [x, y, z, yaw] since roll and pith are estimated on board (see figure 5.3). Moreover the following goal is set:

Goal: *The robot must be able to perform some kind of tasks provided by the user in the most autonomous way possible.*

In other words, the user provides a list of tasks he wants to perform with the robot and the software architecture guide the system by sending mocap values and position set points. The first scheme of the software is represented in figure 6.1. The scheme explains the input output relation, where as input there is a C-

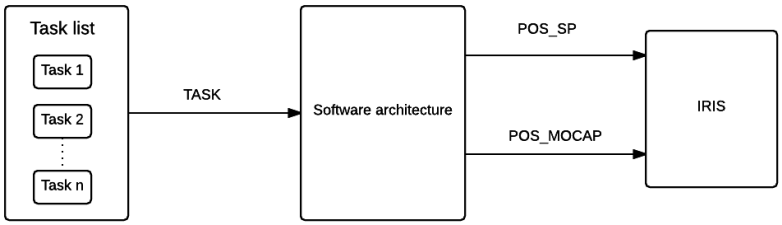


Figure 6.1: Input / Output relation of the software architecture.

struct describing the task and as output MavLink messages for mocap estimate and position set point. Nevertheless, the real structure is a bit different. I decided to put the task list inside the software architecture as a nested component. The main reason for that is simplicity, in the future one may encode the list in a text file as input of the software. See figure 6.2 for the actual implementation.

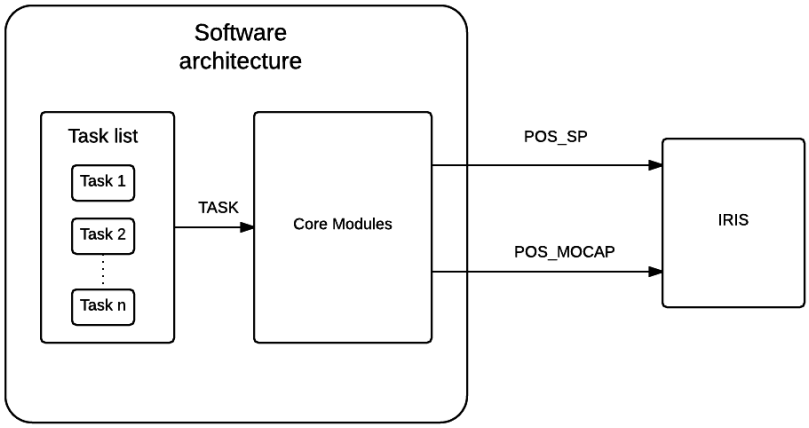


Figure 6.2: Input / Output relation of the software architecture with nested component.

6.2.1 Behavioral architecture

At this point, a way to model the problem is necessary. Let us start from the goal: there a list of tasks and the robot must executes them sequentially and autonomously. Thus, the concept of task arises. The task is defined as *a definite piece of work assigned to the robot and performed by acting on the environment.*

Defined tasks Which kind of tasks may be performed by IRIS? The first step is to define the essential tasks of navigation which are:

- Take off - from the ground or an horizontal plane.
- Land - land on actual position.
- Move - go to a target point in 3-D space.
- Rotate - Change yaw value to a desired one.
- Follow trajectory - Perform a circle around an arbitrary center set in the parameters.

Moreover a fifth task is added but discussed in 7 which is land on a mobile platform.

In order to make things more flexible, each task has a set of parameters used to influence the performance during the execution and to adapt to the environment. A detailed explanation is given in 6.3.4. Furthermore, it is evident that most of the tasks described involve in some way a location in space. Hence the task is modeled in a C-struct with two elements: position and action. In the software the task takes the name **node** which is often used in control stations and graph based applications.

Table 6.2.1 describes the node struct. The position p is itself a struct with four elements (x,y,z,yaw) encoding the 4-D pose in space. The action a is another stuct with a char value to identify the type of action and a parameter array which can be filled with values. The meaning of the parameter array changes depending on the action. Those elements are necessary in order to choose which design pattern is suitable in this environment.

Taking inspiration from biology and observing how simple animals interacts with the environment, robotic schemes and models may be derived. Biologist discovered that animals like frogs, pigeons insects and fishes exhibit different behaviors depending on the sensory inputs they receive from the environment. With this

Node struct			
position p			
	x	double	meters
	y	double	meters
	z	double	meters
	yaw	double	radiants
action a			
	id	char	
	$params$	double[4]	

Table 6.1: Node struct descripton

simple method they are able to survive, hunt and navigate. Two different examples explain the concept very well.

Frogs eyes are able to detect movement. In particular on layer detects small moving objects like flies and another layers detects big object, for example predators. The two layers work in parallel. When the first one is activated, meaning that there is food in the proximity, the frogs jumps towards the pray. On the other hand, when the second layer is triggered, the frogs run away from the object.

The second example involves the navigation of pigeons. When it is sunny, the bird uses the sun to locate himself (behavior 1). When it is cloudy they use the magnetic field of the earth to detect the north (behavior 2). By disturbing the pigeon with artificial light, it get confused only in sunny days. Vice versa, if disturbed with a magnet, it get confused only in cloudy days. This simple experiment lead to an important result: the pigeon switch behavior depending on some triggering inputs just like the frog. More over only one behavior is active.

Behavior definition *A behavior is a mapping of sensory inputs to a pattern of motor actions that are used to achieve a task.*

For the frog, the sensory inputs are be the position of the moving objects while for the pigeon they are the position of the sun and the value of the magnetic field. Moreover, a trigger is present. Those triggers are of different nature depending on the animal, they activate the behaviors or switch from one to the other. Figure 6.3 shows the basic model of the behavior. Inputs and output may be more than one while the activation trigger is always a single signal. From an engineering point of view, imagine the behavior like a process, an algorithm, a thread or a module. Inputs and outputs may be variables, signals or parameters. The triggers are often booleans. The logic that manage the triggers is called **switching logic** and it is a key aspect in the architecture. The switching logic is usually a module that takes inputs from perception modules, and outputs the boolean values of each trigger.

