TÉCNICO LISBOA

Quadcopter Automatic Landing on a Docking Station

Tiago Gomes Carreira

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

Examination Committee

Chairperson: Prof. João Fernando Cardoso Silva Sequeira
Supervisor: Prof. Pedro Manuel Urbano de Almeida Lima
Member of the Committee: Prof. Alexandra Bento Moutinho

October 2013

Agradecimentos

Esta dissertação é o resultado de vários meses de trabalho árduo, de dias e noites de empenho e de muita insistência. Tudo isto não seria possível sem a ajuda, intervenção, presença ou incentivo das pessoas ou entidades que passo a referir.

Primeiramente, gostaria de agradecer ao professor que orientou a minha dissertação, Prof. Pedro Lima, sem o qual nada disto teria sido possível. Obrigado pelas instruções, motivação quando mais precisei e (longas) correcções ao meu trabalho.

Gostaria de agradecer ao Institute for Systems and Robotics (ISR) pela possibilidade de colaboração e ingresso neste projecto. Sinto-me agradecido por ter estado a trabalhar num projecto interessante e de ter visto tantos outros projectos que ajudam a alargar os meus horizontes.

Agradeço a todos os meus colegas de laboratório, Henrique Silva, Filipe Jesus, Duarte Dias e a todos os elementos da equipa do RAPOSA, do SocRob e dos quadcopters, pela abertura e disponibilidade, pela ajuda com as ferramentas de trabalho e dicas no meu trabalho.

A todos os meus colegas de tese: Diogo Rolo, Miguel Serafim, Pedro Santos, Marco Seabra, Joana Sanches, Nuno Marcos, por todas as ajudas, motivações e inspirações.

Não podia deixar de referir todos aqueles que, durante todo o meu percurso académico, me ajudaram a ser o que sou hoje, a todos os meus amigos da TUIST (especialmente ao Nuno Marcos) e do NEEC. Queria ainda deixar o meu agradecimento especial a todos os meus amigos da Confraria (em especial ao Pedro Ganço) que têm sido um grupo fantástico e motivador.

Como não podia deixar de ser, quero agradecer do fundo do coração à Joana Sanches. Foi ela que, onde quer que estivesse, me deu todo o apoio, motivação, força e carinho que tão úteis me foram, e contagiou-me com a vontade de ser mais e melhor, de crescer e de dar passos. Pelo carregar deste ónus que consiste em aturar-me e motivar-me, e por todas as noites sem nunca acabar, o meu obrigado.

Deixo ainda o meu agradecimento a todos os meus amigos em casa, da residência, dos grupos e coros nos quais participo, às pessoas que confiam em mim, às pessoas que todos os dias me ajudam, com especial menção à família Sanches, e a todos aqueles que, apesar de não estarem aqui mencionados, fazem de mim um homem melhor.

Por fim, mas no topo dos agradecimentos, gostaria de agradecer aos meus pais e restante família. Foi um longo percurso, com altos e baixos, fins de semana ausentes e dificuldades superadas. Por todas as ajudas, motivação, compreensão e presença, é com imenso orgulho que posso dizer que, no fim deste longo caminho, apresento o culminar de tantos anos de investimento pessoal. O meu muito obrigado.

Abstract

The goal of this thesis is the control of a quadcopter, with the aim of its spacial stabilization and its autonomous landing. The control approach seeks for full autonomy of the robot, by considering only internal sensors and processing. The method chosen for landing target recognition was the vision-based (with a common camera). Through computer vision, we estimate the quadcopter's relative pose and then proceed to control it and its autonomous landing.

A method was developed to estimate the relative position and orientation of a quadcopter using planar markers and computer vision algorithms. In the end, some tests are performed on the simulator, on which an autonomous quadcopter landing is achieved.

Keywords

Unmanned Aerial Vehicles, Quadcopter, Target Tracking, Vision-Based Control.

Resumo

O objectivo desta tese é o controlo do quadricóptero, com o objectivo da estabilização espacial e de aterragem autónoma do mesmo. O tema do controlo é abordado de uma perspectiva que procura a total autonomia do robô, considerando apenas sensores e processamento internos. O método escolhido para reconhecimento do destino de aterragem foi o da visão (através de uma câmara comum). Através do processamento da imagem, procura-se estimar a posição espacial do quadricóptero, para depois se proceder ao controlo do mesmo e da sua aterragem.

Foi desenvolvido um método de estimação da posição e orientação relativas do quadricóptero com recurso a marcadores planos e algoritmos de visão computacional. Por fim, são feitos testes em simulador, no qual é conseguida uma aterragem autónoma do quadricóptero.

Palavras Chave

Veículos Aéreos Não Tripulados, Quadricóptero, Seguimento de Referências, Controlo Baseado em Visão.

Contents

List of Figures

List of Tables

Abbreviations

AR Augmented Reality

AVA Aplicaciones de la Visión Artificial

CM Center of Mass

DOF Degrees of Freedom

IMU Inertial Measurement Unit

ISR Institute for Systems and Robotics

LRF Laser Range Finder

OpenCV Open Source Computer Vision

PID Proportional-Integral-Derivative

PnP Perspective-n-Point

RGB Red, Green and Blue

ROS Robot Operating System

trpy Thrust, Roll, Pitch, Yaw

UAV Unmanned Aerial Vehicle

UCO Universidad de Córdoba

URDF Unified Robot Description Format

VBHEA Vision-Based Horizontal Estimation Algorithm

VBPEA Vision-Based Pose Estimation Algorithm

WRT With Respect To

1

Introduction

Contents

1.1 Motivation

The field of robotics has been growing for the last few years and it is has a very important role nowadays, with an increasing investment due to its potential for applications. A particular field of robotics is the aerial robotics, especially Unmanned Aerial Vehicles (UAVs). One UAV that is being commonly used is the quadcopter – Figure 1.1. This robot is being very used because it is versatile and it has a relatively simple model (when a set of simplifying assumptions is taken and is valid).

Figure 1.1: Real quadcopter

One of the problems of this kind of vehicles is autonomy. The energy needed to perform the flight is stored in batteries, which are heavy and therefore limit the autonomy of the robot. Even with fully charged batteries and in a controlled environment, the quadcopter is able to fly for only a few minutes (the quadcopter used in this thesis can fly up to 20 minutes). If it carries some more weight (like a camera or some kind of measuring device) this time will be gradually reduced. It is obvious the need to charge the vehicle batteries very often. Today, this charging job has to be done with direct human interaction, i.e. someone must land the quadcopter manually.

The motivation to do this procedure automatically is obvious, and making the quadcopter able to autonomously land opens a lot of possibilities. The most obvious is the possibility to let the quadcopter charge itself by docking on a charging station, without the constant need for human intervention.

1.2 Literature Review

In the last few years, the usage and deployment of UAVs (also known as drones) have been growing, from hobby to military applications. Some of those applications include surveying, maintenance and surveillance tasks, transportation and manipulation, and search & rescue ([1, 2]).

Within the UAVs field, there are several possible platforms. Some of these vehicles are meant to be used within a controlled environment, for simple tasks or lightweight objects lifting. Those UAVs frequently use electrical motors, are very light and have a small autonomy. Some others are meant for endurance tests, heavy cargo and long trips, which use fuel engines and are much heavier (e.g. [3]). Among the previous types of UAVs, there are the fixed-wing and multi-rotors UAVs, and within multi-rotors there are the usual helicopters (two propellers), quadcopters with four propellers (Quad-Copter UX-401L from UAVision[4]), hexacopters with six propellers (MikroKopter[5]), octocopters with eight propellers (OctoCopter UX-801 from UAVision[4]) and even a 16-rotors as the E-Volo's Volocopter

VC200[6]. There are indeed many options, as can be consulted in the list of UAVs in [7]. From all this choices, the quadcopter platform has been the most used one, as it is one of the most flexible and adaptable platforms for undertaking aerial research.

Since the quadcopter is being so used, there are many tutorials and guides on how to acquire, build, control, measure and use this platform, such as [8–12]. Even within the same kind of structure there are many possibilities, mainly when referring to open-source autopilots, as it is stated in [9]. Despite the variety of choices, the aim of them all is the same – having a quadcopter flying and being able to easily control the 6 DOF with only four variables.

In order to correctly control this originally unstable vehicle, a very good model of the system and also a feedback control system are required. The usual navigation device available on a UAV is an Inertial Measurement Unit (IMU) ([13]), which uses accelerometers, gyroscopes and magnetometers to provide information about the attitude of the vehicle. But since this unit works with linear accelerations and angular velocities, the output information will suffer from accumulated error because an integrator is continuously adding (even if small) errors, which leads to drift between the system estimate and the real pose of the vehicle. For these reasons, the common sensors available on the quadcopter are not enough for the purpose of this work.

With this kind of information available, the quantity of users of a quadcopter platform are also growing and so are the need of better performance. As stated before, the autonomy of a quadcopter is very poor for several applications and the automatic self recharging is a clear necessity. As such a process requires very precise operations, it is needed a very precise control as well. Currently, there is a very accurate system that is able to interact with an UAV and get its full pose with so much precision that even some acrobatic moves are possible to perform on a quadcopter, such as multi-flips moves, as developed by D'Andrea et al. [14]. The Vicon motion capture system ([15]) is so good it is often used as a ground truth system (mentioned in [8, 9, 11, 14, 16–20]). This motion tracking system is based on multiple cameras, with very high frame rates (from 120 to 1000 fps) capable of detecting tiny movements of the quadcopter. In a well calibrated environment, it is possible to perform the acrobatics viewed in [21]. All this work provides top results, but it comes with a disadvantage – the need of external devices to track and control the vehicles.

The Institute for Systems and Robotics, Instituto Superior Técnico's Search and Rescue project [2] is one of the applications where the Vicon-like systems cannot be used. In such kind of environments there is the need of another system, capable to provide a good estimation for the (relative) position and orientation of the quadcopter. There are some options, related with Augmented Reality (AR) which may give a good solution, mostly because this kind of systems require a much more simple structure and calibration: a regular calibrated (and one only) camera, an AR software and a printed marker board, which could be printed on top of the RAPOSA robot[1], from the same project. At the moment there are a few very good solutions for AR processing, namely the ArUco [22] (described in Section 3.2.1) or the ARToolKit [23][24]. These systems are intended to apply computer vision techniques in order to compute the position and orientation of a markers board With Respect To (WRT) the camera. This kind

[1] `http://mediawiki.isr.ist.utl.pt/wiki/RAPOSA_robot`, consulted on October 2013

of solution is very interesting because it performs in real time and thus it may be used in an onboard processing environment.

Although the kind of sensor systems stated here provide absolute pose of the quadcopter, in this thesis, the relative pose estimation problem will be addressed. As an autonomous landing on a station is intended, only a relative pose to the station is enough to design a controller. Furthermore, a relative localization system is more useful for its simplicity and it is often possible to implement it.

1.3 Objectives and Contributions

The objective of this thesis consists on making the quadcopter to perform an autonomous landing on a docking station. It is also a requirement that the quadcopter performs this task without relying on some highly calibrated environment, thus using only on board devices to achieve this goal.

Specifically, it is used a quadcopter equipped with a downwards regular camera and an onboard processing unit (Pandaboard). In order to help the identification of the docking station, a markers board is used. The image captured by the camera is processed and the markers must be detected and identified; computer vision algorithms are applied for estimating the pose of the quadcopter. This pose estimation is then used for controlling the quadcopter's twist and performing the required maneuvers to achieve the desired goal: lead the quadcopter to land on the docking station autonomously.

In the conclusion of this thesis, the following steps were achieved:

- Visual-based pose estimation algorithms implementation and reliability analysis;
- Autonomous landing algorithm;
- Fully autonomous concept implementation, without external devices.

1.4 Thesis Outline

In Chapter 2 the basic concepts are reviewed, by introducing the quadcopter and its physics model, and by reviewing the basics on image processing, computer vision and control analysis.

The Chapter 3 is where the development part is stated. It explains the vision-based algorithms developed, the controllers used and the conceptual solutions for the initial problems.

The hardware and software implementations are stated in Chapter 4. The configurations chosen, the markers used and the system schematics is also included in this chapter. It has also the description of some tests made over the algorithms used, in order to classify the quality of the chosen methods.

The last chapter focus on testing the developed algorithms and showing their results.

2

Basic Concepts

Contents

2.1 Introduction

In this Chapter, some fundamental notions of physics and computer vision will be reviewed in the perspective of stability and notation for the remainder of the thesis.

The first Section 2.2 glances through the physical model of the quadcopter and its control variables. It includes some assumptions made to simplify the model and how the system is controlled.

In Section 2.3 image processing and computer vision concepts are stated. It includes the basic geometric elements definitions, the camera model and how to relate the 3D coordinates with an image. It also includes some information about image processing.

The last Section 2.4 includes some basics about controllers, the PID controller and how to implement it in this context.

2.2 Quadcopter Model

The quadcopter (Figure 1.1 on page 2) is one particular type of multi-rotor aerial robot, with four propellers mechanically attached to a rotor. The structure of the quadcopter is a symmetrical, cross configuration, with the propellers fixed and parallel. Two propellers are rotating clockwise (front and rear propellers) and the other two propellers are rotating counterclockwise (left and right propellers). All the propellers are built so their air flow points downwards for the described rotation direction so they get an upward force.

With this structure, it will be always considered that all rotors are linked through a rigid link and the only variables are the rotors' velocities.

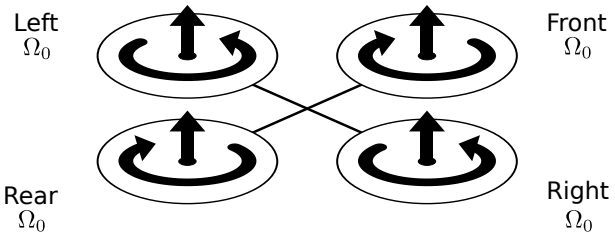


Figure 2.1: Quadcopter's rotors configurations and rotations, in hovering

2.2.1 Frame Definitions

To describe a 6 Degrees of Freedom (DOF) rigid body it is usual to define two reference frames [25]: Ground inertial reference (G-Frame) and Body-fixed reference (B-Frame), Figure 2.2.

The G-Frame $(\boldsymbol{O_G}, X_G, Y_G, Z_G)$ is a fixed right-hand referential with X_G pointing towards North, Y_G pointing towards West and Z_G pointing upwards.

The B-Frame $(\boldsymbol{O_B}, X_B, Y_B, Z_B)$ is a moving right-hand referential with X_B pointing towards quadcopter's front, Y_B pointing towards quadcopter's right and Z_B pointing downwards. The origin of this Frame is on the quadcopter's Center of Mass (CM).

