UNIVERSITA'
DEGLI STUDI
DI BERGAMO

DIPARTIMENTO DI INGEGNERIA

Ingegneria Informatica

ARDrone Parrot 2.0: Marker detection and following

Yamuna Maccarana - matr.1014766
Luca Calomeni - matr.1010939
Manuel Facchinetti - matr.1011222

ANNO ACCADEMICO 2014/2015

Contents

List of Figures

Chapter 1

Introduction

This project has been developed as completion to the course of Robotics with Professor Brugali, at the University of Bergamo, placed in Dalmine. It investigates interesting aspects of software architectures for robot navigation, of 3D mapping with mobile robots and of mobile robots' kinematic.

Through the building of this project, important matters such as component based software architectures, middleware for distributed control system, software model based planning, paradigms control for autonomous robots, mobile robots' kinematic and sensors for movement calculus (odometry), marker based localization, 3D mapping, movement planning and robots' axes control have been examined in depth.

As far as it is a completion to the course, this project has been planned as a robust and flexible starting point for further extensions and adaptation. In fact, with this manual the user will be able to understand each step of the project and, moreover, they will be capable to extend or modify this tool for a specific need.

Acknowledgements First of all, we would like to thank our teacher, Davide Brugali, Associate Professor at University of Bergamo, who started us toward this project, stimulated us and attended upon us for the whole duration of our work. Then, we mutually want to thank ourselves, because we found friends more than workmates in each other, and this made things much easier and sometimes even hilarious.

Figure 1.1: Universitatis bergomensis studium logo.

Chapter 2

Foreword

2.1 Aims

The aim of this project is to provide a basic flexible tool to control ARDrone Parrot 2.0 and similar devices. This software is meant to give the user a solid start point for further extensions.

In chapter 3 background knowledges are exhaustively explained. This project has been conceived for a robotic environment.

2.2 Equipment

Attached to this documentation there is some software containing the unibg.robotics.ardrone.marker.follower implementation. In case of loss, it is possible to find repositories and documentation online[1].

For further information about configurations and used libraries see chapter 5.

As the entire project has been developed to work on a quadricopter, make sure that your device is similar to ARDrone Parrot 2.0[2].

Chapter 3

Background

Before getting to the heart of the matter, it is helpful to introduce basic concepts that are essential to be known and understood. Here are overviews of libraries, architectures and hardware related to this project.

3.1 ARDrone Parrot

The ARDrone (fig.3.1) is a quadricopter comprising four engines, the chassis, of cross shape, which task is to host the electronic board in its center and the motors in each one of the four extremities. The chassis appears to be light and strong so it is suitable to support the entire structure of the quadricopter.

Figure 3.1: ArDrone Parrot.

Each engine mounts a reduction gear and a propeller. The propellers are divided into two groups, each of which shall belong to two diametrically opposite axis. Their distinction is made according to their direction of rotation. For this purpose, the

propellers are distinguished:

- front and rear with counterclockwise rotation;
- lateral with clockwise rotation.

3.1.1 Embodiment

Talking about specific features of ARDrone Parrot 2.0 it is possible to understand why it is suitable for testing this project. Hot topics are:

- chassis;
- cover;
- motors;
- battery;
- motherboard;
- navigationboard.

Next, the main features of the hardware device will described.

Chassis The entire structure is supported by means of a chassis (fig.3.2), to form a cross, in carbon, which task is to accommodate the four brushless motors ends, and a poly-propylene foam (EPP) structure in the center.

Figure 3.2: Chassis.

Cover, indoor and outdoor Externally, two covers can be applied (fig.3.3). The first application is for indoor spaces and is used to prevent the propellers scratching against the walls. The second cover protects the battery and is used for outdoor applications.

Figure 3.3: Covers.

Motors The four brushless motors, which are connected to the propellers by means of a organ reduction (fig.3.4), are used to perform the maneuvers of quadricopter. Each motor is mounted on a plastic material highly resistant PA66, which contains a control board of the motor (brushless BLCB control board); overall, the system is also equipped with a cut-out function which stops the four motors in the case one of the propellers encounters an obstacle.

Figure 3.4: Motors and propellers.

Battery The ARDrone uses a charged 1500mAh, 11.1V LiPo battery (fig.3.5) to fly. When the drone detects a low battery voltage, it first sends a warning message to the user, then automatically lands. If the voltage reaches a critical level, the whole system is shut down to prevent unexpected behaviours.

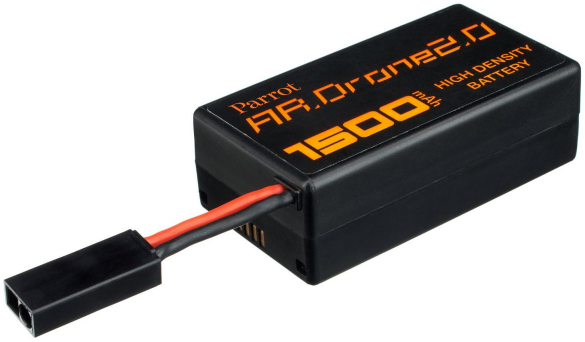

Figure 3.5: Battery.

MotherBoard The motherboard (fig.3.6) is composed of:

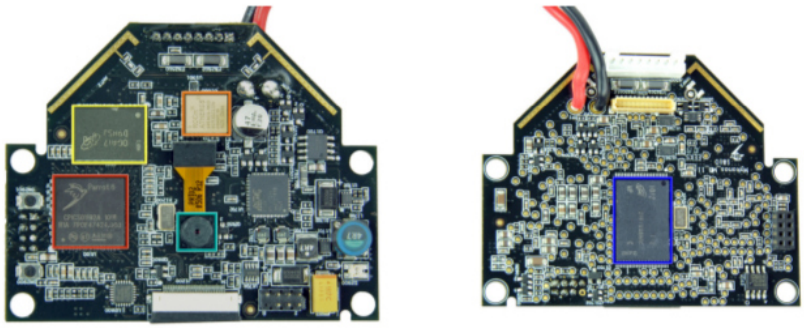

Figure 3.6: MotherBoard

- a DRAM cache;
- a Wi-Fi unit, ROCm Atheros AR6102G-BM2D wireless;

- a processor Parrot P6 (32bit ARM9-core, with frequency to 468MHz), which runs on a real-time operating system based on Linux; its job is to manage resources and acquire the data stream coming from the two cameras and process it;
- a vertical camera, used in navigation algorithms to measure the vertical velocity;
- a connector for the front camera, which has an objective wide angle with a fov of 75° × 60°; its output is a signal with VGA resolution of 1280 x 720 pixels, and 30 fps (for more specifics see next chapter);
- a SMSC USB3317 USB driver for the mini USB connector useful for connect additional add-ons like GPS.

NavigationBoard The NavigationBoard (fig.3.7) uses a 16-bit PIC microcontroller (highlighted with red border) with a frequency of 40 MHz, that is the interface to sensors: 3-axis accelerometer, 2-axis gyroscope vertical precision and two ultrasound sensors.

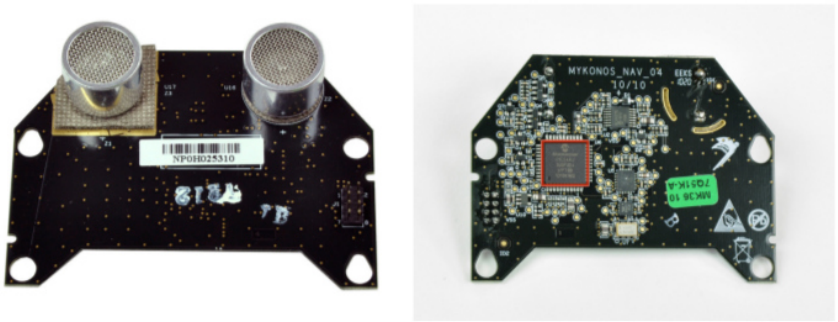

Figure 3.7: NavigationBoard

Ultrasonic sensors are used to estimate the altitude, vertical movements and the depth-of the scene. It can measure distances of 6 m at a frequency of 25 Hz. The PIC microcontroller manages the ultrasonic transmitter and digitizes the signal receiver. The accelerometer is a BOSH BMA150 3-axis which uses a A/D converter 10 bits, and has a range of +/- 2g. Its return value corresponds to the acceleration of the gravity minus the quadricopter's. The two-axis gyroscope is a Invensense IDG500, which Analog signal is digitized by A/D converter 10-bit PIC, and can measure the velocity of rotation up to 500°/s. A gyroscope, more accurate on the vertical axis, is an Epson XV3700, with auto-zero to minimize the drift route.

3.1.2 Camera

In our project, the front camera is one of the main components because it captures and sends images to our system. Acquired images are processed by the system, that will pilot the drone via topic publications. The ARDrone Parrot 2.0 has a HD Camera with specific features such as the following ones:

- 720p - 30FPS;
- wide angle lens: 92° diagonal;
- H264 encoding base profile;
- low latency streaming;
- JPEG photo capture.

More than numbers, what is needed is the possibility to acquire data and images faster than the process of movement of the drone. This is an essential feature for this software to have the certainty of a correct functionality.

Thanks to its 720p, 30FPS and to its low latency streaming, the front camera of ARDrone Parrot 2.0 is the perfect candidate to test the correct functionality of our project.

However, it is necessary to highlight the fact that the above features are the ones of the front camera; instead, the bottom camera is a lower quality camera and does not satisfy our prerequisites due to its low definition.

Camera axes The drone camera has a standard reference system, which z axis comes out of the center of camera. The camera reference system is shown in figure 3.8.

The information acquired is processed by the system which considers the relation between camera and drone system (depicted in fig.3.9), converting them correctly.

The system, via tf data, relates the captured images reference system with the drone one.

tf *tf* is a package that lets the user keep track of multiple coordinate frames over time. It maintains the relationship between coordinate frames in a tree structure buffered in time, and lets the user transform points, vectors, and so on, between any two coordinate frames at any desired point in time.