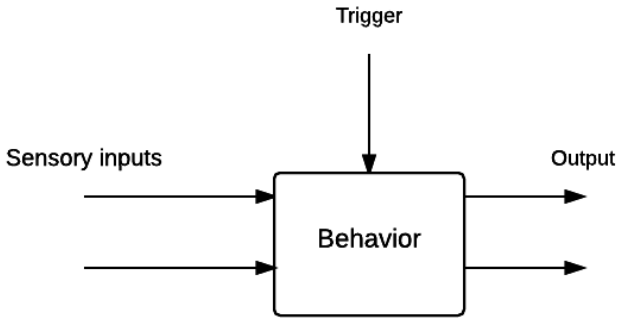


Figure 6.3: Scheme of the behavior.

Figure 6.4 describes a behavioral machine, n behaviors are connected to n trigger signals coming out from the switching module. The logic can be implemented in different ways. It could be a combinatorial circuit made by logical ports, some mathematical functions or a state machine but its output must be a set of n boolean values.

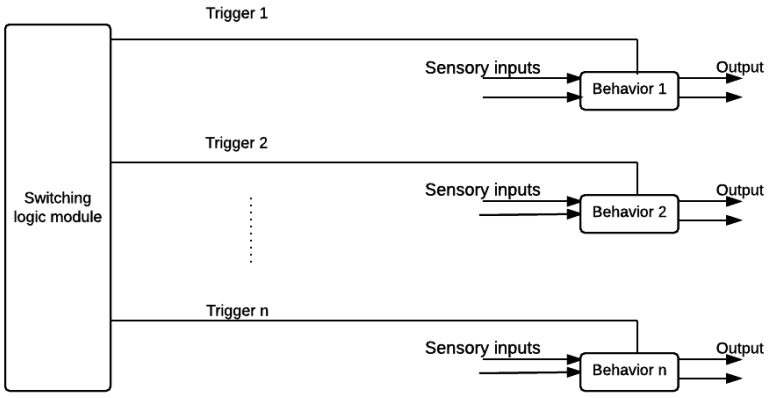


Figure 6.4: Behavioral system with switching logic.

Concurrent behaviors One aspect I find very interesting is the concept of concurrent behaviors. Two or more behaviors are concurrent when they are active at the same time. This approach may solve elegantly different problems but we must pay some attention. Imagine the frog looking around. At some point an insect and a predator appear together in the field of view activating two

behaviors. She is hungry, but more important, she does not want to be the meal of someone else. In this case behaviors can be prioritized (the most important is executed). A second way is to include this scene in the switching logic where behaviors inhibit others. Moving may inhibit landing for example.

On the other hand let us picture the following situation: a mobile robot moving on the ground and going towards a target. The behavior *move straight* is activated and the robot goes in the direction of the goal. At certain point the perception module senses an obstacle in front and activates the behavior *avoid obstacle*. The two behaviors are concurrent since we assume that there is no inhibition in the switching logic. The first output is to spin the wheels with the same speed (move straight) but the second behavior commands the steering wheels to turn and avoid the collision. The to outputs add up and the result is the robot avoiding the obstacle. When the perception module does not sense anymore the obstruction, the second behavior is turned off and the robot continues moving towards the target.

It seems pretty trivial but this approach has a lot of potential. It saves resources turning off and on different modules depending on the situation. Switching logic can be implemented on hardware thus optimizing the performances. Moreover the combinations of simple behaviors may solve complex tasks.

For this case, in order to solve concurrency issues, I used a simple rule: **concurrent behaviors are possible only if they act on different dynamics**. For example, move and rotate can be activated together because one changes x, y and z while the other changes yaw set points. Follow trajectory and move cannot be concurrent because they act on the same dynamics. One may need to follow a trajectory, like a circle around a point of interest, and film with the camera the center thus keeping the nose always pointing to the target. This is solved combining *follow trajectory* with *rotate*. Although it is not implemented in this way, potentially the landing on a mobile platform task may be defined by two concurrent behaviors. The first is *move* or track the platform, and the second is *land*. The more behaviors are added, the more combinations are possible.

Implementing behaviors The switching logic is implemented as a finite state machine which is presented in section 6.3.5. Behaviors are functions with the following inputs: the robot state (x,y,z,yaw), the actual position setpoint(x,y,z,yaw) and the node value. As output they give the new position setpoint which is updated. Figure 6.5 shows the data flow for general set of behaviors, the switching logic is not shown for simplicity. Regarding implemented behaviors, the switching logic is almost a one to one map. Each presented task has its own behavior called with the same name. In this version of the software, the only concurrency appears in *follow trajectory* task which performs a circle and keeps the nose always at the