The ability to correctly define the quadcopter pose and twist is crucial.

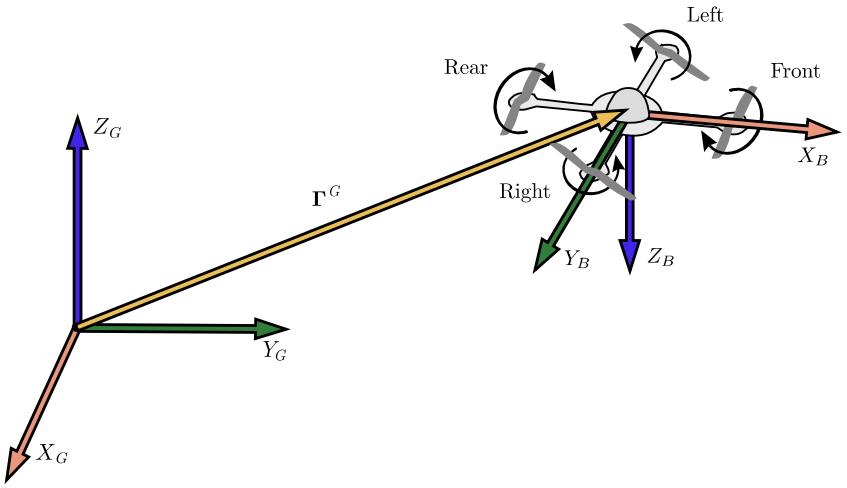
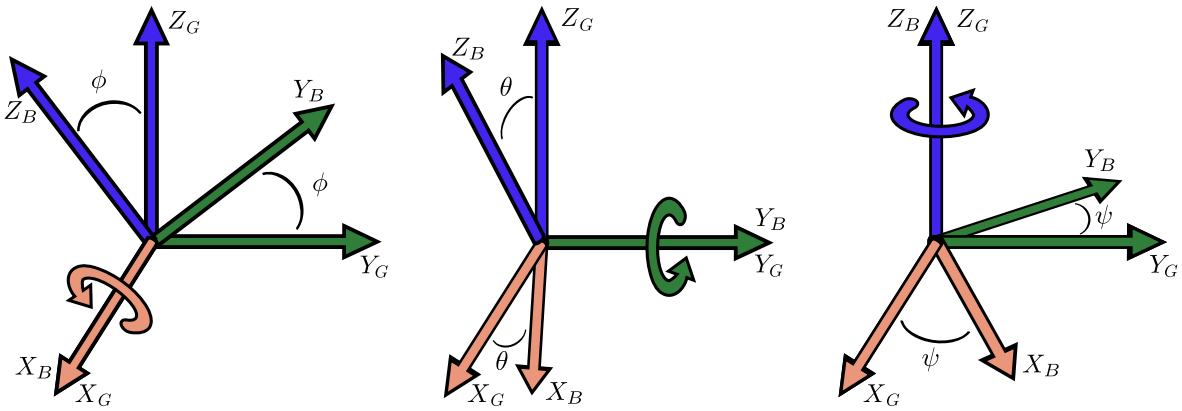


Figure 2.2: Frames and references considered (G-Frame and B-Frame)

The pose of the quadcopter, usually defined in the G-Frame, is a 6 DOF vector composed by a linear position and an angular position: $\boldsymbol{\xi} = (\boldsymbol{\Gamma^G}, \boldsymbol{\Theta^G}) = (X, Y, Z, \phi, \theta, \psi)$, with $\boldsymbol{\Gamma^G} = (X, Y, Z)[m]$ being the usual Cartesian coordinates, and $\boldsymbol{\Theta^G} = (\phi, \theta, \psi)[rad]$ being the roll, pitch and yaw which are rotations around X^G, Y^G and Z^G, respectively (Figure 2.3).



(a) Roll Rotation **(b)** Pitch Rotation **(c)** Yaw Rotation

Figure 2.3: Possible rotations around the three axis (X, Y, Z)

The twist of the quadcopter is usually defined in the B-Frame. It is a 6 DOF vector composed by a linear velocity and an angular velocity: $\boldsymbol{\nu} = (\boldsymbol{V^B}, \boldsymbol{\omega^B}) = (u, v, w, p, q, r)$, with $\boldsymbol{V^B} = (u, v, w)[m/s]$ being the linear velocity and $\boldsymbol{\omega^B} = (p, q, r)[rad/s]$ the angular velocity, both WRT B-Frame.

Therefore, it is possible to define a complete representation of the body in the space, given the Equations (2.1) and (2.2):

$$\boldsymbol{\xi} = (\boldsymbol{\Gamma^G}, \boldsymbol{\Theta^G}) = (X, Y, Z, \phi, \theta, \psi) \tag{2.1}$$

$$\boldsymbol{\nu} = (\boldsymbol{V^B}, \boldsymbol{\omega^B}) = (u, v, w, p, q, r) \tag{2.2}$$

The G-Frame and the B-Frame are related to each other by a rotation and a translation, ie. a rigid body transformation. The translation from G-Frame to B-Frame is the vector $\boldsymbol{\Gamma^G}$ itself. The rotation from G-Frame to B-Frame is a rotation matrix given by Equation (2.3).

$$\begin{aligned} \boldsymbol{R_\Theta} = \boldsymbol{R(\psi)}\ \boldsymbol{R(\theta)}\ \boldsymbol{R(\phi)} = \begin{bmatrix} c_\psi & -s_\psi & 0 \\ s_\psi & c_\psi & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} c_\theta & 0 & s_\theta \\ 0 & 1 & 0 \\ -s_\theta & 0 & c_\theta \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & c_\phi & -s_\phi \\ 0 & s_\phi & c_\phi \end{bmatrix} \Leftrightarrow \\ \Leftrightarrow \boldsymbol{R_\Theta} = \begin{bmatrix} c_\psi c_\theta & -s_\psi c_\phi + c_\psi s_\theta s_\phi & s_\psi s_\phi + c_\psi s_\theta c_\phi \\ s_\psi c_\theta & c_\psi c_\phi + s_\psi s_\theta s_\phi & -c_\psi s_\phi + s_\psi s_\theta c_\phi \\ -s_\theta & c_\theta s_\phi & c_\theta c_\phi \end{bmatrix} \end{aligned} \tag{2.3}$$

where c_x and s_x mean $cos(x)$ and $sin(x)$ respectively. In order to define orientation, this equation uses the Z-Y-X (yaw-$pitch$-$roll$) Euler angles. This this rotation axis sequence is used from now on.

Therefore, to transform any generic position (coordinate) P_x^B in B-Frame to P_x^G in G-Frame, it is applied a rigid-body transformation, defined in matrix form as a multiplication of a $4{\times}4$ matrix by a $4{\times}1$ matrix

$$\begin{bmatrix} \boldsymbol{P_x^G} \\ 1 \end{bmatrix} = \begin{bmatrix} \boldsymbol{R_\Theta} & \boldsymbol{\Gamma^G} \\ \boldsymbol{0}_{1\times3} & 1 \end{bmatrix} \cdot \begin{bmatrix} \boldsymbol{P_x^B} \\ 1 \end{bmatrix} \tag{2.4}$$

The inverse operation is also possible if it is considered the inverse of the operation in (2.4)[2]:

$$\begin{bmatrix} \boldsymbol{P_x^G} \\ 1 \end{bmatrix} = \begin{bmatrix} \boldsymbol{R_\Theta} & \boldsymbol{\Gamma^G} \\ \boldsymbol{0}_{1\times3} & 1 \end{bmatrix} \cdot \begin{bmatrix} \boldsymbol{P_x^B} \\ 1 \end{bmatrix} \Rightarrow \begin{bmatrix} \boldsymbol{P_x^B} \\ 1 \end{bmatrix} = \begin{bmatrix} \boldsymbol{R_\Theta} & \boldsymbol{\Gamma^G} \\ \boldsymbol{0}_{1\times3} & 1 \end{bmatrix}^{-1} \cdot \begin{bmatrix} \boldsymbol{P_x^G} \\ 1 \end{bmatrix} \tag{2.5}$$

A rotation is also possible to be represented by other notations, like Quaternions or through the Rodrigues' rotation formulas. Quaternions are the most used notation in graphical computation because they perform better than rotation matrices. Although these notations are not studied in this thesis, they will be here stated some basics about them because they are used in Robot Operating System (ROS)[26] and ArUco [22].

A Quaternion (q) may be expressed as a four-dimension vector ($(w, x, y, z) \in \mathbb{R}^4$) or as a number in an "hypercomplex" space ($q \in \mathbb{H}$) with three different imaginary units: i, j and k ([27]). For the matters of numerical stability, all quaternions should be normalized. To rotate an orientation, it is just necessary to multiply the Quaternion rotation by the Quaternion orientation (remembering that multiplication order matters, according to Equations (2.8)).

$$q \in \mathbb{H} = w + xi + yj + zk \tag{2.6}$$

This "hypercomplex" space must verify the following conditions:

$$\begin{cases} i^2 = j^2 = k^2 = -1 \\ ij = k, \quad jk = i, \quad ki = j \\ ij + ji = 0, \quad ik + ki = 0, \quad jk + kj = 0 \end{cases} \tag{2.7}$$

The "hypercomplex" ($\mathbb{H}$) has also its own algebra. Let $Q_1 = (w_1, x_1, y_1, z_1)$ and $Q_2 = (w_2, x_2, y_2, z_2)$, then the addition and multiplication operations are defined as followings:

[2] The validity of this statement is not covered in this thesis, for the matter of simplicity

$$Addition \qquad Q_1 + Q_2 = (w_1 + w_2, x_1 + x_2, y_1 + y_2, z_1 + z_2) \tag{2.8a}$$

$$Multiplication \qquad Q_1 \cdot Q_2 = \begin{cases} (w_1 w_2 - x_1 x_2 - y_1 y_2 - z_1 z_2, \\ w_1 x_2 + x_1 w_2 + y_1 z_2 - z_1 y_2, \\ w_1 y_2 - x_1 z_2 + y_1 w_2 + z_1 x_2, \\ w_1 z_2 + x_1 y_2 - y_1 x_2 + z_1 w_2) \end{cases} \tag{2.8b}$$

$$Multiplication\ by\ a\ scalar \qquad \alpha \cdot Q_1 = (\alpha w_1, \alpha x_1, \alpha y_1, \alpha z_1). \tag{2.8c}$$

The Rodrigues' rotation consists on a vector direction that defines the rotation axis, and whose intensity is the angle turned over the rotation axis [28]. This notation is far more intuitive for people than rotation matrices or quaternions. To rotate a vector $\boldsymbol{v}$ through a rotation $\boldsymbol{k}^*$ (both in Rodrigues' rotations) it is performed the following formula (Rodrigues' rotation formula):

$$\boldsymbol{v_{rot}} = \boldsymbol{v} cos(\theta) + [\boldsymbol{k}]_x \boldsymbol{v} sin(\theta) + \boldsymbol{k}\boldsymbol{k}^T \boldsymbol{v}(1 - cos(\theta)) \tag{2.9}$$

where $\theta = norm(\boldsymbol{k}^*)$ and $\boldsymbol{k}$ is the normalized vector $\boldsymbol{k}^*$. The notation $[\ ]_x$ is a skew symmetric matrix such as:

$$\boldsymbol{k} = \begin{bmatrix} k_1 \\ k_2 \\ k_3 \end{bmatrix} \quad \Rightarrow \quad [\boldsymbol{k}]_x = \begin{bmatrix} 0 & -k_3 & k_2 \\ k_3 & 0 & -k_1 \\ -k_2 & k_1 & 0 \end{bmatrix}. \tag{2.10}$$

This skew symmetric matrix allows to compute the cross product in such a way that $[\boldsymbol{k}]_x \boldsymbol{v} = \boldsymbol{k} \times \boldsymbol{v}$.

Although quaternions and Rodrigues's rotations are used withing the implementation code, only the Euler angles are used for describing the quadcopter's model, as defined in Equation (2.3).

2.2.2 Quadcopter Kinematics

As the quadcopter is a sub-actuated system (only four actuators to control 6 DOF) there is a need to redefine the actuation variables. Luckily, it is relatively easy to define four distinct movements that the quadcopter is able to do in order to achieve the desired results. To do this we will consider the four basic movements:

1. Throttle ($U_1[N]$)

   This force is directly related with a linear acceleration along Z_B-axis. By increasing (or decreasing) the absolute value of the angular velocity of all propellers by the same amount, it is created a vertical force that pushes up (or down) the body of the quadcopter. If it is perfectly hovering, this force will cause the quadcopter to go up (or down) with still X and Y position. The Figure 2.4 illustrates this movement.

2. Roll ($U_2[N.m]$)

   This torque is directly related with an angular acceleration along X_B-axis. By maintaining the front and rear rotors' speeds and increasing the left one while decreasing the right one by the

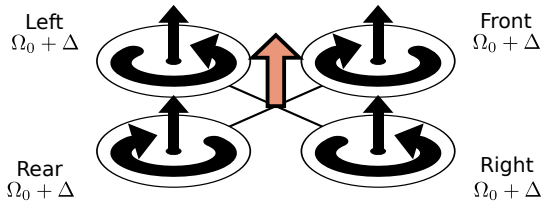


Figure 2.4: Quadcopter's throttle movement

same amount, the body will rotate positively along X_B-axis. The negatively roll will be achieved by decreasing the left rotor's speed while increasing the right rotor's one by the same speed. At a first approximation, this movement will cause only a roll angle acceleration, but the thrust force will have an horizontal component that will make the quadcopter also move right (if positive roll is applied). The Figure 2.5 illustrates this movement.

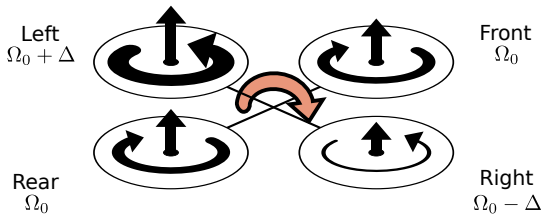


Figure 2.5: Quadcopter's roll movement

3. Pitch ($U_3[N.m]$)

   This torque is directly related with an angular acceleration along Y_B-axis. Similar to the previous one, the pitch movement is achieved by maintaining the left and right rotors' speeds and increasing the front one while decreasing the rear one by the same amount. This will impose the body a positive rotation along Y_B-axis. The negatively pitch will be achieved by decreasing the front rotor's speed while increasing the rear rotor's one by the same speed. At a first approximation, this movement will cause only a pitch angle acceleration, but, like the roll movement, the thrust force will have an horizontal component that will make the quadcopter also move back (if positive pitch is applied). The Figure 2.6 illustrates this movement.