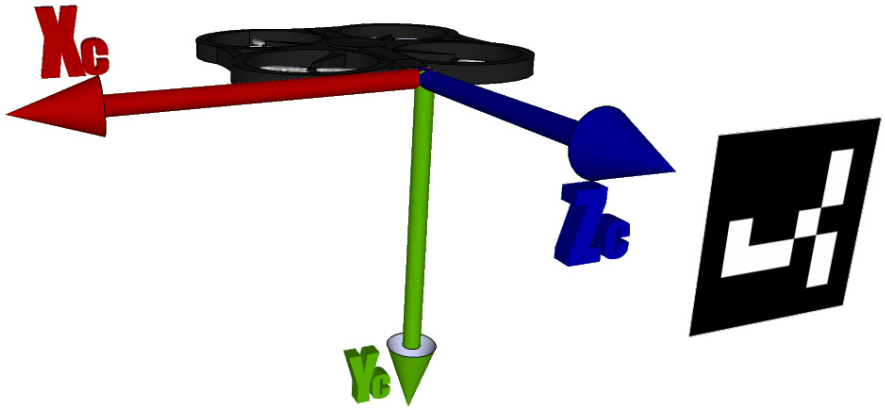

Figure 3.8: Camera system reference

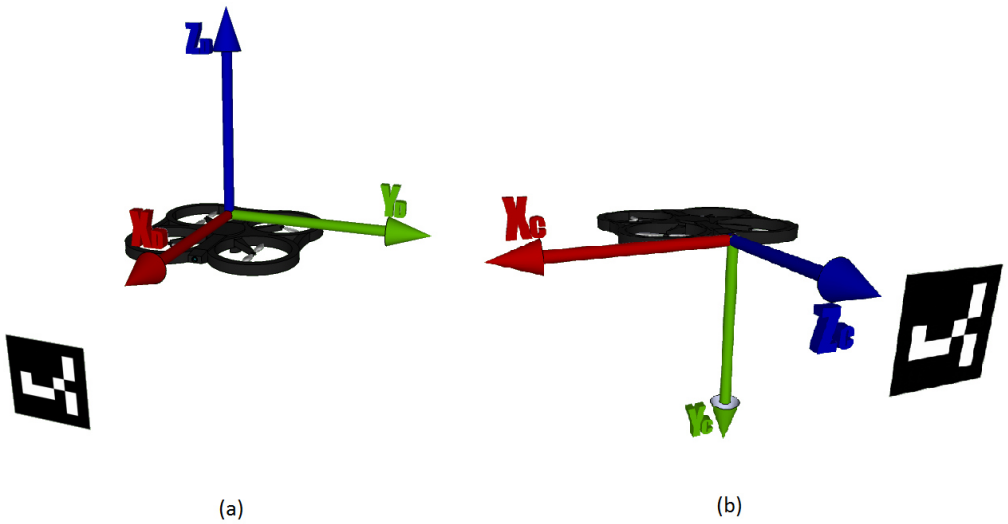

Figure 3.9: (a)Drone system reference (b)Camera system reference

3.1.3 Wi-Fi network and connection

The ARDrone can be controlled from any client device supporting the Wi-Fi ad-hoc mode. The following process explains its functionality:

- the ARDrone creates a Wi-Fi network with an ESSID usually called ardrone2_parrot and self allocates a free, odd IP address;
- the user connects the client device to this ESSID network;
- the client device requests an IP address from the drone DHCP server;
- the AR.Drone DHCP server grants the client with an IP address which is 192.168.1.x;
- the client device can start sending requests the AR.Drone IP address and its services ports. The client can also initiate the Wi-Fi ad-hoc network. If the drone detects an already-existing network with the SSID it intented to use, it joins the already-existing Wi-Fi channel.

3.1.4 Kinematics

A quadrotor[8] consists of a rigid body driven by four propellers (see fig. 3.10), each one of them developing a mechanical force T_i, and a torque M_i. It is generally ac-

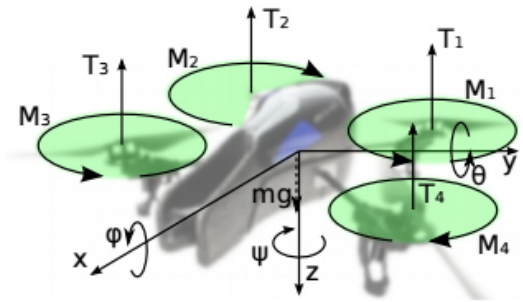

Figure 3.10: Kinematics

cepted to approximate their mechanical effort as $T_i = kT\omega^2$ and $M_i = kM\omega^2$, where ω, kM and kT are the propeller speed and two propeller characteristic constants. The usual mid-level control approach involves using: the average thrust $\sum_{i=1}^{4} T_i$ as

control variable for the altitude, the average torque $\sum_{i=1}^{4} M_i$ as control variable for the yaw heading of the vehicle, and the roll and pitch angles that tilt the average thrust to obtain two linear acceleration commands in the horizontal plane.

There are four possible basic movements of quadricopter namely: throttle, pitch, roll, yaw, depending on the speed of rotation of each propeller.

Throttle (fig.3.11) is defined as the vertical displacement along the z axis (the reference system integral with the quadricopter). In the case of flight fixed point (hovering mode) the angular velocity of the four propellers is identical. Same for the acceleration in the case in which the quadricopter increases (salt) or decreases (down) its share.

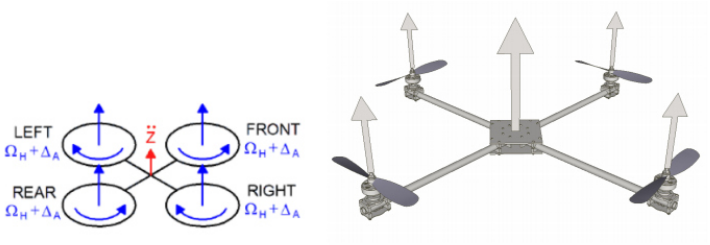

Figure 3.11: Throttle.

The rotation around the x axis (the system reference integral with the quadricopter) is called **Pitch** (fig.3.12). This movement takes place, increasing the speed of one of the engines placed on the x axis, decelerating diametrically opposite that of the motor, and maintaining a constant speed of the other pair of engines.

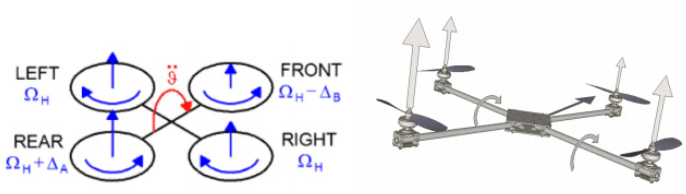

Figure 3.12: Pitch.

Roll (fig.3.13) is defined as rotation around the y axis (of the system of reference integral with the quadricopter). The principle of generation of such movement is the same pitch, but in this case the motors along the y axis vary the speed.

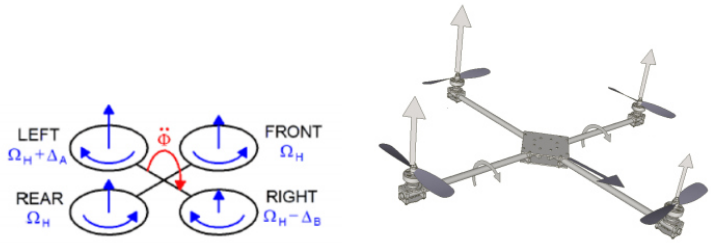

Figure 3.13: Roll.

If the goal is to maintain the constant height and obtain a rotation around the z axis (the reference system integral with the quadricopter) is the case of **Yaw** (fig.3.14). It increases the speed of rotation of the motors along the x axis, while it decreases that of motors along the y axis by the same amount, or vice versa depending on whether you wants a rotation positive or negative.

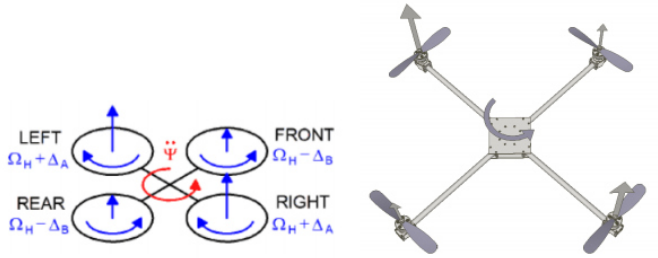

Figure 3.14: Yaw.

The max speed measured was 11.11m/s and the altitude was 199.03m.

3.2 Robot Operating System (ROS)

The robotics community has made impressive progress in recent years. Reliable and inexpensive robot hardware, from land-based mobile robots, to quadrotor helicopters, to humanoids, is more widely available than ever before. Perhaps even more impressively, the community has also developed algorithms that help those robots run with increasing levels of autonomy. This project introduces a software platform called Robot Operating System, or ROS, that is intended to ease some of these difficulties. The official description of ROS[3] is:

ROS is an open-source, meta-operating system for robot. It provides the services expected from an operating system, including hardware abstraction, low level device control, implementation of commonly used functionality, message passing between processes, and package management. It also provides tools and libraries for obtaining, building, writing, and running code across multiple computers.

Thus, the Robot Operating System is a component-based infrastructures provided to control lower part of robots (hardware). It provides libraries and tools to help software developers create robot applications.

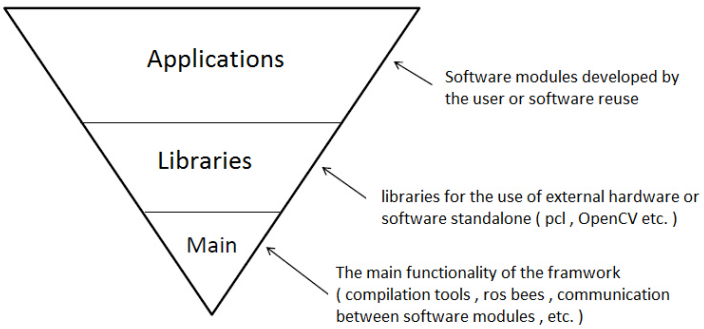

Figure 3.15: Component-based infrastructures.

Roscore The roscore is the fundamental element that is responsible for coordinating the nodes connected to it by managing parallel execution and communication between them.

Rosnode The rosnode consists of code developed by the user, or is a software package provided by ROS.

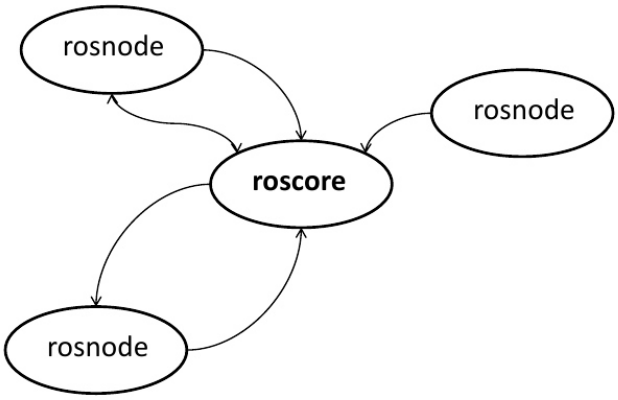

Figure 3.16: Main elements ROS Architecture.

Communication Protocols Communication is one of the most important elements of the system ROS. Easy exchanging of messages between nodes makes it possible to program multi-threading, ignoring the issues related to policy Up Sync and communication between processes. ROS could be useful due to its message-based publish-subscriber peer-to-peer communication.

Publish/subscribe: writing a message on a topic provided by the roscore. All nodes that want to receive the message can request it to roscore.

Service: a node sends a request to all nodes able to satisfy it. These nodes will receive a response.

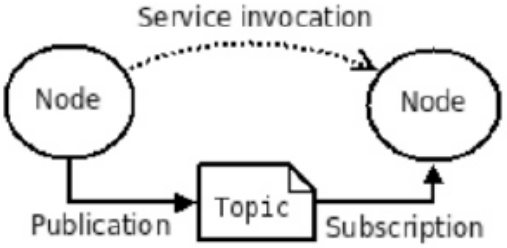

Figure 3.17: Message-based publish-subscriber peer-to-peer communication.

3.3 ArUco

The marker detection process of ArUco is described next:

- apply an Adaptive Thresholding so as to obtain borders (fig.3.18);

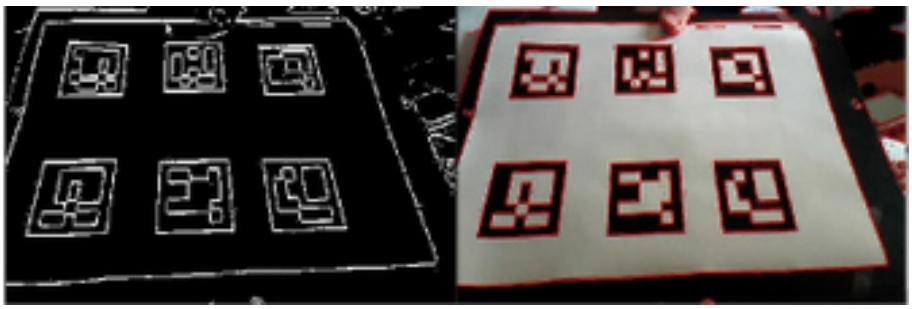

Figure 3.18: Adaptive Thresholding:obtain borders.