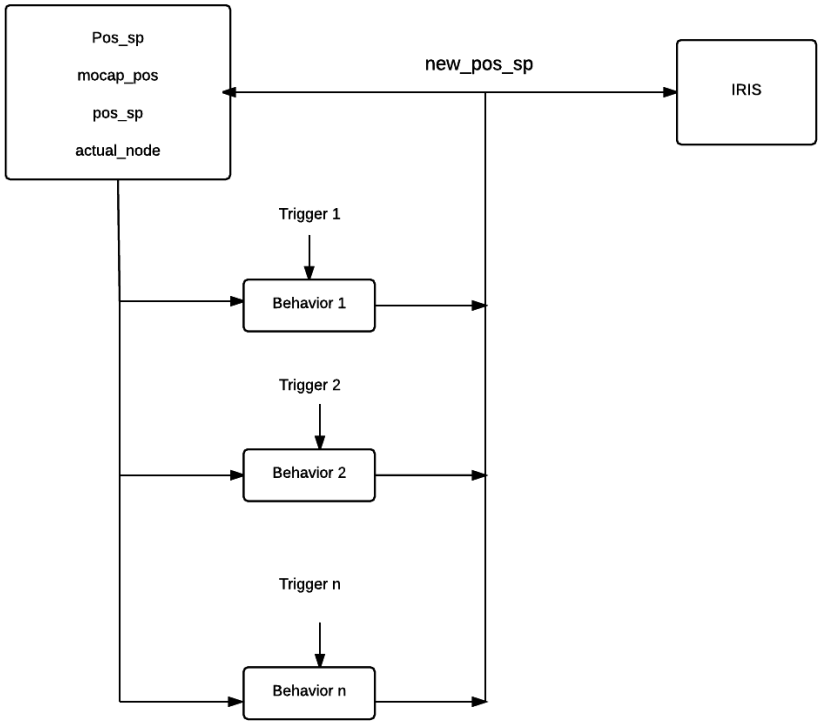


Figure 6.5: Input/Output relations with particular adopted scheme. Switching logic is not depicted.

center. Depending on the actual task, the switching module activates different behaviors and the map is shown in table 6.2.

For prototyping and simplicity reasons, the mobile landing has its own behavior but two concurrent behaviors can be assigned to it, namely *move* and *land*. Moreover, many improvements can be done. For example one can decide to activate or not the *rotate* behavior in the follow trajectory task by passing a value in the params array. This architecture can be largely expanded from its actual state by adding behaviors, concurrencies and tasks.

Actual task	Activated behaviors
take off	take_off
land	land
move	move
follow trajectory	follow_traj ; rotate
rotate	rotate
land on a mobile platform	plat_land

Table 6.2: Switching logic mapping.

6.3 Software description

At this stage, the actual implementation of the concepts explained previously can be described. The structure of the application is a multi-threaded scheme. A thread of execution is a sequence of programmed instructions that can be managed independently by a scheduler. Different threads implement different modules or processes running simultaneously and independently. Within the Qt environment, multi-threading is easily implemented through the QThread class. By inheriting its properties we can redefine its *run()* method and design the wanted algorithm. The advantages of having a multi-threaded architecture are several. Threads can be started and stopped as many times we need. Moreover, if a thread fails or get stuck, the others continue running since they are all independent. This is a key aspect in consideration of the fact that there are essential modules which cannot be blocked by other threads.

This scheme presents also a main drawback. When two or more threads are running in parallel, they may need to access to the same variable. As result they could interfere with each other, thus synchronization is essential and some precaution is a must.

To summarize, we can write threads by inheriting properties by the QThread class and overloading its *run()* function with our custom code. By calling the start method, the thread is started and *run()* is executed, usually an infinite loop is implemented with a custom rate. Each loop runs in parallel, hence if one gets stuck the others continue spinning.

Another powerful tool is the connect method. This method is member of each class in Qt since they are derived from the same one. Every class has signals and slots. Slots are just methods while signals are booleans. With connect, we can relate the signal of one class with the slot of another class. Meaning that, if the signal of the first class at certain point goes up then the slot of the second class is executed. It can be done within threads so callbacks can be partially simulated. With this first overview, the reader should have a rough idea on the working principle of the system. Infinite loops are running simultaneously and

independently and each one represent a module. The following custom module are implemented, they represent the main ingredients and the core of the application:

- NatNet Reciever
- Position Dispacher
- Manual Control
- Automatic Control
- Executioner
- Commander

The *NatNet Reciever* and *Position Dispatcher* are also called service module. The Reciever task is to read the value of the mocap estimate and save it on a global variable. The Dispatcher takes the estimate and the position set point, packs them in MavLink messages and send them trough a socket interface via usb to the radio module.

Manual Control module incorporates an user interface through which the user can change manually position set point during flight. It runs on the same thread of the service modules, namely the main thread which is created at the starting of the entire application.

Automatic Control stores the algorithms for every behavior and execute them depending on the switching logic. It is the central member of the application.

Executioner module implements the switching logic as a finite state machine.

Commander is the lower level module. It is the only one which has the permission to update the position set point value.

Communication Communication between modules is done with a simple method. Every common data is stored as a global variable. Producers update the variable while consumers read it from the global space. This approach is not optimal but very simple to implement. With the use of the QMutex class we are able to lock and unlock variables in the read/write operations.

Note: service modules were previously implemented by Tommaso Falchi Delitalia, in his PhD work, and slightly modified by me in order to adapt them with the rest of the application.

6.3.1 General scheme

More technical details can now be presented. As already mentioned, the design started with service modules implemented. In that implementation the communication management with global variables was used and I continued on that direction.

Implementation The actual implementation is explained in this section. First of all, let us introduce the class MavState. This class is used to store mocap values and position set points. The elements of the class are 3 coordinates of the position in space, 4 values for the quaternion orientation and one value for the yaw in radians. The class with members and methods is shown in table 6.3.

members			
	x , y , z	position in 3D space	meters
	yaw	value for the yaw	radians
	qw , qx ,qy qz	values for the orientation	quaternion
methods			
	setPosition(x,y,z)	set values for the position	
	setOrientation()	set values for orientation passed like quaternions or RPY angles	
	getPosition(x , y , z)	get value of position	
	getOrientation()	get value of quaternion or RPY angles	

Table 6.3: MavState Class

The full scheme is represented in figure 6.6. We must give some consideration on the implementation of system. The mocap estimate is saved in a MavState global variable, potentially every module may access it. The used entries are position (x , y , z) and yaw (retrieved from quaternion).

The automatic Control implements the behavioral part and calculates a new position set point (position and yaw). The new setpoint is pushed in a vector called auto command. In the same way, Manual control pushes the new set point in the manual command vector. The Commander module checks both vector and evaluate which command is good to be sent. This rule is used: **if the two vectors are non empty, then the command sent will be the one in the manual vector**. It basically prioritize the manual command over the automatic ones and update the position set point in the global space. The executioner implements the switching logic; according to the actual state of the robot and the actual action to be performed it decides whether stay in the same node or switch to the next one. It then updates the values of every triggering signal which are saved in the global space.