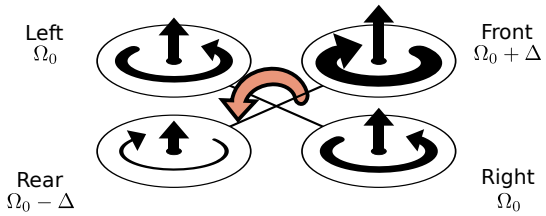


Figure 2.6: Quadcopter's pitch movement

4. Yaw ($U_4[N.m]$)

   This torque is directly related with an angular acceleration along Z_B-axis. This one takes advantage of the fact that the front and rear propellers rotate clockwise while the left and right propellers rotate counterclockwise. Decreasing the front and rear rotors' speeds and increasing the left and the right ones by the same amount, creates an unbalance within the torque in the Z_B-axis, thus

creating a rotation along the same axis, originating a change in the quadcopter's direction, turning right (with positive yaw in the B-Frame and negative in the G-Frame). The Figure 2.7 illustrates this movement.

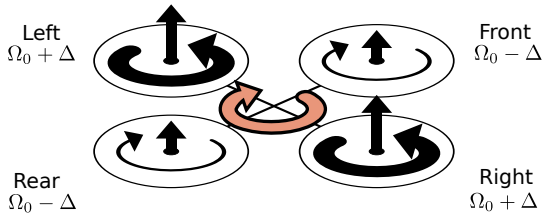


Figure 2.7: Quadcopter's yaw movement

Since the variables that are possible to control directly are the propellers' velocities (through the autopilot), a relation from these $\boldsymbol{U_B} = (U_1, U_2, U_3, U_4)$ to the propellers' velocities $\boldsymbol{\Omega} = (\Omega_1, \Omega_2, \Omega_3, \Omega_4)$ is needed, which is stated in (2.11) (explained in detail by [25]). Since this is a closed solution, it can abstract the direct variables, thus simplifying the system intended to control.

$$\begin{cases} \Omega_1^2 &= \frac{1}{4b}U_1 - \frac{1}{2bl}U_3 - \frac{1}{4d}U_4 \\ \Omega_2^2 &= \frac{1}{4b}U_1 - \frac{1}{2bl}U_2 + \frac{1}{4d}U_4 \\ \Omega_3^2 &= \frac{1}{4b}U_1 + \frac{1}{2bl}U_3 - \frac{1}{4d}U_4 \\ \Omega_4^2 &= \frac{1}{4b}U_1 + \frac{1}{2bl}U_2 + \frac{1}{4d}U_4 \end{cases} \Leftrightarrow \begin{cases} \Omega_1 &= \sqrt{\frac{1}{4b}U_1 - \frac{1}{2bl}U_3 - \frac{1}{4d}U_4} \\ \Omega_2 &= \sqrt{\frac{1}{4b}U_1 - \frac{1}{2bl}U_2 + \frac{1}{4d}U_4} \\ \Omega_3 &= \sqrt{\frac{1}{4b}U_1 + \frac{1}{2bl}U_3 - \frac{1}{4d}U_4} \\ \Omega_4 &= \sqrt{\frac{1}{4b}U_1 + \frac{1}{2bl}U_2 + \frac{1}{4d}U_4} \end{cases} \tag{2.11}$$

where $l[m]$ is the distance between quadcopter's center and the center of a rotor, $b[Ns^2]$ is the aerodynamic contribution for thrust and $d[Nms^2]$ is the aerodynamic contribution for drag. All this parameters are constant, verifying the closed form in the previous sentence.

The differential kinematics equation of a generic 6 DOF rigid body is given by (2.12).

$$\dot{\boldsymbol{\xi}} = \boldsymbol{J_\Theta \nu} \tag{2.12}$$

where $\dot{\boldsymbol{\xi}}$ is the time derivative of $\boldsymbol{\xi}$, ie. the generalized velocity WRT G-Frame. The matrix $\boldsymbol{J_\Theta}$ is defined in (2.13), which is composed by the sub-matrix $\boldsymbol{R_\Theta}$ defined in Equation (2.3) on page 8 and by a transfer matrix $\boldsymbol{T_\Theta}$, defined in (2.14) [29].

$$\boldsymbol{J_\Theta} = \begin{bmatrix} \boldsymbol{R_\Theta} & \boldsymbol{0_{3\times3}} \\ \boldsymbol{0_{3\times3}} & \boldsymbol{T_\Theta} \end{bmatrix} = \begin{bmatrix} c_\psi c_\theta & -s_\psi c_\phi + c_\psi s_\theta s_\phi & s_\psi s_\phi + c_\psi s_\theta c_\phi & 0 & 0 & 0 \\ s_\psi c_\theta & c_\psi c_\phi + s_\psi s_\theta s_\phi & -c_\psi s_\phi + s_\psi s_\theta c_\phi & 0 & 0 & 0 \\ -s_\theta & c_\theta s_\phi & c_\theta c_\phi & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & s_\phi t_\theta & c_\phi t_\theta \\ 0 & 0 & 0 & 0 & c_\phi & -s_\phi \\ 0 & 0 & 0 & 0 & s_\phi / c_\theta & c_\phi / c_\theta \end{bmatrix} \tag{2.13}$$

$$\boldsymbol{T_\Theta} = \begin{bmatrix} 1 & s_\phi t_\theta & c_\phi t_\theta \\ 0 & c_\phi & -s_\phi \\ 0 & s_\phi / c_\theta & c_\phi / c_\theta \end{bmatrix} \tag{2.14}$$

with c_x, s_x and t_x are $cos(x)$, $sin(x)$ and $tan(x)$ respectively.

Therefore, the kinematics equations for the quadcopter are described as in Equations (2.15).

$$\dot{\boldsymbol{\Gamma}}^G = \boldsymbol{R_\Theta} \cdot \boldsymbol{V^B} \Leftrightarrow \begin{bmatrix} \dot{X} \\ \dot{Y} \\ \dot{Z} \end{bmatrix} = \begin{bmatrix} c_\psi c_\theta & -s_\psi c_\phi + c_\psi s_\theta s_\phi & s_\psi s_\phi + c_\psi s_\theta c_\phi \\ s_\psi c_\theta & c_\psi c_\phi + s_\psi s_\theta s_\phi & -c_\psi s_\phi + s_\psi s_\theta c_\phi \\ -s_\theta & c_\theta s_\phi & c_\theta c_\phi \end{bmatrix} \cdot \begin{bmatrix} u \\ v \\ w \end{bmatrix} \tag{2.15a}$$

$$\dot{\boldsymbol{\Theta}}^G = \boldsymbol{T_\Theta} \cdot \boldsymbol{\omega^B} \quad \Leftrightarrow \quad \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} = \begin{bmatrix} 1 & s_\phi t_\theta & c_\phi t_\theta \\ 0 & c_\phi & -s_\phi \\ 0 & s_\phi / c_\theta & c_\phi / c_\theta \end{bmatrix} \cdot \begin{bmatrix} p \\ q \\ r \end{bmatrix} \tag{2.15b}$$

2.2.3 Quadcopter Dynamics

The dynamics of a 6 DOF rigid-body takes into consideration the mass of the body $m[kg]$ as well as its inertia matrix $\boldsymbol{I}[Nms^2]$. The matrix $\boldsymbol{I}$ is calculated empirically, through a process described in [25]. If it is considered the axis of B-Frame coincident with the principal axes of inertia, this matrix $\boldsymbol{I}$ becomes a diagonal matrix:

$$\boldsymbol{I} = \begin{bmatrix} I_{xx} & 0 & 0 \\ 0 & I_{yy} & 0 \\ 0 & 0 & I_{zz} \end{bmatrix} \tag{2.16}$$

where I_{XX} denotes the moment of inertia around the x-axis when the body is rotated around the x-axis and analogously to I_{YY} and I_{ZZ}.

Therefore, the dynamics of the quadcopter can be described in Equation (2.17), using Newton's Law and Newton-Euler's Law.

$$\begin{bmatrix} m\,\boldsymbol{eye(3)} & \boldsymbol{0_{3\times3}} \\ \boldsymbol{0_{3\times3}} & \boldsymbol{I} \end{bmatrix} \begin{bmatrix} \dot{\boldsymbol{V}}^B \\ \dot{\boldsymbol{\omega}}^B \end{bmatrix} + \begin{bmatrix} \boldsymbol{\omega}^B \times (m\boldsymbol{V}^B) \\ \boldsymbol{\omega}^B \times (\boldsymbol{I}\boldsymbol{\omega}^B) \end{bmatrix} = \begin{bmatrix} \boldsymbol{F}^B \\ \boldsymbol{\tau}^B \end{bmatrix} \tag{2.17}$$

where $\boldsymbol{eye(3)}$ is an identity 3×3 matrix, $\boldsymbol{0_{3\times3}}$ is a 3×3 matrix full with zeros, $\dot{\boldsymbol{V}}^B$ is the time derivative of the linear velocities vector $\boldsymbol{V}^B$ WRT B-Frame (a linear acceleration in B-Frame), $\dot{\boldsymbol{\omega}}^B$ is the time derivative of the angular velocities vector $\boldsymbol{\omega}^B$ also WRT B-Frame (an angular acceleration in B-Frame), $\boldsymbol{F}^B = (F_x, F_y, F_z)[N]$ is a vector with the force applied in $(x, y, z)^B$ direction on the B-Frame and $\boldsymbol{\tau}^B = (\tau_x, \tau_y, \tau_z)[N.m]$ is a vector with the torques applied in in each axis x, y, z.

Equation (2.17) is rewritten with the linear part separated from the angular part and, if the cross product is rewritten using the skew symmetric matrix, we reach the Equations (2.18).

$$\boldsymbol{F}^B = m(\dot{\boldsymbol{V}}^B + \left[\boldsymbol{\omega}^B\right]_\times \cdot \boldsymbol{V}^B) \quad \Leftrightarrow \quad \begin{bmatrix} F_x \\ F_y \\ F_z \end{bmatrix} = m \left(\begin{bmatrix} \dot{u} \\ \dot{v} \\ \dot{w} \end{bmatrix} + \begin{bmatrix} wq - vr \\ ur - wp \\ vp - uq \end{bmatrix} \right) \tag{2.18a}$$

$$\boldsymbol{\tau}^B = \boldsymbol{I} \cdot \dot{\boldsymbol{\omega}}^B + \left[\boldsymbol{\omega}^B\right]_\times \cdot (\boldsymbol{I}\boldsymbol{\omega}^B) \quad \Leftrightarrow \quad \begin{bmatrix} \tau_x \\ \tau_y \\ \tau_z \end{bmatrix} = \begin{bmatrix} \dot{p}\, I_{xx} \\ \dot{q}\, I_{yy} \\ \dot{r}\, I_{zz} \end{bmatrix} + \begin{bmatrix} qr(I_{zz} - I_{yy}) \\ rp(I_{xx} - I_{zz}) \\ pq(I_{yy} - I_{xx}) \end{bmatrix} \tag{2.18b}$$

In order to complete this set of equations, the vector $\boldsymbol{U_B}$ defined in Section 2.2.2 and the gravity force ($g[m/s^2]$) must be added to the model. Considering this, it is possible to figure a new set of equations:

$$\boldsymbol{F}^B = \begin{bmatrix} F_x \\ F_y \\ F_z \end{bmatrix} = \boldsymbol{R}_{\boldsymbol{\Theta}}^T \cdot \begin{bmatrix} 0 \\ 0 \\ -m\ g \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ U_1 \end{bmatrix} = \begin{bmatrix} m\ g\ s_\theta \\ -m\ g\ c_\theta\ s_\phi \\ -m\ g\ c_\theta\ c_\phi \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ U_1 \end{bmatrix} \tag{2.19a}$$

$$\boldsymbol{\tau}^B = \begin{bmatrix} \tau_x \\ \tau_y \\ \tau_z \end{bmatrix} = \begin{bmatrix} U_2 \\ U_3 \\ U_4 \end{bmatrix} \tag{2.19b}$$

To conclude, Equations (2.20) have the complete [3] quadcopter's mathematical model, based in Equations (2.15), (2.18) and (2.19).

$$\dot{\boldsymbol{\Gamma}}^G = \begin{bmatrix} \dot{X} \\ \dot{Y} \\ \dot{Z} \end{bmatrix} = \boldsymbol{R}_{\boldsymbol{\Theta}} \cdot \boldsymbol{V}^B = \begin{bmatrix} c_\psi c_\theta & -s_\psi c_\phi + c_\psi s_\theta s_\phi & s_\psi s_\phi + c_\psi s_\theta c_\phi \\ s_\psi c_\theta & c_\psi c_\phi + s_\psi s_\theta s_\phi & -c_\psi s_\phi + s_\psi s_\theta c_\phi \\ -s_\theta & c_\theta s_\phi & c_\theta c_\phi \end{bmatrix} \cdot \begin{bmatrix} u \\ v \\ w \end{bmatrix} \tag{2.20a}$$

$$\dot{\boldsymbol{\Theta}}^G = \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} = \boldsymbol{T}_{\boldsymbol{\Theta}} \cdot \boldsymbol{\omega}^B = \begin{bmatrix} 1 & s_\phi t_\theta & c_\phi t_\theta \\ 0 & c_\phi & -s_\phi \\ 0 & s_\phi / c_\theta & c_\phi / c_\theta \end{bmatrix} \cdot \begin{bmatrix} p \\ q \\ r \end{bmatrix} \tag{2.20b}$$

$$\dot{\boldsymbol{V}}^B = \begin{bmatrix} \dot{u} \\ \dot{v} \\ \dot{w} \end{bmatrix} = \sum \boldsymbol{F}^B = \begin{bmatrix} v\ r - w\ q \\ w\ p - u\ r \\ u\ q - v\ p \end{bmatrix} + g \cdot \begin{bmatrix} s_\theta \\ -c_\theta\ s_\phi \\ -c_\theta\ c_\phi \end{bmatrix} + \frac{1}{m} \cdot \begin{bmatrix} 0 \\ 0 \\ U_1 \end{bmatrix} \tag{2.20c}$$

$$\dot{\boldsymbol{\omega}}^B = \begin{bmatrix} \dot{p} \\ \dot{q} \\ \dot{r} \end{bmatrix} = \boldsymbol{I}^{-1} \cdot \sum \boldsymbol{\tau}^B = \begin{bmatrix} qr\ \frac{I_{yy} - I_{zz}}{I_{xx}} \\ rp\ \frac{I_{zz} - I_{xx}}{I_{yy}} \\ qp\ \frac{I_{xx} - I_{yy}}{I_{zz}} \end{bmatrix} + \begin{bmatrix} \frac{1}{I_{xx}}\ U_2 \\ \frac{1}{I_{yy}}\ U_3 \\ \frac{1}{I_{zz}}\ U_4 \end{bmatrix} \tag{2.20d}$$

[3] It's taken as assumption that ideal rotors are used and their velocities are converted from vector $\boldsymbol{U_B}$ as stated in (2.11)

2.3 Vision in the Loop

In order to process images some basic understanding about cameras and digital storage of image is needed. Besides the simple image analysis, there is also the need to use computer vision results so as to perceive the three-dimensional structure of the world.

