MASTER'S THESIS

Developing a ROS Enabled Full Autonomous Quadrotor

Ivan Monzon
2013

Master of Science in Engineering Technology
Engineering Physics and Electrical Engineering

Luleå University of Technology
Department of Computer Science, Electrical and Space Engineering

LULEÅ UNIVERSITY OF TECHNOLOGY

Developing a ROS Enabled Full Autonomous Quadrotor

Iván Monzón

Luleå University of Technology
Department of Computer science,
Electrical and Space engineering, Control Engineering Group

27th August 2013

Abstract

The aim of this Master Thesis focuses on: a) the design and development of a quadrotor and b) on the design and development of a full ROS enabled software environment for controlling the quadrotor. ROS (Robotic Operating System) is a novel operating system, which has been fully oriented to the specific needs of the robotic platforms. The work that has been done covers various software developing aspects, such as: operating system management in different robotic platforms, the study of the various forms of programming in the ROS environment, evaluating building alternatives, the development of the interface with ROS or the practical tests with the developed aerial platform.

In more detail, initially in this thesis, a study of the ROS possibilities applied to flying robots, the development alternatives and the feasibility of integration has been done. These applications have included the aerial SLAM implementations (Simultaneous Location and Mapping) and aerial position control.

After the evaluation of the alternatives and considering the related functionality, autonomy and price, it has been considered to base the development platform on the ArduCopter platform. Although there are some examples of unmanned aerial vehicles in ROS, there is no support for this system, thus proper design and development work was necessary to make the two platforms compatible as it will be presented.

The quadrotor's hardware has been mounted on an LTU manufactured platform, made through 3D printing, and its functionality has been evaluated in real environments. Although an aluminium platform has been also evaluated and tested with less satisfactory results.

For proper operation of the whole system, a connection between the quadrotor and the ground station has been established. In this case, an alternative connection between computers (for the case of an on board computer is mounted on the aircraft) or connection between computer and ArduCopter (for the case of no on board computers) have been designed.

A series of algorithms to perform the quadrotor control autonomously has been also implemented: Navigation with way points, rotation control and altitude control. The novelty of the proposed activities is that for the first time all these control modules have been designed and developed under the ROS system and have been operated in a

networked manned from the ground station.

Finally, as it will be presented, a reader module for the adopted inertial measurement unit, currently under development by the University of Luleå (KFly), has also been developed. This device, although tested in controlled laboratory environments, has not yet became part of the quadrotor, but in the near future is expected to serve as a replacement to the on board computer.

PREFACE

This project has been done as a degree thesis to finish my studies in Industrial Engineering with a major in Industrial Automation and Robotics by the Zaragoza University (Spain). The project has been developed and presented in the Luleå University of Technology with the team of the Department of Computer science, Electrical and Space engineering; whom I thank for their support and collaboration.

It was a hard and long journey for almost six month, full of delays and problems. But now I can say it was worth it.

Therefore, I would like to thank the Erasmus project to give me the chance to go to this wonderful Swedish city, Luleå.

Finally, I would like to thank too George Nikolakopoulos, my supervisor in this endeavor, for his support and optimism.

Contents

CHAPTER 1

Introduction

1.1 Robots

1.1.1 History

Robotics has came to our world, and it has been to stay. Since Heron of Alexandria edited *Pneumatica* and *Automata* twenty two centuries ago, many things changed in our approach to the robotics and more on how robotics changed our world. At that time, Robots have been considered magic, illusions to entertain an ignorant public. Now, robotics is part of our daily life, and continuously serve to entertain, although now they target at a much more cultured public, more specialized and thirsty to be aware of what happens after each of the movements of the robots.

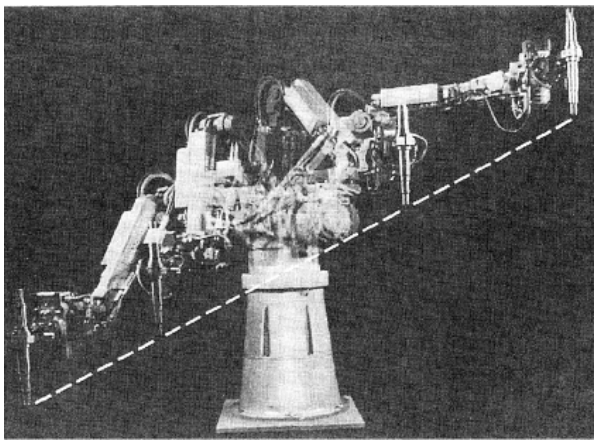

Figure 1.1: Cincinnati Milacron T3 Industrial Robot

Furthermore, it has exploited many different avenues of growth. When we began to

see that robotics was able to help us in our tasks (mid-50s), there was a revolution, in the moment we realized that we did not need to create things, but we were also able to create things to work for us.

As you can see in these few lines, robotics is a science with history, but it was asleep for a long time. Therefore, it can be stated that the evolution in robotics has been started half a century ago and from that point until now this evolution has been unstoppable.

In the early stages of this evolution, progress was mechanical and hydraulic mostly. But after about 18 years, this began to change. Cincinnati Milacron created the first computer-controlled robot (in Figure 1.1), and at this time robotics began to grow without boundaries.

In parallel to the robotic evolution, the significant advances in the field of computer science has launched robotics to a fantastic world to discover. Now it was possible to program much more complex tasks than before in minimum space, while it was not necessary to create unitask robots, as it was possible even to teach them to work and reprogram operations as often as needed.

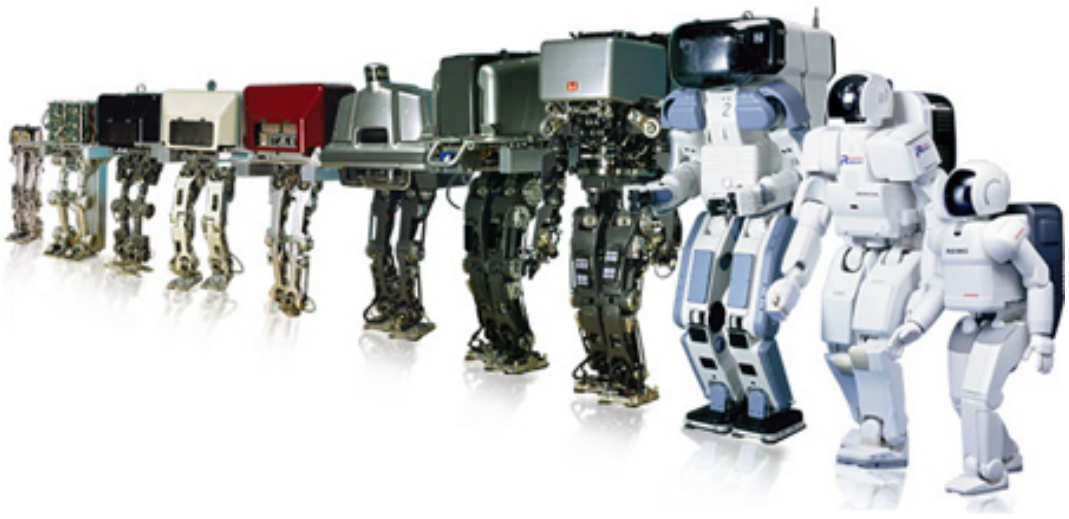

Figure 1.2: Honda ASIMO and its predecessors

From that day until today the history of robotics is already known. Broadly speaking, there are industrial robots in all fairly large factories in the developed world. Painting robots, welding robots, cutting robots, etc. (such as PUMA [1] or Kuka [2]). It should be mentioned that is only the part that was born and evolved with those first industrial robots in the mid 20th century.

At the same time we began to investigate about intelligence and autonomous robots.

In 1987 DARPA showed that they could create an autonomous vehicle [3] that using laser and computer vision creates a path to move between two points. In 1997 the first computer was able to beat a chess master in a game (Kasparov vs. Deep Blue [4]). Asimo (in Figure 1.2) [5], the Honda humanoid robot is created in 2000.

1.1.2 Robots in service of man

From this historical point and until the beginning of the new century, the XXI century, a significant change in the way of thinking towards robots took place. The robotics ceases to be a exclusive domain of research and it is becoming something commercial, something that can be close to the ordinary people.

In the world of toys it begins to arise a variety of robotic pets with basic interaction capabilities. On medical fields, wheelchairs managed with the mind, rehabilitation robots, prostheses controlled by nerve endings or telesurgery (the da Vinci Surgical System).

Figure 1.3: Roomba, the iRobot cleaning robot

Those robots can be found also in industrial tasks, inspection robots, etc.

In domestic areas, kitchens that run unattended, automatic cleaners (Roomba [6], in Figure 1.3); systems that control the temperature, humidity, or light in the house; refrigerators that detect a shortage of some product and it automatically ask the supermarket; inspection robots to check and fix the pipes; or home theatre systems that adapt the sound to the room geometry and to the playback features.

In parallel to these activities, the research efforts have focused in two branches: a) the military, and b) the civilian. In military matters, DARPA has been since its inception the reference entity. In 1994 they developed the MQ-1 Predator [7], one of the first unmanned aerial vehicles. Also emerged from their laboratories the most current Boeing X-45 (2004)

[8]. Currently, they are developing the ACTUV (autonomous military vessel) [9], the BigDog and the LS3 (4-legged autonomous robots) [10], the EATR (an autonomous vehicle which is able to run on any biomass) and the future Vulture (a UAV able to be in the air for five years).

Even with a much smaller funding, civil investigations have been no less. The European project AWARE [11] has designed autonomous unmanned aerial robots, distributed and able to cooperate with each other for intervention in disasters and fires. Researchers at IFM-GEOMAR have created the largest fleet of underwater exploration robots in Europe [12]. The Google driverless car is capable of running autonomously both city and road [13], while brands such as Volvo, BMW and Audi also have similar projects underway.

A very interesting topic that significantly added intelligence in the field of autonomous robots has been the SLAM (Simultaneous Localization and Mapping) concept. SLAM addresses the problem of robot to traverse a previously unknown terrain. The robot makes a relative observation on the surrounding environment to reconstruct a map of that environment so it can create a motion path to follow. If that map is available, the robot uses the SLAM information to position itself properly in it. Robots of all kinds (as you can see in Figure 1.4) make use of this technique for orientation.

(a) Undersea (b) Underground (c) Other planets

Figure 1.4: Robots using SLAM

1.2 UAVs (Unmanned Aerial Vehicle)

Among all this astonishments range of robotic possibilities, the research that concerns us here is specifically focused on the problem of autonomous flying robots.

UAVs, or unmanned aerial vehicles, have been in our world since the First World War, but at that time they were very focused on military environments. Then, UAVs have been used as a safer way to fly (at least for the side that controlled it), since no pilot was required. Moreover, for this reason, you could make smaller aircraft, more aerodynamic, more agile, and with a less fuel consumption.

A UAV, with greater or lesser degree of intelligence, can communicate with the controller or the base station and transmit: the image data (thermal, optical, etc..), information regarding its status, the position, the airspeed, the heading, the altitude or any parameter of its telemetry. Usually, these vehicles are armed with systems that in the case of failure of any of the on board components or program, are able to take corrective actions and/or alert the operator. In the most intelligent cases, they are able to take "imaginative" actions if they tackle a unexpected problem.

1.2.1 Types

As in every field, UAVs can be sorted using many features. Size, range, flying principle or utility would be the most successful.

The Figure 1.5 shows some of the current UAVs. From cheap and small toys that you can buy anywhere as the RC planes; to big military projects such as the Predator. They can be remotely piloted, as the Bell Eagle Eye [14], or autonomous, as the AAI shadow [15].

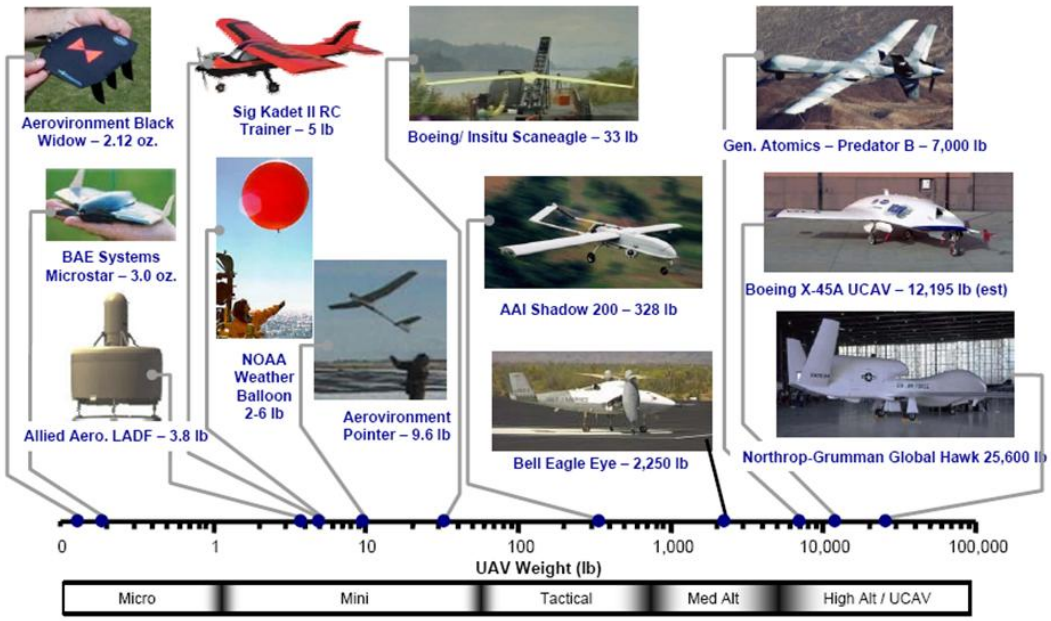

Figure 1.5: Diferents commercial UAVs [16]

Focusing on the low cost UAVs, in Figure 1.6 we can see a classification of these machines by the flying principle.

Fixed wing UAVs have the advantage of a balanced consumption; the simple mechanism; and the stealth operation, because they can fly without mobile parts and gliding,

while the mobility is not too good. They are good for flying from one place to another, but not to much more.