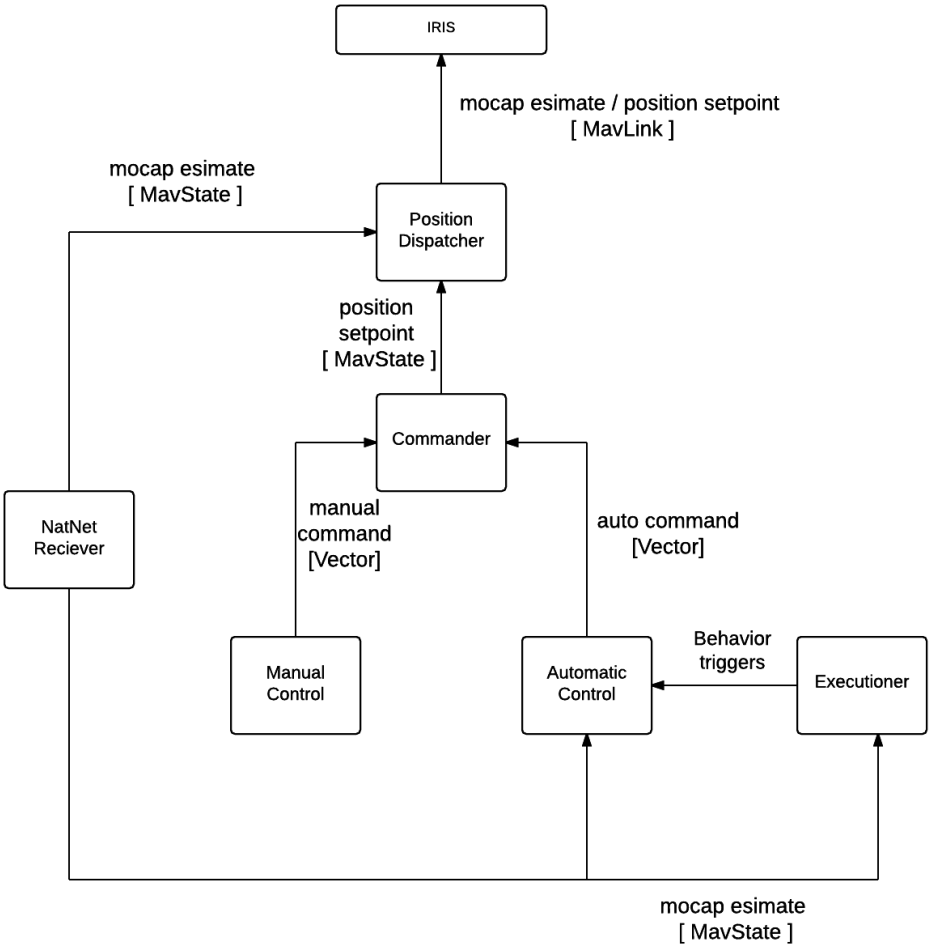


Figure 6.6: Implementation of the system.

For sake of clarity, in figure 6.6 messages are represented by arrows. The connections are only logical because potentially every module can access those signals since they are stored in global space. In addition, the node list defined by the user and the id of the actual node which is executing are stored globally.

6.3.2 Service modules

Those modules run on the main thread and they use the signal/slot feature embedded in Qt.

NatNet Reciever

Using the proprietary SDK from NaturalPoint, this module parses the position of the center of mass and the orientation of every rigid body tracked by motive. In the actual state of implementation, only two bodies are saved. The value is saved in MAvState classes contained in the namespace g with the following convention:

- Rigid body number 1 is saved in g::state
- Rigid body number 2 is saved in g::platform

The order is very important, the robot must be labeled as one and the mobile platform as two. This is easy to check with motive. Every time g::state is saved, the signal dataUpdate is emitted.

Position Dispatcher

The signal dataUpdate is connected to the slot sendPosition() of the Position Dispatcher. As consequence, every time the robot state is updated, sendPosition() is executed. This function takes the state and the postition setpoint (saved by the commander in g::setPoint variable) and fills two mavlink packages. The state MavLink package is predefined for Optitrack, while for the set point I used the Vicon dedicated package. The same message cannot be used since PX4 creates a uOrb topic for each mavLink type received, thus they would be written on the same topic.

When the messages are ready, trough a socket interface, they are sent to the radio module. The radio has a dispatch interval of 100 ms, meaning that after sending data, we must wait 100 ms. If messages arrive in this interval they are ignored.

6.3.3 Manual Control

Manual Control is started at the beginning in the main thread. It features a simple user interface shown in figure 6.7 which translates the position set point on three axis and rotates th yaw. The home button issues a setpoint in the center of the room. The starting setpoint is $[0, 0, -1]$ for position and π for the yaw. Thus, **the robot must be placed on that location** on the horizontal plane and yaw if one wants to fly it by manual control. If not placed correctly, the take off could be dangerous. The start and stop button run an auxiliary thread which I used for prototyping and testing.

Each time the user click on a translational button, the set point is moved by 0.1 meters on that direction in earth frame. The yaw is changed by steps of $\pi/18$ radians.

Figure 6.7: Manual user interface.

6.3.4 Automatic Control

This module is the core of the application. It implements the behavioral pattern explained in section 6.2.1. It is defined with the class *Automatic* and it has one member: *AutoThread*. This thread implements a *run()* function which is started by clicking on the button AUTO. The loop runs at 10 Hz. When the thread starts the system is in automatic mode, thus it executes the nodes present in the list (stored globally) in succession. Every behavior is implemented as a function and the structure of the loop is: The structure is simple, while spinning, we read each value of ¡*task*¿_*sig* coming from the switching logic and activate different portion of code. In particular, we run specific functions depending on the behavior. Those function modify the *comm* variable which is then published in the command vector. The signal publish() at line 23 is connected with the commander slot which is asked to check to check the command vectors.