2.3.1 Geometric Primitives

Firstly, an image is considered as a representation of one perspective of the world, in array form. In computers, images are usually defined as an array and each entry is a pixel. Each pixel contains three values, three intensities of three colors: Red, Green and Blue (RGB).

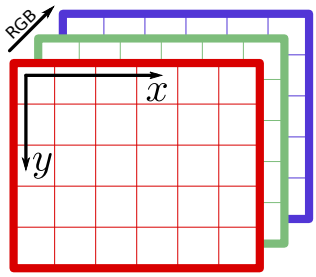


Figure 2.8: Digital representation of an RGB image

Although an image is conceptually thought as a 2D array, it is actually a 3D array, with the third dimension being the RGB values. This concept is represented in Figure 2.8. Thus, each pixel can be represented by a pair of (x, y) coordinates, with $\boldsymbol{x}$ being a pixel and $\tilde{\boldsymbol{x}}$ being the same pixel written in homogeneous coordinates, used further in this thesis. Therefore, a 2D Point is defined as:

$$\boldsymbol{x} = (x, y) \tag{2.21a}$$

$$\tilde{\boldsymbol{x}} = \lambda(x, y, 1) \tag{2.21b}$$

where λ is a scale factor needed to convert from homogeneous coordinates back to non-homogeneous coordinates. This scale factor is always implied during all equations listed further in this thesis, although it is hidden for simplicity.

In order to describe a shape within the image, mathematical lines definitions are required. A line may be written in homogeneous coordinates as $\tilde{\boldsymbol{l}} = (a, b, c)$ and the equation that defines 2D lines is in (2.22).

$$\tilde{\boldsymbol{x}}^T \cdot \tilde{\boldsymbol{l}} = ax + by + c = 0 \tag{2.22}$$

Real world coordinates are 3D, hence 3D primitives must also be defined. A 3D point $\boldsymbol{X}$ may also be represented in homogeneous coordinates $\tilde{\boldsymbol{X}}$:

$$\boldsymbol{X} = (X, Y, Z) \tag{2.23a}$$

$$\tilde{\boldsymbol{X}} = \lambda(X, Y, Z, 1) \tag{2.23b}$$

Just as before, the scale factor λ is omitted from now on.

The plane $\tilde{\boldsymbol{W}} = (A, B, C, D)$ equation in 3D is very similar to a line in 2D, with the plane equation defined as:

$$\tilde{\boldsymbol{X}}^T \cdot \tilde{\boldsymbol{W}} = AX + BY + CZ + D = 0 \tag{2.24}$$

Defining lines in 3D is not so easy as in 2D, but they can be written with two points, expressing the line as a linear combination of the two points:

$$\tilde{\boldsymbol{L}} = (1 - \lambda)\boldsymbol{X_1} + \lambda \boldsymbol{X_2} \tag{2.25}$$

if there is a restriction $0 \leq \lambda \leq 1$ then the equation represents the line segment joining $\boldsymbol{X_1}$ and $\boldsymbol{X_2}$

2.3.2 Image Transformations

There are some transformations that can be performed in points or even objects on an image and they are useful for e.g. to relate multiple cameras, to model motion and to calculate poses. Here is defined a set of 2D planar transformations and their properties. This set of transformations is schematized in Figure 2.9 and the image properties preserved by each transformation are summarized in Table 2.1.

- Translation transformation

  Translations only transforms the position of an object, maintaining its orientation, geometry and dimensions. It has two DOF: (x, y). Its equation is defined in (2.27a).

- Euclidean transformation

  The Euclidean transformation, also known as *2D rigid body motion* or as *rotation + translation* transformation, performs a simple movement of the object, rotating and translating it. It preserves geometry and all dimensions. It has three DOF: (x, y, θ). Its equation is defined in (2.27b).

- Similarity transformation (scaled rotation)

  The similarity transformation is also known as *scaled rotation* performs an Euclidean transformation with a scale factor. It does not require $a^2 + b^2 = 1$, as the rotation matrix $\boldsymbol{R}$ in an Euclidean transformation. This transformation does not preserve the dimensions of the object but preserves its geometry. It has four DOF: (x, y, θ, α). Its equation is defined in (2.27c).

- Affine transformation

  This transformation performs a linear operation on the image, distorting the object. This operation still preserves lines and parallelism between lines. It has six DOF: $(x, y, a_{11}, a_{12}, a_{21}, a_{22})$. Its equation is defined in (2.27d).

- Projective transformation

  The projective transformation is also known as *perspective transform* or *homography*, operates on homogeneous coordinates so that the inhomogeneous coordinates of a point are stated in Equations (2.26).

$$x' = \frac{h_{11}x + h_{12}y + h_{13}}{h_{31}x + h_{32}y + h_{33}} \tag{2.26a}$$

$$y' = \frac{h_{21}x + h_{22}y + h_{23}}{h_{31}x + h_{32}y + h_{33}}. \tag{2.26b}$$

It has eight DOF: $(h_{11}, h_{12}, h_{13}, h_{21}, h_{22}, h_{23}, h_{31}, h_{32})$; one of $\tilde{\boldsymbol{H}}$ entries is just a scale factor, so h_{33} may be chosen considering the other entries of $\tilde{\boldsymbol{H}}$. Homography is a very common transformation present in real images captured for a camera, and it is particularly useful when it is necessary to straighten an object on an image, like a squared shape, as illustrated in Figure 2.9 (used further in Section 3.2.1).

$$Translation \quad \boldsymbol{x}' = \boldsymbol{x} + \boldsymbol{t} \quad \tilde{\boldsymbol{x}}' = \begin{bmatrix} \boldsymbol{I}_{2\times2} & \boldsymbol{t} \\ \boldsymbol{0}_{1\times2} & 1 \end{bmatrix} \tilde{\boldsymbol{x}} \quad \boldsymbol{I}_{2\times2} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad \boldsymbol{t} = \begin{bmatrix} x \\ y \end{bmatrix} \tag{2.27a}$$

$$Euclidean \quad \boldsymbol{x}' = \boldsymbol{R}\boldsymbol{x} + \boldsymbol{t} \quad \tilde{\boldsymbol{x}}' = \begin{bmatrix} \boldsymbol{R} & \boldsymbol{t} \\ \boldsymbol{0}_{1\times2} & 1 \end{bmatrix} \tilde{\boldsymbol{x}} \quad \boldsymbol{R} = \begin{bmatrix} cos(\theta) & -sin(\theta) \\ sin(\theta) & cos(\theta) \end{bmatrix} \tag{2.27b}$$

$$Similarity \quad \boldsymbol{x}' = \alpha\boldsymbol{R}\boldsymbol{x} + \boldsymbol{t} \quad \tilde{\boldsymbol{x}}' = \begin{bmatrix} \alpha\boldsymbol{R} & \boldsymbol{t} \\ \boldsymbol{0}_{1\times2} & 1 \end{bmatrix} \tilde{\boldsymbol{x}} \quad \alpha\boldsymbol{R} = \begin{bmatrix} a & -b \\ b & a \end{bmatrix} \tag{2.27c}$$

$$Affine \quad \boldsymbol{x}' = \boldsymbol{A}\boldsymbol{x} + \boldsymbol{t} \quad \tilde{\boldsymbol{x}}' = \begin{bmatrix} \boldsymbol{A} & \boldsymbol{t} \\ \boldsymbol{0}_{1\times2} & 1 \end{bmatrix} \tilde{\boldsymbol{x}} \quad \boldsymbol{A} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \tag{2.27d}$$

$$Projective \quad \begin{matrix} not\ linear \\ see\ (2.26) \end{matrix} \quad \tilde{\boldsymbol{x}}' = \tilde{\boldsymbol{H}}\tilde{\boldsymbol{x}} \quad \tilde{\boldsymbol{H}} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \tag{2.27e}$$

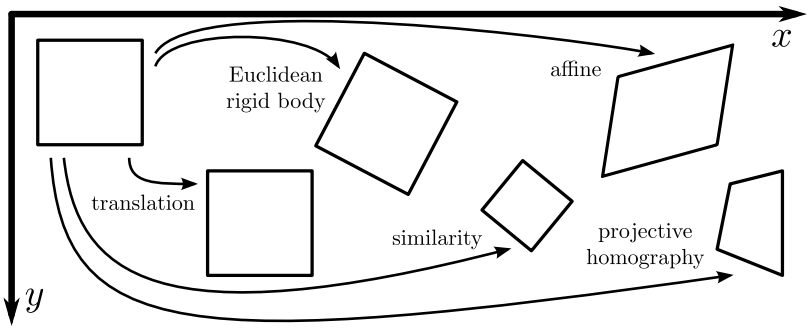


Figure 2.9: Basic set of 2D planar transformations

Transformation	# DOF	Position	Orientation	Length	Geometry	Parallelism	Lines
Translation	2	✗	✓	✓	✓	✓	✓
Euclidean	3	✗	✗	✓	✓	✓	✓
Similarity	4	✗	✗	✗	✓	✓	✓
Affine	6	✗	✗	✗	✗	✓	✓
Projective	8	✗	✗	✗	✗	✗	✓

Table 2.1: Invariance of image objects properties, according to each transformation in 2D

This set of transformations can also be done in 3D (imagine a cube, instead of a square) with similar equations. All properties remain the same except for the number of DOF. The set of transformations in 3D is expressed in Table 2.2.

Transformation	# DOF	Matrix form	
Translation	3	$\tilde{\boldsymbol{X}}' = \begin{bmatrix} \boldsymbol{I}_{3\times3} & \boldsymbol{T} \\ \boldsymbol{0}_{1\times3} & 1 \end{bmatrix} \tilde{\boldsymbol{X}}$	$\boldsymbol{I}_{3\times3} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \boldsymbol{T} = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$
Euclidean	6	$\tilde{\boldsymbol{X}}' = \begin{bmatrix} \boldsymbol{R} & \boldsymbol{t} \\ \boldsymbol{0}_{1\times3} & 1 \end{bmatrix} \tilde{\boldsymbol{X}}$	$\boldsymbol{R} = \begin{bmatrix} c_\psi c_\theta & -s_\psi c_\phi + c_\psi s_\theta s_\phi & s_\psi s_\phi + c_\psi s_\theta c_\phi \\ s_\psi c_\theta & c_\psi c_\phi + s_\psi s_\theta s_\phi & -c_\psi s_\phi + s_\psi s_\theta c_\phi \\ -s_\theta & c_\theta s_\phi & c_\theta c_\phi \end{bmatrix}$
Similarity	7	$\tilde{\boldsymbol{X}}' = \begin{bmatrix} s\,\boldsymbol{R} & \boldsymbol{T} \\ \boldsymbol{0}_{1\times3} & 1 \end{bmatrix} \tilde{\boldsymbol{X}}$	
Affine	12	$\tilde{\boldsymbol{X}}' = \begin{bmatrix} \boldsymbol{A} & \boldsymbol{T} \\ \boldsymbol{0}_{1\times3} & 1 \end{bmatrix} \tilde{\boldsymbol{X}}$	$\boldsymbol{A} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$
Projective	15	$\tilde{\boldsymbol{X}}' = \tilde{\boldsymbol{H}}\tilde{\boldsymbol{X}}$	$\tilde{\boldsymbol{H}} = \begin{bmatrix} h_{11} & h_{12} & h_{13} & h_{14} \\ h_{21} & h_{22} & h_{23} & h_{24} \\ h_{31} & h_{32} & h_{33} & h_{34} \\ h_{41} & h_{42} & h_{43} & h_{44} \end{bmatrix}$

Table 2.2: Transformations in 3D

2.3.3 Pinhole Camera Model

Starting with the device that is able to capture the real world to a 2D structure, we review here the most used simplification of a camera model – the pinhole camera. This model aims to translate point $\tilde{\boldsymbol{X}} = (X, Y, Z, 1)[m]$ in real world into image points $\tilde{\boldsymbol{x}} = (x, y, 1)[m]$. It is based on the assumption that all the light passes through a single, infinitesimal point and projects on the sensor, forming the image that everyone is familiar with. The pinhole model approximation doesn't need to consider the focusing issue and it considers that enough light passes through the hole.

To convert a 3D point into a 2D point, a simple perspective projection is able to do it:

$$x = -f\frac{X}{Z} \tag{2.28a}$$

$$y = -f\frac{Y}{Z} \tag{2.28b}$$

where $f[m]$ is the focal distance (in Figure 2.10). In matrix form, this projection is in (2.29), where $\boldsymbol{\Pi_0}$ is the standard projection matrix.

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = f \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \quad \Leftrightarrow \quad \tilde{\boldsymbol{x}} = f\,\boldsymbol{\Pi_0}\tilde{\boldsymbol{X}} \qquad \boldsymbol{\Pi_0} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{2.29}$$

In order to transform metric points $\tilde{\boldsymbol{x}}$ into pixels points $\tilde{\boldsymbol{x}}'$ in image, it must be considered a set of parameters – the *intrinsic parameters* (denoted as a matrix $\boldsymbol{K}$). The intrinsic parameters are four: focal distances (f_x and f_y [pixels]) and optical center (O_x and O_y [pixels]). There is usually a fifth intrinsic

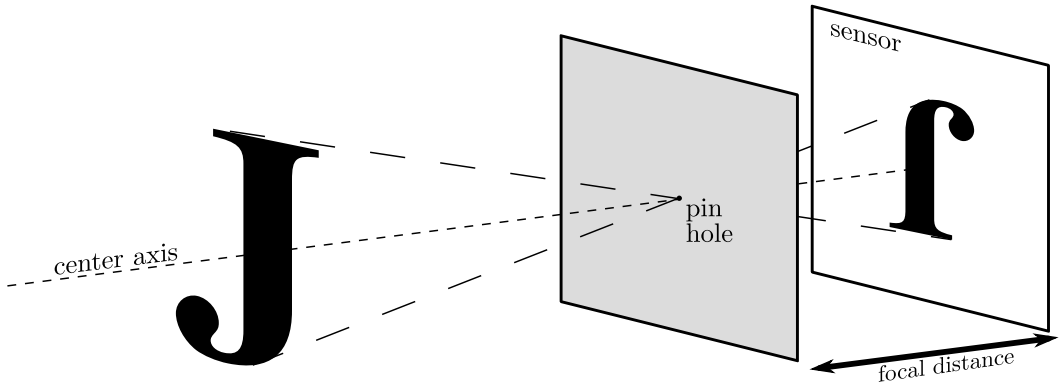


Figure 2.10: Pinhole camera draw

parameter γ which is the skew coefficient but as it is often 0, sometimes it is omitted. The conversion from metric form into pixels form (u, v) is done recurring to Equation (2.30).