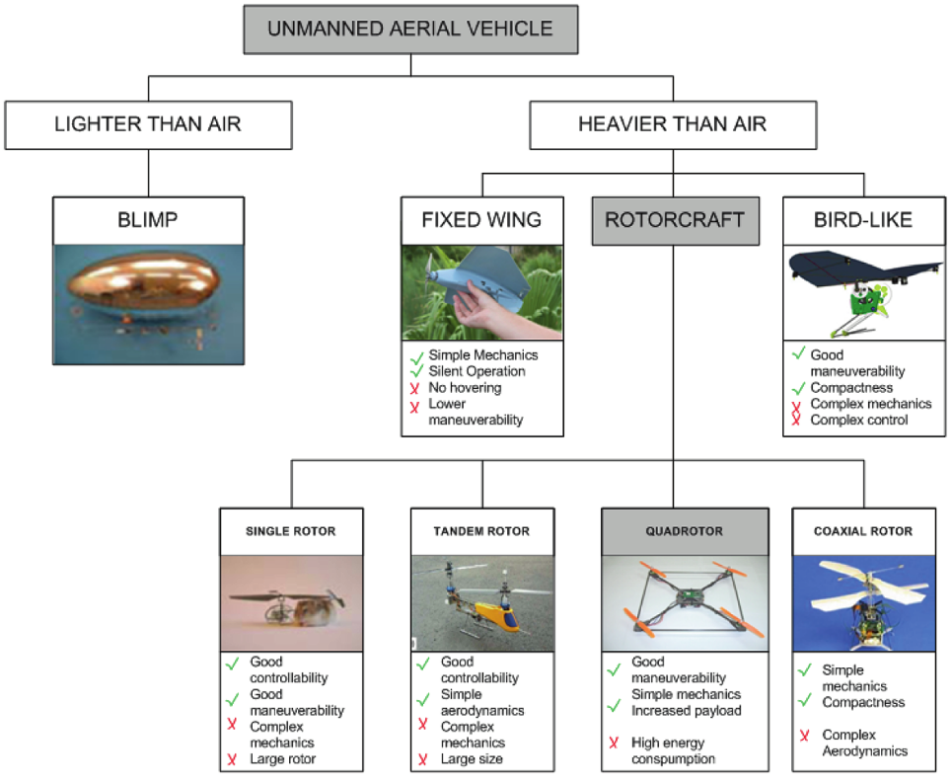

Figure 1.6: Types of UAVs [17]

Rotorcraft UAVs usually have a higher consumption and complex mechanics but they have higher handling capabilities. They can operate in inhospitable places, in small spaces and complex paths.

In this group, the multi rotor helicopter is a special member. It has some problems with the energy consumption as its batteries do not last long, but it outperforms almost every other UAV on manoeuvrability, simplicity of mechanics and even they are easier to drive.

From now, we are going to focus in the autonomous and low cost vehicles for its high growth potential and, more specifically, in autonomous multi rotor helicopters. They can make decisions faster than humans and most important, if they lose the ground connection, they can continue working.

The evolution continues with the size, with small flying robots, more precise jobs (as in [18]) can be done, go through smaller paths or be more stealth. Also a cooperation robot matrix is possible with small size robots.

For example, if we consider a SLAM UAV, a smaller one could perform lower flights and could achieve a really good world reconstruction, reconstructions that before you had to make by hand (or terrestrial vehicles) with cameras. In addition, with them, you can have a bigger z-dimension that helps us to reconstruct high buildings, mountains, trees, etc. Nowadays, you can mix an aerial reconstruction with a ground one with the same vehicle, thereby reducing time and money and making it more accurate.

Figure 1.7: Quadrotor

1.3 Quadrotors

For some years now, one new kind of UAV has begun to have a lot of attention, and well-deserved recognition, the quadrotor. Until that time, UAVs were heavier and not enough agile than it should be (helicopters [19] and [20]).

A quadrotor has a really good stability in the air (by assuming on board good sensors). It can keep still in the air as a helicopter, but it also can do sharp turns or fly upside down without any problem, and it is lighter than helicopters and planes.

A quadrotor, also called a quadrotor helicopter or quadcopter, is a multicopter that is lifted and propelled by four rotors. As you can see in Figure 1.7, it is small, between 10 cm and 150 cm. The bigger the helicopter is, the more autonomy may have, because if the propeller is bigger it can rotate slower to lift and the efficiency is better, while the smaller the size is, the more agile it is.

A quadrotor is a stable aircraft, therefore, it can be used indoors, but it can also be a dangerous machine, because it is fast, and it can turn and change direction quickly. Thus, losing control of the quadrotor can destroy itself or its environment or harm someone.

In our lab, and to test the code in a safe environment, a vertical wire with bumpers (in Figure 1.8) has been built. The quadrotor is hooked up on it and it can rotate and

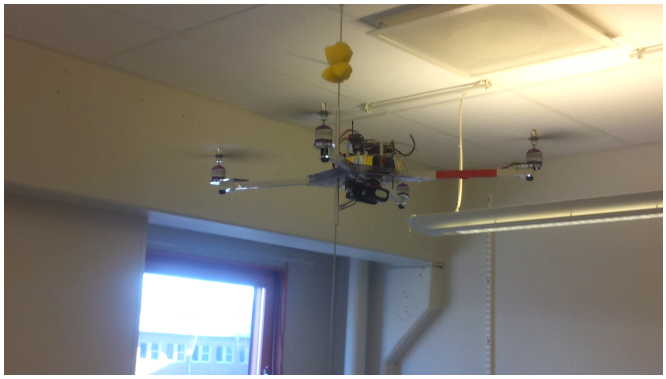

Figure 1.8: LTU lab with a safety system for quadrotors

lift. There has been also a safety net between the quadrotor and the operators.

1.4 The Problem

The aim of this Master Thesis is to develop a fully ROS enabled quadrotor and experimentally demonstrate the capabilities of the platform on a realistic flight scenarios, such as attitude and altitude stabilisation and position hold. ROS (Robotic Operating System) is an open source distributed robotics platform designed to accelerate robotic research and development by supporting reusable and modular components able to implement a variety of low and high level functionalities, while during the last 3 years it has received significant attention from the scientific community, as the most prestigious research groups in the area of robotics are putting efforts in this field.

Currently, there are already some existing ROS enabled software implementations for quadrotors but these software implementations are platform depended and they aim mainly in performing a tele-operation task, which means that a user should always fly the quadrotor from a distance. The results of this Master Thesis will focus in a novel approach, within the ROS field, where the aim is to create a quadrotor being able to perform autonomous tasks, such as: attitude and altitude stabilisation in a fully ROS enabled environment. The performance of the proposed software architecture will be evaluated in both simulations and extended experimental tests. The main contribution of this Thesis will be in the design and implementation of ROS enabled software control modules, while covering various software developing aspects, such as: operating system management, forms of programming in the ROS environment, software building alternatives, ROS interfaces and finally various practical software implementation issues with the developed aerial platform.

1.5 Goals

The general goal for this Mater Thesis is to extend the vast amount of ROS libraries and the novel approach in designing robotic applications, quadrotors in this case, by creating a proper modular software architecture that will allow the autonomous operation of a quadrotor, like attitude and altitude stabilisation, without the need of human interference. The control modules will be based on the classical PID control architecture, but the novelty of the current implementation will be that all these control modules will be developed to be ROS enabled and will be synchronised with the rest of the ROS operating modules for achieving proper flight performance.

To achieve the previous goal the current Thesis has been divided in a number of related activities that include the development of the experimental platform, the design and development of the ROS enabled software architecture and the final extended experimental verifications in real environments.

To be able to design a full ROS enabled software architecture, a broad knowledge of the ROS software is considered necessary, which should include both the understanding of its performance and handling as well as the knowledge to program in both C++ and Python, and the ability to modify and adapt existing modules, create new ones and synchronise the whole software architecture of the system. More analytically the goals of this Master Thesis will be the following ones:

1. Literature study in the area of ROS enabled robots and on ROS documentation and functionalities, especially for the case of Unmanned Aerial Vehicles (UAV).

2. Experimental quadrotor platform customization and adaptation to ROS needs.

3. Programming and synthesizing ROS functionalities and system architecture for allowing the autonomous flight (such as attitude and altitude stabilisation), including sensor integration and motor control.

4. Developing ROS enabled control modules, such as On/Off, P and PID controllers.

5. Comparison of ROS modules and experimental evaluation of the quadrotor's flight performance.

CHAPTER 2

ROS

2.1 Robot Operating System

Since the advent of the Internet, developer communities have become an essential part of scientific discovery. Now, it is not necessary to be a professional researcher to get into fields, until recently restricted to professional areas.

Now, just as a hobby, you can learn and research in any imaginable field. Instructables or even Youtube make this task easier. But if we break away from the general services, there is a world of possibilities.

And that is where ROS comes in. ROS allows people all over the planet, people with any level of knowledge, to insert into the robotics world. ROS is not just an operating system, is a complete ecosystem, nourished code and tools that help us to give life to our creations. But above all, it is a community. A system such as ROS helps the spread of ideas, the free use of knowledge to achieve faster progress in this technological world.

2.1.1 Definition

Robot Operating System, or ROS, as it is known, is an open source (distributed under the terms of the BSD license) software framework for robot software development. It provides libraries and tools to help software developers create robot applications. Moreover, it provides hardware abstraction, device drivers, libraries, visualizers, message-passing, package management, and more functionalities for delivering fast integrated solutions.

In general "A system built using ROS consists of a number of processes, potentially on a number of different hosts, connected at runtime in a peer-to-peer topology." [21]

ROS framework is also specifically developed to reuse drivers and code from another robots. This functionality is being achieved by encapsulating the code in standalone libraries that have no dependencies on ROS.

In some cases, ROS can be used only to show configuration options and to route data into and out of the respective software, with as little wrapping or patching as possible.

At the end of the day, the most important reason for using ROS and not other framework is that it is part of a collaborative environment. Due to the big amount of robots and artificial intelligence software that have been developed, the collaboration between researchers and universities is essential. To support that development ROS provides a package system; the ROS package which is a directory containing an XML file to describe the package and the related dependencies.

At the time of writing, several thousand ROS packages exist as open source on the Internet, and more than fifteen hundred are directly exposed in `http://www.ros.org/browse`.

2.1.2 Robots running ROS

Figure 2.1: Modlab's CKBots

Since 2007, when ROS was initially released, there have been one hundred different ROS enabled robots in the market. This covers a considerable number of types of robots, mobile robots, manipulators, autonomous cars, humanoids, UAVs, AUVs, UWVs and even some uncategorizable robots as CKBot (in Figure 2.1). The complete list of ROS enabled robots can be located here: `http://www.ros.org/wiki/Robots` (Some of them can be seen in Figure 2.2).

This allow us to use this platform for example in a iRoomba or in a Lego NXT without changing nothing in the code, so that you can focus more in new developments, without the need for time-consuming initial settings.

Figure 2.2: ROS enable robots

In the field of quadrotor there has not been much content in ROS servers with which we could help. Right now, there are 5 universities fully involved in this field, as it will be presented in the sequel.

On one hand, the CCNY Robotics Lab in New York is developing along with AscTec the Pelican and the Hummingbird, which includes the complete software packages in the repositories of ROS. The University of Pennsylvania has a long list of experiments with Penn Quadrotors, including aerobatics flights or light performances with quadrotors swarms.

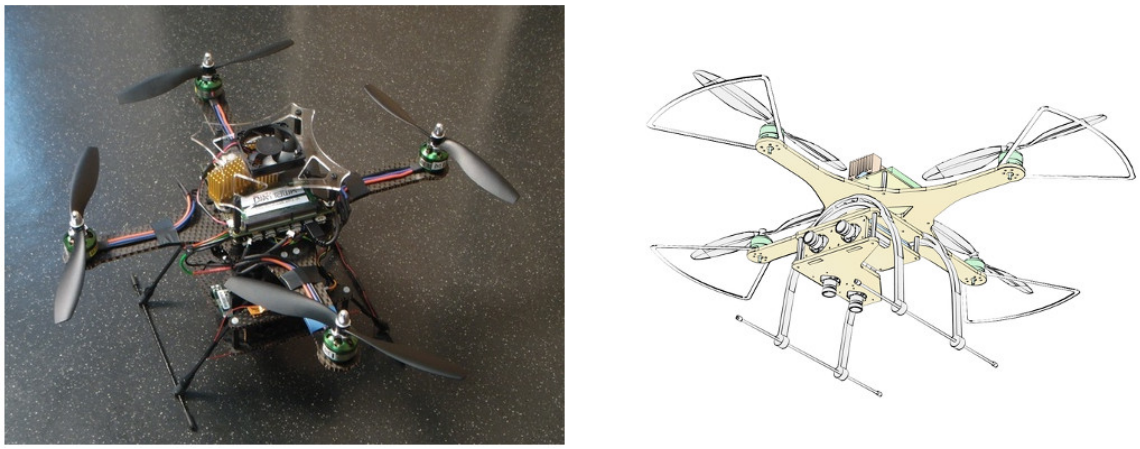

Figure 2.3: PixHawk Cheetah BRAVO

PixHawk Cheetah BRAVO (in Figure 2.3) is being developed from the ETH Zurich, it uses a stereo camera system to be able to perform indoor navigation. The Autonomy Lab, at the Simon Fraser University in Canada, is developing a ROS-based driver for the Parrot AR.Drone (in Figure 2.4), one of the few commercial quadrotors. Finally the project STARMAC (in Figure 2.5) by Stanford University aims to explore cooperation in a multi-vehicle system with a multi-agent controller.

Figure 2.4: Parrot AR.Drone

In every case, except with PixHawk, specific hardware is used, and in some cases hardware built for the experiment, such as low level boards or IMUs. This restricts the ROS utilization in specific hardware, and thus in this thesis a more broader and hardware independent ROS development approach has been investigated.

2.1.3 Nomenclature

Everything on the ROS system is based on nodes, messages, topics and services. This allows us to have a simple code structure and to follow a schematic way of work.

The nodes

The *nodes* are processes that execute a specific task. It is like a software module, while nodes can use the *topics* to communicate each other and use the services to do some simple and external operations. The nodes usually manage low complexity tasks. One moves the motors, another does the SLAM, another handles the laser, etc.

Figure 2.5: One of the STARMAC vehicles in flight

It is possible to see the nodes in a graphical representation easily with the *rxplot* command, as for example it has been presented in Figure 2.6 for the case of *sr_hand* program, which is used to access and experiment with Shadow Robot's hardware, where the *shadowhand* shares data with the logout ROS node (*rosout*).

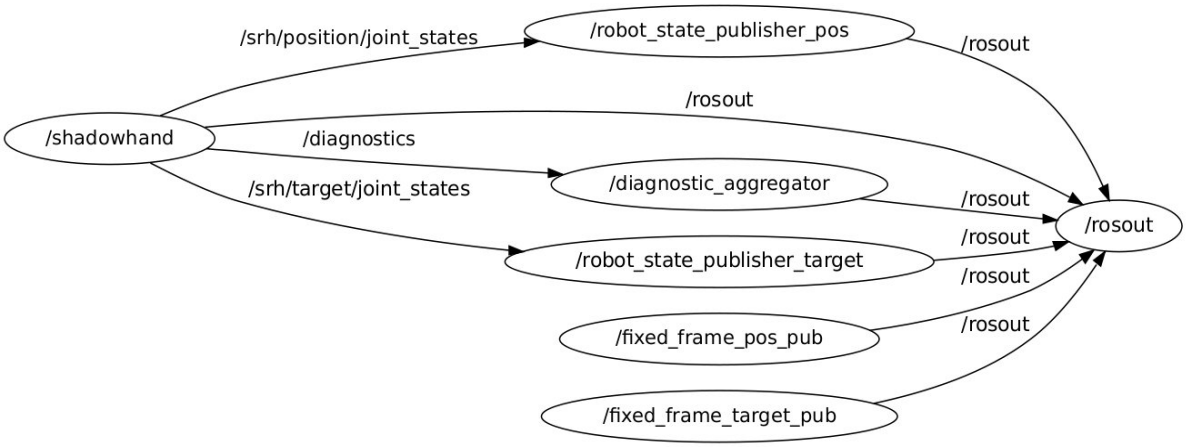

Figure 2.6: Graphical nodes representation

Topics

Topics are the places which nodes use to exchange *messages*. In Figure 2.6 the name on the arrows are the exchanged messages.