- find contours. After that, not only the real markers are detected but also a lot of undesired borders. The rest of the process aims to filter out unwanted borders;

- remove borders with an small number of points (fig.3.19);

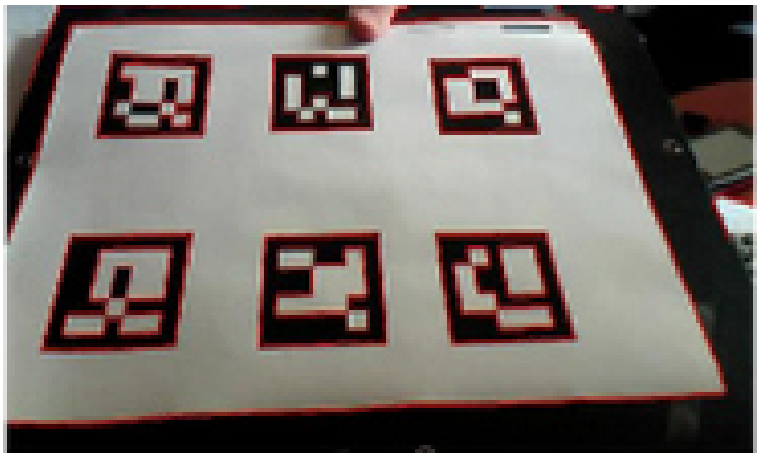

Figure 3.19: Filter out unwanted borders.

  Polygonal approximation of contour and keep the concave contours with exactly 4 corners (i.e., rectangles) (fig.3.20).

- sort corners in anti-clockwise direction;

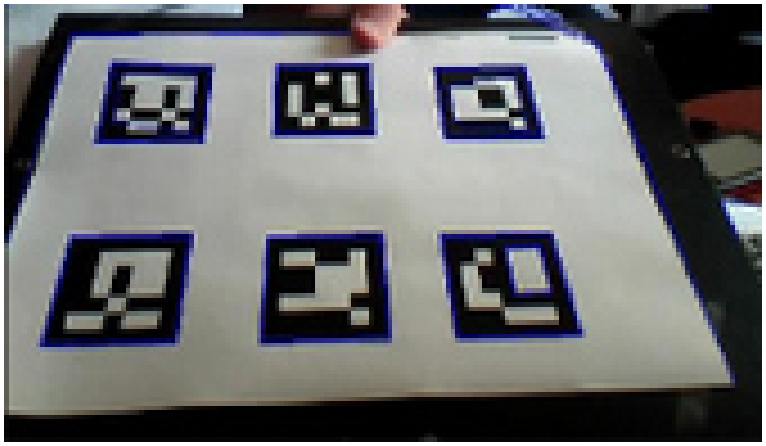

Figure 3.20: Concave contours.

- remove too close rectangles. This is required because the adaptive threshold normally detects the internal an external part of the marker border. At this stage, we keep the most external border (fig.3.21);

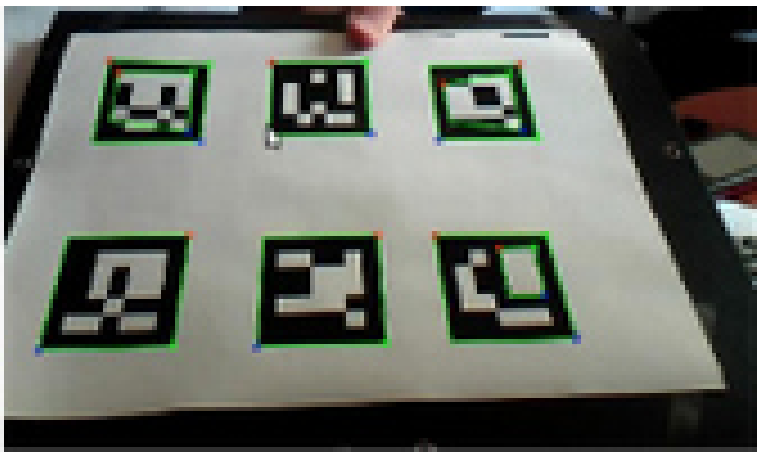

Figure 3.21: External border.

- marker identification, consisting of:

  1. removing the projection perspective so as to obtain a frontal view of the rectangle area using an homography (fig.3.22);
  2. thresholding the area using Otsu. Otsu's algorithms assumes a bimodal distribution and finds the threshold that maximizes the extra-class variance while keeping a low intra-class variance;
  3. identification of the internal code. If it is a marker, then it has an internal code. The marker is divided in a 6x6 grid, of which the internal 5x5

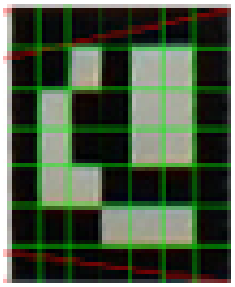

Figure 3.22: External border detail.

cells contains id information. The rest corresponds to the external black border. Here, first it is checked that the external black border is present. Afterwards, the internal 5x5 cells are read and it is checked if they provide a valid code (it might be required to rotate the code to get the valid one). For the valid markers, corners are refined using subpixel interpolation. In the end, if camera parameters are provided, it is computed the extrinsic of the markers to the camera.

Alternatively it is possible to use ARToolkit[6] but in this project we preferred to use Aruco as it is the most recent and most documented library.

3.3.1 Markers

Each marker has an internal code given by 5 words of 5 bits each. The codification employed is a slight modification of the Hamming Code. In total, each word has only 2 bits of information out of the 5 bits employed. The other 3 are employed for error detection. As a consequence, we can have up to 1024 different ids. The main difference with the Hamming Code is that the first bit (parity of bits 3 and 5) is inverted. So, the id 0 (which in hamming code is 00000) becomes 10000 in our codification. The idea is to prevent a completely black rectangle from being a valid marker id with the goal of reducing the likelihood of false positives with objects of the environment. Due to ARDrone Parrot 2.0 camera definition, it is suggested to use 16x16 cm markers to allow their identification at a satisfying distance: small markers may not be detected if too far from the Drone.

Chapter 4

Case Study

In this project, a software application that allows the ARDrone to autonomously recognize and follow a given marker, maintaining a specific distance, is developed. In our case study, the used marker has id=36, but for any future developments on the project, this value can be easily modified to recognize markers with different id.

The structure of the software has been developed with an agile approach, which allows the addition of new features.

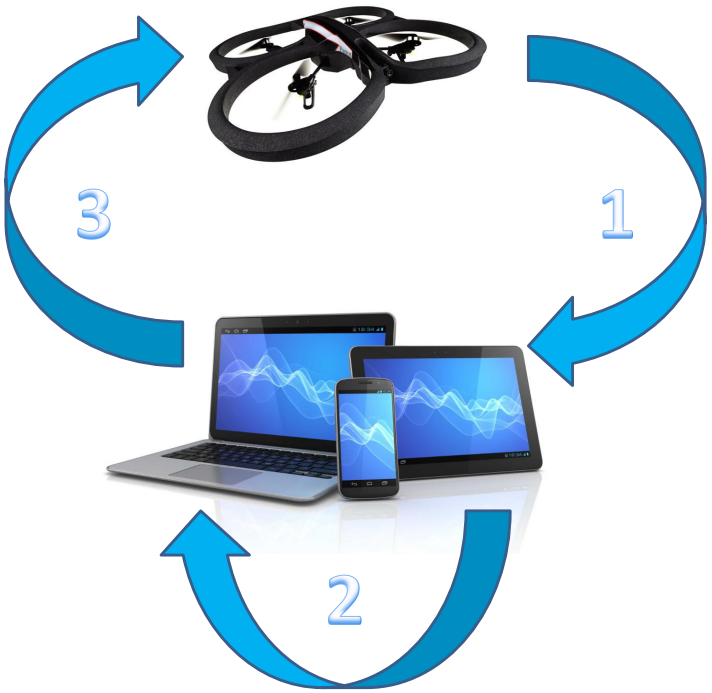

Figure 4.1: Ardrone-pc data flow.

4.1 Marker based navigation

The project is based on four software modules: Parrot Driver, Camera Driver, Marker Detector and Follower.

Camera Driver and Parrot Driver are two software components (fig.4.2) which are provided by the *ardrone_autonomy* library and allow the exchange of information with the drone through the ROS ardrone_driver node, which is the driver's executable node. The exchange of messages is done on appropriate topics. Marker Detector and Follower are ROS nodes created specifically for this project purpose, which communicate with the root node (roscore).

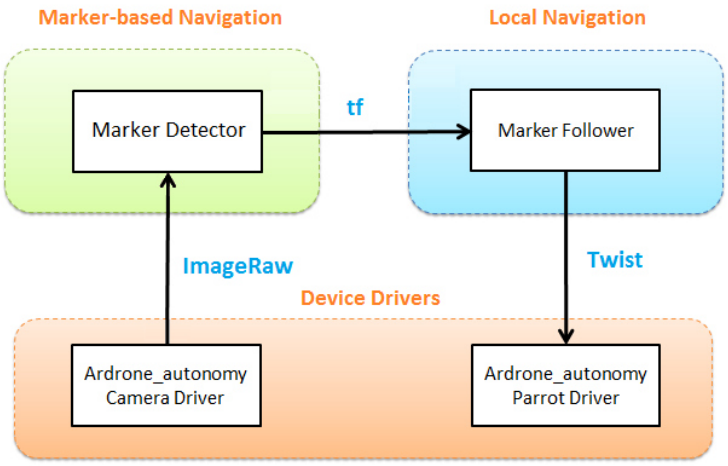

Figure 4.2: Marker based navigation.

During the development phase of the project a problem with device connection came up. In fact, during communication between Drone and PC there was a delay that compromised the correct sending of commands caused by drone Wi-Fi connection problem. The phenomenon can be seen from the behaviour of the quadricopter which does not properly follow the marker. However, in ideal conditions, without lag, tracking is correct.

Ardone_autonomy Camera Driver First, the driver node sends to the */ardrone/front/image_raw* topic the captured and processed image by the drone camera (fig.4.3).

Figure 4.3: ARDrone to pc data flow.

Marker Detector Then, the marker detector takes as input the captured image and its processing by subscribing to the /ardrone/front/image_raw topic (fig.4.4).

Figure 4.4: Pc data flow.

Extrapolating the tf data through ArUco, this node is able to recognize the id of the selected marker and calculate a correct transform frame.

Marker Follower The Marker Follower receives the tf and processes it (for further information see chap.6.3.1). Still, the computation logic is depicted in fig.4.4. When the node has calculated the position of the marker, it converts position and axis and sends a new velocity command to the Parrot Driver on */cmd_vel* topic (fig.4.5).

Figure 4.5: Pc to ARDrone data flow.

Ardone_autonomy Parrot Driver Parrot Driver is the node that drives the motors of the drone, it receives the twist and makes sure that the quadricopter moves to the new location. A summit of the logic can be seen in fig.4.6.

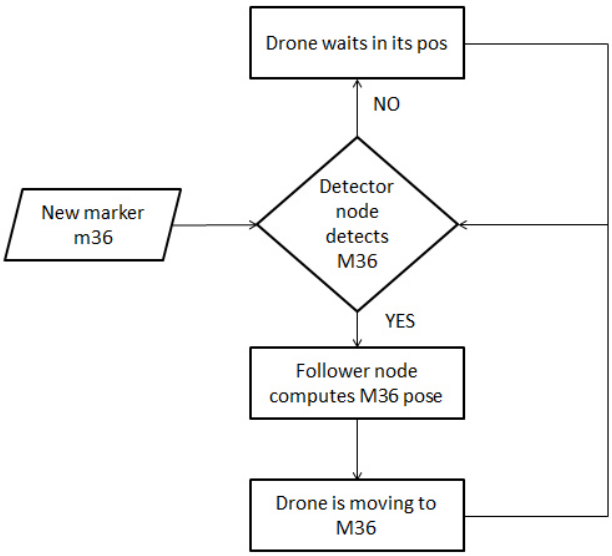

Figure 4.6: Marker detection flowchart.