At this point we can explore deeply how behaviors are implemented.

Take off Taking off is pretty simple. When the behavior is activated for the first iteration, it saves the actual horizontal position and yaw. Then a set point is issued on the vertical of the starting position at a fixed altitude with the initial yaw.

Move The move action is a bit more complex. At each loop spin the 3D position error is calculated, namely the difference between the goal and the actual position. This difference results in a vector $\boldsymbol{V_e} = \boldsymbol{P_g} - \boldsymbol{P_r}$ originating on the robot position $\boldsymbol{P_r}$ and going towards the goal P_g. Let us define a positive constant α and let

Algorithm 1 Brief overview of the automatic thread.

```
1: while true do
2:    comm = g :: setPoint  {save previous set point}
3:    if take_off_sig then
4:       take_off();
5:    end if
6:    if move_sig then
7:       move();
8:    end if
9:    if land_sig then
10:      land();
11:   end if
12:   if rotate_sig then
13:      rotate();
14:   end if
15:   if follow_traj_sig then
16:      rotate();
17:      follow_traj()
18:   end if
19:   if land_plat_sig then
20:      land_plat();
21:   end if
22:   autoCommand.pushback(comm) {fill the command vector}
23:   publish()
24: end while
```

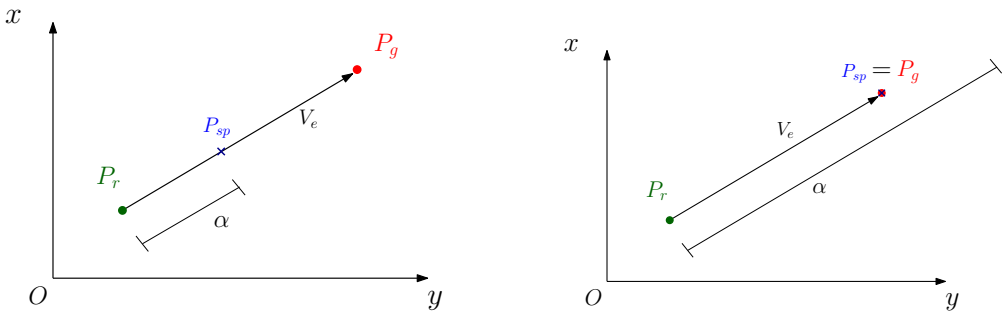


Figure 6.8: Move function. On the left the limited position set point in blue, on the right the case with no scaling.

$\boldsymbol{u_e} = \frac{\boldsymbol{V_e}}{\|\boldsymbol{V_e}\|}$. Then the following method is used to calculate the position set point P_{sp} to be sent to the commander:

$$\boldsymbol{P_{sp}} = \begin{cases} \boldsymbol{P_g}, & \text{if } \|\boldsymbol{V_e}\| \leq \alpha \\ \boldsymbol{P_r} + \alpha \boldsymbol{u_e}, & \text{if } \|\boldsymbol{V_e}\| > \alpha \end{cases} \tag{6.1}$$

The principle of this function is depicted in figure . The reason why we limit the position set point is because of the effect of the proportional term in the position controller. Issuing a far set point, the error is high and the desired velocity becomes big. Hence, in order to avoid this kind of effect, which results in bad flight dynamics, we limit the distance of the issued position set point from the robot by a distance α.

Land Landing maneuver consists in slowly increasing the z (the z axis is upside down, increasing z the height decreases) while maintaining the horizontal coordinates of the last position set point(where the robot is). Thus, only z coordinate of g::setPoint is changed. The trick used simulates kind of velocity control. The z coordinate of the set point is calculated depending on the horizontal error between the robot and the vertical of the target landing point and it is given at a fixed vertical distance with the robot. That means that when the robot descend, the set point maintain a fixed distance with robot position. This results in a fixed vertical error an a constant vertical velocity. Let h_e being the horizontal error of the robot from the landing vertical and $P_r(3)$ the z coordinate of the robot, then the following empirical value are found to be good:

$$z_{sp} = \begin{cases} P_r(3) + 0.3, & \text{if } 0.5 < h_e \leq 0.8 \\ P_r(3) + 0.5, & \text{if } h_e \leq 0.5 \end{cases} \tag{6.2}$$

When the height of the robot is sufficiently small, the rotor are set to idle by issuing a set point under the ground and the robots land. Note that the vertical set point sums up with the robot position because z axis is pointing towards earth. Moreover, horizontal set points are slightly compensated in order to increase precision. Let $\boldsymbol{P_t} = [x, y]^T$ the landing target on the horizontal plane and $\boldsymbol{P_{sp}} = [x_{sp}, y_{sp}]^T$ the actual set point command, then the compensation is the following:

$$\boldsymbol{P_{sp}} = \boldsymbol{P_t} + K_p \boldsymbol{e_p} \tag{6.3}$$

Where K_p is the correction gain and $\boldsymbol{e_p} = \boldsymbol{P_t} - \boldsymbol{P_r}$ with $\boldsymbol{P_r}$ the robot horizontal position. In this way we increase the landing precision emulating an increase on position gains.

Rotate Rotate action is done by sending yaw set points in radians. Let us define two functions: *increase_yaw()* and *decrease_yaw()*. The actual implementation is not discussed but a brief presentation is given. The first function issue a yaw set point which is equal to the actual yaw plus a delta (rotate a bit clockwise), on the other hand *decrease_yaw()* rotate a bit counterclockwise. With those two elementary methods we can design an algorithm which, given a generic yaw set point in radians, rotates the robot in the direction which spans the smallest angle to the target.