Every device that wants to be public (or that wants to publish something) has to create a topic. This topic behaves like a mailbox; when a device has a new sensor input, it sends it to the mailbox. On the other hand, each program that wants to know what happens with some device has to be subscribed to this topic (the mailbox). In this way, if the topic receives new data, it sends a signal to their subscribers; and they actualize their values.

This system of publisher-subscriber is not enough for some tasks than need a feedback, and for theses specific tasks the *service* functionality should be utilized. For that tasks is the **service**.

2.1.4 Applications, sensors and more

Very much researches use the open source system to share their publications and their programs. This behavior can be exploited only if there is a system to reuse and share the code. And ROS is this system. The way in which ROS is based helps to create a file sharing system. With ROS is possible to create an independent code to move an arm or to read a sensor. This code could be used for any robot that has to read a sensor or that has a similar arm.

ROS dependencies are a large ecosystem with several software packages.

In this place it is possible to find both, whole systems to move a given robot and drivers for specific applications, sensors or actuators. Some of these useful "whole system" packages are:

- **Shared Control for a Quadrotor Platform**, which is used to facilitate inspection of vertical structures using Quadrotors. It uses simple controls related to the structure in order to simplify the operators commands.
- **AscTec drivers** provide autopilot tools to manage a UAV.
- **Kinton apps**, which uses the Pelican Quadrotor from AscTec to the project ARCAS (Aerial Robotics Cooperative Assembly System). In this project, they use SLAM and recognition techniques for path planning and cooperative aerial manipulation.
- **Starmac ROS package** is the software used in the STARMAC project.
- **Hector packages**, which will be explained later in the Section 2.3.

Moreover, there are other packages that provide just one, or some, functionalities as:

- **PR2 Teleop**. Although PR2 is a complete software package for one robot, individual packets that make up its stack are quite useful and easy to use separately. PR2 Teleop is a package that provides some tools to control the *cmd_vel* topic, which is generic topic used by ROS to control the robot speed with 6 parameters (3 linear speeds and 3 angular speeds), from a joystick, a game pad or just a keyboard. It is also easily reprogrammable.

- **Humanoid localization** can be used on UAVs for localization of objects in a known map in order to avoid obstacles in defined routes.

- **Quadrotor Altitude Control and Obstacle Avoidance** is a software that makes use of a Kinect camera on board to perform its task.

- **Laser Drivers** is a stack that contains drivers for laser range finders, including Hokuyo SCIP 2.0-compliant and SICK models. It publishes the *scan* topic, which it will be utilized in Section 3.2.2.

- **USB Cam** is a stack that provides drivers for most standard USB camera. There are many of these generic drivers to fit perfectly with a camera system.

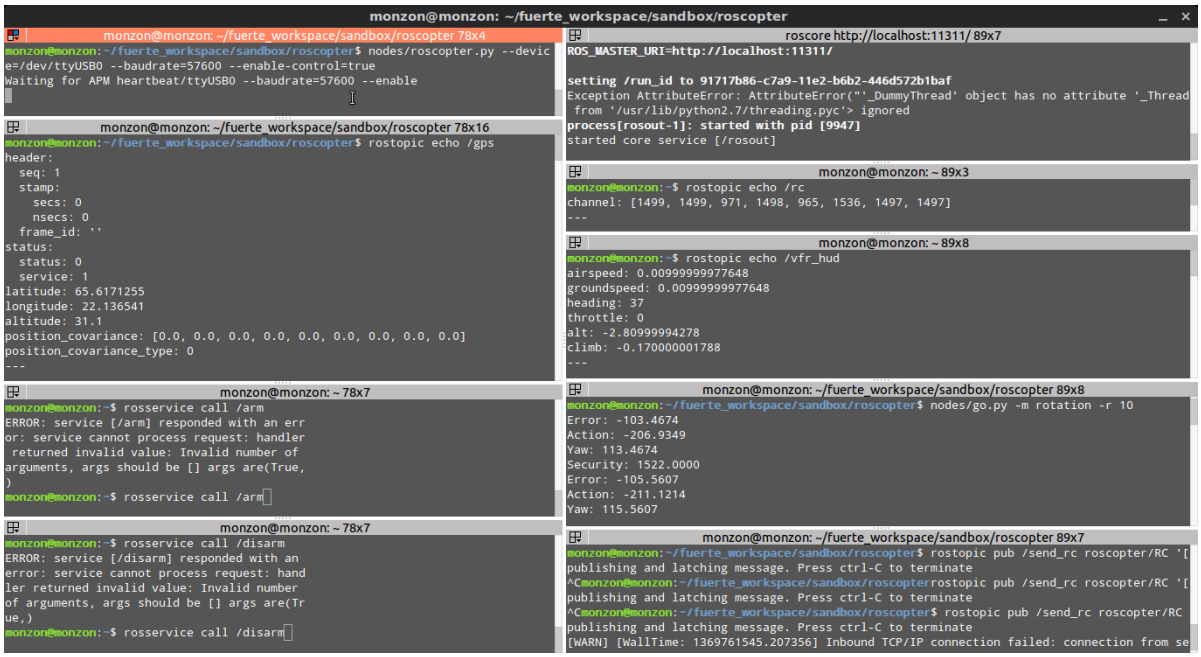

Figure 2.7: ROS terminal running several programs at the same time

In general there are many drivers and packages for a variety of applications, and thus a detailed search needs initially to be performed `http://www.ros.org/`

Since one of the advantages of ROS is its ability to launch many nodes in parallel (as you can see in Figure 2.7), you can launch these packages to control a specific device of your robot. After that you have to launch your own code collecting all the topics you need to operate the robot.

In the example of Figure 2.7, in the first row, there is a node with RosCopter [22] (connecting the ROS system with ArduCopter), there is a terminal window to read the */gps* parameter (rostopic echo /gps), the other two terminal windows have services to start or finish the service */arm*. In the second row, there are a *roscore* (in order to start the ROS system), two topic readers (*/rc* and */vfr_hub*), one python program (sending rotation commands) and one topic publisher (overwriting the rc channel).

2.1.5 Setup

For setting up ROS, the first thing to do is to stabilize a connection between machines. Then, master and host name have to be set in order to each machine know their role.

In order to be possible to utilize ROS, a *roscore* has to be started. This is a collection of nodes and programs that are needed for a ROS-based system. It also will start up a ROS master, that will manage the links between nodes monitoring publishers and subscribers to topics and services. Once these nodes are located each other, they communicate mutually peer-to-peer. Moreover, a parameter server will be utilized, that will be a share space to store and recover parameters at runtime, and a *rosout* logging node.

In cases where the ROS program runs over several machines, you should start a *roscore* just in one of them, which will be in the master machine.

Once the system is running, it is possible to launch a program, node, script or whatever in needed.

2.2 First tests to understand the platform

The first ROS learning test has been carried out with a **TurtleBot** (in Figure 2.8), this is a ground robot, Roomba-based, that thanks to a Kinect camera can sense and interact with the environment. This robot mounts ROS on a laptop located on it. Other Ubuntu laptop is used as a workstation machine. The robot is used to understand how the ROS nodes system works and to start working with a ground robot, always simpler in learning tasks.

The first step is to establish communication between both machines (the workstation and the robot) in order to create a network in the whole ROS system. For this, it is used the ***slattach*** command to create a network point with the serial port that the XBee is using.

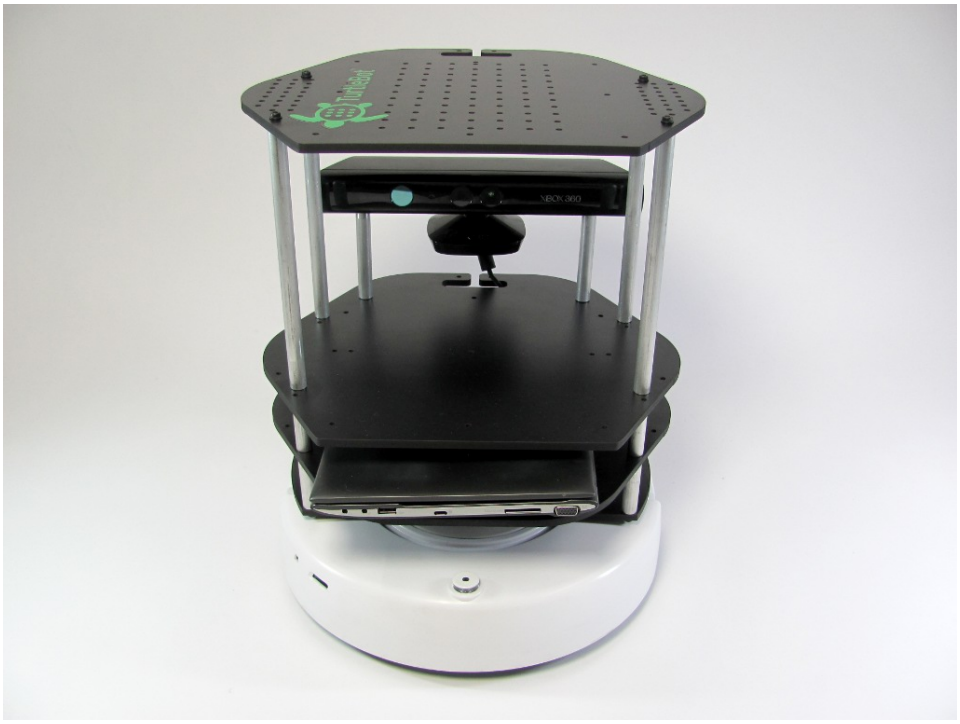

Figure 2.8: TurtleBot v1.5

```
sudo slattach -s 57600 /dev/ttyUSB0
```

Then, it is created a connection peer-to-peer between the IPs as:

```
sudo ifconfig sl0 10.0.0.1 pointopoint 10.0.0.2 netmask 255.255.255.0
```

Of course, the port in the first command have to be the same port where the XBee has been connected and the IPs in the second command have to be, first, the machine where the command has been written and, second, the other machine. These commands have to be executed in both machines.

The baudrate selected in the first command can be chosen by the operator (between 1200 and 115200, even more if a patched *slattach* command is used). The only limitation is that the rate has to be the same in both ZigBee radios. In this case, and because there aren't any problems with the batteries, the higher rate without problems (57600) will be set.

All these processes can be set automatically, as it follows, if some of the PCs haven't screen writing this code in the */etc/network/interfaces*, then the network interface will be set up each time that you start the quadrotor.

```
auto sl0
iface sl0 inet static address 10.0.0.2
pointopoint 10.0.0.1
netmask 255.255.255.0 mtu 512 pre-up /usr/local/bin/slattach -s 57600 -p slip /dev/
    ttyUSB0 & pre-up sleep 1 post-down kill `cat /var/lock/LCK..ttyUSB0`
```

When the network is properly configured it is possible to use the ***ssh*** command to use the terminal on the other machine.

```
ssh machine-name@machine-IP
```

The second step is to configure the master and the host name in every machine. Then you can open an instance of *roscore* in the master, start the services that you need and start the programs.

The first experimentation has been performed by evaluating the SLAM code, while the commands have been:

```
Terminal 1 (master):
roscore

Terminal 2 (bot):
sudo service turtlebot start

Terminal 3 (master):
rosrun turtlebot_dashboard turtlebot_dashboard&

Terminal 2 (bot):
roslaunch turtlebot_navigation gmapping_demo.launch

Terminal 3 (master):
rosrun rviz rviz -d `rospack find turtlebot_navigation`/nav_rviz.vcg

Terminal 4 (bot):
roslaunch turtlebot_teleop keyboard_teleop.launch

Move in the SLAM area

Terminal 4 (bot): (first kill the process with Ctrl+C)
rosrun map_server map_saver -f /tmp/my_map
```

If you try to do a lot of turns, quick acceleration and intense breaking the odometry losses its location and the results are presented in the next SLAM map (lab room map) in Figure 2.9.

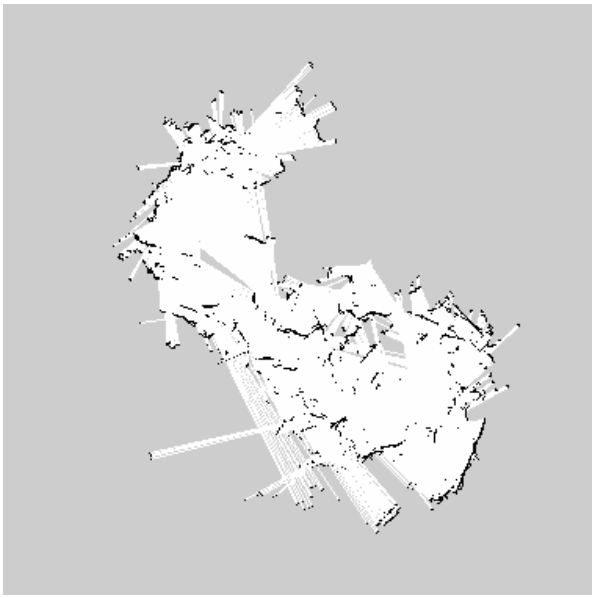

Figure 2.9: Bad map created by TurtleBot

In case that the previous experimentation is being repeated by driving the robot in a straight line, better results can be achieved and with a greater accuracy, as it can be presented in Figure 2.10 which is a corridor map at LTU.

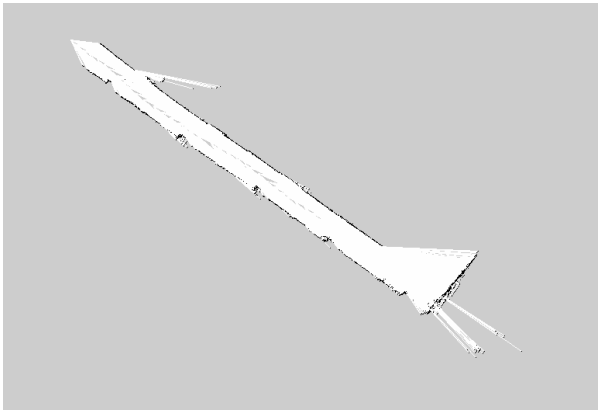

Figure 2.10: Good map created by TurtleBot

2.3 Evaluating ROS: Hector Quadrotor package

Hector [23] is a collection of ROS stacks (a **stack** is a collection of packages that provides a functionality) originally developed by the Technische Universität Darmstadt (Germany). These stacks supply several tools to simulate or to interact with robots. SLAM, location or modelling are some of them.

In this collection, the hector_quadrotor stack is going to be utilized with that tool, it is going to be able to moderate (with the physical parameters) a 3D quadrotor model (presented in Figure 2.11) in Gazebo and to have a 3D reconstruction of the world with a SLAM tool.