For further information, please see chapter 6.

4.2 Environment

Due to its camera definition (recall chap.3.1.2), in normal light condition, it is verified that a 16x16cm marker can be detected at a finite distance and with a certain angle. More in details, since the camera of the drone drone has a wide-angle lens with field amplitude equal to 92 degrees, this means that the marker to identify must fall within +/- 46 degrees from the focal point of the camera. As regards the maximum distance to which the marker is seen clearly depends on the depth of field. The depth of field is affected by three factors: focal length, diameter of the diaphragm and camera distance. With a marker of at least 16x16cm it was verified that the maximum distance to which it is correctly identified, in the ideal condition, is about 3m, as explained in next section.

4.3 ARDrone Parrot 2.0 viewing angle

The ARDrone Parrot 2.0 viewing angle is depicted in fig.4.7.

In particular, the maximum distance for the marker detection is 3 meters, with a deferment of 40cm in both up and down directions and of 60cm in both left and right directions.

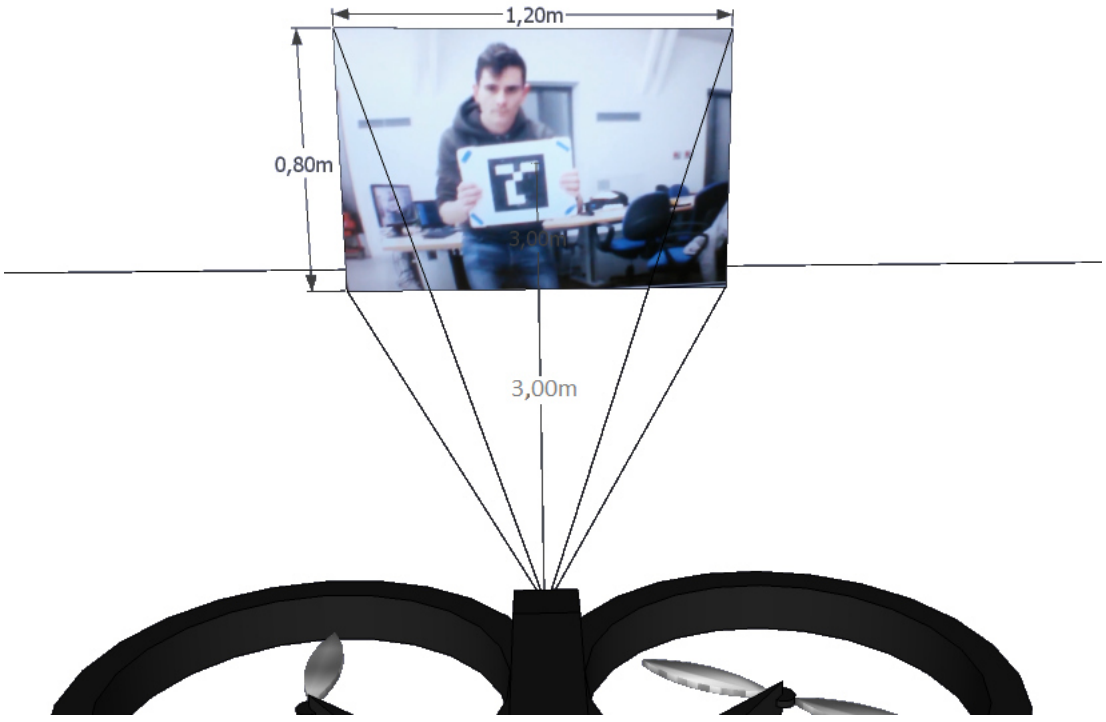

Figure 4.7: ARDrone effective field of view.

Chapter 5

Configuration and folder explanation

5.1 Operating Conditions

Operating system used: Ubuntu 14.04 64 bit.

Other supported operating systems: systems with ROS installed and Wifi communication unit.

Drone: ARDrone Parrot 2.0[2].

Other supported drones: similar to ARDrone (in term of communication and dynamics, topics and commands).

5.2 Ubuntu Repositories

First, please configure your Ubuntu repositories to allow "restricted", "universe", and "multiverse". You can follow the Ubuntu guide[9] for instructions on doing this.

Then, setup your computer to accept software from packages.ros.org with the sequent line of code:

$ sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu trusty main" > /etc/apt/sources.list.d/ros-latest.list'

Then set your keys as follows:

$ wget https://raw.githubusercontent.com/ros/rosdistro/master/ros. key -O - | sudo apt-key add -

Before proceeding with the installation, make sure that the Debian package index is up-to-date:

$ sudo apt-get update

If you are using Ubuntu Trusty 14.04.2, you may have to install some additional system dependencies:

$ sudo apt-get install xserver-xorg-dev-lts-utopic mesa-common-dev-lts-utopic libxatracker-dev-lts-utopic libopenvg1-mesa-dev-lts-utopic libgles2-mesa dev-lts-utopic libgles1-mesa-dev-lts-utopic libgl1-mesa-dev-lts-utopic libgbm-dev-lts-utopic libegl1-mesa-dev-lts-utopic

5.3 Libraries and Tools in ROS

There are many different libraries and tools in ROS. We provided four default configurations to get you started. You can also install ROS packages individually. Desktop-Full Install: (Recommended) : ROS, rqt, rviz, robot-generic libraries, 2D/3D simulators, navigation and 2D/3D perception.

Command reported here:

$ sudo apt-get install ros-indigo-desktop-full

5.4 Environment setting

To find all available packages, use:

$ apt-cache search ros-indigo

Before you can use ROS, you will need to initialize rosdep. rosdep enables you to easily install system dependencies for source you want to compile and it is required to run some core components in ROS. To do so, please use:

$ sudo rosdep init

$ rosdep update

It is convenient if the ROS environment variables are automatically added to your bash session every time a new shell is launched:

$ echo "source /opt/ros/indigo/setup.bash" » ∼/.bashrc

$ source ∼ /.bashrc

To change the environment of the current shell, enter:

$ source /opt/ros/indigo/setup.bash

A useful tool for work into ROS is rosinstall. Rosinstall is a frequently used command-line tool in ROS that is distributed separately. It enables you to easily download many source trees for ROS packages with one command. To install this tool on Ubuntu, run:

$ sudo apt-get install python-rosinstall

5.5 Installing OpenCV

OpenCV (Open Source Computer Vision) is a library of programming functions mainly aimed at real-time computer visio. To install this library, follow these steps:

1. Open a terminal.
2. Install some packages that OpenCV neeeds:

   $ sudo apt-get update && sudo apt-get upgrade

   $ sudo apt-get install build-essential libgtk2.0-dev libjpeg-dev libtiff4-dev libjasper-dev libopenexr-dev cmake python-dev python-numpy python-tk libtbb-dev libeigen2-dev yasm libfaac-dev libopencore-amrnb-dev libopencore-amrwb-dev libtheora-dev libvorbis-dev libxvidcore-dev libx264-dev libqt4-dev libqt4-opengl-dev sphinx-common texlive-latex-extra libv4l-dev libdc1394-22-dev libavcodec-dev libavformat-dev libswscale-dev
3. Download the OpenCV-2.4.10 package with the next command:

   $ wget http://sourceforge.net/projects/opencvlibrary/files
4. Extract the package.
5. Create a new folder that will help us to compile and install OpenCV:

   $ cd opencv-2.4.10

   $ mkdir build

   $ cd build
6. Compile and install OpenCV:

   $ cmake -D WITH_TBB=ON -D BUILD_NEW_PYTHON_SUPPORT=ON
   -D WITH_V4L=ON -D INSTALL_C_EXAMPLES=ON -D INSTALL_PYTHON_EXAMPI
   -D BUILD_EXAMPLES=ON -D WITH_QT=ON -D WITH_OPENGL=ON
   ..

   $ make

   $ sudo make install
7. Make sure you have installed everything correctly by running the following example:

   $ cd opencv-2.4.10/build/bin

   $ ./cpp-example-calibration_artificial

5.6 Installing ArUco

Before proceeding to install the ArUco library you need to install the OpenCV, if not already installed (please see chapter 5.5).

After installing OpenCV, you can proceed to install the ArUco library following these steps:

1. Open a terminal and download the ArUco library:

   $ wget http://sourceforge.net/projects/aruco/files/1.2.4/ aruco-1.2.4.tgz

2. Extract the package and create a new folder useful to compile and install:

   $ tar zxvf aruco-1.2.4.tgz

   $ cd aruco-1.2.4

   $ mkdir build

   $ cd build

3. Compile and install ArUco:

   $ cmake ..

   $ make

   $ sudo make install

4. Now test ArUco as follows:

   $ cd aruco-1.2.4/build/utils

   $ ./aruco_create_board 5:2 board.png board.yml

5. Next you will need to calibrate your camera with OpenCV. This process will generate a 'camera.yml' file in "OpenCV-directory/build/bin" that you have to copy and paste in "aruco-1.2.4/build/utils". To calibrate the room you need to follow the following steps:

   (a) Download the calibration board [7] (as the one in fig.5.1).

   (b) Open a terminal and go to path "opencv-2.4.10/build/bin":
   $ cd opencv-2.4.10/build/bin

   (c) Use:
   $ ./calibration 0 -w 9 -h 6 -s 0.025 -o camera.yml -op -oe
   where:

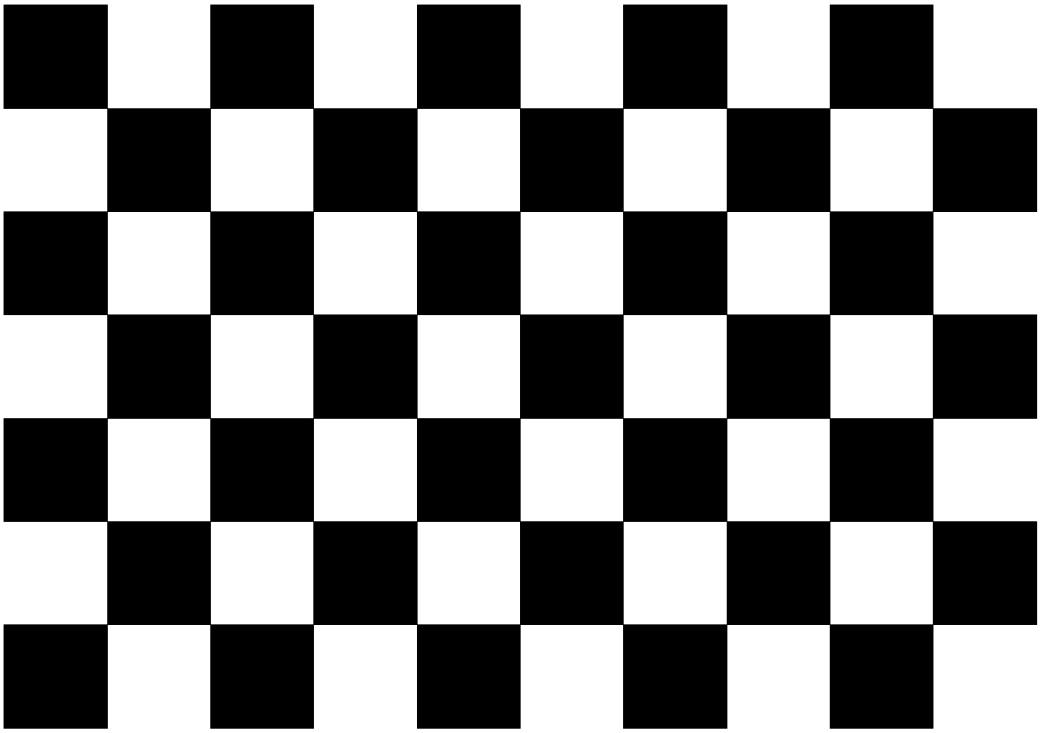

Figure 5.1: Calibration board.

i. 0 is the parameter of your camera. To see what is the number of your camera, open a new terminal and write: ls /dev/video*. If you see /dev/video0 the number you have to write is 0; for /dev/video1, number 1; and so on ...

ii. - w 9. Is the width of your board, in my case 9.

iii. - h 6. Is the height of your board.

iv. - s 0.024. Size in meters of the squares.

v. - o camera.yml. The output calibration file that will be created.