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & \gamma & O_x \\ 0 & f_y & O_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad \Leftrightarrow \quad \tilde{\boldsymbol{u}} = \boldsymbol{K}\tilde{\boldsymbol{x}} \qquad \boldsymbol{K} = \begin{bmatrix} f_x & \gamma & O_x \\ 0 & f_y & O_y \\ 0 & 0 & 1 \end{bmatrix} \tag{2.30}$$

To complete the camera model, we also need to consider that the origin of the camera usually differs from the origin of the world. In order to take this into consideration, a rotation matrix $\boldsymbol{R}_{3\times3}$ and a translation vector $\boldsymbol{T}_{3\times1}$ must be added to the previous equations, thus forming the matrix denoted as *projection matrix* ($\boldsymbol{\Pi}_{4\times3}$). Therefore, the Equation (2.31) is the full perspective model.

$$\tilde{\boldsymbol{u}} = \boldsymbol{\Pi}\tilde{\boldsymbol{X}} \Leftrightarrow \tilde{\boldsymbol{u}} = \boldsymbol{K} \cdot \begin{bmatrix} \boldsymbol{R} & \boldsymbol{T} \end{bmatrix} \cdot \tilde{\boldsymbol{X}} \Leftrightarrow \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & \gamma & O_x \\ 0 & f_y & O_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} R_{11} & R_{12} & R_{13} & T_x \\ R_{21} & R_{22} & R_{33} & T_y \\ R_{31} & R_{32} & R_{33} & T_z \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{2.31}$$

This matrix $\boldsymbol{\Pi}$ is defined up to a scale so it has eleven DOF.

The task of determining the intrinsic parameters is called *calibration of a camera*. As the intrinsic parameters only depend on the camera itself (do not depend on its position and orientation), the calibration process in this thesis is simplified by using a computational tool publicly available [4]. The process consists in taking a set of pictures of a calibration board and let the algorithm calculate its intrinsic parameters. This tool also gives additional parameters: lens distortions $(k_1, k_2, p_1, p_2, k_3)$. This non linear distortion is modeled and applied as in Equations (2.32).

[4] *camera_calibration* package under ROS [30]

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} \boldsymbol{R} & \boldsymbol{T} \end{bmatrix} \cdot \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{2.32a}$$

$$\begin{cases} x' = x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) + 2p_1 xy + p_2(r^2 + 2x^2) \\ y' = y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) + 2p_2 xy + p_1(r^2 + 2y^2) \end{cases} \tag{2.32b}$$

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & \gamma & O_x \\ 0 & f_y & O_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} \tag{2.32c}$$

where $r = x^2 + y^2$ and the Equation (2.32b) is responsible for the distortion model.

2.3.4 From 2D to 3D

As previously written, the camera matrix $\boldsymbol{\Pi}$ is a composition of three matrices: $\boldsymbol{K}$, $\boldsymbol{R}$ and $\boldsymbol{T}$. In order relate 3D to 2D and vice versa, the matrix $\boldsymbol{\Pi}$ is required. If $\boldsymbol{K}$ is already known, the 3D pose perception of an object identified in the image, resumes to find both $\boldsymbol{R}$ and $\boldsymbol{T}$ i.e. the translation and the rotation from the camera to the object reference. This process is usually done by matching a few points of the image with the corresponding 3D points in the world and calculate which transformation is responsible for the image captured. This method is known as Perspective-n-Point (PnP) problem. There are several approaches on how to solve the PnP problem, some methods are iterative methods, some others are not.

This topic and methods are very well explained and compared by Moreno-Noguer et al.[31]. They write the coordinates of the n 3D points as a weighted sum of four virtual control points, thus reducing the problem to estimating the coordinates of the control points in the camera frame, which can be done by expressing these coordinates as weighted sum of the eigenvectors of a 12×12 matrix and solving a small constant number of quadratic equations to pick the right weights. This paper shows that this method has a complexity $O(n)$ and therefore is very useful to fast and inexpensive computation.

In order to find the correct orientation and translation using four points detected in the image (the four corners of a square, further analysed) we need the four 3D positions of those points. As the absolute position of the square is unknown (unless a previous and boring calibration process is done) there are some facts to consider. To solve this P4P problem, it is usually assumed that the coordinates of the points in 3D space are known. If a squared figure is taken, it could be assumed that the square is horizontally centered in the origin of the world frame, and then its corners coordinates are known. This trick is used further in Section 3.2.1.

To illustrate how this problem is solved, one can imagine the four points of a square in the image plane, projected as lines in 3D, and trying to fit a square with fixed dimensions, in such a way that the four corners must coincide with the lines, as illustrated in Figure 2.11.

Unfortunately, due to mathematical issues, the correct identification of the pose solving the PnP problem is very difficult to achieve when the points have a special configuration, specifically when four

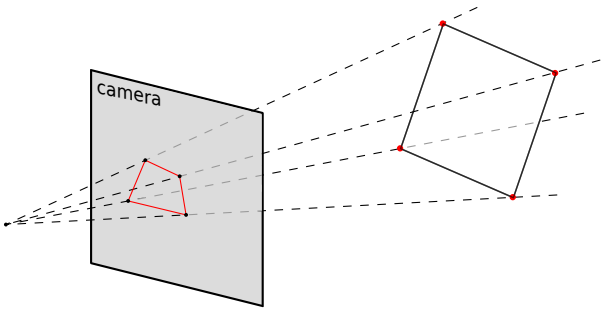


Figure 2.11: Fitting a 3D object in an image projection

corners of a square are equidistant from the camera (front view of the square). This problem will spoil the ability to correct identify the pose of a camera in certain conditions, as it will be explained further in Chapter 3.

2.3.5 Image Processing

The analysis of an image is essential in computer vision. The analysis usually starts by detecting edges or colors but it can be extended to objects identification, shapes detection, objects matching, etc... It can also be used with a stream of images (video) to detect cars, people moving or even a terrorist inside an airport. The uses of image processing and computer vision is very diverse. In this thesis, only a small part of the literature about computer vision and image processing will be reviewed.

One basic operation over an image is filtering. A filter may be used for noise, reduction, image quality improving, background subtraction, finding contours, etc... In order to find known shapes (like squares) it is essential to find contours and lines. This may be achieved by applying filters in an image.

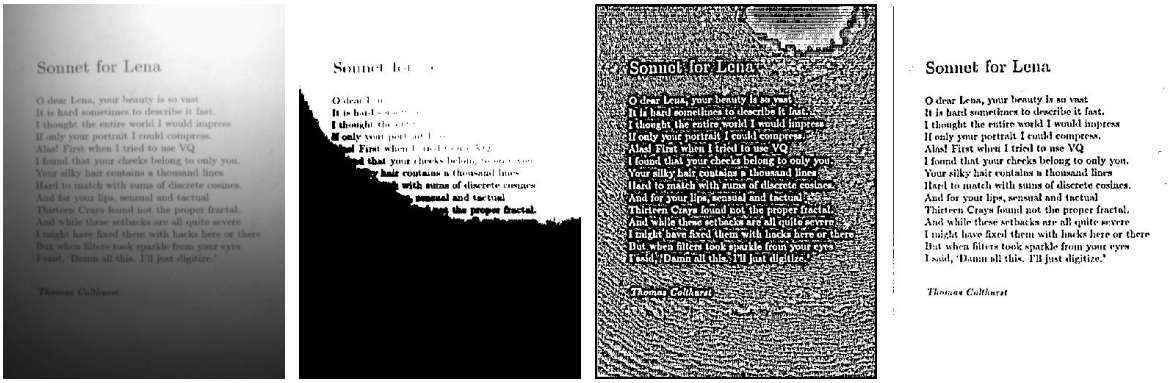
A linear filter may be computed through the *convolution operation(*$*$*)*. The filtering operation consists on a mask $M(m,n)$ (or *kernel*) which is convoluted with the image. The result is a filtered image. The convolution operation is described in Equation (2.33)[5].

$$\boldsymbol{J}(m,n) = \boldsymbol{I} * \boldsymbol{M} = \sum_{p}\sum_{q} \boldsymbol{I}(p-m, q-n) \cdot \boldsymbol{M}(p,q) \tag{2.33}$$

A useful processing method is the *adaptive thresholding*. The global threshold method consist in analysing only the pixels with a greater then threshold value, discarding all the others. This process brings problems when the image has a strong illumination gradient. The adaptive thresholding tries to solve this issue by analysing the neighbourhood of each pixel in order to find a good threshold. The method to find the threshold can be a mean (bigger or smaller), c-means, median, etc... A good example where the adaptive threshold is well used can be viewed in Figure 2.12.

As previously mentioned, finding contours with filters is very handful when trying to find squares (needed in Section 3.2.1). To do this, there are several options to consider. A simple one is using a *Sobel Mask* like (2.34).

[5]The equation is only true if a non-usual matrix coefficients notation is used, with $\boldsymbol{M}(0,0)$ being the center of the mask matrix, instead of the upper left corner

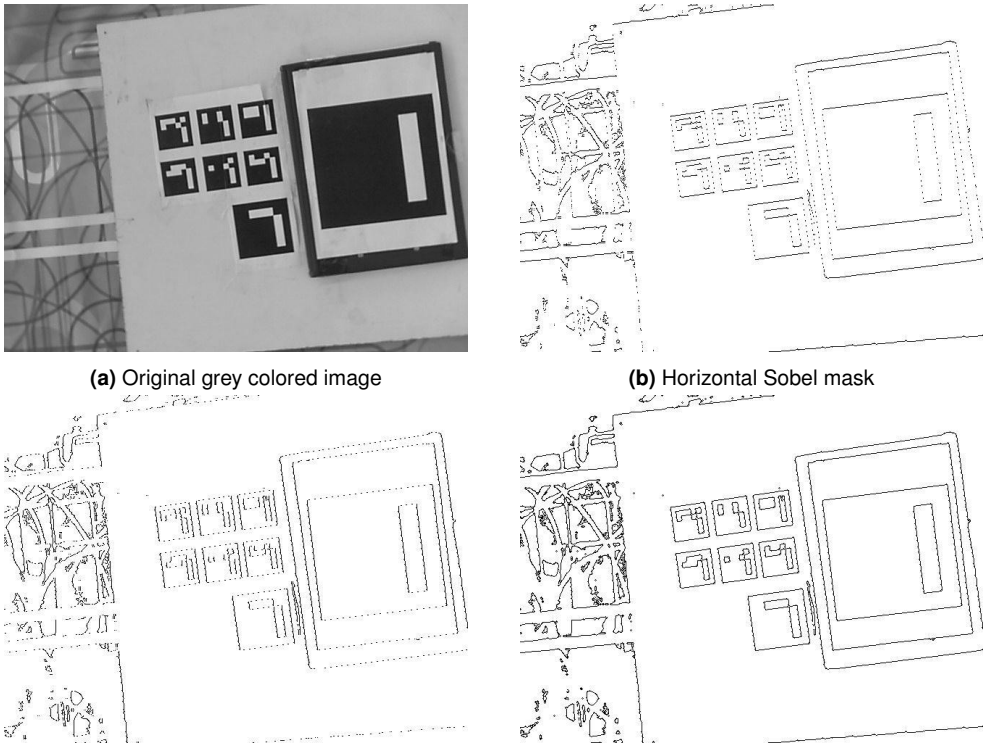


(a) Original grey colored image **(b)** Global thresholding **(c)** Adaptive thresholding (mean 7×7) **(d)** Adaptive thresholding (c-mean $C = 7$)

Figure 2.12: Image with a strong illumination gradient, filtered by different threshold filters

$$\boldsymbol{M}_{Sobel1} = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \qquad \boldsymbol{M}_{Sobel2} = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \qquad \boldsymbol{M}_{Sobel3} = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix} \tag{2.34}$$

which, after applying a threshold limit, produce results according to Figure 2.13.



(a) Original grey colored image **(b)** Horizontal Sobel mask

(c) Vertical Sobel mask **(d)** A different Sobel mask

Figure 2.13: Images filtered with a convolution with a Sobel mask

The OpenCV library uses a different and more complex algorithm, which will not be here explained but can be consulted in [32].

2.4 Control

A controller interacts with the system in order to lead its output to exhibit the desired behaviour, through the manipulation of the input variables, and by having access to a feedback signal of the system's outputs and/or state variables.

One of the most used controller is the Proportional-Integral-Derivative (PID) controller, often used for its simplicity and effectiveness. The controller tries to minimize the error $e(t) = x_{goal} - x(t)$, based on the system's output $x(t)$ and on the desired state for the system x_{goal}, and actuates on the system's input variables. This controller has three constant parameters K_p, K_i and K_d, used to respectively tune the proportional, the integral and the differential components, as it is suggested in Equation (2.35) and it is schematized in Figure 2.14.

$$u(t) = K_p\, e(t) + K_i \int_0^t e(\tau)d\tau + K_d\, \frac{d}{dt}e(t) \tag{2.35}$$

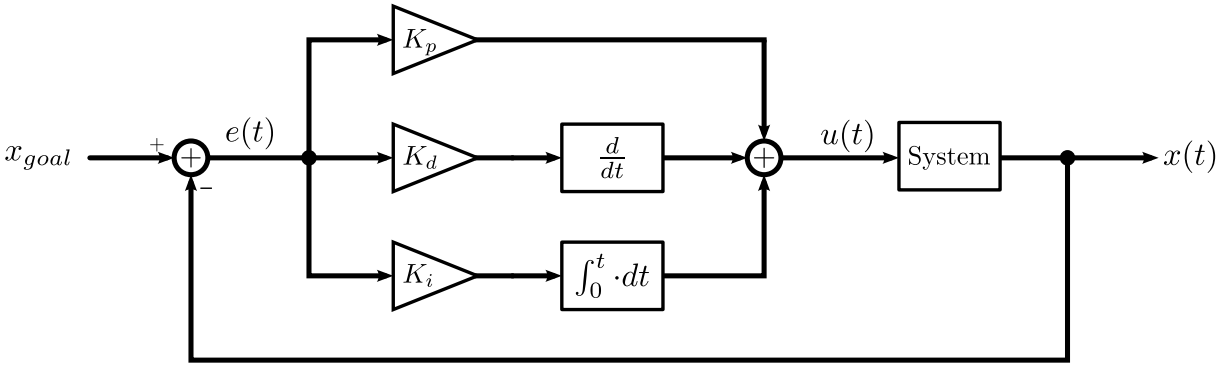


Figure 2.14: Block diagram for PID controller

Each one of the three components has a specific job.