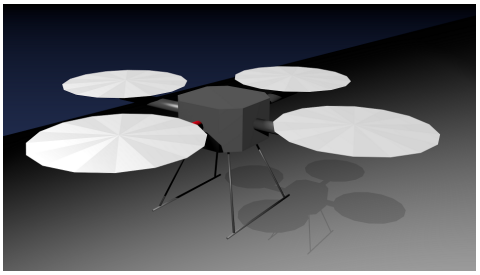

Figure 2.11: Hector quadrotor render

In Figure 2.12 you can see the Gazebo representation (on the left in the figure), as well as the world representation provides by Rviz (on the right in the figure). Gazebo is a multi-robot simulator which generates both realistic sensor feedback and physically plausible interactions between objects (it includes an accurate simulation of rigid-body physics). While Rviz is a 3D visualization environment that is part of the ROS visualization stack. On this figure it has been also presented a vision range representation (the cone on the left).

For utilizing the Hector platform, the first thing to do is to start the world.

```
roslaunch hector_quadrotor_demo indoor_slam_gazebo.launch
```

Then, and thanks to the easy access and interoperability between nodes, it is possible to use private code to take the control using the topic "*cmd_vel*" to control the quadrotor speed.

```
rosrun hector_quadrotor_controller simulation
```

In a first stage, and through that platform, a simulator has been developed to try the produced controllers and for a better understanding of the quadrotor behaviour.

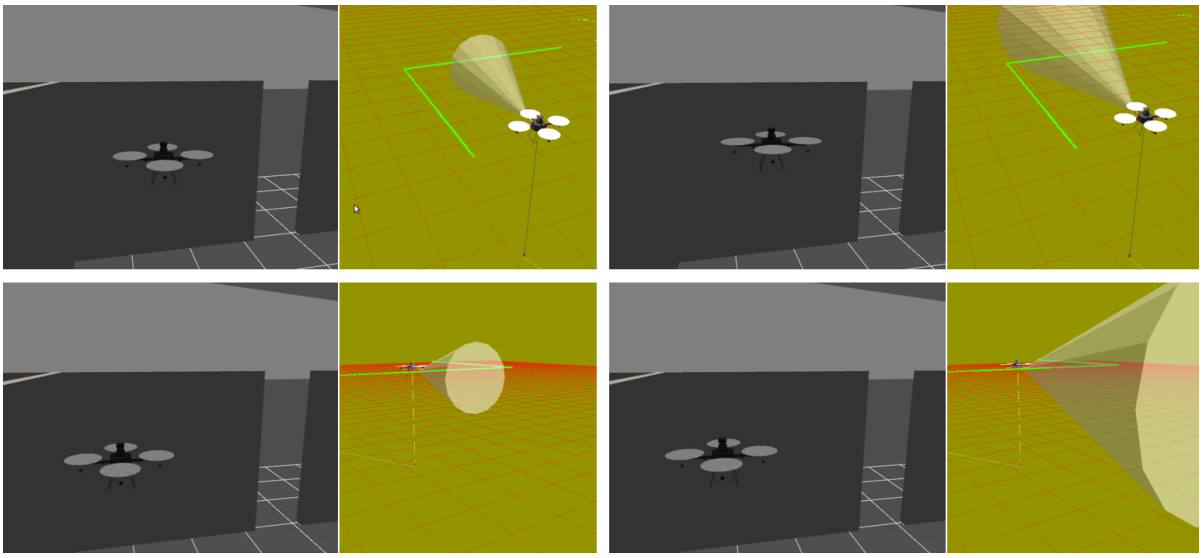

Figure 2.12: Hector quadrotor simulation

For this purpose, a control code has been developed to carry out the way point navigation and all the positioning codes. With this code you have to set some goals (as many as are needed), the precision and the speed. You can edit the proportional, integrative and derivative constants too. For the moment, all of these parameters needs to be set in the code file (cpp) and a *rosmake* needs to be performed. It is also possible to modify every parameter at running time (but then it is not possible to modify the number of goals) when the program is being executed with this structure.

```
1 rosrun package program parameter:=new_parameter
```

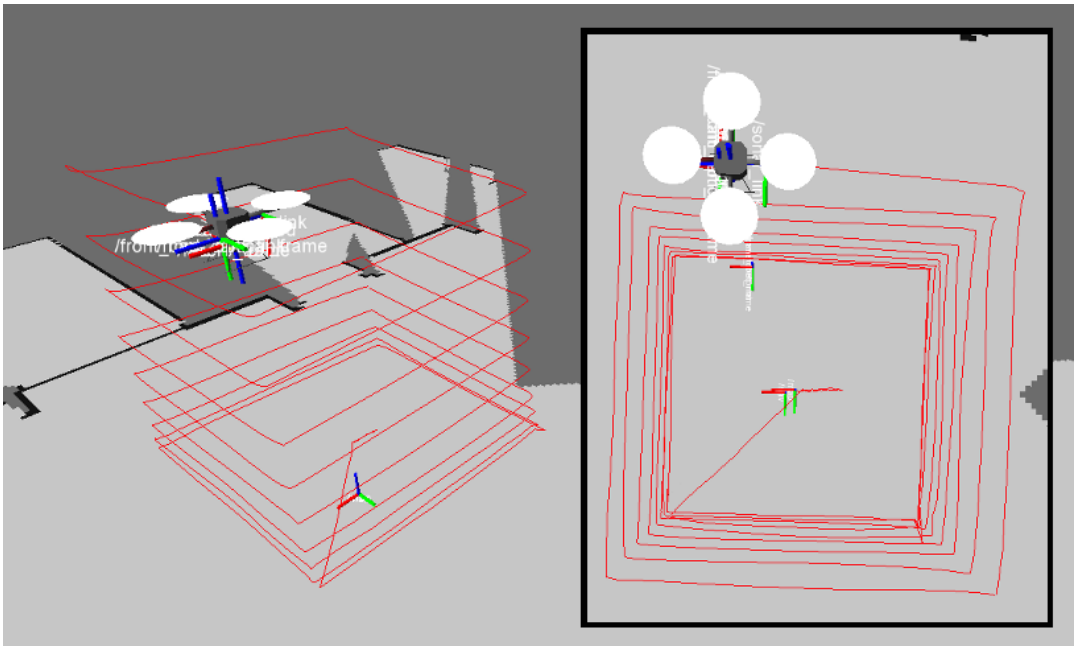

Figure 2.13: Hector quadrotor simulation

This controller can move the quadrotor in X, Y or Z speed direction, it also controls that orientation to be always stable and North. For performing this simulation, the complete code (in C++) has been included in the Appendix 1.2.1.

In Figure 2.13 the simulated quadrotor following an almost perfect cube (the red line is the path followed) is being presented. For this goal, the PID constants to have a stable system have been set as the equation 2.1 shows.

$$\kappa_p = 0.5 \qquad \kappa_i = 0.0002 \qquad \kappa_d = 0.00005 \tag{2.1}$$

The next simulation test was to try to control the simulator movement with the real quadrotor while performing true interoperability between platforms and networks.

This simulator took the Euler angle outputs directly from the experimental quadrotor platform to show the same movement in the virtual one. The Z axis can be rotated to rotate in the simulation, the X axis to advance in Y direction or Y axis to advance in X direction. Just as it would move if the quadrotor motors roll, pitch or yaw in these directions.

In order to achieve this we need to use a controller to move the virtual quadrotor from the original position (the virtual position) to the goal position (the real position) similar as before.

For these tests,the experimental frame has been utilized (in Figure 2.14) and an IMU, the k-Fly, which has been designed and printed by Emil Fresk. This frame has been printed in our lab with a handmade 3D printer, and it has been tested its strength to axial tension and bending.

The code (in C++) for this second simulation can be located in Appendix 1.2.2.

Figure 2.14: Prototype quadrotor frame

CHAPTER 3

Building the Quadrotor

3.1 Hardware

3.1.1 ArduCopter

ArduCopter (presented in Figure 3.1) is an easy to setup and easy to fly platform for multirotors and helicopters, it is a basic RC multicopter which can be found in the today market. Thanks to Arduino and the great community that both have it is possible to find lots of modifications to the platform which can be easily adapted to specific applications.

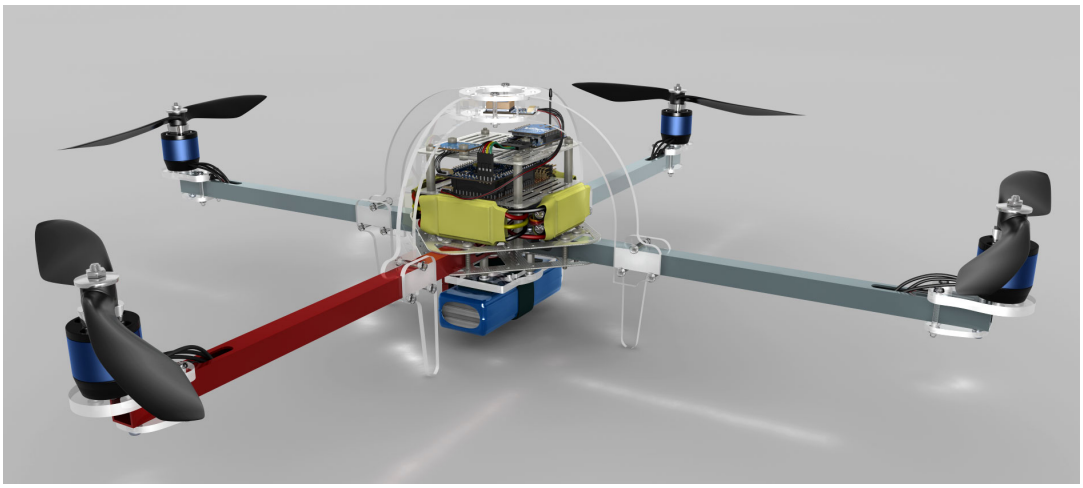

Figure 3.1: Arducopter quadrotor

ArduCopter is mainly a modified Arduino board (ArduPilot Mega) with an IMU board

attached. The IMU (inertial measurement unit), that provides 3 accelerometer and 3 gyroscope directions, also has an attached magnetometer (with 3 degrees of freedom) and an XBee unit to support wireless communication. Moreover, this board can provide a quaternion [24] system with which you can calculate an Euler angle [25] system and measure how the quadrotor is positioned. A GPS attached also to the ArduPilot Mega board is used to estimate the 3D position of the quadrotor. Finally, there is a power distribution board controlled by the ArduPilot Mega board, this low level board controls the ESC (Electronic Speed Controller) of the motors. In Figure 3.2 there is a scheme of this parts.

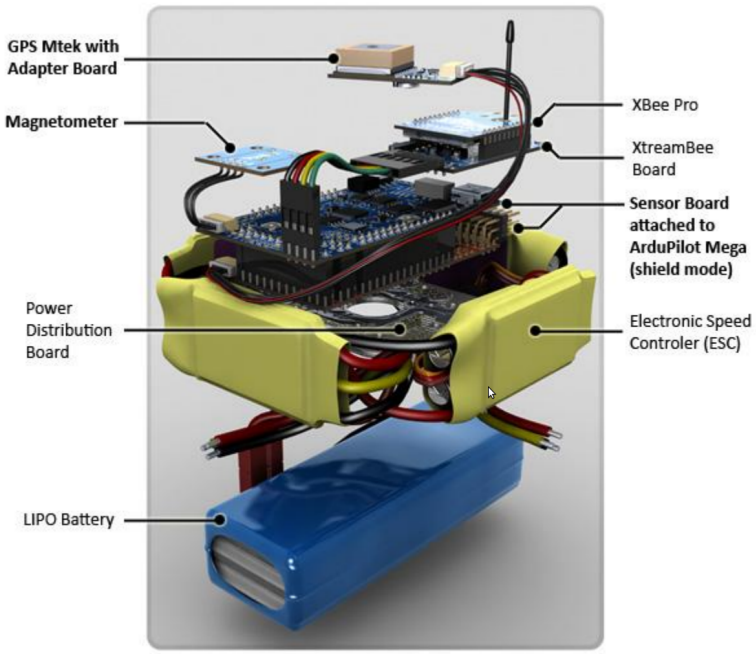

Figure 3.2: ArduCopter components

One of the most important parts in the quadrotor are the propellers, how they are mounted and how they can give lift to the quadrotor.

The flying stabilization in that kind of flying machine is achieved by two opposite forces [26]. The force that pushes the UAVs down is the gravity, an opposite force needs to be created of same magnitude to get hold the vehicle flying. That force is generated by the propellers that create a low pressure area over the top surface of them.

These propellers are able to change the quadrotor angles. In Figure 3.3 (a) a sketch of the quadrotor structure is presented in black. The Ω symbol represents the angular speed, and the Δ symbol represents the increase in speed. $\ddot{\Phi}$, $\ddot{\Theta}$ and $\ddot{\Psi}$ represent the angular acceleration created in that moment in X, Y and Z directions. The fixed-

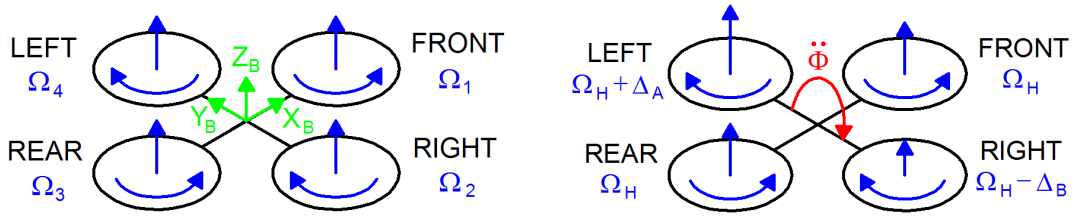

Figure 3.3: a) Simplified quadrotor motor in hovering (left) and b) Roll movement (right)

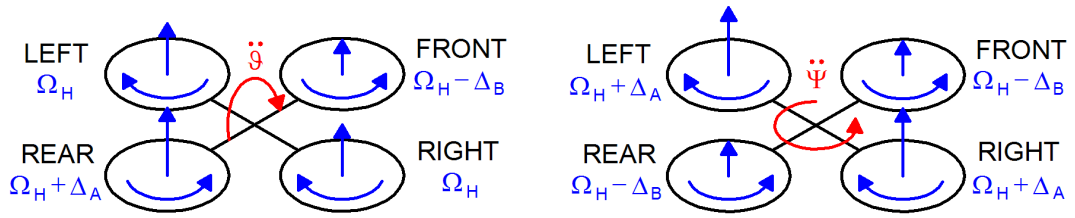

Figure 3.4: a) Pitch movement (left) and b) Yaw movement (right)

body B-frame is shown in green and in blue is represented the angular speed of the propellers. Roll movement is provided by increasing (or decreasing) the left propeller speed and by decreasing (or increasing) the right one. It leads to a torque with respect to the X_B axis which makes the quadrotor turn. Figure 3.3 (b) shows the roll movement on a quadrotor sketch. Pitch movement is very similar to the roll and is provided by increasing (or decreasing) the rear propeller speed and by decreasing (or increasing) the front one. It leads to a torque with respect to the Y_B axis which makes the quadrotor turn. Figure 3.4 (a) shows the pitch movement on a quadrotor sketch. The Yaw moment is provided by increasing (or decreasing) the front-rear propellers' speed and by decreasing (or increasing) that of the left-right couple. It leads to a torque with respect to the Z_B axis which makes the quadrotor turn. The yaw movement is generated thanks to the fact that the left-right propellers rotate clockwise, while the front-rear ones rotate counter-clockwise. Hence, when the overall torque is unbalanced, the helicopter turns on itself around Z_B. Figure 3.4 (b) shows the yaw movement on a quadrotor sketch. [27]

Working with ArduCopter

In this thesis we will use a default ArduCopter quadcopter that will be connected with a ground workstation through XBee [28]. The MAVlink [29] serial connection will provide data about attitude (Euler angles and Euler angle speeds), vfr_hub (ground speed, air speed, altitude and climb rate), GPS, RC (the values of the 8 radio frequency channels).