(d) Next, you will see a window with the vision of your camera. You will have to put the printed chessboard in front of your camera in different positions to calibrate (see fig.5.2).

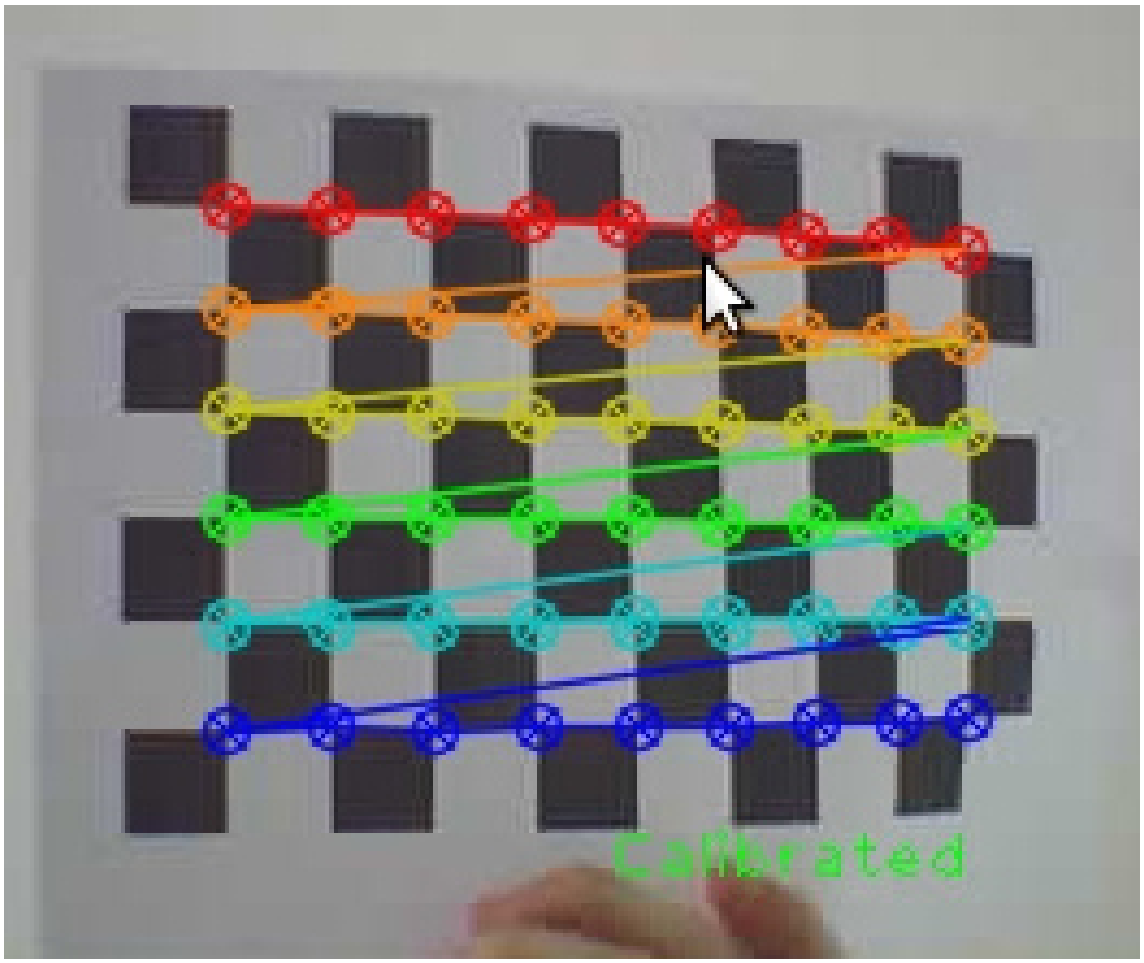

Figure 5.2: Calibration.

6. Next, you have to print the board that is in "aruco-1.2.4/build/utils/board.png" and execute the following commands to detect a single marker:

$ cd aruco-1.2.4/build/utils

$ ./aruco_test live camera.yml 0.025

The board "board.png" looks like fig.5.3 and the screen shot in fig.5.4 is the output of complete board, due to a correct configuration.

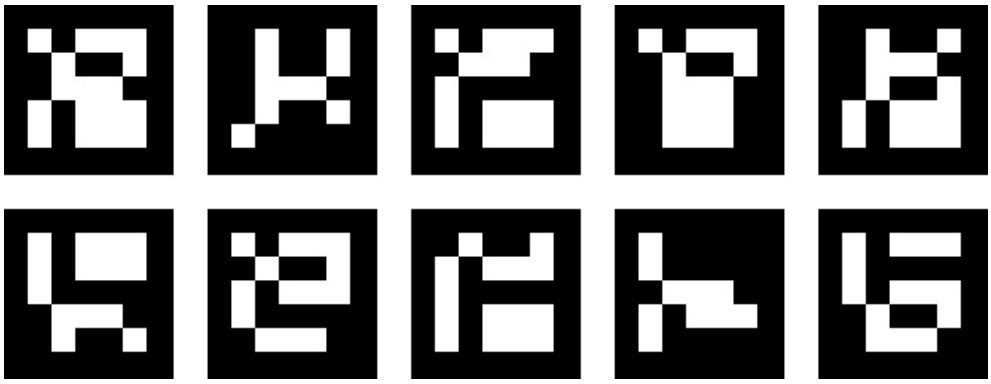

Figure 5.3: Board.

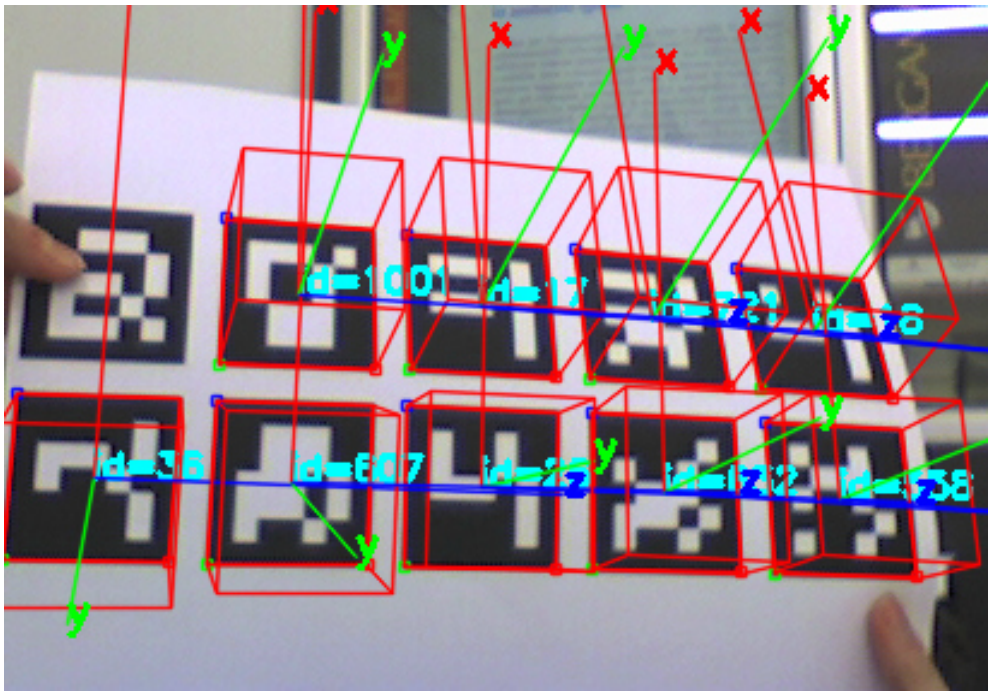

Figure 5.4: Output of complete board.

5.7 Workspace creation

Now, after installing ArUco and OpenCV library, it is possible to create a ROS workspace. Make sure that you have your environment properly setup. A good way to check is to ensure that environment variables like ROS_ROOT and ROS_PACKAGE_PATH are set:

$ export | grep ROS

To set the environment variables set setup.*sh files. To do so:

$ source /opt/ros/indigo/setup.bash

You will need to run this command on every new shell you open to have access to the ROS commands, unless you add this line to your .bashrc. To create a new workspace digit:

$ mkdir -p /catkin_ws/src

$ cd /catkin_ws/src

$ catkin_init_workspace

$ cd /catkin_ws/

$ catkin_make

Before continuing source the new setup.*sh file:

$ source devel/setup.bash

5.7.1 Package creation

First change to the source space directory of the workspace you created previously:

$ cd /catkin_ws/src

Now use the catkin_create_pkg script to create a new package named package_name which depends on std_msgs, nav_msgs, geometry_msgs,sensor_msgs rospy, rospy, roscpp, sensor_msgs and std_msgs tf:

$ catkin_create_pkg package_name std_msgs nav_msgs geometry_msgs sensor_msgs rospy roscpp sensor_msgs std_msgs tf

This will create a folder package_name which contains a package.xml and a CmakeList.txt, which has been partially filled out with the information given by catkin_create_pkg.

Now you need to build the packages in the catkin workspace:

$ cd /workspace

$ catkin_make

After the workspace has been built, a similar structure has been created in the devel subfolder as you usually find under /opt/ros/ROSDISTRO_NAME, in our case /opt/ros/indigo.

To add the workspace to your ROS environment you need to source the generated setup file:

$. /catkin_ws/devel/setup.bash

Before continuing remember to source your environment setup file if you have not already. On Ubuntu it would be something like this:

$ source /opt/ros/indigo/setup.bash

In the end, in the build folder of each package, one command is needed:

$ cmake ..

This will create the structure needed to execute the make command, which has to be launched after the src code modification:

$ make

Now an executable file is created in /build/devel/lib/package_name folder.

5.8 Installing ardrone autonomy

ardrone_autonomy[4] is a ROS driver for Parrot ARDrone quadricopter. This driver is based on official ARDrone SDK version 2.0 and supports both ARDrone 1.0 and 2.0.

Installation On supported Ubuntu platform and for ROS Indigo you can install the driver using:

$ apt-get install ros-*-ardrone-autonomy

where * indicates your distribution.

Compile and Install from Source The bundled ARDrone SDK has its own build system which usually handles system wide dependencies itself. The ROS package depends on these standard ROS packages: roscpp, image_transport, sensor_msgs, tf, camera_info_ manager and std_srvs. The installation follows the same steps needed usually to compile a ROS driver using catkin. Clone (or download and unpack) the driver to the src folder of a new or existing catkin workspace (e.g /catkin_ws/src) and use catkin_make to compile. Assuming you are compiling for ROS Indigo:

$ cd /catkin_ws/src

$ git clone https://github.com/AutonomyLab/ardrone_autonomy.git -b hydro-devel

$ cd /catkin_ws/catkin_make

If you need to compile for older ROS distros or you are bound to use rosbuild, please check the fuerte-devel branch.