- Proportional component

  The proportional component consists on reducing an error by applying a force proportional to the error $u_p(t) = K_p\, e(t)$. This is the most important part of the PID controller, since it tells how far the system's output from the desired value. Since it acts based only on the present state of the system, it may present an oscillatory behaviour, more obvious on fast systems. Moreover, the system's output may present a steady state error, when the forces applied on the system and the force applied by the controller are in equilibrium, which are not corrected by the proportional component.

- Integral component

  This component acts not only using present values, but also using past values of the system's output. The influence of the integral component gets stronger as the system keeps away from the target state along the time, thus correcting the steady state error. Although this error gets corrected, the integral component increases the speed of response as well as the instability on the system. This instability is usually not desired and may be compensated using the derivative component.

- Derivative component The derivative component acts on the derivative of the error, thus predicting the future behaviour of the system's output and limiting the response of the controller, bringing stability to the system.

The comparison between each component effect can be observed in Figure 2.15. The previous characteristics can be observed, like the overshoot of both P and PI controllers, the steady state on the P and PD controllers and the oscillatory behaviour on the P and PI controllers. The PID gathers all this characteristics, thus performing better.

The complementarity of these three components make the PID controller a good controller and a common choice for fast implementations. Nevertheless, this controller does not guarantee an optimal control nor system's stability.

For the matters of analysis and concept proving, the PID controller performs good enough on the quadcopter's system and therefore the PID controller is the chosen method for controlling the quadcopter.

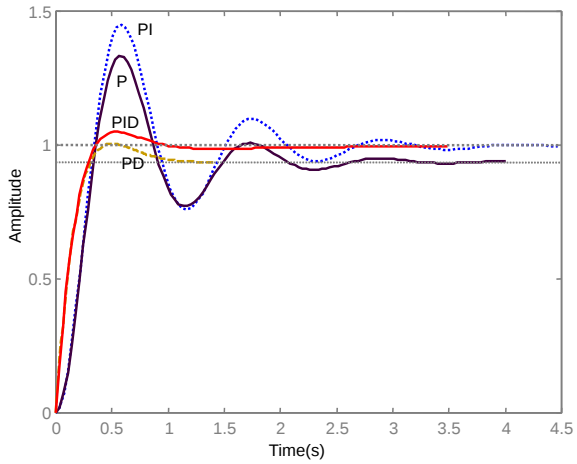


Figure 2.15: Comparison between the time response of P, PD, PI and PID controllers

3

Vision Based Autonomous Control

Contents

3.1 Introduction

In order to build a good controller able to land a quadcopter on a visually observed target, the first step is to get a good estimation for its pose. The location and orientation of the aerial vehicle, relative to a visually observed target, are essential.

In this chapter we introduce and discuss the methods used for quadcopter target-tracking using vision in the control loop and the options taken. The algorithms that are essential to this work are also discussed here. It will be discussed how the detection of the target is made and how it is interpreted in the system, how the control of the quadcopter is designed and what strategies are used to add more reliability to the controller.

The integration of all algorithms and the implementation of them in the quadcopter is addressed in the end of the chapter.

For better understanding the frames involved in this chapter, Figure 3.1 shows the three used frames: B-Frame, quadcopter's body frame (top); img-Frame, camera's image frame (middle); and G-Frame, Ground frame (bottom), coincident with the reference point on the markers board, mentioned in Section 3.2.2.

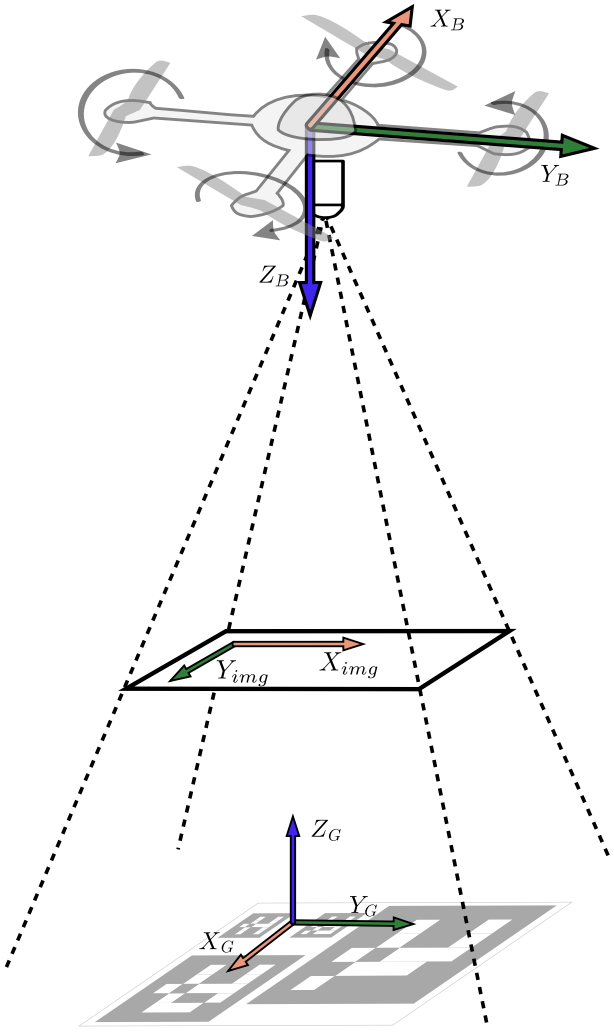


Figure 3.1: Gathering all frames
B-Frame (quadcopter's body), img-Frame (camera's image) and G-Frame (ground)

3.2 Vision-Based Target Localization by a Quadcopter

In order to control a system like the quadcopter, very reliable sensor is needed that can provide a good estimation about the system's state. Like most of mobile robots, the dead-reckoning approach, by integrating the IMU sensor data, is very noisy and it gets compromised very quickly. The fact that the quadcopter is a very unstable system makes the dead-reckoning even worse, discarding completely its usage (only) to control the quadcopter. Nevertheless, it would not suit the need for relative localization, as it is intended in this thesis.

The onboard webcam confers quadcopter a full autonomy and it allows to operate the quadcopter in almost any environment, without the need to setup a set of cameras or to calibrate the environment lighting. As long as the camera is previously calibrated (just needed once) and the set of markers are perfectly known (each marker size and their position, relative to each other), this system is able and ready to operate. The usage of markers allows an easy and fast computation so it can be used in real time.

The vision block will operate based on the detection of a specific AR marker (or set of markers), that will be identified as the target. In order to consider a docking station as the target, a markers board must be used and attached to the docking station. As the markers board can be easily made, by simply printing a sheet paper, this procedure is very easy to do.

The algorithms used for detection and identification of the markers board are reviewed in the next section. The detection of the camera's pose is based on the previously known geometry and sizes of the markers, by trying to find the link between the observations and the physical constraints, defined by the markers geometry.

3.2.1 ArUco Library

To detect the marker with a regular webcam (RGB camera) a library called ArUco was used, developed by Aplicaciones de la Visión Artificial (AVA) from the Universidad de Córdoba (UCO). This library is "a minimal library for Augmented Reality applications based on Open Source Computer Vision (OpenCV)" [22] and has an API for developing in C++, very useful in this work.

ArUco works with the concepts stated in Section 2.3 in order to get real world 3D information by analysing an image. It searches the image and tries to find a well known marker, which consists on a printed 2D mark, black and white colored and in a squared shape (Figure 3.2).

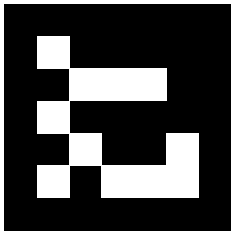


Figure 3.2: ArUco marker example (id number 201)

The marker can be seen as a 7×7 boolean matrix, with the outer cells filled with black color (which makes a perfect square, easy to find with image processing). The remaining 5×5 matrix is a 10-bits coded ID (up to 1024 different IDs), where each line represents a couple of bits. Each line has only 2 bits of information out of the 5 bits, with the other 3 being used for error detection. These extra 3 bits add asymmetry to the markers, i.e. only a few valid markers are symmetric (e.g. Figure 3.3), which allow a unique orientation detection for most of the markers. The codification used is a slight modification of the Hamming Code (the first bit is inverted to avoid a valid black square). So, any ArUco AR marker can be created by converting a number to binary, splitting into 5 groups of two bits and by putting each couple in one line of the marker, from the top to the bottom. The example in Figure 3.2 is the number 201, which is (00 11 00 10 01) in binary. It is easy to confirm the code by referencing to the information in Table 3.1.

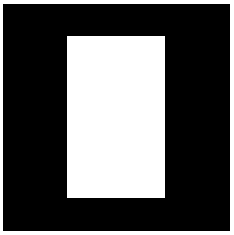


Figure 3.3: ArUco marker example (id number 1023) where symmetry is present

Number	Binary	Marker code
0	00	
1	01	
2	10	
3	11	

Table 3.1: Codification of an ArUco marker

The ArUco library processes the image supplied and detects all markers and each one's ID, as well as their position and orientation in the 3D world, relative to the camera. The open source code of ArUco is based in OpenCV, which is a library highly optimized for image processing. Therefore, all the calculations are performed in fractions of seconds so it can be used in real time applications. The main code is not very complex and the markers detection is performed like:

1. Converting color image to gray image
2. Apply adaptive thresholding
3. Detect contours
4. Detect rectangles
   (a) Detect corners
   (b) Detect linked corners
   (c) Consider only figures with four connected corners
5. For all detected (possible) markers

(a) Calculate homography (from corners)

(b) Threshold the area using Otsu, which assumes a bimodal distribution and finds the threshold that maximizes the extra-class variance while keeping a low intra-class variance.

(c) Detect and identify a valid marker, which respects Table 3.1, and if not detected tries the four rotations

6. Consider only valid markers

7. Detect extrinsic parameters (by supplying the calibration matrix, distortion matrix and physical markers dimensions)

The extrinsic parameters are calculated with the help of an OpenCV function: `solvePnP()` which uses the method described in Section 2.3.4. For each marker, the four corners in the image and their respective 3D coordinates are provided to the algorithm, which will be:

$$\boldsymbol{X_{1-4}} = \begin{bmatrix} -d/2 & d/2 & d/2 & -d/2 \\ d/2 & d/2 & -d/2 & -d/2 \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{3.1}$$

where $d[m]$ is the dimension of the side of the printed squared marker. As it can be observed, all the four points have their coordinate $Z = 0$ and their (X, Y) coordinates are disposed as a square, which means that the marker is considered to be horizontal, in the origin of the world reference, as suggested in Figure 3.4 and so the extrinsic parameters will be the rotation and translation of the camera relative to the marker.

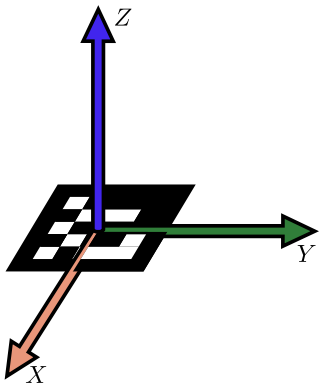


Figure 3.4: A marker at the world's reference

This ArUco library has also a built-in tool that allows to plot a projection of a 3D cube in the image, as if the cube was laid on the square. This tool is helpful for development and debugging because it makes it easier for a person to confirm the extrinsic parameters calculated by the algorithm. An example of the usage of this tool can be observed in Figure 3.5.

The ArUco library is also able to work with a markers board (a set of markers with a fixed size and relative displacement), which is used to reduce the noise from the detection (more markers, more information, less error). The board is created with ArUco itself and has the limitations of the library, i.e. all the markers have the same size, and the distances from each other are fixed and proportional to the

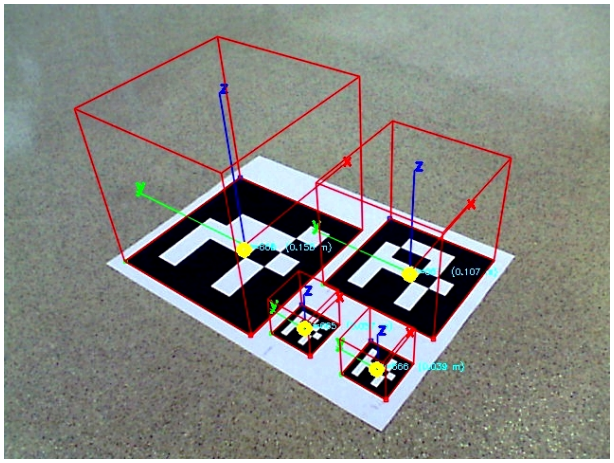
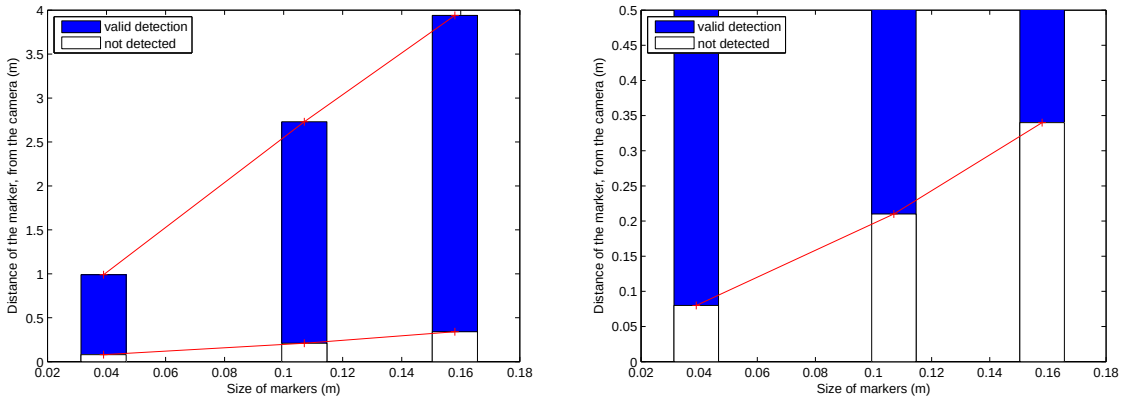


Figure 3.5: The visualization of a correctly detected marker, using a real camera

size of the markers. This is a small disadvantage and not very useful in this work. The reason why and how it was overcome is reviewed in the next Section 3.2.2.