This serial connection also provides heartbeat (systems can use this message to track if the system is alive); navigation controller output (with information about goals and errors); hardware status (board voltage); raw IMU (acceleration, gyroscope and compass values); pressure; and some more parameters which have not been used in this system, while the same connection will be used to send the required throttle, and the euler angles.

In Figure 3.5 it is shown that it is possible to create a network between several aerial robots and ground stations using XBee devices.

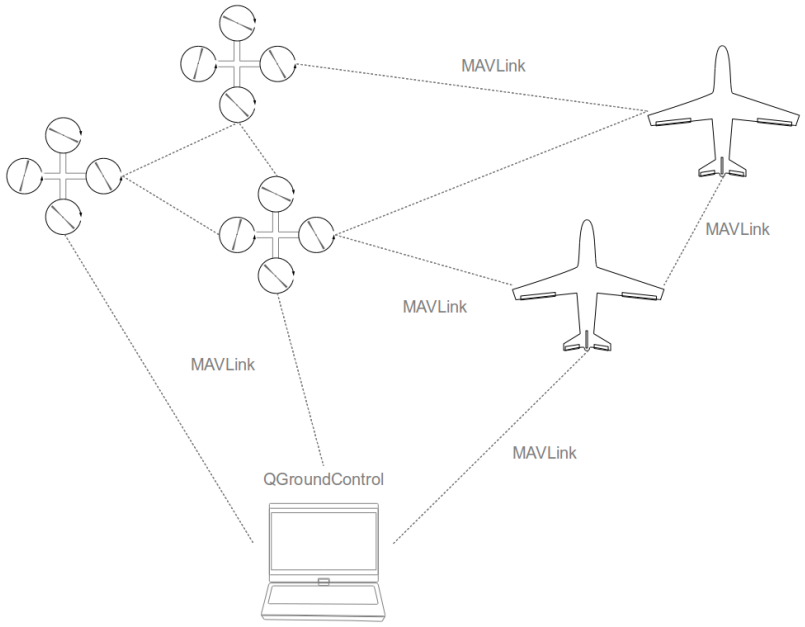

Figure 3.5: MAVlink connection between flying machines and a ground station [30]

3.1.2 XBee

XBee (in Figure 3.6) Digi International radio module is being displayed that uses the ZigBee protocol (it is based on an IEEE 802 standard).

For this thesis, it is needed a device with a low consumption. The normal flying time for this kind of UAVs usually is lower than 15 minutes, so any extension in the autonomy is a big improvement as is also needed a long working range. The quadrotor can fly autonomously, but it is usually good to keep the telemetry running to see what happened in the air and to overcome some minor problems.

Therefore, an XBee Pro 60mW Wire Antenna - Series 1 (802.15.4) has been selected. It is a cheap (less that $40) and reliable device with more than 1 km connectivity range

Figure 3.6: XBee module

and a low consumption.

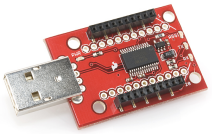

Figure 3.7: XBee Explorer

For the interface of this XBee with the workstation an *XBee Explorer Dongle* (in Figure 3.7) has been selected, while for the interface with the ArduCopter the selection has been an *XBee Explorer Regulated* through the UART0 port.

3.1.3 Positioning Systems

Figure 3.8: GPS module

A MTek GPS plus an adapter board (in Figure 3.8) has been used to control the quadcopter outdoor with a sufficient accuracy (about 3 meters) and a good stability.

This device along with a barometer, that uses the pressure to know the altitude and a ultrasonic sensor (LV-EZ0) allows to have a good position control in every moment.

The GPS and the barometer are used for the absolute position, while the sonar is used for a relative position. This helps the platform to take off and landing autonomously without any external help [31].

3.1.4 Remote Control

A Futaba Remote Control is used in case of emergency. If something happens and the quadrotor looses the control, the operator can switch to manual control just changing by the mode in the remote control, or using another mode, is possible to perform automatic emergency landing (the code in the Appendix 1.3.2). By setting a specific height, the quadrotor can go to height zero, progressively reducing the motors power and using intermediate height way points to stabilize.

3.2 Code Architecture and Evaluations in Real Tests

3.2.1 Interface

The interface between ArduCopter and ROS is a python code to connect the workstation machine to ArduCopter using XBee and then to create the topics and services to establish good communication. The code can be found in Appendix A.1.

In that code it is created a connexion between the XBee in the ArduCopter and the XBee in the workstation using MAVlink protocol. It starts the publishers for the GPS, RC (radio control signal), state (armed, GPS enable and current mode), vfr_hub (air speed, ground speed, orientation, throttle, altitude, and climb rate), attitude (Euler angles and speeds) and raw_IMU and the subscribers for the send_rc (it will be the topic used to send the commands). It also starts the services for arm and disarm the quadrotor.

Now, in the main it is created the node ("roscopter") and it is stored the MAVlink message with the function *recv_match*. The message is checked, and it is published in the right topic.

On the other hand, each time a message is sent to *send_rc*, a callback is launched to send the data to the ArduCopter.

3.2.2 Simulation

Figure 3.9 block diagram is being presented, which represents the overall simulation that combines the virtual quadrotor with the real one, situation described in Section 2.3. The pure simulation diagram would be just removing from the "*/attitude*" topic to the right.

In that case, the control is managed by the *pr2_teleop* package. That package uses the keyboard interruptions to modify the *cmd_vel* topic, which controls the speed in almost all ROS robots.

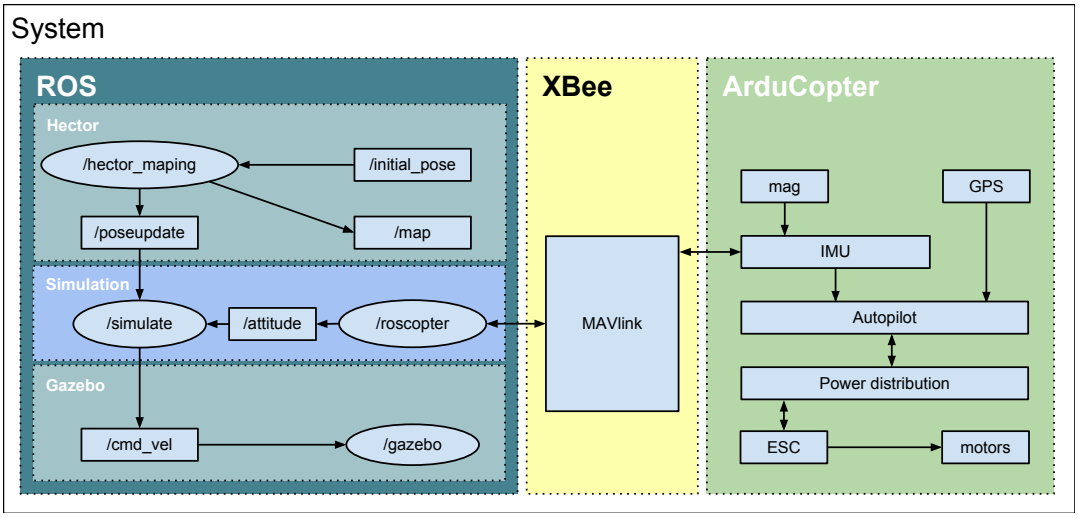

Figure 3.9: Simulation nodes diagram

In the ROS module, the blue one, circles are programs (or nodes) and squares are topics. That module uses three basic programs to carry out the task. In the middle area, the simulate node is the responsible of managing the controller. It reads data from the ArduCopter (through the RosCopter node) and from Hector quadrotor and actuates in the quadrotor speed command (roll, pitch and yaw speed).

In the upper part, the hector_maping node creates the world and manages the SLAM map that the UAV is seeing. It has the physical parameters to shape the robot and it places the robot in Gazebo. Hector uses the */poseupdate* topic to share the position in the 3 axes and the quaternion. Gazebo and hector_maping are also connected with a */scan* topic (omitted in the picture to improve the clarity). With this topic, hector knows how the world is in order to draw its SLAM map. Hector also needs a */initial_pose*. This allows the utilization of any kind of map without problems. This initial pose has to be a position without any other object.

On the other hand, in the bottom, **Gazebo** manages the physics (with the provided parameters) in the simulation and visualizes the quadrotor in a 3D map.

The simulation code can be located in the Appendix 1.2.2.

In this program, every active part is in the callbacks. There are two of them. The first one is launched when there is an update on the topic "*attitude*" that stores the euler angles and their velocities. The second one is launched when the update happens on the

topic "*poseupdate*" that stores the simulator quaternion.

The "poseupdate" callback transforms the quaternion to the roll, pitch, yaw euler angles and stores them in global variables. The "attitude" callback is in charge of controlling.

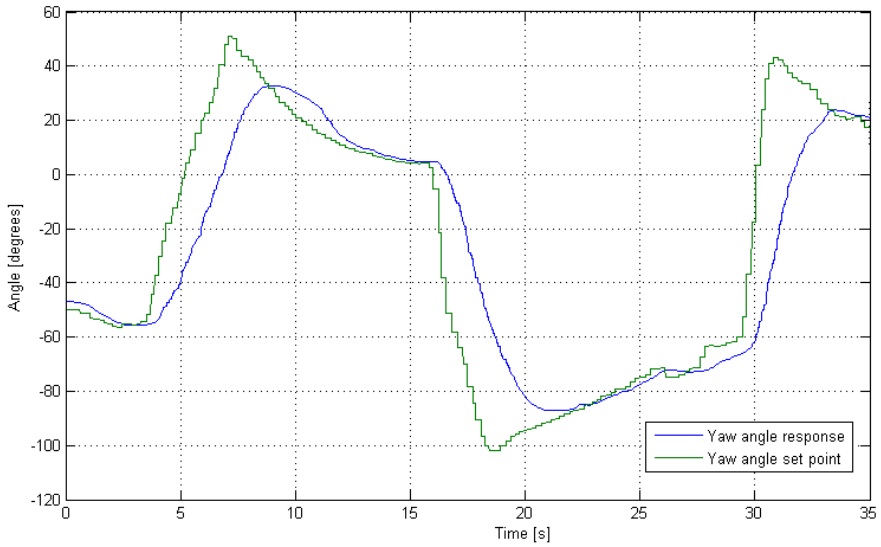

Figure 3.10: Response of manual rotation in simulation

In Figure 3.10 you can see how the yaw angle in the virtual quadrotor changes when the real quadrotor angle is changed. It cannot be a good representation of how the control is because is hard to have a stable input signal (the goal is by hand), but you can see a fast response and a perfect tracking.

3.2.3 Flying code

Different programs have been developed to move the quadrotor and all of them can be called with different parameters in the function *go* (the code (in python) in Appendix 1.3.1). In this way, it is possible to navigate with way points, to rotate to a specific angle or to go to a specific altitude. There is also a safety code for the take off or landing of the quadrotor.

Figure 3.11 is a representation of how the quadrotor performs its task.

It consists of three modules. The first one, red in the picture (left), is the ROS system that has sensor measurement (global position, euler angles, RC channels and the state) and can actuate on the roll and pitch angles and the yaw speed through the RC parameters. It uses two programs to perform its task. The main one is */go*. It is the control

program and it is used to close the loop. Instead, */roscopter* is just a interface (as it has been presented in Section 3.2.1), it bridges both platforms taking messages from one side and leaving to the other side.

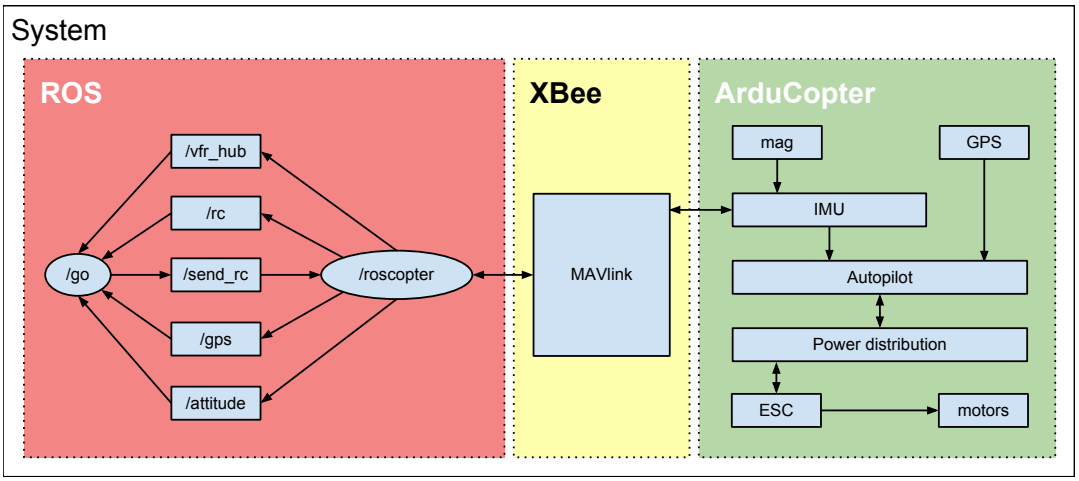

Figure 3.11: Control nodes diagram

The second one, in yellow (middle), is a *MAVlink* interface that is responsible for taking data in both sides and send them to the other. Between */roscopter* and *MAVlink* are carrying out the task of converting ArduCopter data into ROS data.

The third part, in green (right), is the ArduCopter system. It has an autopilot module, it is the smart device on board, which can manage simple control tasks and it has the sensor measurements and can control the movement of the motors. It receives data from the IMU and the GPS inputs. It has also an inertial measurement unit and receiving the data from a external magnetometer. Finally, the ArduCopter system mounts a power distribution board controlled by the autopilot module. It can give the command signals to the motors' controllers.

3.3 Controller

We will use the ROS platform to perform the control, because topics and nodes are good tools to have a discrete control. We don't need to create a periodical loop nor a discretization because the data is sent periodically from ArduPilot to the ROS system, through MAVlink. This means that the ROS program will launch the topic update callbacks periodically too (when a new message appear in the topic).