How to run The driver's executable node is ardrone_driver. You can either use:

$ rosrun ardrone_autonomy ardrone_driver

or put it in a custom launch file with your desired parameters.

Topics list To check the correct installation and connection functionality, type:

$ rostopic echo /tf

to receive all tf data sent by the drone.

5.8.1 Deepening of the topics: tf, imageRaw

tf is a standardized protocol for publishing transform data to a distributed system, helper classes and methods for:

- publishing coordinate frame data: TransformBroadcaster;
- collecting transform data and using it to manipulate data: Transformer, TransformListener, tf::MessageFilter, ... ;
- currently: Only tree(s) of transformations, but any robot(s) / sensor layout;
- conversion functions, mathematical operations on 3D poses.

Both ARDrone 1.0 and 2.0 are equipped with two cameras. One frontal camera pointing forward and one vertical camera pointing downward. This driver will create three topics for each drone: *ardrone/image_raw*, *ardrone/front/image_raw* and *ardrone/bottom/image_raw*. Each of these three are standard ROS camera interface and publish messages of type image transport. The driver is also a standard ROS camera driver, therefore if camera calibration information is provided either as a set of ROS parameters or appropriate *ardrone_front.yaml* and/or *ardrone_bottom.yaml*, the information will be published in appropriate camera_info topics.

5.9 Installing unibg.robotics.ardrone.marker.follower software

To install our software, it is suggested to recall section 5.7 and to insert unibg .cpp node files in the correct folder. In particular, MarkerFollower.cpp file shall be put in unibg_maker_follower package, in the src folder and MarkerDetector.cpp file in unibg_maker_detector package, still in src folder. Please remember to put camera.yml file in the unibg_maker_follower/build/devel/unibg_maker_follower folder.

More in detail, after a correct configuration and installation (after cmake .. and make commands in the build folder of each package), the overall view of this project structure is depicted in fig.5.5.

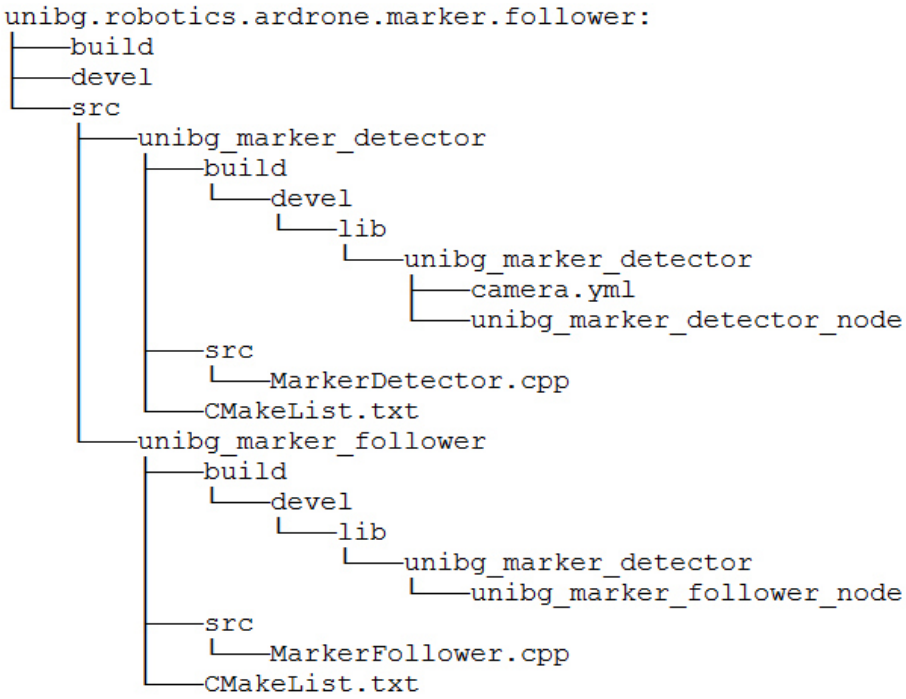

Figure 5.5: Workspace structure.

More catkin_generated folders will be present but not useful to understand our project.

5.9.1 CMakeList.txt

In the end, before compiling, the CMakeList.txt in the main folder of each package has to be modified, adding the following lines:

```
# Aruco - see http://www.uco.es/investiga/grupos/ava/node/26
SET(CMAKE_MODULE_PATH $CMAKE_INSTALL_PREFIX/lib/cmake/ )
find_package(aruco REQUIRED )
target_link_libraries(unibg_marker_detector $aruco_LIBS)


# Ardrone
SET(CMAKE_MODULE_PATH $CMAKE_INSTALL_PREFIX/lib/cmake/ )
find_package(ardrone_autonomy REQUIRED )

# Declare a cpp executable
add_executable(unibg_marker_detector src/UnibgMarkerDetector.cpp)

# Specify libraries to link a library or executable target
target_link_libraries(unibg_marker_detector $catkin_LIBRARIES)
```

Chapter 6

Development

This project has been developed to provide a robust architecture to communicate with an ARDrone Parrot 2.0 in order to receive information about its position, speed and direction in the frame of reference, about eventual observed markers and their coordinates compared to the drone's camera, to elaborate this information and to send new instructions (twist) to make it move in the environment.

To do so, the project has been divided in four major work packages (WP) of reception, detection, elaboration and dispatching of information:

- WP1 - reception of images from ARDrone camera;
- WP2 - detection of markers from images;
- WP3 - elaboration of markers position in relation to ARDrone position;
- WP4 - dispatching of commands to ARDrone.

WP 1 and 2 are implemented in one ROS node named *unibg_marker_detector_node* included in *unibg_marker_detector* package, instead, WP 2 and 3 are implemented in another ROS node named *unibg_marker_follower_node* included in *unibg_marker_follower* package.

A workspace visualization can be found in fig.6.1.

As ARDrone Parrot 2.0 continuously sends information on more than ten topics, there is the necessity to filter data in order to obtained just what needed.

During the development of this project, already existing ROS drivers such as the *ardrone_autonomy*[4] have been taking into account.

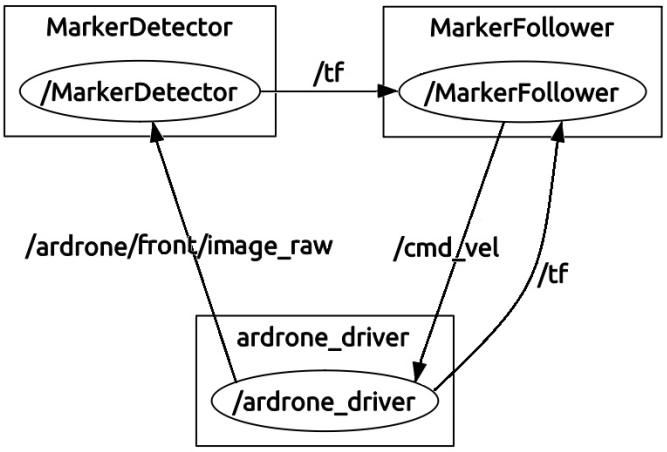

Figure 6.1: Rqt_graph.

6.1 Receiving visual information from ARDrone

The first step is to receive visual information from ARDrone. This is achieved through the subscription to a dedicated topic named */ardrone/front/image_raw*. This topic is used to receive images in image raw format from the drone camera. Implementation in listing 6.1 shows how the subscription happens.

Listing 6.1: Subscription to image_raw.

```
1 #include <ros/ros.h>
2 #include <image_transport/image_transport.h>
3 #include <iostream>
4 ros::NodeHandle nh;
5 image_transport::ImageTransport it(nh);
6 image_transport::Subscriber image_sub = it.subscribe
    ("/ardrone/front/image_raw", 1000,
    imageConverterCallback);
```

In this listing, line 3 calls *imageConverterCallback* callback, that is needed to convert the image format to OpenCV (see listing 6.2). In particular, a cv_bridge::CvImagePtr is used to help conversion. This type of variable contains a cv::Mat object as field.

Listing 6.2: imageConverterCallback callback.

```
1 // Includes for image conversion
2 #include <cv_bridge/cv_bridge.h>
3 #include <sensor_msgs/image_encodings.h>
4 #include <opencv2/imgproc/imgproc.hpp>
```

```
5  using namespace cv;
6  // Global parameters
7  cv::Mat InImage;
8  cv_bridge::CvImagePtr cv_ptr;
9  void imageConverterCallback(const
       sensor_msgs::ImageConstPtr& msg){
10   // Image conversion through cv_bridge
11   try{
12     cv_ptr = cv_bridge::toCvCopy(msg,
           sensor_msgs::image_encodings::BGR8);
13   } catch (cv_bridge::Exception& e){
14     ROS_ERROR("cv_bridge_exception:%s", e.what());
15     return;
16   }
17   InImage = cv_ptr->image;
18 }
```

In fact, to start using ArUco[5] library, a cv::Mat object is needed. Next section will explain how this library works and why it has proved to be the best library that could be chosen at the state of the art.

6.2 Marker detection

To detect markers in the figure, the idea is to denote each colour in the image with a value. Close colours with very similar values can be assumed to be part of an edge. Edges are the basic components to build the environment and, depending on their disposition, they can highlight a marker shape. Based on its side dimensions, markers position and orientation can be defined.

Due to its usability and scalability, the ArUco library immediately proved to be suitable for this project needs. A recall of its features can be found in chap.3.3.

As guaranteed by this library, a single line is enough to benefit from its provided function of markers detections (see listing 6.3).

Listing 6.3: ArUco markers detection.

```
1 // Includes for aruco library
2 #include <aruco/aruco.h>
3 #include <aruco/cvdrawingutils.h>
4 #include <opencv2/highgui/highgui.hpp>
```

```
5  using namespace aruco;
6  CameraParameters TheCameraParameters;
7  TheCameraParameters.readFromXMLFile("camera.yml");
8  vector<Marker> Markers;
9  try{
10   MarkerDetector MDetector;
11   // Marker detection
12   MDetector.detect(InImage, Markers,
        TheCameraParameters, TheMarkerSize);
```

In this implementation, the method *detect* is used, passing as arguments the input image obtained from the previous conversion, the markers vector where to save detected markers, the camera parameters obtained from the camera calibration and the marker size which side is expressed in $m \cdot 10^{-1}$.

At this time, a vector containing markers is filled. If it is not empty, a specific marker is sought. In this case, the selected marker is the one with id = 36. Marker id can be selected in the code from global variables (line 2, listing 6.4). If found, boundaries are drawn in the image (with line colour and width as passed parameters) to show the user that ArUco is properly working (line 7, listing 6.4).

Listing 6.4: Marker 36 seeking.

```
1  Marker MyMarker;\\
2  int MarkerNumber = 36;
3  for(unsigned int i=0;i<Markers.size();i++){
4    // Select only markers with id = 36
5      if(Markers[i].id == MarkerNumber){
6        MyMarker = Markers[i];
7        MyMarker.draw(InImage,Scalar(0,0,255),2);
8      }
9    }
```

Then, image is shown with marker info (listing 6.5).

Listing 6.5: Image shown.

```
1  static const std::string OPENCV_WINDOW = "Image_window";
2  cv::imshow(OPENCV_WINDOW,InImage);
```

In the end, the marker transfer frame is extracted after its detection. Before proceeding with the next section, a recall to chap.3.1.2 is suggested. In particular, referring to fig.3.9, following reasoning has been carried out.