Due to the resolution and the size of the camera image, the detection of a marker is only valid within a distance range from the camera to the marker. When in short distances from the camera to the marker, the camera is not able to see the whole marker if it is too big. When in long distances, the resolution of the camera does not allow a pattern recognition and therefore it is not possible to identify the marker. In Figure 3.6, they were used squared markers with 39, 107 and 158 mm on the side.



(a) Distance range for markers detection

(b) Minimum distance for markers detection

Figure 3.6: Distance range for markers detections, function of markers sizes

3.2.2 Improvements over ArUco

One of the problems when considering the usage of markers on the target is the size of the marker. As it is intended to be seen by a camera on the quadcopter, and it is supposed to take off and land on the docking station, and during the maneuvers the camera may be very far or very close to the marker, the size of it being a relevant issue. If the marker is too big, when the quadcopter is close to the marker the camera won't be able to see it (it will easily get out of the view) but if the marker is too small, the camera won't be able to see it when the quadcopter is high. Instead of choosing a middle size marker (that would be bad in both cases) it was found a better solution. The group of markers sizes chosen

```
## id = size_in_m
## id = size_in_m (disp_x,disp_y,disp_z)

0 = 0.195 (-0.013,-0.176,0.01)
4 = 0.088
3 = 0.089 (-0.097,0.014,0)
6 = 0.057 (0,0,0)
```

Table 3.2: File containing all markers configurations

in Figure 3.6 allow a detection range from ~8 cm (approximately the distance from the camera to the ground when the quadcopter is landed) to almost 4 m.

By having multiple markers with different sizes, it is possible to detect a marker when the quadcopter is far from the target (bigger marker) and when it is closer (smaller marker). As the ArUco itself do not provide the ability to detect multiple markers if they have different dimensions, some changes to the open source code were made.

In ArUco, each marker is a collection of four points (corners) and some other attributes. To meet this new need, it was added to the marker a dimension attribute. This attribute is defined after the marker's id is known (Item 5c of the algorithm procedures list in Section 3.2.1). In order to correctly define the dimension of the marker, there is a correspondence list, with the size of each existent ID. Because this dimensions list should be versatile, this functionality was implemented via an external file, which lists the existing IDs and their correspondent size.

As it is explained in the next Section 3.3, besides the markers side information, we also needed to provide the program the information about the relative position of each marker, which is stored in a text file. The file has one marker information per line and each line may have the ID and its size, or the ID, size and position. Comments are also valid with '#'. An example file is in Table 3.2.

As long as there is a small preparation of the markers system, this system can run in any place, without further measures *in situ*. The simplest way to prepare the markers system is getting one big board (in this thesis, a A4 sheet paper was used) with a collection of markers and measuring their real sizes and displacements. This procedure is needed just once. With this new functionality, the VBPEA is able to process more than one size of markers simultaneously.

The creation of the VBPEA implies another change in ArUco, explained in Section 3.3. This group of new features constitute the VBPEA block used in this work.

3.3 Vision-Based Pose Estimation

3.3.1 Vision-Based Pose Estimation Algorithm (VBPEA)

As it is written in Section 3.2.1, the VBPEA block returns a position and an orientation of each marker (a pose of each marker), related to the camera frame. Although is is very interesting, it still does not provide the current pose of the camera. To obtain this, another step is needed. There is the information

about the pose of the marker WRT the camera frame (Cam-Frame) but we need to find the pose of the camera WRT the ground frame (G-Frame).

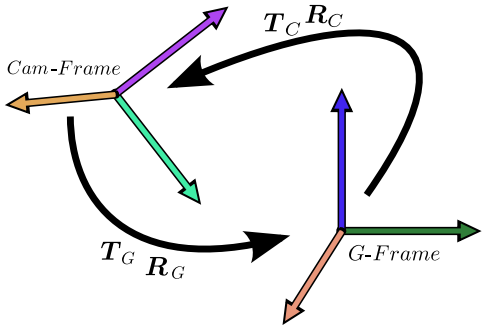


Figure 3.7: Relations between $Cam\text{-}Frame$ and $G\text{-}Frame$

In terms of what it is sketched in Figure 3.7, the VBPEA provides the $\boldsymbol{R}_G$ and $\boldsymbol{T}_G$, i.e. the parameters for a rigid body transformation from the Cam-Frame to the G-Frame. By applying a rotation matrix property $\boldsymbol{R}^{-1} = \boldsymbol{R}^T$, the inverse operation is easily calculated in Equation (3.2).

$$\begin{aligned} \boldsymbol{x}' = \boldsymbol{R}\boldsymbol{x} + \boldsymbol{t} \Leftrightarrow \boldsymbol{R}^{-1}(\boldsymbol{x}' - \boldsymbol{t}) = \boldsymbol{x} \Leftrightarrow \\ \Leftrightarrow \boldsymbol{x} = \boldsymbol{R}^T(\boldsymbol{x}' - \boldsymbol{t}) \Leftrightarrow \boldsymbol{x} = \boldsymbol{R}^T\boldsymbol{x}' - \boldsymbol{R}^T\boldsymbol{t} \end{aligned} \tag{3.2}$$

Looking at the Figure 3.7, the new rotation matrix is $\boldsymbol{R}_C = \boldsymbol{R}_G^T$ and the new translation vector is $\boldsymbol{T}_C = -\boldsymbol{R}_G^T \cdot \boldsymbol{T}_G$. Applying this new equation to the vision based estimation algorithm, the new output is the camera pose WRT the G-Frame.

But if the algorithm runs just like this, there will be inconsistencies in the detected extrinsic parameters, because each detected marker gives the camera pose as if it is the origin of the G-Frame. The objective then is that the markers are considered not to be in the origin of the G-Frame, but they must be considered to be on their real relative positions (after deciding what is the reference frame).

As already stated in Section 2.3.4, the `solvePnP()` function will calculate the position and rotation of the marker from a set of correspondences between 3D points to 2D image points. In order to achieve the multiple markers pose detection it is made a slight change in the algorithm. Instead of calculating each marker position individually, all the corners of all markers will be considered when using the `solvePnP()` function. With this change, the pose will be calculated from a set of correspondences between $N\times4$ 3D points to $N\times4$ 2D image points, where N is the number of markers detected in the image. These sets of points must be built considering the set of $N\times4$ 2D image points is the corners of the markers detected in the image and the set of $N\times4$ 3D points is built considering the real positions of each mark, relative to the reference point. The algorithm which describes how to build the set of 3D points is described in Table 3.3.

These modifications bring three achievements:

- increase of the precision on calculating the camera pose, due to the increased number of markers used.
- the pose is calculated based on one reference point, instead of multiple markers points;

```
for each marker i=0:N_markers {
        halfSize = Marker(i).size / 2.0;
        Xmarker  = Marker(i).center.x;
        Ymarker  = Marker(i).center.y;
        Zmarker  = Marker(i).center.z;

        Point(0 + i*4).x = -halfSize  + Xmarker;
        Point(0 + i*4).y = -halfSize  + Ymarker;
        Point(0 + i*4).z =      0     + Zmarker;
        Point(1 + i*4).x = -halfSize  + Xmarker;
        Point(1 + i*4).y =  halfSize  + Ymarker;
        Point(1 + i*4).z =      0     + Zmarker;
        Point(2 + i*4).x =  halfSize  + Xmarker;
        Point(2 + i*4).y =  halfSize  + Ymarker;
        Point(2 + i*4).z =      0     + Zmarker;
        Point(3 + i*4).x =  halfSize  + Xmarker;
        Point(3 + i*4).y = -halfSize  + Ymarker;
        Point(3 + i*4).z =      0     + Zmarker;
}
```

Table 3.3: Method to build the set of 3D points of the markers

- the pose provided by the `solvePnP()` function is given WRT the reference frame;

To maximize the performance of the algorithm, more than one marker detected every time is desired. When the camera is far enough from the markers board, it is easy to view all the markers on the board, but when the camera gets too close, the bigger markers will certainly go off the image. In order to avoid getting only one marker in the image when the quadcopter is too low on height and still have a small markers board, there are two small markers close to the reference point, so that it is possible to see two markers, even if the quadcopter is very low, and still have an A4 sized markers board. The board used for the experiments is described in Section 4.2 and it can be visualized in Appendix A.2.

In conclusion, the VBPEA is able to retrieve a pose of the camera; by processing the markers visualized through the camera, it is able to deal with multiple markers with different sizes and also take advantage of viewing multiple markers. As long as a correct information about the relative location of the markers is provided (if the markers are on a board it is easy), the VBPEA is able to retrieve the pose relative to a reference point defined by the user.

3.3.2 Vision-Based Horizontal Estimation Algorithm (VBHEA)

As it is stated before, the VBPEA takes care of getting the pose of the quadcopter by inspection of the image viewed by its bottom camera. But as it is stated in Section 2.3, there is a problem with the calculated pose when the camera is right above the marker, leading to miscalculations of the quadcopter's pose. This becomes a major problem since the objective is landing the quadcopter on the top of the target. So, lets analyse deeper how the pose estimation quality is affected by the position of the camera WRT the G-Frame.

The trust of the estimations depends on the angle of the markers board. When analysing this, it is possible to confirm (in Chapter 5) that only the (X, Y) estimation is significantly affected, but the

Z estimation is still very good. This means that it is not possible to control the quadcopter position on hovering with just the information from the VBPEA, but it is possible to control the height of the quadcopter with those data.

After having this information, we reach the conclusion that developing a different way of controlling the position of the quadcopter on hovering is needed. The approach taken consists on determining the (X, Y) position by removing some DOF of the quadcopter, namely the pitch and roll that are considered to be zero and locking the Z^* component, which is given by the VBPEA. Applying this on the mathematical equations from Section 2.3, the rotation matrix becomes much more simple and then the solution can be obtained from Equation (3.3). Because the translations vector $\boldsymbol{T}$ (position of the camera WRT the marker frame) should be written WRT the rotated frame, a rotation operation is also applied on $\boldsymbol{T}$.

$$\begin{gathered}\lambda\tilde{\boldsymbol{u}} = \boldsymbol{K}\cdot\begin{bmatrix}\boldsymbol{R} & -\boldsymbol{RT}\end{bmatrix}\cdot\tilde{\boldsymbol{X}} \Leftrightarrow \lambda\tilde{\boldsymbol{u}} = \boldsymbol{K}\cdot(\boldsymbol{RX}-\boldsymbol{RT}) \Leftrightarrow \boldsymbol{K}^{-1}\lambda\tilde{\boldsymbol{u}} = \boldsymbol{RX}-\boldsymbol{RT} \Leftrightarrow \\ \Leftrightarrow \boldsymbol{RT} = \boldsymbol{RX}-\boldsymbol{K}^{-1}\lambda\tilde{\boldsymbol{u}} \Leftrightarrow \boldsymbol{T} = \boldsymbol{R}^{-1}(\boldsymbol{RX}-\boldsymbol{K}^{-1}\lambda\tilde{\boldsymbol{u}}) \Leftrightarrow \boldsymbol{T} = \boldsymbol{X}-\boldsymbol{R}^{-1}\boldsymbol{K}^{-1}\lambda\tilde{\boldsymbol{u}} \Leftrightarrow \\ \Leftrightarrow \begin{bmatrix}T_x\\T_y\\T_z\end{bmatrix} = \begin{bmatrix}X\\Y\\Z\end{bmatrix} - \begin{bmatrix}c_\psi & -s_\psi & 0\\ s_\psi & c_\psi & 0\\ 0 & 0 & 1\end{bmatrix}^{-1}\begin{bmatrix}f_x & \gamma & O_x\\ 0 & f_y & O_y\\ 0 & 0 & 1\end{bmatrix}^{-1}\cdot\lambda\begin{bmatrix}u\\v\\1\end{bmatrix}\end{gathered} \tag{3.3}$$

As mentioned in Equations (2.21) and (2.23), there is a λ scale factor when dealing with homogeneous coordinates. This factor must be defined in order to solve this equation. In order to define this factor, the Z^* variable is used in Equation (3.4), by introducing the constraint $T_z = Z^*$. Because $\boldsymbol{K}$ is an upper triangular matrix and its entry $(3,3)$ is equal to 1, some properties can be used in order to simplify this equation.

$$\begin{gathered}Z^* = T_z \;\Rightarrow\; Z^* = Z - \boldsymbol{K}^{-1}_{(3,3)}\cdot\boldsymbol{R}^{-1}_{(3,3)}\cdot\lambda\cdot 1 \Leftrightarrow \\ \Leftrightarrow Z^* = Z - 1\cdot 1\cdot 1\cdot\lambda \Leftrightarrow \lambda = Z - Z^*\end{gathered} \tag{3.4}$$

By gathering both Equations (3.3) and (3.4), we get Equation (3.5).

$$\boldsymbol{T} = \boldsymbol{X}-\boldsymbol{R}^{-1}\boldsymbol{K}^{-1}\lambda\tilde{\boldsymbol{u}} \Leftrightarrow \begin{bmatrix}T_x\\T_y\\T_z\end{bmatrix} = \begin{bmatrix}X\\Y\\Z\end{bmatrix} - \begin{bmatrix}c_\psi & -s_\psi & 0\\ s_\psi & c_\psi & 0\\ 0 & 0 & 1\end{bmatrix}^{-1}\begin{bmatrix}f_x & \gamma & O_x\\ 0 & f_y & O_y\\ 0 & 0 & 1\end{bmatrix}^{-1}\cdot(Z-Z^*)\cdot\begin{bmatrix}u\\v\\1\end{bmatrix} \tag{3.5}$$

This equation doesn't take into account the distortion parameters or the small angles of roll and pitch; nevertheless, it serves the purpose of giving a good approximation about the (X, Y) displacement of the quadcopter and therefore allowing to control the robot on hovering. This algorithm is referred as Vision-Based Horizontal Estimation Algorithm (VBHEA).

3.4 Control Algorithms

3.4.1 Vision-Based Controller

This controller is based on both VBPEA and VBHEA. The VBPEA is responsible for detecting the markers on the image and compute the Z coordinate of the camera. Within the process of identifying the marker, the algorithm computes also the yaw angle of the camera, as referred in Section 3.3.1. These two variables (Z, Yaw) will be inputs for the VBHEA, as well as the image.