Several controllers have been tested to get a quick response, avoiding as much as possible overshooting, and achieve a fast settlement.

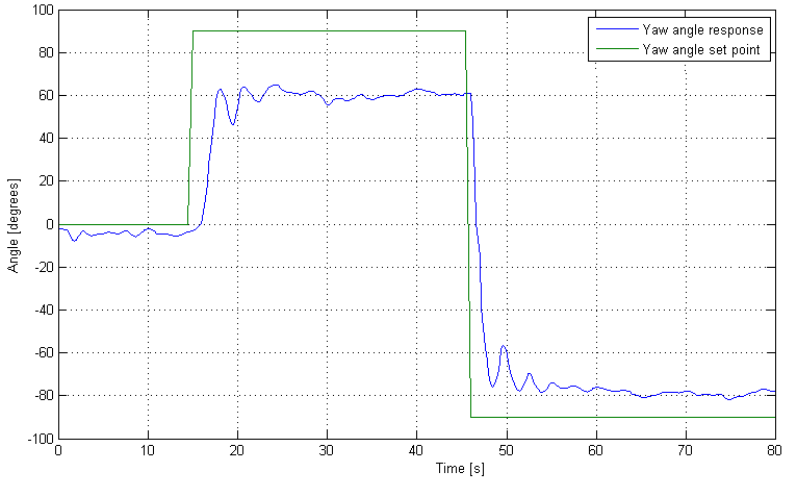

Figure 3.12: Step response of rotation in [90,-90] range

In the first attempt a proportional control was used to test the similarity between the real performance of the quadrotor and the performance in the simulation. However, the test results show a non-unity gain and an offset in the response as can be seen in Figure 3.12. This is due to the appearance of external perturbations and the speed threshold of the motors. Here, we can see the angle never reach the desired value as the motors have no enough velocity to start the movement. It is also possible to see a bigger error in the positive zone caused by a little imbalance in zero speed.

With some adjustments more in the proportional constant is possible to save most of the speed thresholds but still do not have a unity gain (in Figure 3.13), and thus a PID controller will be applied.

So the second attempt was using a PID controller [32]. With this kind of controller you can add the cumulative error in the action (with the integrative part). The longer the UAV is within an error, the biggest is the action. And you can add the future behaviour for the error in the action using the rate of change of the error (with the derivative part). Finally, the following constants (equation 3.1) have produced an acceptable behaviour.

$$\kappa_p = 2 \qquad \kappa_i = 0.1 \qquad \kappa_d = 0.07 \tag{3.1}$$

As it is possible to see now in Figure 3.14, if the change is not too much (less than 120°), the response is perfect. It uses 3 seconds to reach the goal with 10% of error. After 16 seconds the error oscillate between 2 and 3 %.

If the change is higher (at second 50) it has some overshooting (about 50 degrees). Now it uses 10 seconds to reach the goal with 10% of error. After 2 seconds more, the

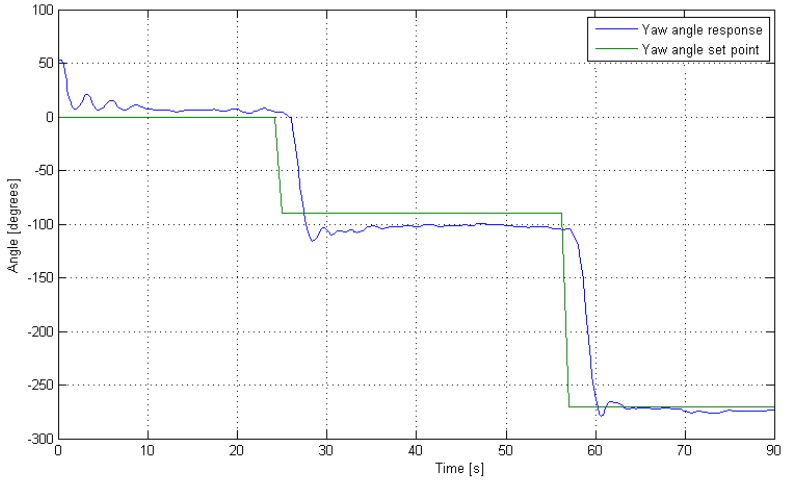

Figure 3.13: Step response of rotation in [90,-90] range with a bigger constant

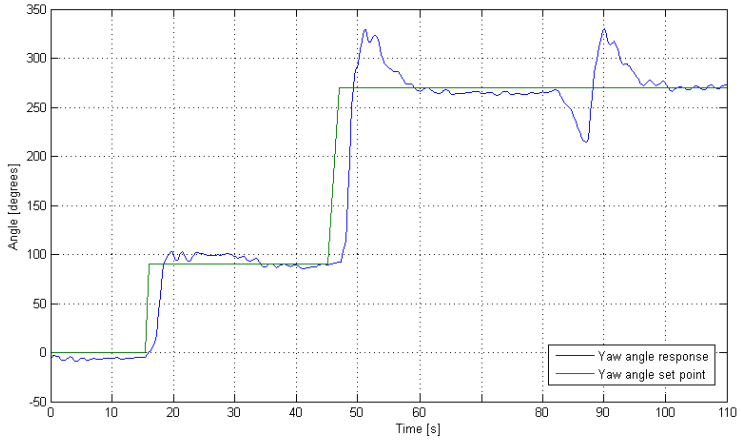

Figure 3.14: Step response and perturbation of rotation in [90,-90] with a PID

error is lower that 3%.

Furthermore, this system can correct the perturbations quite soon (there was a manual quadcopter rotation at second 83).

Algorithm 1: Control algorithm

read data
store prev_error
calculate error: goal - data
calculate proportional action: κ_p * *error*
calculate integral: prev_integral + κ_i * *error*
calculate derivative: κ_d * *(error - prev_error)*
calculate action: Proportional + Integral + Derivative
ajust the action to the rc level
publish the action

In the presented codes, as it has been stated, topic callbacks are used in order to carry out the control instead of loops. So the Algorithm 1 is launched each time that "attitude" callback has an update in its topic.

That algorithm uses three correcting terms, whose sum is the control action. The **proportional** action produces an absolute change in the energy of the system, which modifies the response time.

The **integral** action helps to remove the residual steady-state error. It maintains a count of the cumulative error, and it adds to the action in order that the slightest errors can be eliminated. The **derivative** action predicts system behavior and thus improves settling time and stability of the system.

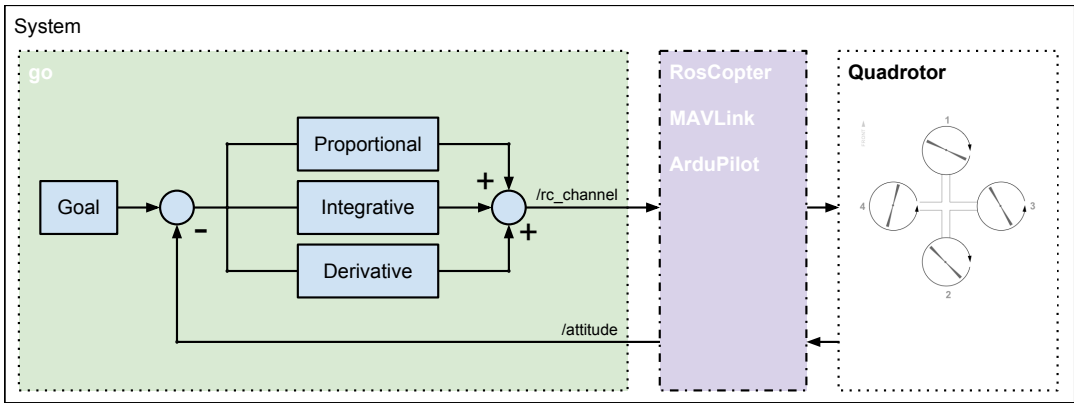

Figure 3.15: Control diagram

In the case of the waypoint control, the roll controls the Y coordinate, the pitch controls the X coordinate and the throttle controls the Z coordinate. The implementation of this

controller is presented in Figure 3.15. In this representation, the interfaces have been simplified, since the focus has been provided in the control code *go*.

CHAPTER 4

Future work

The work you can do in six months is limited, and therefore the work that lies ahead is enormous.

After the waypoint navigation codes and, derived from it, paths generated navigation, will allow us to trace routes of action for SLAM.

Having ensured the proper operation of navigation codes would have to start working with SLAM tasks. The hardware for this purpose would be a USB camera connected to the onboard computer. By reconstruction software that take advantage of the quadrotor own motion for a stereoscopic image.

This reconstruction would also used for route planning and obstacle avoidance.

Future research will address how the robot performs its task. While perform its task completely in local (on-board computer without help from the ground station) is the option that seems more complete. Would be equally satisfactory options. One alternative would be the remote processing of data sending the images to the ground station. It would perform the 3D reconstruction and would send the new routes to the quadrotor. The second one would be to record such images for off-line 3D reconstruction for use as a map on other platforms.

This two alternatives could help in the quadrotor consumption. In this case, you could avoid the onboard computer. The main problem of this course is that the ground station and the quadrotor would have to have always a stable connection.

CHAPTER 5

Conclusion

The aim of this Master Thesis was to design and develop a quadrotor that will have the merit to operate under a fully ROS enabled software environment. As it has been presented in the Thesis, based on an existing flying frame, a quadrotor has been designed and integrated, where all the necessary hardware components, including the motor controllers, the IMU, the wireless communication link based on the xBees and the GPS have been properly adjusted and tuned for allowing the proper flight capabilities of the platform. Except from the first necessary stage of developing the flying platform, the main focus has been provided in the design and implementation of the necessary ROS enabled software platform, that would be able to connect all the sub-components and support novel ROS enabled control algorithms for allowing the autonomous or semi-autonomous flying.

As it has been indicated in the introduction, there are already some existing ROS enabled software implementations for quadrotors but these software implementations are platform depended and they aim mainly in performing a tele-operation task, which means that a user should always fly the quadrotor from a distance. The results of this thesis have been presented a novel approach where the quadrotor is able to perform autonomous tasks, such as: attitude and altitude stabilisation in a fully ROS enabled environment. The results have been performed in both simulation and extended experimental studies.

The software developments that have been performed have covered various software developing aspects, such as: operating system management in different robotic platforms, the study of the various forms of programming in the ROS environment, the evaluation of software building alternatives, the development of the ROS interface and finally various practical software implementation issues with the developed aerial platform.

Upon completion of this project, two different configurations (aluminium and plastic) of the quadrotor have been delivered. The ArduCopter has been utilised and integrated in the ROS platform while as it has been presented in the thesis, a series of codes have been designed for the project, ranging from establishing the communication with the ground station to the way points following, with the most important to be the

software implementation of the ROS enabled PID controllers for the attitude and altitude stabilisation of the quadrotor.

Although the tests in a real environment have not had all the desirable extent due to the luck of having a full controllable laboratory environment, the developed software architecture for the attitude and altitude stabilisation of the quadrotor have been extended evaluated and the obtained experimental results have proven the overall feasibility of the novel ROS enabled control structure implementation and the achievement of a acceptable flying performance for the quadrotor.

The potential of such autonomous quadrotors is enormous, and it can be even bigger when the size of these devices gets miniaturized, with a relative decrease in the overall hardware cost. For these quadrotors, a proper real time and modular operating system is needed in order to extend the capabilities of these platforms in higher standards. With this thesis it has been experimentally demonstrated that ROS is a proper operating system to manage all the requirements that might arise, while it has the unique merit of being an open source software, allowing the sharing of drivers and programs among the developer community, for speeding the developing phase.

APPENDIX A

Program codes

A.1 RosCopter

```
#!/usr/bin/env python
import roslib; roslib.load_manifest('roscopter')
import rospy
from std_msgs.msg import String, Header
from std_srvs.srv import *
from sensor_msgs.msg import NavSatFix, NavSatStatus, Imu
import roscopter.msg
import sys,struct,time,os

sys.path.insert(0, os.path.join(os.path.dirname(os.path.realpath(__file__)), '../mavlink/pymavlink'))


from optparse import OptionParser
parser = OptionParser("roscopter.py [options]")

parser.add_option("--baudrate", dest="baudrate", type='int',
                  help="master port baud rate", default=57600)
parser.add_option("--device", dest="device", default="/dev/ttyUSB0", help="serial device")
parser.add_option("--rate", dest="rate", default=10, type='int', help="requested stream rate")
parser.add_option("--source-system", dest='SOURCE_SYSTEM', type='int',
                  default=255, help='MAVLink source system for this GCS')
parser.add_option("--enable-control",dest="enable_control", default=False, help="Enable listning to control messages")

(opts, args) = parser.parse_args()

import mavutil

# create a mavlink serial instance
master = mavutil.mavlink_connection(opts.device, baud=opts.baudrate)

if opts.device is None:
    print("You must specify a serial device")
    sys.exit(1)

def wait_heartbeat(m):
```

```
    '''wait for a heartbeat so we know the target system IDs'''
    print("Waiting for APM heartbeat")
    m.wait_heartbeat()
    print("Heartbeat from APM (system %u component %u)" % (m.target_system, m.
        target_system))


#This does not work yet because APM does not have it implemented
#def mav_control(data):
#    '''
#    Set roll, pitch and yaw.
#    roll                    : Desired roll angle in radians (float)
#    pitch                   : Desired pitch angle in radians (float)
#    yaw                     : Desired yaw angle in radians (float)
#    thrust                  : Collective thrust, normalized to 0 .. 1 (float)
#    '''
#    master.mav.set_roll_pitch_yaw_thrust_send(master.target_system, master.
    target_component,
#                                                          data.roll, data.pitch,
    data.yaw, data.thrust)
#
#    print ("sending control: %s"%data)


def send_rc(data):
    master.mav.rc_channels_override_send(master.target_system, master.target_component,
        data.channel[0],data.channel[1],data.channel[2],data.channel[3],data.channel[4],
        data.channel[5],data.channel[6],data.channel[7])
    print ("sending rc: %s"%data)


#service callbacks
#def set_mode(mav_mode):
#    master.set_mode_auto()

def set_arm(req):
    master.arducopter_arm()
    return True

def set_disarm(req):
    master.arducopter_disarm()
    return True

pub_gps = rospy.Publisher('gps', NavSatFix)
#pub_imu = rospy.Publisher('imu', Imu)
pub_rc = rospy.Publisher('rc', roscopter.msg.RC)
pub_state = rospy.Publisher('state', roscopter.msg.State)
pub_vfr_hud = rospy.Publisher('vfr_hud', roscopter.msg.VFR_HUD)
pub_attitude = rospy.Publisher('attitude', roscopter.msg.Attitude)
pub_raw_imu =  rospy.Publisher('raw_imu', roscopter.msg.Mavlink_RAW_IMU)
if opts.enable_control:
    #rospy.Subscriber("control", roscopter.msg.Control , mav_control)
    rospy.Subscriber("send_rc", roscopter.msg.RC , send_rc)

#define service callbacks
arm_service = rospy.Service('arm',Empty,set_arm)
disarm_service = rospy.Service('disarm',Empty,set_disarm)


#state
gps_msg = NavSatFix()
```

```
95 def mainloop():
       rospy.init_node('roscopter')
97     while not rospy.is_shutdown():
           rospy.sleep(0.001)
99         msg = master.recv_match(blocking=False)
           if not msg:
101            continue
           #print msg.get_type()
103        if msg.get_type() == "BAD_DATA":
               if mavutil.all_printable(msg.data):
105                sys.stdout.write(msg.data)
                   sys.stdout.flush()
107        else:
               msg_type = msg.get_type()
109            if msg_type == "RC_CHANNELS_RAW" :
                   pub_rc.publish([msg.chan1_raw, msg.chan2_raw, msg.chan3_raw, msg.
                       chan4_raw, msg.chan5_raw, msg.chan6_raw, msg.chan7_raw, msg.
                       chan8_raw])
111            if msg_type == "HEARTBEAT":
                   pub_state.publish(msg.base_mode & mavutil.mavlink.
                       MAV_MODE_FLAG_SAFETY_ARMED,
113                                  msg.base_mode & mavutil.mavlink.
                                       MAV_MODE_FLAG_GUIDED_ENABLED,
                                   mavutil.mode_string_v10(msg))
115            if msg_type == "VFR_HUD":
                   pub_vfr_hud.publish(msg.airspeed, msg.groundspeed, msg.heading, msg.
                       throttle, msg.alt, msg.climb)
117
               if msg_type == "GPS_RAW_INT":
119                fix = NavSatStatus.STATUS_NO_FIX
                   if msg.fix_type >=3:
121                    fix=NavSatStatus.STATUS_FIX
                   pub_gps.publish(NavSatFix(latitude = msg.lat/1e07,
123                                          longitude = msg.lon/1e07,
                                           altitude = msg.alt/1e03,
125                                        status = NavSatStatus(status=fix, service =
                                               NavSatStatus.SERVICE_GPS)
                                           ))
127            #pub.publish(String("MSG: %s"%msg))
               if msg_type == "ATTITUDE" :
129                pub_attitude.publish(msg.roll, msg.pitch, msg.yaw, msg.rollspeed, msg.
                       pitchspeed, msg.yawspeed)


131
               if msg_type == "LOCAL_POSITION_NED" :
133                print "Local Pos: (%f %f %f) , (%f %f %f)" %(msg.x, msg.y, msg.z, msg.vx
                       , msg.vy, msg.vz)

135            if msg_type == "RAW_IMU" :
                   pub_raw_imu.publish (Header(), msg.time_usec,
137                                     msg.xacc, msg.yacc, msg.zacc,
                                        msg.xgyro, msg.ygyro, msg.zgyro,
139                                     msg.xmag, msg.ymag, msg.zmag)


141


143
# wait for the heartbeat msg to find the system ID
145 wait_heartbeat(master)

147
# waiting for 10 seconds for the system to be ready
```

```
149 print("Sleeping for 10 seconds to allow system, to be ready")
    rospy.sleep(10)
151 print("Sending all stream request for rate %u" % opts.rate)
    #for i in range(0, 3):
153
    master.mav.request_data_stream_send(master.target_system, master.target_component,
155                                     mavutil.mavlink.MAV_DATA_STREAM_ALL, opts.rate, 1)

157 #master.mav.set_mode_send(master.target_system,
    if __name__ == '__main__':
159     try:
            mainloop()
161     except rospy.ROSInterruptException: pass
```