6.2.1 Camera-Marker transformation frame

The last step of WP 2 is to define a tf between the front camera and the marker and to send it on */tf* topic. To do so, information is extracted from the detected marker, as shown in listing 6.6.

Linear transform parameters are obtained from Tvec vector and are expressed in meters. Angular parameters are obtained through *cv::Rodrigues()* method in order to obtain angles with respect to camera coordinates. Then, a tf::Transform object is initialize with *setOrigin* and *setRotation* methods. In the end, information are properly published on /tf topic, specifying the subjects of the transform frame: */ardrone_base_frontcam* and *marker*.

Listing 6.6: Camera-Marker tf definition.

```
1  // Includes for tf
2  #include <tf/transform_broadcaster.h>
3  // Tf fields survey
4  float x_tf = MyMarker.Tvec.at<Vec3f>(0,0)[0] * 10;
5  float y_tf = MyMarker.Tvec.at<Vec3f>(0,0)[1] * 10;
6  float z_tf = MyMarker.Tvec.at<Vec3f>(0,0)[2] * 10;
7  cv::Mat rot_mat(3,3,cv::DataType<float>::type);
8  cv::Rodrigues(MyMarker.Rvec,rot_mat);
9  float pitch = -atan2(rot_mat.at<float>(2,0),
       rot_mat.at<float>(2,1));
10 float yaw = acos(rot_mat.at<float>(2,2));
11 float roll = -atan2(rot_mat.at<float>(0,2),
       rot_mat.at<float>(1,2));
12 // Tf definition
13 tf::Transform transform;
14 transform.setOrigin(tf::Vector3(x_tf,y_tf,z_tf));
15 transform.setRotation(tf::createQuaternionFromRPY
       (yaw,pitch,roll));
16 // Tf publishing
17 static tf::TransformBroadcaster br;
18 br.sendTransform(tf::StampedTransform(transform,
       ros::Time::now(), "/ardrone_base_frontcam",
       "marker"));
```

Please, notice that it is not necessary to define the node frequency: whenever a frame is published, the node receives it and calls *imageConverterCallback* callback.

Whit this section, the unibg_marker_detector_node has been totally shown and explained. For further details see src code. The next section will explain WP 3 and 4, implemented in another unibg_marker_follower_node.

6.3 Elaboration and movement planning

In WP 3, what implemented is the elaboration of the received tf between the ARDrone base frontcam and the marker. Subscription to /tf topic and reception of new selected transfer frames is reported in listing 6.7.

Listing 6.7: Selected tf reception.

```
1  #include <ros/ros.h>
2  #include <tf/transform_listener.h>
3  tf::TransformListener listener;
4  ros::NodeHandle n;
5  tf::StampedTransform transform;
6  try{
7    listener.lookupTransform("/ardrone_base_frontcam",
       "marker", ros::Time(0), transform);
8  } catch (tf::TransformException &ex){
9  // If no tf are received: break operations untill next
     loop
10   ROS_ERROR("%s",ex.what());
11   continue;
12 }
```

Starting from this information, the necessity is to convert a position (the one of the marker with respect to the camera) into a velocity command to be sent to the drone.

Next section will explain the drone and camera axis definition and conversion.

6.3.1 ARDrone-Marker transformation frame

In fact, camera axis do not correspond to drone axis.

To better show the necessity of a conversion, fig.6.2 compares the two reference systems, where X_D, Y_D and Z_D denote the drone axis and X_C, Y_C and Z_C denote the camera axis.

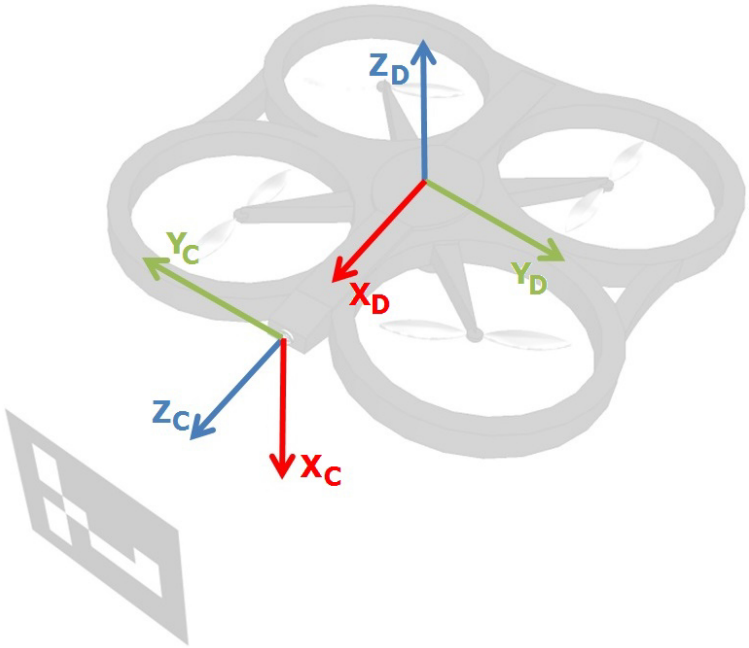

Figure 6.2: Drone vs camera axis detail.

It is clear that a roto-translation is needed. In particular two rotation are necessary to convert camera axis to drone axis, as shown in fig.6.3. In fig.6.4 rotations are depicted more in details.

The first rotation defines X'_C, Y'_C and Z'_C axis through the following rotation matrix:

$$R(_{Z_D,90^o}) = \begin{bmatrix} cos(90^o) & -sin(90^o) & 0 & 0 \\ sin(90^o) & -cos(90^o) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The second rotation defines X''_C, Y''_C and Z''_C through the following rotation matrix:

$$R(_{X_D,90^o}) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos(90^o) & -sin(90^o) & 0 \\ 0 & sin(90^o) & cos(90^o) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

To complete the overlap of the axes is required a 20cm translation of the camera reference system to the ARDrone center. Thus, the complete roto-translation matrix describes the relationship between the camera reference system and that of the

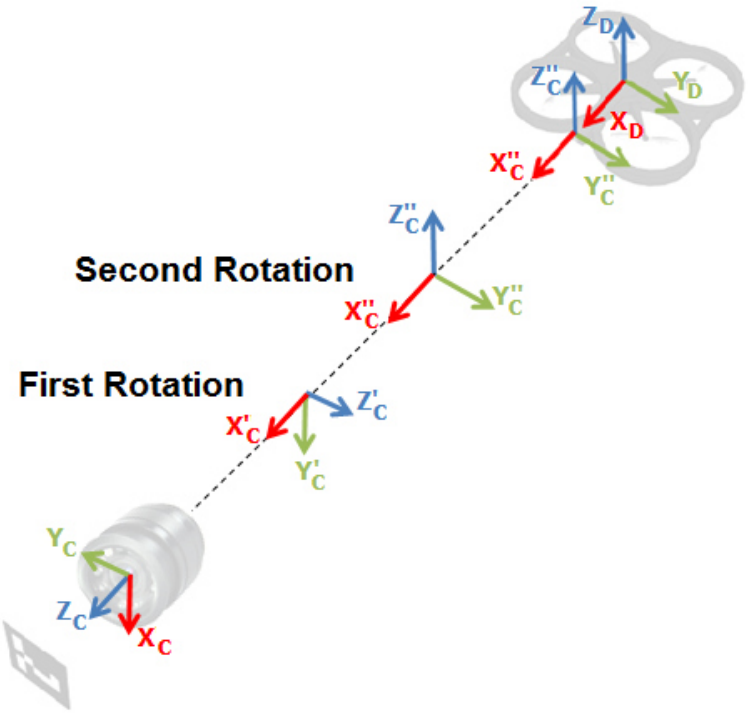

Figure 6.3: Camera axis roto-translation.

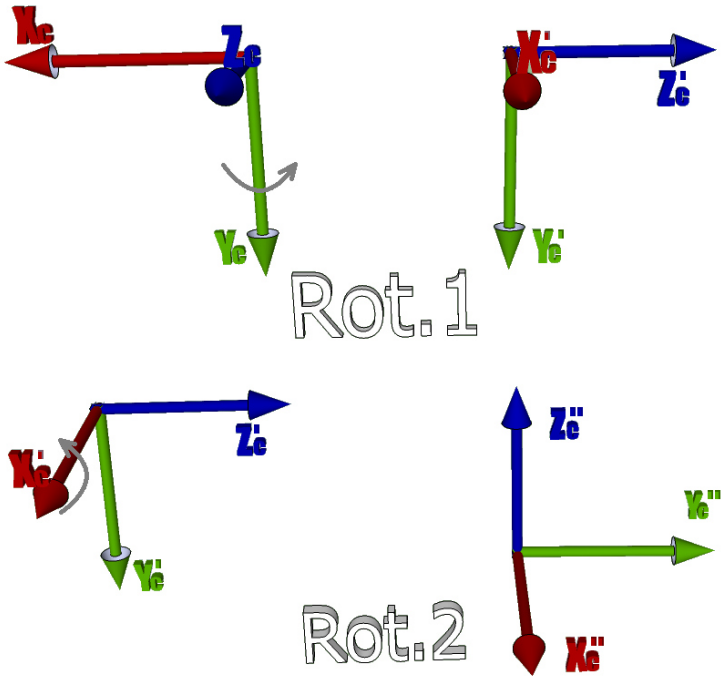

Figure 6.4: Camera axis rotations detail.

drone, and is defined as:

$$H = R(_{Z_D},_{90^o})R(_{X_D},_{90^o})T(_{X_D},_{20cm}) \tag{6.1}$$

For example, to represent the position of a generic point Q_C in the camera reference system into the drone reference system, it is necessary to multiply its position vector in the camera reference system by the roto-translation matrix H, to obtain Q_C, which defines the position of the point in the drone reference system:

$$Q_D = H \cdot Q_C \tag{6.2}$$

Nevertheless, this simple conversion describes the direction of the movements to be executed by the drone, but does not define the real twist to be sent to the drone. The speed modules depend on the control law introduced in the next chapter.

6.3.2 Control law

Once the roto-translation has been achieved, a reasoning on tf and velocity command full scale has been made. In fact, velocity commands are given with a range between 0 and 1 and are expressed in m/s, tf instead, are expressed in meters and, as shown in chap.4.3, can reach values of 3 meters in the distance, 0,6 meters in left-right direction and 0,4 meters in up-down direction. Moreover, it is necessary to calibrate velocity commands in order not to make the drone fly to fast for the position control loop or to slow for the marker following, or it will loose the marker from its viewing angle. In addition, another factor has to be taken into account: the need to dispatch velocity commands for a determined time (see section 6.4.1). Also, it is necessary to provide the drone the possibility of a quick change of direction also in case of wind (if operating outdoors) which brings a limitation in particular on up-down direction velocity, due to ARDrone Parrot 2.0 weight and power features. In the end, this drone behaviour has been carefully observed while moving and what came out is an auto-regulation to stabilise its flight. This brings more issues to be solved while sending new commands.

In order to control the drone flight after its takeoff, it is necessary to publish a message of type *geometry_ msgs::Twist* to the *cmd_ vel* topic. In particular, a twist message has three angular components and three linear components. As far as this project does not manage angular components, only linear components have to be managed. For this reason, one proportional controller has been used for each direction, for a total of three P controllers, as depicted in fig.6.5.