Firs of all, we must recall few concepts. In section 4.2 the yaw angle is defined as $-\pi < \psi < \pi$. Imagine a top view of the earth frame with x axis vertical and going upwards and y axis horizontal pointing to the right. The yaw is the angle is positive on the right half of the plane and negative on the left half. This property works also in body frame meaning that: each direction on the right side of the robot has a positive yaw and each direction on the left side of the robot has a negative yaw (in body frame, be careful). Positive yaw in robot frame means a clockwise rotation while negative yaw means a counterclockwise rotation (we have functions for that).

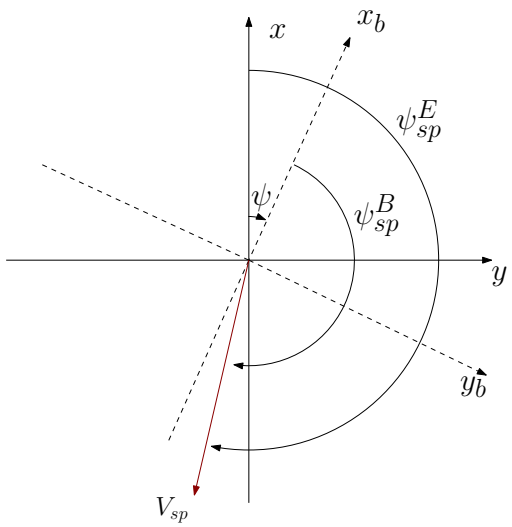
That said, the idea is simple. First we express the desired direction (yaw set point) in robot frame and then we check whether it lies on the right or left side by looking at the yaw sign. For expressing these procedure in equations let $R_z(\psi)$ be the rotation matrix on z by an angle ψ and $\boldsymbol{V_{sp}^E} = [\cos(\psi_{sp}), \sin(\psi_{sp}), 0]^T$ a unit vector encoding the desired direction in earth frame calculated from the yaw set point ψ_{sp}, then the desired direction in body frame is:

$$\boldsymbol{V_{sp}^B} = R_z(\psi)^T \boldsymbol{V_{sp}^E} \tag{6.4}$$

where ψ is the actual yaw of the robot. Next step is to extract the yaw of the vector $\boldsymbol{V_{sp}^B}$ which is expressed in body frame by the following expression:

$$\psi_{sp}^B = atan2(V_{sp}^B[1], V_{sp}^B[0]) \tag{6.5}$$

if $\psi_{sp}^{B} > 0$ it means that the shortest path to point at the desired yaw is to rotate clockwise, otherwise the rotation is counterclockwise. Figure 6.9 shows the quantities involved in the procedure. In the case of the figure we have a negative desired yaw in earth frame but positive in body frame. That means that the shortest path is clockwise and it is easily verifiable just looking at the image.



V_e

Figure 6.9: Involved frames in rotation task.

Moreover, the yaw set point can be provided in radians or as a directional vector.

6.3.5 Executioner

The last component is the Executioner. This module implements the switching logic and it contains the definition of each nodes. It is defined by a class with one member: ExecThread. This thread is started together with the automatic thread and it runs a loop until all the tasks are performed. The position of the actual node is stored in an integer variable i in the global space. At the beginning $i = 0$, the executioner increases this variable every time an action is completed. The automatic thread reads i and looks in the list what is the actual node, from which it reads the value of position and parameters. Those value correspond to the sensory inputs of the activated behavior.

This process can be explained clearly using a state machine diagram. The machine has two states, one is called $ACTUAL$ in which the output is the actual value i and the other is $NEXT$ in which i is increased by 1, meaning that we can

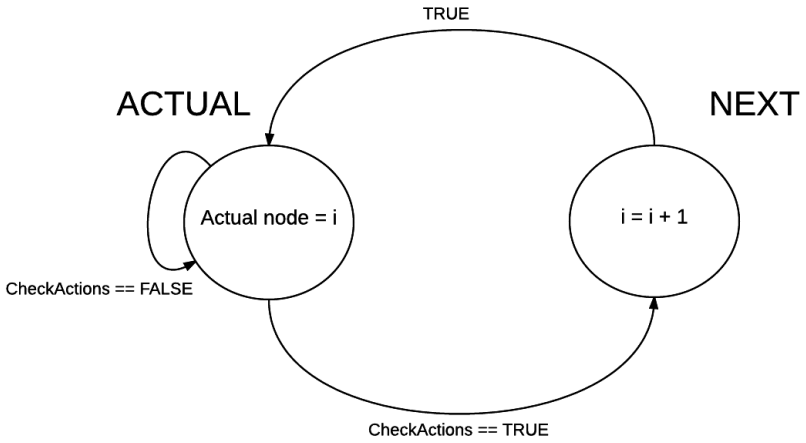


Figure 6.10: State machine diagram of the executioner module.

pass to next node. The initial state is *ACTUAL*; until the function checkActions() returns false, the state remains the same. When checkActions() returns true the state changes to *NEXT*, i is increased and then it switches back to *ACTUAL*. Those operations are done at 10 Hz. The function *checkActions()* reads the actual state of the system and, depending on the actual action, checks a number of stopping conditions (i.e. when te robot is near the target point, move task finishes. When the vertical velocity is zero and the robot is close to the ground, land task is performed.) returning ture if they are satisfied, false otherwise.

The state machine relies on boolean signals to communicate with other modules. Those signals are organized in global namespaces, one for each action. For example, the namespace *executioner::move* has two important boolean members: one is *move_sig* and the other is *move_done*. The first one correspond to the behavior trigger (see algorithm 6.3.4), this signal is set to true when the index i points to a node in which the move action is encoded, to false otherwise. Figure 6.10 explains those concepts.

Informations on actions are given in table 6.4.

6.4 Experiments and results

The first experimental stage was to test each behavior singularly, after some more complex node lists were tested successfully.