A.2 Simulate

1.2.1 Simulation

```
#include <ros/ros.h>
#include <geometry_msgs/Twist.h>
#include <geometry_msgs/PoseWithCovarianceStamped.h>

float goal[6][3]={{0,0,0},{1,1,0},{-1,1,0},{-1,-1,0},{1,-1,0},{1,1,0}};
float tolerance = 0.05;

double kp = 0.5;
double ki = 0.0002;
double kd = 0.00005;

float w;

float error_x = 0;
float error_y = 0;
float error_z = 0;
float error_w = 0;
float prev_error_x = 0;
float prev_error_y = 0;
float prev_error_z = 0;
float prev_error_w = 0;
float rise = 1;
float nonstop = true;

float proportional_x = 0;
float proportional_y = 0;
float proportional_z = 0;
float proportional_w = 0;
float integral_x = 0;
float integral_y = 0;
float integral_z = 0;
float integral_w = 0;
float derivative_x = 0;
float derivative_y = 0;
float derivative_z = 0;
float derivative_w = 0;
float action_x = 0;
float action_y = 0;
float action_z = 0;
float action_w = 0;

geometry_msgs::Point real;
geometry_msgs::Twist twist;

bool must_exit = false;
int waypoint_number = 0;

void odoCallback(const geometry_msgs::PoseWithCovarianceStamped::ConstPtr& msg)
{
  real.x=msg->pose.pose.position.x;
  real.y=msg->pose.pose.position.y;
  real.z=msg->pose.pose.position.z;
  w=msg->pose.pose.orientation.z;

  prev_error_x = error_x;
  prev_error_y = error_y;
  prev_error_z = error_z;
```

```
  prev_error_w = error_w;
59 error_x = goal[waypoint_number][0] - real.x;
   error_y = goal[waypoint_number][1] - real.y;
61 error_z = (goal[waypoint_number][2] + 0.001 * rise) - real.z;
   error_w = 0 - w;
63
   proportional_x = kp * error_x;
65 proportional_y = kp * error_y;
   proportional_z = kp * error_z;
67 proportional_w = kp * error_w;
   integral_x += ki * error_x;
69 integral_y += ki * error_y;
   integral_z += ki * error_z;
71 integral_w += ki * error_w;
   derivative_x = kd * (error_x - prev_error_x);
73 derivative_y = kd * (error_y - prev_error_y);
   derivative_z = kd * (error_z - prev_error_z);
75 derivative_w = kd * (error_w - prev_error_w);

77 action_x = proportional_x + integral_x + derivative_x;
   action_y = proportional_y + integral_y + derivative_y;
79 action_z = proportional_z + integral_z + derivative_z;
   action_w = 10 * proportional_w + integral_w + derivative_w;
81
   twist.linear.x = action_x;
83 twist.linear.y = action_y;
   twist.linear.z = action_z;
85 twist.angular.z = action_w;

87 ROS_INFO("Error X: %0.2f \n", error_x);
   ROS_INFO("Error Y: %0.2f \n", error_y);
89 ROS_INFO("Error Z: %0.2f \n", error_y);
   ROS_INFO("W: %0.2f \n", w);
91 ROS_INFO("Action X: %0.2f \n", action_x);
   ROS_INFO("Action Y: %0.2f \n", action_y);
93 ROS_INFO("Action Z: %0.2f \n", action_y);
   ROS_INFO("Action W: %0.2f \n", action_y);
95 ROS_INFO("Voy hacie el objetivo %d \n", waypoint_number+1);

97
   if ((fabs(error_x) < tolerance) && (fabs(error_y) < tolerance)) {
99   if (must_exit == true)
     {
101    twist.linear.x = 0;
       twist.linear.y = 0;
103    twist.linear.z = 0;
       twist.angular.z = 0;
105    exit(0);
      }
107   else
      {
109    waypoint_number += 1;
       rise += 1;
111   }
   }
113
   if (waypoint_number == (sizeof(goal)/sizeof(goal[0])))
115 {
     if (nonstop)
117  {
       waypoint_number = 1;
119  }
     else
```

```
121     {
          waypoint_number = 0;
123       must_exit = true;
        }
125   }
}
127
int main(int argc, char **argv)
129 {

131   ros::init(argc, argv, "test_viapoint");

133   ros::NodeHandle np;
      ros::NodeHandle nh;
135   ros::Publisher pub_vel = nh.advertise<geometry_msgs::Twist>("cmd_vel", 1);

137   ros::Subscriber sub = np.subscribe("poseupdate", 1000, odoCallback);

139   ros::Rate loop_rate(10);
      int count = 0;
141
      while (ros::ok())
143   {
        pub_vel.publish(twist);
145     ros::spinOnce();
        loop_rate.sleep();
147     ++count;
      }
149 }
```

1.2.2 Simulation with real input

```
#include <ros/ros.h>
#include "roscopter/RC.h"
#include <roscopter/Attitude.h>
#include <sensor_msgs/Imu.h>
#include <geometry_msgs/Quaternion.h>
#include <geometry_msgs/Twist.h>
#include <geometry_msgs/PoseWithCovarianceStamped.h>
#include <signal.h>
#include <termios.h>
#include <stdio.h>
#include "boost/thread/mutex.hpp"
#include "boost/thread/thread.hpp"

#define PI 3.14159

#define KEYCODE_SPACE 0x20

struct euler
{
  float roll;
  float pitch;
  float yaw;
} ;

geometry_msgs::Twist vel;
geometry_msgs::Quaternion virtual_quaternion;

euler real_angles;
euler virtual_angles;

float kp_p = 0.09;
float kp_y = 0.09;
float kp_r = 0.09;

float ki_p = 0.00004;
float ki_y = 0.00004;
float ki_r = 0.00004;

float kd_p = 0.00001;
float kd_y = 0.00001;
float kd_r = 0.00001;

float yaw_prev_error = 0;
float pitch_prev_error = 0;
float roll_prev_error = 0;

float yaw_error = 0;
float pitch_error = 0;
float roll_error = 0;

float yaw_action = 0;
float pitch_action = 0;
float roll_action = 0;

float yaw_proportional = 0;
float pitch_proportional = 0;
float roll_proportional = 0;

float yaw_integral = 0;
float pitch_integral = 0;
```

```
61 float roll_integral = 0;

63 float yaw_derivative = 0;
   float pitch_derivative = 0;
65 float roll_derivative = 0;

67 float toDegrees(float radians)
   {
69   return radians*180/PI;
   }
71
   float toRads(float degrees)
73 {
     return degrees*PI/180;
75 }

77 int diffAngle(float target, float source)
   {
79   int a;
     target = (int) target;
81   source = (int) source;
     a = target - source;
83   a = (a + 180) % 360 - 180;
     return a;
85 }

87 euler QuaternionToRoll(float x, float y, float z, float w)
   {
89   float test = x * y + z * w;
     float roll, pitch, yaw;
91   euler solution;
     if (test > 0.499) { // singularity at north pole
93     pitch = (2 * atan2(x, w));
       yaw = (PI / 2);
95     roll = 0;
       solution.roll = toDegrees(roll);
97     solution.pitch = toDegrees(pitch);
       solution.yaw = toDegrees(yaw);
99     return solution;
     }
101  if (test < -0.499) { // singularity at south pole
       pitch = (-2 * atan2(x, w));
103    yaw = (-PI / 2);
       roll = 0;
105    solution.roll = toDegrees(roll);
       solution.pitch = toDegrees(pitch);
107    solution.yaw = toDegrees(yaw);
       return solution;
109  }
     float sqx = x * x;
111  float sqy = y * y;
     float sqz = z * z;
113  pitch = atan2(2 * y * w - 2 * x * z, 1 - 2 * sqy - 2 * sqz);
     yaw = asin(2 * test);
115  roll = atan2(2 * x * w - 2 * y * z, 1 - 2 * sqx - 2 * sqz);
     solution.roll = toDegrees(roll);
117  solution.pitch = toDegrees(pitch);
     solution.yaw = toDegrees(yaw);
119  return solution;
   }
121
   void odoCallback(const roscopter::Attitude& msg)
123 {
```

```
125   real_angles.roll=toDegrees(msg.roll);
      real_angles.pitch=toDegrees(msg.pitch);
127   real_angles.yaw=toDegrees(msg.yaw);

129   yaw_prev_error = yaw_error;
      yaw_error = diffAngle(real_angles.yaw,virtual_angles.yaw);
131   yaw_proportional = kp_y * yaw_error;
      yaw_integral += ki_y * yaw_error
133   yaw_derivative = kd_y * (yaw_error - yaw_prev_error)

135   yaw_action = yaw_proportional + yaw_integral + yaw_derivative

137   pitch_prev_error = pitch_error;
      pitch_error = real_angles.pitch - virtual_angles.pitch;
139   pitch_proportional = kp_p * pitch_error;
      pitch_integral += ki_p * pitch_error
141   pitch_derivative = kd_p * (pitch_error - pitch_prev_error)

143   pitch_action = pitch_proportional + pitch_integral + pitch_derivative

145   roll_prev_error = roll_error;
      roll_error = real_angles.roll - virtual_angles.roll;
147   roll_proportional = kp_r * roll_error;
      roll_integral += ki_r * roll_error
149   roll_derivative = kd_r * (roll_error - roll_prev_error)

151   roll_action = roll_proportional + roll_integral + roll_derivative

153   vel.angular.z = yaw_action;
      vel.linear.x = pitch_action;
155   vel.linear.y = roll_action;
      printf("Action: %0.2f \n", yaw_action);
157   printf("Error: %0.2f \n", yaw_error);
      printf("Roll: %0.2f \n", real_angles.yaw);
159   printf("Virtual roll: %0.2f \n", virtual_angles.yaw);
}
161
void odoCallback2(const geometry_msgs::PoseWithCovarianceStamped::ConstPtr& msg)
163 {
      virtual_quaternion.x=msg->pose.pose.orientation.x;
165   virtual_quaternion.y=msg->pose.pose.orientation.y;
      virtual_quaternion.z=msg->pose.pose.orientation.z;
167   virtual_quaternion.w=msg->pose.pose.orientation.w;

169   virtual_angles = QuaternionToRoll(virtual_quaternion.x,virtual_quaternion.y,
          virtual_quaternion.z,virtual_quaternion.w);
}
171

173 int main(int argc, char **argv)
{
175     ros::init(argc, argv, "simulate");
        ros::NodeHandle n;
177     ros::NodeHandle np;
        ros::NodeHandle nh;
179     ros::Publisher pub_vel = n.advertise<geometry_msgs::Twist>("cmd_vel", 1000);

181     ros::Subscriber attitude_sub = np.subscribe("attitude", 1000, odoCallback);
        ros::Subscriber imu_virtual_sub = nh.subscribe("poseupdate", 1000, odoCallback2);
183
        ros::Rate loop_rate(10);
185     int count = 0;
```

```
187     while (ros::ok())
        {
189         pub_vel.publish(vel);
            ros::spinOnce();
191         loop_rate.sleep();
            ++count;
193     }
        return 0;
195 }
```