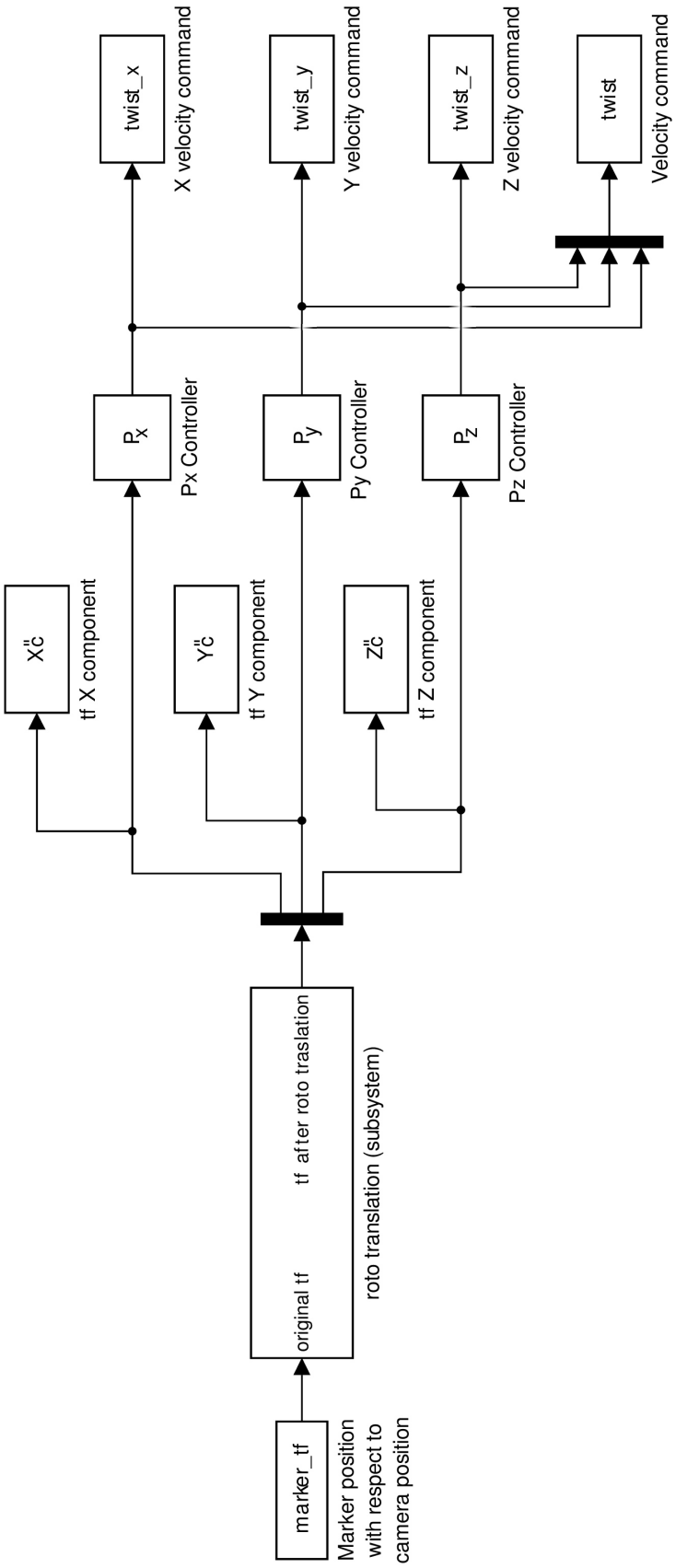

Figure 6.5: Complete control system.

The P controllers act also as unity measures converter, because their P_X, P_Y and P_Z parameters are expressed in s^{-1}.

For everything that has been considered, the best fitting parameters were established to be the following:

- Forward-backward drone controller parameter:

  $P_X = 0.1$

- Left-right drone movement controller parameter:

  $P_Y = 0.2$

- Up-down drone movement controller parameter:

  $P_Z = 2.0$

Thus, the final control law will be defined as follows:

- Forward-backward drone velocity command:

  $twist_X = final\_tf_X \cdot P_X$

- Left-right drone movement velocity command:

  $twist_Y = final\_tf_Y \cdot P_Y$

- Up-down drone velocity command:

  $twist_Z = final\_tf_Z \cdot P_Z$

where $final\_tf_X$, $final\_tf_Y$ and $final\_tf_Z$ are the final transform components between marker and drone (after roto-translation) and $twist_X$, $twist_Y$ and $twist_Z$ define the velocity command components.

In the end, for what said in the previous chapter, the following relation is included in the implementation:

$$final\_tf = H \cdot tf \tag{6.3}$$

where tf, composed of linear and angular components, is the initial transform value between marker and camera (before roto-translation).

What just explained is implemented in listing 6.8.

Listing 6.8: Tf-velocity commands scaling.

```
1 geometry_msgs::Twist twist;
2 // Forward-backward: quick enough: factor = 0.1
3 twist.linear.x=(transform.getOrigin().z()-1)*0.1;
4 // Left-right: quick enough: factor = 0.2
5 twist.linear.y=-transform.getOrigin().x()*0.2;
6 // Up-down: really slow: factor = 2.0
7 twist.linear.z=-transform.getOrigin().y()*2.0;
```

Yet, another control has to be implemented: in case of marker loss, due also to connection issues, the drone must not maintain its movement, but instead, it has to stop and keep looking for the marker. This control is made simply through the introduction of a twist variable containing the last twist. Measures sensibility is in terms of μm, which makes almost impossible to have the same tf between the drone and the camera at distance of a few milliseconds. So, if the new twist is calculated to be equal to the last, this indicates the loss of connection: then twist will be set to zero, waiting for a new different sighting. What just explained is implemented in listing 6.9.

Listing 6.9: Twist control loop.

```
1 if(twist == twist_old)
2   twist = null;
3 else
4   twist_old = twist;
```

Now that twists have been properly initialized, next section will show how to send commands to ARDrone.

6.4 Sending commands to ARDrone

Three different phases are detected and defined by three states:

1. takeoff;
2. marker following;
3. landing.

The drone will takeoff, land or emergency stop/reset by publishing an Empty ROS messages to the following topics: *ardrone/takeoff*, *ardrone/land* and *ardrone/reset* respectively.

So, once received the first tf message, the drone will be ready to takeoff. To launch the drone, see listing 6.10.

Listing 6.10: Takeoff phase.

```
#include <std_msgs/Empty.h>
ros::Publisher pub = n.advertise <std_msgs::Empty>
   ("/ardrone/takeoff", 1);
double time_start=(double)ros::Time::now().toSec();
// Send command for three seconds
while ((double)ros::Time::now().toSec()< time_start+3.0){
  pub.publish(std_msgs::Empty()); // Launches the drone
  ros::spinOnce();
}
```

Please note the necessity of publishing takeoff messages not just once but for a selected time interval. This depends on connection issues due to channels Wi-Fi overlapping.

Now, the drone is on flight and ready to follow the marker.

6.4.1 Navigation data commands

Information to be sent to the drone will be published to the */cmd_vel* topic. The message has to be a geometry_msgs::Twist, as shown in listing 6.11, with properly scaled values (between 0 and 1), as described previously.

Listing 6.11: Onflight marker followint phase.

```
// Loop rate
ros::Rate rate(100.0);
ros::Publisher pub = n.advertise <geometry_msgs::Twist>
   ("/cmd_vel", 1);
double time_start=(double)ros::Time::now().toSec();
// Send command for 0.1 second
while ((double)ros::Time::now().toSec()< time_start+0.1){
  pub.publish(twist);
  ros::spinOnce();
}
```

Dispatching commands

Once again, to see a proper behaviour of the drone without loosing control on it, the command has to be sent for a while. The value of the interval depends also on the node frequency loop. The idea is to make the drone move for a little time laps in one direction, with the attempt not to make it too nervous in its movements but also quick responsive to marker position variation.

Landing and phase changing control

The last phase is the one of the drone land. To make the drone land, this project provides the functionality of send a land command pressing a key. This is made through a concurrent thread that monitors keys pressing (see listing 6.12).

Listing 6.12: Thread for input commands.

```
#include <boost/thread.hpp>
using namespace boost;
// If user presses anything: go to next phase
void checkCin(){
  // state 0:  ready to takeoff
  char follow_marker;
  std::cin >> follow_marker;
  state = 1; // state 1: follow marker
  char land;
  std::cin >> land;
  state = 2; // state 2: land
}
int main(int argc, char **argv){
  boost::thread t(&checkCin);
}
```

Once the second key has been pressed, the drone is ready to land the same way it tookoff (listing 6.13).

Listing 6.13: Landing phase.

```
ros::Publisher pub = n.advertise <std_msgs::Empty>
   ("/ardrone/land", 1);
double time_start=(double)ros::Time::now().toSec();
// Send command for three seconds
```

```
4 while ((double)ros::Time::now().toSec()< time_start+3.0){
5   pub.publish(std_msgs::Empty()); // Land the drone
6   ros::spinOnce();
7 }
```

Whit this section, also the unibg_marker_follower_node has been totally shown and explained. Next sections will explain how to manipulate code to obtain different behaviours and to implement further extensions.

Chapter 7

User extension

This chapter explains how to manipulate src code in order to obtain different behaviours.

Changing selected marker id

To change the selected marker id, it is enough to change it in line 2 of listing 6.4. Please recall which id numbers are supported, reading section 3.3.1.

Switch camera

To switch camera from front to bottom, it is necessary to recalibrate and pass new camera parameters outside src code (camera.yml file).
New axis conversion and twist command definition has to be made, in listing 6.8.
Moreover, one more command to be launched from terminal is needed to actually switch the drone camera data flow, while ardrone_autonomy is running:

$ rosservice call /ardrone/togglecam

Further consideration on ARDrone path

This project is a simple tool to control up-down, backward-forward and left-right drone movements. Of course, more complex control can be implemented, also integrating other existing functionalities or calculating a complex path (please see chap.8.1 for further development concepts).

However, all that concerns drone navigation is defined during twist definition in listing 6.8 and could even be substituted by a whole new ROS node for different

techniques integration.

Any other idea may be shared on github[1].

Chapter 8

Final consideration

This project has been created as a basic tool to be extended for specific purposes. In fact, it is simply adjustable to satisfy specific intention such as, for example, adding new methods or override already existing ones. This would seamlessly integrate new abilities with the already existing environment.

8.1 Future direction

As introduced in section 7, a possible future development regards the ARDrone path, considering also angles and orientation of the marker. This would bring to a definition of a more sophisticated algorithm to define the drone movements also in terms of orientation.

For example, pursuant to a marker rotation, the drone may be able to reach a correct position following a circumference path.

Moreover, this project can be integrated with other projects developed in the Robotics lab of the University of Study of Bergamo, in which the drone follows a pre-defined path in respect with static markers. For example, a mixed pre-defined and human-following path can be generated.

Also, another challenging goal could be the one of coordinating a marker-based navigation with other techniques such as map-based localization and navigation.

Appendix A

Code listings

Bibliography

[1] GitHub online repository

Find src on github.com/yamunamaccarana

[2] ARDrone Parrot 2.0

Go to official website: cdn.ardrone2.parrot.com

[3] ROS

The Open Robot Control Software Project

See orocos.org

[4] An AutonomyLab developement:

ardrone_autonomy

Go to github.com/AutonomyLab/ardrone_autonomy

[5] ArUco library

www.uco.es/investiga/grupos/ava/node/26

[6] ARToolKit library

www.hitl.washington.edu/artoolkit/

[7] Calibration board

http://docs.opencv.org/_downloads/pattern.png

[8] ARDrone Identification and Navigation Control at CVG-UPM

J. Pestana, J. L. Sanchez-Lopez, I. Mellado-Bataller, Changhong Fu and P. Campo

[9] Ubuntu instruction guide

https://help.ubuntu.com/community/Repositories/Ubuntu