The performances of the task rotate and take off are already presented in section 5.4.2. Figure 5.9 shows the take off maneuver; as already pointed out,

Action	id (char)	parameters (double[4])
take off	't'	params[0] = final height
move	'm'	params[0] = alpha params[1] = hovering time on target
rotate	'r'	params[0] = angle valid
land	'l'	params[0] = touchdown speed params[1] = ground height
follow trajectory	'c'	

Table 6.4: Actions coding and parameters

the response is very slow at the first takeoff of the flight since the saturation of the integral on z axis, which compensates for gravity, is pretty slow. Figure 5.10 show the rotation maneuver in which a constant yaw rate is implicitly set being the yaw set point a line over time. The convergence is very good.

The move behavior is also partially tested and analyzed in section 5.4.2. Figure 5.7 and 5.7 shows the move maneuver with no scaling factor α but what happens if we introduce it? The expectation is to see the set point with the shape of a ramp depending on the scaling factor value. Four move nodes were placed in the node list (beside take off and land), each node is the vertex of a square centered in (0,0) with the edge 1.6 meters long. Image 6.11 shows the move function with the scaling term $\alpha = 1$ while figure 6.12 represent the same experiment with $\alpha = 0.6$.

Figure 6.11: Move with alpha 1

We can easily see the difference between the two experiments. The gap between the position and the set point is bigger with higher scaling factor but the

Figure 6.12: Move with alpha 0.6

maneuver is faster with more oscillations. The small oscillations of the set points are due to the fact that the error takes into account the robot position, which oscillates by definition inducing this phenomenon. From second 65 the landing procedure begins, the oscillations of the set point the set point compensation for more accuracy explained in the landing section.

Chapter 7

Landing on a mobile platform

Summary

Chapter 8

Conclusions

Write the conclusions here...

Appendix A

Extra

Write here...

References

[1] Mocap Optitrack ROS package. http://wiki.ros.org/mocap_optitrack, note = Accessed: 18-08-2015, year = 2015,. 66

[2] Gyroscopic stabilizer. The Sperry Gyroscopic Stabilizer, 1915. published on Flight magazine, available on http://www.flightglobal.com/pdfarchive/view/1915/. 4

[3] 3D Robotics store. https://store.3drobotics.com/products, 2015. Accessed: 16-08-2015. 6, 30

[4] Airware control station. http://www.airware.com/products/ground-control-software, 2015. Accessed: 18-08-2015. 12

[5] ARDrone website. http://ardrone2.parrot.com/, 2015. Accessed: 18-08-2015. 6

[6] Ardupilot Firmware. http://ardupilot.com/, 2015. Accessed: 16-08-2015. 8, 9

[7] ArduPilot Mission Planner. http://planner2.ardupilot.com/, 2015. Accessed: 18-08-2015. 12

[8] Ardupilot website. http://www.ardupilot.co.uk/, 2015. Accessed: 16-08-2015. 8

[9] Ascending Technologies. http://www.asctec.de/en/, 2015. Accessed: 18-08-2015. 6

[10] DJI website. http://www.dji.com, 2015. Accessed: 15-08-2015. 6

[11] Extended Kalman Filter for the PX4 autopilot. https://pixhawk.org/firmware/apps/attitude_estimator_ekf, 2015. Accessed: 16-08-2015. 52

[12] Flamewheel built by Lorenz Meier. https://pixhawk.org/platforms/multicopters/dji_flamewheel_450, 2015. Accessed: 19-08-2015. 6

[13] FlightPlot visualizaion tool. https://pixhawk.org/dev/flightplot, 2015. Accessed: 18-08-2015. 57

[14] Helicopter hisorical website. http://www.helicopter-history.org/, 2015. Accessed: 20-08-2015. 4

[15] IRIS, 3d Robotics. https://store.3drobotics.com/products/iris, 2015. Accessed: 15-08-2015. 17

[16] MAVLink protocol description. http://qgroundcontrol.org/mavlink/start, 2015. Accessed: 15-08-2015. 20

[17] NuttX website. http://nuttx.org/, 2015. Accessed: 16-08-2015. 19

[18] Optihub image. http://www.fontysvr.nl/mediawiki/index.php/OptiTrack_FLEX:V100R2, 2015. Accessed: 15-08-2015.

[19] OptiTrack website. http://www.optitrack.com/, 2015. Accessed: 15-08-2015. 8, 23, 24, 26

[20] PixHawk board description. https://pixhawk.org/modules/pixhawk, 2015. Accessed: 15-08-2015. 18

[21] PixHawk project at ETH. https://pixhawk.ethz.ch/, 2015. Accessed: 19-08-2015.

[22] Predator image. http://www.airforce-technology.com/projects/predator-uav/predator-uav6.html, 2015. Accessed: 20-08-2015.

[23] PX4 autopilot description. https://pixhawk.org/firmware/source_code, 2015. Accessed: 15-08-2015. 9, 19

[24] PX4 build environment. https://pixhawk.org/dev/toolchain_installation_lin, 2015. Accessed: 16-08-2015. 29

[25] PX4 Firmware Github. https://github.com/PX4/Firmware, 2015. Accessed: 16-08-2015. 9, 19

[26] QGroundControl website. http://qgroundcontrol.org/, 2015. Accessed: 19-08-2015. 12

[27] Qt libraries. http://www.qt.io/developers/, 2015. Accessed: 2-09-2015. 66

[28] qualisys website. http://www.qualisys.com/, 2015. Accessed: 16-08-2015. 7

[29] Reference frames. https://pixhawk.org/dev/know-how/frames_of_reference, 2015. Accessed: 16-08-2015. 32

[30] Ros org. http://www.ros.org/, 2015. Accessed: 16-08-2015. 66

[31] Software architectures and behavioral systems, motivation and use cases. https://msdn.microsoft.com/en-us/library/bb245656.aspx, 2015. Accessed: 2-09-2015. 64