A.3 Control code

1.3.1 Go

```
#!/usr/bin/env python
from __future__ import print_function
import roslib; roslib.load_manifest('roscopter')
import rospy
from std_msgs.msg import String
from sensor_msgs.msg import NavSatFix, NavSatStatus, Imu
import roscopter.msg
import sys,struct,time,os
import time

from optparse import OptionParser
parser = OptionParser("go.py [options]")

parser.add_option("-m","--mode", dest="mode",
                  help="Working mode: waypoints, goup, rotation", default="waypoints")

parser.add_option("-w","--waypoints", dest="goal_waypoints", type='float',
                  help="Array with the waypoints [x1,y1,z1,x2,y2,z2...]", default
                      =[0,0,0])

parser.add_option("-a","--altitude", dest="altitude", type='float',
                  help="Altitude goal", default=1.0)

parser.add_option("-r","--angle", dest="goal_deg", type='int',
                  help="yaw angle goal", default=180)

(opts, args) = parser.parse_args()

waypoint_number = 0

def rc(data):
    global security
    global throttle
    security = data.channel[5]
    throttle = data.channel[2]

def landing(rate):
    if rate < 1450 :
        for x in range(rate, 1300, -1):
            for x2 in range(x-5,x+5):
          channel = [0,0,x2,0,0,0,0,0]
          pub_rc.publish(channel)
          time.sleep(0.05)
    else:
        for x in range(1450, 1300, -1):
            for x2 in range(x-5,x+5):
          channel = [0,0,x2,0,0,0,0,0]
          pub_rc.publish(channel)
          time.sleep(0.05)
    channel = [0,0,1000,0,0,0,0,0]
    pub_rc.publish(channel)

def security_vel(channel):

    channel_sec = channel

    if (security < 1200):
```

```
  print ("Changing to secure mode")
        channel = [0,0,0,0,0,0,0,0]
        pub_rc.publish(channel)
    elif (1200 < security < 1700):
  for i in range(channel[2]-5, channel[2]+5):
      channel_sec[2] = i
            pub_rc.publish(channel_sec)
    elif (security > 1700):
  landing(throttle)
        os._exit(1)
    else:
  for i in range(990, 1010):
            channel = [0,0,i,0,0,0,0,0]
            pub_rc.publish(channel)


if opts.mode == "waypoints":
    def gps(data):
  global error_x
  global error_y
  global error_z
  global Integral_x
  global Integral_y
  global Integral_z

        x = data.x
        y = data.y
        z = data.z

  prev_error_x = error_x
  prev_error_y = error_y
  prev_error_z = error_z
        error_x = opts.goal_waypoints[3*waypoint_number+0] - x
        error_y = opts.goal_waypoints[3*waypoint_number+1] - y
        error_z = opts.goal_waypoints[3*waypoint_number+2] - z

        action_x = kp * error_x
        action_y = kp * error_y
        action_z = kp * error_z

        position = {'x':x, 'y':y, 'z':z}
        goal = {'gx':opts.goal_waypoints[3*i+0], 'gy':opts.goal_waypoints[3*i+1], 'gz':
            opts.goal_waypoints[3*i+2]}
        error = {'ex':error_x, 'ey':error_y, 'ez':error_z}
        action = {'ax':action_x, 'ay':action_y, 'az':action_z}

        print ("Position: %(x).0.4f, %(y).0.4f, %(z).0.4f" % position)
        print ("Goal: %(gx).0.4f, %(gy).0.4f, %(gz).0.4f" % goal)
        print ("Error X, Y, Z: %(ex).0.4f, %(ey).0.4f, %(ez).0.4f" % error)
        print ("Actions X, Y, Z: %(ax).0.4f, %(ay).0.4f, %(az).0.4f" % action)

  Proportional_x = kp * error_x
  Proportional_y = kp * error_y
  Proportional_z = kp * error_z
  Integral_x += ki * error_x
  Integral_y += ki * error_y
  Integral_z += ki * error_z
  Derivative_x = kd * (error_x - prev_error_x)
  Derivative_y = kd * (error_y - prev_error_y)
  Derivative_z = kd * (error_z - prev_error_z)

  action_x = Proportional_x + Integral_x + Derivative_x
  action_y = Proportional_y + Integral_y + Derivative_y
```

```
119   action_z = Proportional_z + Integral_z + Derivative_z

121         rc_action_roll = 1500 + action_y
            rc_action_pitch = 1500 + action_x
123         rc_action_throttle = 1000 + action_z
            channel = [rc_action_roll,rc_action_pitch,rc_action_throttle,0,0,0,0,0]
125
            security_vel(channel)
127
            global waypoint_number
129         if error_x < tolerance and error_y < tolerance and error_z < tolerance:
                if must_exit == 1:
131               landing(throttle)
      os._exit(1)
133       else
              waypoint_number += 1
135
    if waypoint_number == (len(opts.goal_waypoints) / 3):
137       waypoint_number = 0
          must_exit = 1
139
      kp = 2
141   ki = 0.1
      kd = 0.07
143   error = 0
      Integral = 0
145   tolerance = 0.1
      must_exit = 0
147   gps_sub = rospy.Subscriber("gps", NavSatFix, gps)

149 elif opts.mode == "goup":
      def vfr_hud(data):
151   global error
    global Integral
153         global security

155         altitude = data.alt

157   prev_error = error
            error = goal - altitude
159         action = kp * error

161   Proportional = kp * error
    Integral += ki * error
163 Derivative = kd * (error - prev_error)

165   action = Proportional + Integral + Derivative

167         rc_action = 1000 + action
            channel = [0,0,rc_action,0,0,0,0,0]
169
    security_vel(channel)
171
      kp = 2
173   ki = 0.1
      kd = 0.07
175   error = 0
      Integral = 0
177   goal = opts.altitude
      vfr_hud_sub = rospy.Subscriber("vfr_hud", roscopter.msg.VFR_HUD, vfr_hud)
179
  elif opts.mode == "rotation":
181   def attitude(data):
```

```
        global error
        global Integral
        global throttle

        yaw = data.yaw * 180 / 3.14

        prev_error = error
        error = goal - yaw
        error = (error + 180) % 360 - 180 #only in yaw
        Proportional = kp * error
        print ("Error: %0.4f"%error)
        print ("Action: %0.4f"%Proportional)
        print ("Yaw: %0.4f"%yaw)
        print ("Security: %0.4f"%security)

        Integral += ki * error
        Derivative = kd * (error - prev_error)

        action = Proportional + Integral + Derivative

  rc_action = 1480 + action
        channel = [0,0,1300,rc_action,0,0,0,0]
        channel = [0,0,security,rc_action,0,0,0,0]

  security_vel(channel)

        f = open('yaw_data','a')
        ticks = time.time()
        to_file = {'time': ticks, 'yaw': yaw, 'goal': goal, 'error': error, 'prev_error'
            : prev_error, 'Proportional': Proportional, 'Integral': Integral, '
            Derivative': Derivative}
        f.write("%(time)d %(yaw)0.4f %(goal)0.4f %(error)0.4f %(prev_error)0.4f %(
            Proportional)0.4f %(Integral)0.4f %(Derivative)0.4f \n"%to_file)

    kp = 2 # 1.75
    ki = 0.1
    kd = 0.07
    error = 0
    Integral = 0
    goal = opts.goal_deg  #in rads
    attitude_sub = rospy.Subscriber("attitude", roscopter.msg.Attitude, attitude)

security = 0
pub_rc = rospy.Publisher('send_rc', roscopter.msg.RC)
rc_sub = rospy.Subscriber("rc", roscopter.msg.RC, rc)

def mainloop():

    pub = rospy.Publisher('rolling_demo', String)
    rospy.init_node('rolling_demo')

    while not rospy.is_shutdown():
        rospy.sleep(0.001)

if __name__ == '__main__':
    try:
        mainloop()
    except rospy.ROSInterruptException: pass
```

1.3.2 Landing

```
#!/usr/bin/env python
from __future__ import print_function
import roslib; roslib.load_manifest('roscopter')
import rospy
from std_msgs.msg import String
from sensor_msgs.msg import NavSatFix, NavSatStatus, Imu
import roscopter.msg
import sys,struct,time,os
import time

def vfr_hud(data):

    w = data.heading
    altitude = data.alt

    error = 0 - altitude
    error_w = 0 - w
    action = kp * error
    action_w = kp * error_w

  Proportional = kp * error
    Proportional_w = kp * error_w

  action = Proportional
    action_w = 10 * Proportional_w

    rc_action = 1000 + action
    rc_action_w = 1500 + action_w
    channel = [0,0,rc_action,rc_action_w,0,0,0,0]

  security_vel(channel)

kp = 0.5
vfr_hud_sub = rospy.Subscriber("vfr_hud", roscopter.msg.VFR_HUD, vfr_hud)

security = 0
pub_rc = rospy.Publisher('send_rc', roscopter.msg.RC)
rc_sub = rospy.Subscriber("rc", roscopter.msg.RC, rc)

def mainloop():

    pub = rospy.Publisher('landing', String)
    rospy.init_node('landing')

    while not rospy.is_shutdown():
        rospy.sleep(0.001)

if __name__ == '__main__':
    try:
        mainloop()
    except rospy.ROSInterruptException: pass
```

Appendix B

Glossary

AUV Autonomous Unmanned Vehicle

BSD Berkeley Software Distribution

Δ Amount of Increase

ESC Electronic Speed Controller

GPS Global Positioning System

IMU Inertial Measurement Unit

κ_p Proportional Constant used in control

κ_i Integrative Constant used in control

κ_d Derivative Constant used in control

Ω Angular Speed

$\ddot{\Phi}$ Angular Speed in X direction

$\ddot{\Psi}$ Angular Speed in Z direction

PID Proportional-Integral-Derivative controller

UAV Unmanned Aerial Vehicle

UWV Unmanned Water Vehicle

ROS Robotic Operating System

SLAM Simultaneous localization and mapping

$\ddot{\Theta}$ Angular Speed in Y direction

References

[1] Stäubli, "PUMA," http://www.staubli.com/en/robotics/.

[2] K. Robotics, "Kuka," www.kuka-robotics.com/en/.

[3] R. D. Leighty and ARMY ENGINEER TOPOGRAPHIC LABS FORT BELVOIR, *DARPA ALV (Autonomous Land Vehicle) Summary*. Defense Technical Information Center, 1986.

[4] F.-H. Hsu, *Behind Deep Blue: Building the Computer that Defeated the World Chess Champion*. Princeton University Press, Sept. 2002.

[5] B. Llc, *Robotics at Hond: Asimo, Honda E Series, Humanoid Robotics Project, Honda P Series*. General Books LLC, 2010.

[6] iRobot, "Roomba." http://www.irobot.com/us/.

[7] US Air Force, "MQ-1B PREDATOR." http://www.af.mil/information/factsheets/factsheet.asp?fsID=122.

[8] Boeing, "Boeing X-45," http://www.boeing.com/boeing/history/boeing/x45_jucas.page.

[9] navaldrones, "ACTUV," http://www.navaldrones.com/ACTUV.html.

[10] Boston Dynamics, "BigDog," http://www.bostondynamics.com/.

[11] A. Ollero, M. Bernard, M. La Civita, L. Van Hoesel, P. Marron, J. Lepley, and E. de Andres, "AWARE: Platform for Autonomous self-deploying and operation of Wireless sensor-actuator networks cooperating with unmanned AeRial vehiclEs," in *Safety, Security and Rescue Robotics, 2007. SSRR 2007. IEEE International Workshop on*, pp. 1–6, 2007.

[12] (IFM-GEOMAR), Leibniz Institute of Marine Sciences, "Fleet of high-tech robot 'gliders' to explore oceans," jan 2010.

[13] S. Thrun, "Sebastian Thrun: Google's driverless car [Video file]." Retrieved from http://on.ted.com/driverless, Mar. 2011.

[14] Scaled composites, "Bell Eagle Eye TiltRotor UAV." Retrieved from http://web.archive.org/web/20070113222345/http://www.scaled.com/projects/eagleye.html.

[15] OLIVE-DRAB, "RQ-7 Shadow UAV." Retrieved from http://olive-drab.com/idphoto/id_photos_uav_rq7.php.

[16] NewEagleWiki, "Unmanned Systems," http://www.neweagle.net/support/wiki/index.php?title=Unmanned_Systems.

[17] K. Alexis, G. Nikolakopoulos, A. Tzes, and L. Dritsas, "Coordination of Helicopter UAVs for Aerial Forest-Fire Surveillance," in *Applications of Intelligent Control to Engineering Systems* (K. Valavanis, ed.), vol. 39 of *Intelligent Systems, Control, and Automation: Science and Engineering*, pp. 169–193, Springer Netherlands, 2009.

[18] V. Kumar, "Vijay Kumar: Robots that fly ... and cooperate [Video file]." Retrieved from http://www.ted.com/talks/lang/es/vijay_kumar_robots_that_fly_and_cooperate.html, Feb. 2012.

[19] F. Caballero, L. Merino, J. Ferruz, and A. Ollero, "Vision-Based Odometry and SLAM for Medium and High Altitude Flying UAVs," *Journal of Intelligent and Robotic Systems*, vol. 54, no. 1-3, pp. 137–161, 2009.

[20] A. Nemra and N. Aouf, "Robust cooperative UAV Visual SLAM," in *Cybernetic Intelligent Systems (CIS), 2010 IEEE 9th International Conference on*, pp. 1–6, 2010.

[21] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "ROS: an open-source Robot Operating System," in *ICRA Workshop on Open Source Software*, 2009.

[22] pechkas@gmail.com and juancamilog@gmail.com, "ROS interface for Arducopter using Mavlink 1.0 interface." Retrieved from https://code.google.com/p/roscopter/.

[23] T. Graber, S. Kohlbrecher, J. Meyer, K. Petersen, O. von Stryk, and U. Klingauf, "RoboCupRescue 2012 - Robot League Team Hector Darmstadt (Germany)," tech. rep., Technische Universitẗ Darmstadt, 2012.

[24] W. R. Hamilton, "On Quaternions, or on a New System of Imaginaries in Algebra," *Philosophical Magazine*, vol. 25, no. 3, pp. 489–495, 1844.

[25] L. Euler, "Formulae generales pro translatione quacunque corporum rigidorum," pp. 189–207, 1776. http://math.dartmouth.edu/~euler/docs/originals/E478.pdf.

[26] G. Hoffmann, D. Rajnarayan, S. Waslander, D. Dostal, J. S. Jang, and C. Tomlin, "The Stanford testbed of autonomous rotorcraft for multi agent control (STARMAC)," in *Digital Avionics Systems Conference, 2004. DASC 04. The 23rd*, vol. 2, pp. 12.E.4–121–10 Vol.2, 2004.

[27] J. Macdonald, R. Leishman, R. Beard, and T. McLain, "Analysis of an Improved IMU-Based Observer for Multirotor Helicopters," *Journal of Intelligent & Robotic Systems*, pp. 1–13, 2013.

[28] R. Faludi, *Building Wireless Sensor Networks: with ZigBee, XBee, Arduino, and Processing*. O'Reilly Media, 2010.

[29] L. Meier, "MAVLink Micro Air Vehicle Communication Protocol." http://qgroundcontrol.org/mavlink/start, 2009.

[30] L. Meier, "Ground Control Station for small air-land-water autonomous unmanned systems." http://qgroundcontrol.org/about, 2009.

[31] H. Lim, J. Park, D. Lee, and H. Kim, "Build Your Own Quadrotor: Open-Source Projects on Unmanned Aerial Vehicles," *Robotics & Automation Magazine, IEEE, vol.19, no.3, pp.33-45*, Sept. 2012. http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6299172&isnumber=6299141.

[32] K. Ogata, *Modern Control Engineering*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 4th ed., 2001.