[32] uOrb user guide. https://pixhawk.org/dev/shared_object_communication, 2015. Accessed: 18-08-2015. 47

[33] Vicon website. http://www.vicon.com/, 2015. Accessed: 18-08-2015. 8

[34] E. Altug, J. Ostrowski, and R. Mahony. Control of a quadrotor helicopter using visual feedback. *Proceedings of IEEE International Conference on Robotics and Automation*, 1, 2002. 11

[35] S. J. Andrew Zulu. A Review of Control Algorithms for Autonomous Quadrotors. *Open Journal of Applied Sciences*, pages 547–556, 2014. 10

[36] G. Antonelli, E. Cataldi, P. R. Giordano, S. Chiaverini, and A. Franchi. Experimental validation of a new adaptive control scheme for quadrotors MAVs. *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2439–2444, 2013. 11

[37] E. Baird. A universal strategy for visually guided landing. *Proceedings of the National Academy of Sciences*, pages 1–15, 2013. 14

[38] J. Blanco. A tutorial on se (3) transformation parameterizations and on-manifold optimization. Technical report, University of Malaga, Tech. Rep, 2010. 37

[39] S. Bouabdallah. Design and Control of Quadrotors With Application To Autonomous Flying. Master's thesis, ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE, 2007. 9

[40] S. Bouabdallah, a. Noth, and R. Siegwart. PID vs LQ control techniques applied to an indoor micro quadrotor. *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 3:2451–2456, 2004. 11

[41] S. Bouabdallah and R. Siegwart. Full control of a quadrotor. *2007 IEEE/RSJ International Conference on Intelligent Robots and Systems*, (1):153–158, 2007. 10

[42] H. Bouadi, M. Bouchoucha, and M. Tadjine. Sliding mode control based on backstepping approach for an UAV type-quadrotor. *World Academy of Science*, 1:1–6, 2007. 11

[43] G. De Croon and M. Ieee. Controlling Spacecraft Landings With Constantly and Exponentially Decreasing. 51(2):1241–1252, 2015. 14

[44] T. F. Delitalia. Modelling and control of 3DR IRIS Quadcopter. Unpublished, November 2014.

[45] J. Diebel. Representing attitude: Euler angles, unit quaternions, and rotation vectors. *Matrix*, 58:1–35, 2006. 38

[46] B. Erginer and E. Altug. Modeling and PD Control of a Quadrotor VTOL Vehicle. *2007 IEEE Intelligent Vehicles Symposium*, pages 894–899, 2007. 10

[47] M. Faessler, E. Mueggler, K. Schwabe, and D. Scaramuzza. A Monocular Pose Estimation System based on Infrared LEDs. *ICRA*, 2014.

[48] J. Friis, E. Nielsen, R. F. Andersen, and A. Jochumsen. Autonomous Landing on a Moving Platform. Technical report, Aalborg University, Department of electronics systems, 2009. 14, 38

[49] D. Henderson. Euler Angles, Quaternions, and Transformation Matrices. *NASA JSC Report*, 1977.

[50] B. Herisse, T. Hamel, R. Mahony, and F. X. Russotto. The landing problem of a VTOL unmanned aerial vehicle on a moving platform using optical flow. *IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS*, 28(1):1600–1605, 2010. 14

[51] G. Hoffmann, H. Huang, and S. Waslander. Quadrotor helicopter flight dynamics and control: Theory and experiment. *American Institute of Aeronautics and Astronautics*, pages 1–20, 2007. 10

[52] H. Huang, G. M. Hoffmann, S. L. Waslander, and C. J. Tomlin. Aerodynamics and control of autonomous quadrotor helicopters in aggressive maneuvering. *Proceedings - IEEE International Conference on Robotics and Automation*, pages 3277–3282, 2009. 35

[53] D. Izzo and G. D. Croon. Landing with Time-to-Contact and Ventral Optic Flow Estimates. *Journal of Guidance, Control, and Dynamics*, 35:1362–1367, 2012. 14

[54] F. Kendoul, D. Lara, I. Fantoni, and R. Lozano. Real-Time Nonlinear Embedded Control for an Autonomous Quadrotor Helicopter. *Journal of Guidance, Control, and Dynamics*, 30(4):1049–1061, 2007. 43

[55] W. Khalil. *Modeling Identification and Control of Robots*. 37, 38

[56] T. Luukkonen. Modeling and control of Quadcopter. *Independent research project in applied mathematics*, 22:1134–45, 2011. 9

[57] R. Mahony, V. Kumar, and P. Corke. Multirotor Aerial Vehicles: Modeling, Estimation, and Control of Quadrotor. *IEEE Robotics & Automation Magazine*, 19, 2012. 9, 10, 42

[58] D. Mellinger. Minimum snap trajectory generation and control for quadrotors. pages 2520–2525, 2012. 53

[59] Michael, Nathan. Quadrotor Modeling and Control. http://www.cs.cmu.edu/afs/cs.cmu.edu/academic/class/16311/www/s15/syllabus/ppp/Lec08-Control3.pdf, 2014. Accessed: 17-08-2015. 9

[60] J. F. Montgomery, A. H. Fagg, and G. a. Bekey. USC AFV-I: a behavior-based entry in the 1994 international aerial robotics competition. *IEEE expert*, 10:16–22, 1995. 13

[61] E. Mueggler, M. Faessler, F. Fontana, and D. Scaramuzza. Aerial-guided Navigation of a Ground Robot among Movable Obstacles. *SSRR Conference*, 2014.

[62] M. Vendittelli. Elective in Robotics Quadrotor Modeling, Control Problems and approaches. 9, 41

[63] H. Voos. Nonlinear Landing Control for Quadrotor UAVs. Technical report, University of Applied Sciences Ravensburg-Weingarten, Mobile Robotics Lab, December 2008. 14