The VBHEA takes the processed image and translates the correspondences between 2D image points to 3D points, thus providing the (X, Y) coordinates of the camera (using the methods described in Section 3.3.2). Because the VBPEA provides good estimates for (Z, Yaw) and the VBHEA provides better estimates for (X, Y), the group of these two algorithms behaves as one block, providing the full pose of the quadcopter $(X, Y, Z, Yaw, Roll, Pitch)$, where $Roll$ and $Pitch$ are equal to zero, as it was defined earlier in Section 3.3.2.

As the autopilot present in the quadcopter is responsible for controlling each motor from the desired Thrust, Roll, Pitch, Yaw (trpy), we can consider a perfect autopilot controller and ignore the rotors model, for controlling effects. Then, the variables to control the quadcopter will be trpy and so it will be the full controller's output. The simulated quadcopter's autopilot was tested, in Section 5.6.

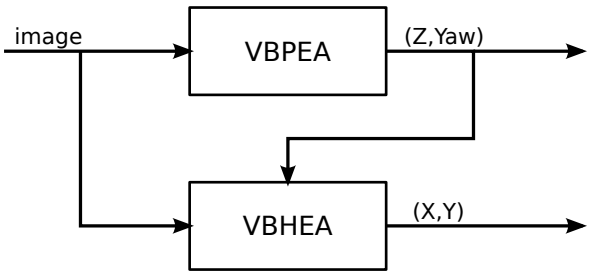


Figure 3.8: Integration of VBPEA with VBHEA in order to obtain the (X, Y, Z, Yaw) on hovering

As stated before, the VBPEA needs to be complemented with the VBHEA. To do this, they are gathered as suggested in Figure 3.8. This gathered block is providing a full pose $(X, Y, Z, \phi = 0, \theta = 0, \psi)$ with the locked pitch and roll, ie: assuming the quadcopter is perfectly horizontal. This allows better stability and therefore allows to a better and smoother control.

3.4.2 Hovering Algorithm

This algorithm uses the VBHEA and applies a PID controller to the (X, Y) position of the quadcopter and stabilize the robot above the target, by keeping a constant height from it. Due to the fact that the error may be not differentiable, the derivative gain is applied directly to the feedback value, instead of the error.

In order to apply a PID controller, a destination point $\boldsymbol{X}_d = (X, Y, Z)_d$ is defined, the one above the target on which the quadcopter will be hovering. With the purpose of aligning the quadcopter reference frame with the world frame, a yaw rotation is applied to the the destination point (originally in the G-Frame), thus considering the destination point WRT the B-Frame.

$$\boldsymbol{X}_d^B = \boldsymbol{R}(\boldsymbol{\psi})\boldsymbol{X}_d^G \tag{3.6}$$

Considering the current position of the quadcopter $\boldsymbol{X} = (X, Y, Z)$, a PID controller is applied on the position error $\boldsymbol{E} = \boldsymbol{X}_d - \boldsymbol{X}$. As the control variables are trpy, the controller defined in Equations (3.7) provides the control for the commanded roll ($\hat{\phi}$) and commanded pitch ($\hat{\theta}$) movements.

$$\hat{\theta} = K_{p_x} E_x - K_{d_x} \frac{dX}{dt} + K_{i_x} \int_0^t E_x \, dt \tag{3.7a}$$

$$\hat{\phi} = K_{p_y} E_x - K_{d_y} \frac{dY}{dt} + K_{i_y} \int_0^t E_y \, dt \tag{3.7b}$$

A PID controller is also applied on yaw error $E_{yaw} = \psi_d - \psi$, in order to maintaining the quadcopter aligned with the marker. This is useful for implementing on a real quadcopter to land on a real docking station, as it probably matters the quadcopter orientation when landed. The commanded yaw ($\hat{\psi}$) is calculated in Equation (3.8).

$$\hat{\psi} = K_{p_{yaw}} E_{yaw} - K_{d_{yaw}} \frac{d\psi}{dt} + K_{i_{yaw}} \int_0^t E_{yaw} \, dt \tag{3.8}$$

For maintaining the quadcopter's height Z, a PID controller is also applied to the height error E_z from the previously defined error vector $\boldsymbol{E}$, thus giving a command thrust $\hat{Z}$.

$$\hat{Z} = K_{p_z} E_z - K_{d_z} \frac{dZ}{dt} + K_{i_z} \int_0^t E_z \, dt \tag{3.9}$$

The Figure 3.9 shows the block diagram of the implementation of this algorithm.

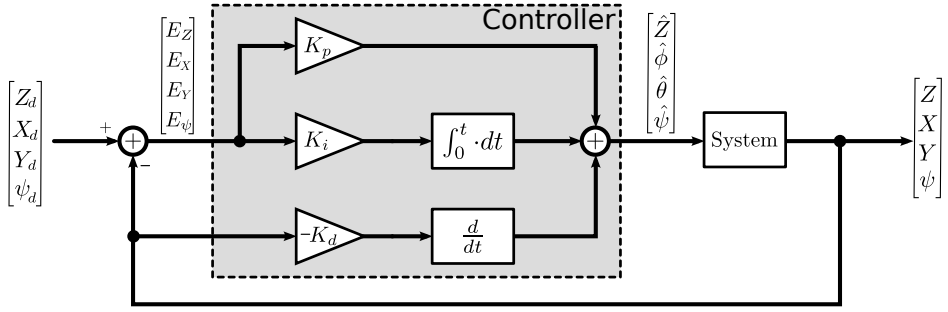


Figure 3.9: PID controller applied to the VBHEA

3.4.3 Landing Algorithm

This algorithm consists on using the PID controller on the Z component of the position, by incrementally changing the destination point for hovering, by decreasing its Z coordinate, after some conditions are verified. This incremental change on the desired Z_d helps maintaining stability, by continuously checking those conditions.

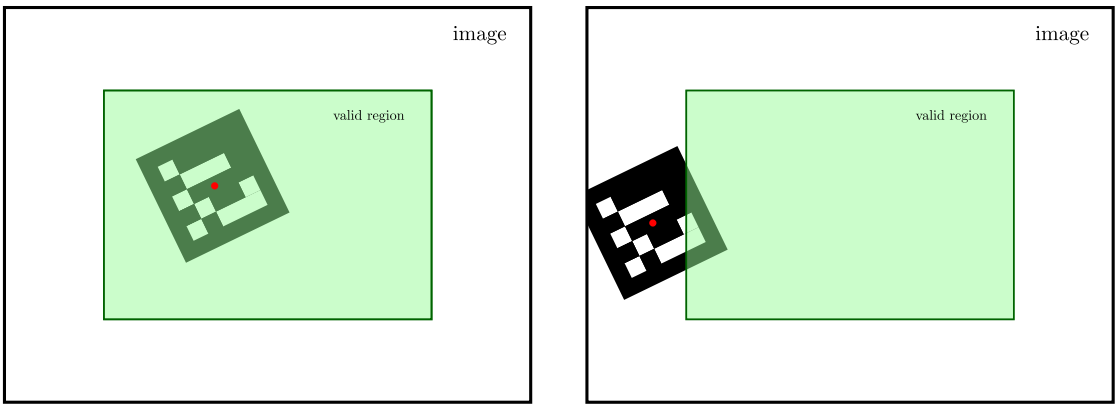
In order for the algorithm starts to decrease the Z_d, the following conditions must be checked:

- variability of the $roll$ and $pitch$ command for the quadcopter;
- position of the reference point within the center of the image;
- image change velocity;
- height of the quadcopter.

To evaluate the commands variability, a low pass filter is applied to the $roll$ and $pitch$, in such a way that when the quadcopter is relatively stable, the variability values are low. The discrete low filter is applied as in Equation (3.10). If the variability is below a threshold, the quadcopter is considered to be stable.

$$y_i = \alpha x_i + (1 - \alpha) y_{i-1} \tag{3.10}$$

To evaluate if the reference point is centered within the image, we project the 3D reference point to the image, using Equation (2.31), and check if the projection is inside a rectangular region in the center of the image. Figure 3.10 represents this validation. The only validated point is the reference point, to avoid further markers getting an invalid tag when they are out (or in the limit) of the image.



(a) Reference point inside the valid region, thus valid

(b) Reference point outside the valid region, thus invalid

Figure 3.10: Validating if reference point is in the image center region
The red dot represents the reference point, in this example coinciding with the marker's center

The evaluation of the image change velocity consists on calculating the velocity of the reference point, in $[pixels.s^{-1}]$, which will provide information about how much the reference point is moving, within the image. This information will help on preventing the image to get too much blur effect. The velocity is calculated as the norm of the difference between the reference point coordinates in consecutive frames, divided by the frames temporal space Δt, as stated in Equation (3.11). When the quadcopter is higher, the velocities within the image will be slower and it will be tagged as valid for landing. This effect is desired because if the quadcopter is high enough, even if it is not completely stable from the image point of view, the blur effect is smaller than when the quadcopter is lower. When the velocity is below a certain threshold, the image is considered to be stable.

$$v_i = \frac{\|\boldsymbol{u_i} - \boldsymbol{u_{i-1}}\|}{\Delta t} = \frac{\left\| \begin{bmatrix} u \\ v \end{bmatrix}_i - \begin{bmatrix} u \\ v \end{bmatrix}_{i-1} \right\|}{\Delta t} \tag{3.11}$$

The Z_d will be decreased only if its height is close from the Z_d, by less than a threshold Z_{thresh}. E.g. if $Z_d = 0.80$ and the threshold $Z_{thresh} = 0.10$, the Z_d will not get lower while the quadcopter is above $Z = 0.90$.

It was experimentally observed that as soon as any part of the robot touches the ground, this will affect the IMU which will make the Paparazzi to respond. This results in unexpected behaviour and instability. Therefore, it is needed to find some solution.

The solution for safely landing is consists on turning off the rotors when the quadcopter is close enough of the ground. On experimental trials, it was tested that the quadcopter can be turned off safely even if it is 15cm from the ground. Having this measure as the maximum, the landing algorithm is ready to turn off the rotors when it is 5cm from the ground. This height guarantees that it will not have that unexpected behaviour and still lands safely.

If all the previous conditions are verified, the quadcopter is considered able to go down; then the Z_d is decreased by a step Z_{step}. This process will continue until the quadcopter is lower than a limit distance from the ground. When it reaches the limit, the quadcopter's rotors are turned off.

The landing task also implies the hovering task. By gathering both algorithms it is possible to achieve the goal of landing the quadcopter. In Figure 3.11 it can be observed the flowchart when landing the robot.

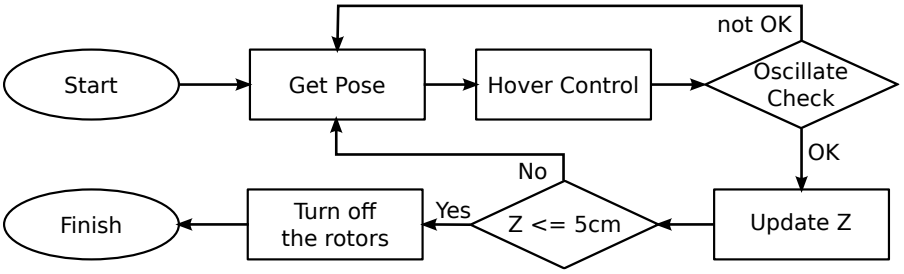


Figure 3.11: Landing algorithms flowchart

3.5 Integration and Implementation

3.5.1 Assembling

As this quadcopter itself does not have an inbuilt camera, it was attached to it a regular webcam. It is fixed on the bottom of the robot's body, at approximately 8cm from the very bottom of it (what touches the ground). The configuration is considered to be perfectly downwards directed, with the upper part of the image coincident with the X of the quadcopter, ie: the front direction of the quadcopter corresponds to the top of the image. Figure 3.12 shows how the image frame is related with the quadcopter frame.

All the control and processing is integrated under ROS and they are distributed as nodes and each node performs one task. The several nodes developed and working in this thesis are listed below.

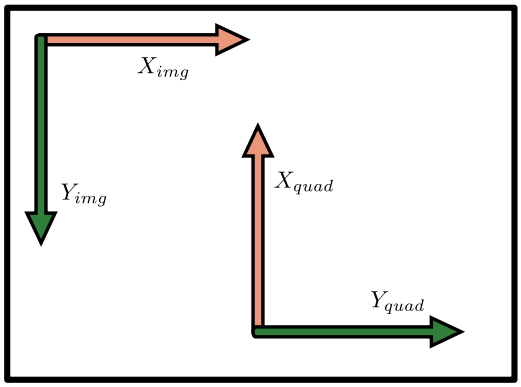


Figure 3.12: Relation between camera image and quadcopter frame

- **usb_cam**

  Node responsible for reading raw data from the USB camera and converting it to a ROS image data, which can be read by any ROS node and by *cv_bridge*, a library used to convert ROS images to OpenCV images.

- **image_view**

  Node used for visualization only. It reads a ROS image and creates a window which shows the image.

- **Gazebo**

  Physics simulator with a graphical interface, allowing a friendly-user understanding about the interactions of the objects simulated.

- **quad_sim**

  Node responsible to spawn a virtual quadcopter on Gazebo and model its dynamics. It also has a simulated camera which is broadcasting an image from the bottom of the quadcopter.

- **Vision-Based Pose Estimation Algorithm (VBPEA)**

  This is one of the most important nodes which reads an image from the quadcopter's onboard camera and locates a special marker, processing it and providing a 3D full pose of the quadcopter.

- **Vision-Based Horizontal Estimation Algorithm (VBHEA)**

  This node, in cooperation with the VBPEA, provides a more accurate location of the quadcopter when it is horizontally hovering a marker. Unlike the VBPEA which provides a full pose, this node only provides (X, Y) location of the quadcopter.

- **pose_fusion**

  This node is responsible for fusing the information about the estimated pose. It reads the pose estimated from odometry, VBPEA and VBHEA and outputs the considered pose of the quadcopter.

- **quad_hover_control**

This controller node is a PID controller on the position of the quadcopter. Receives a destination point and tries to minimize the distance from that point, by the means of a PID controller. This node outputs the control variables for the quadcopter: $(\phi, \theta, \dot{\psi}, thrust)$.

- **quad_landing_control**

  This is the node that makes the decisions and defines what should be the destination point for the quad_hover_control. It will decrease the Z component of the destination point in order to make the quadcopter to land. It is also responsible to shut down the controllers and turn off the rotors when the quadcopter is very close to the ground, making it stop on the target.

The full architecture using all nodes is represented in Figure 3.13. Conceptually, the only difference between the simulation and the real operation is only on the camera image (the simulated camera image does not need any conversion) and the interface for the real quadcopter communications, of course, which is hided for simplicity.

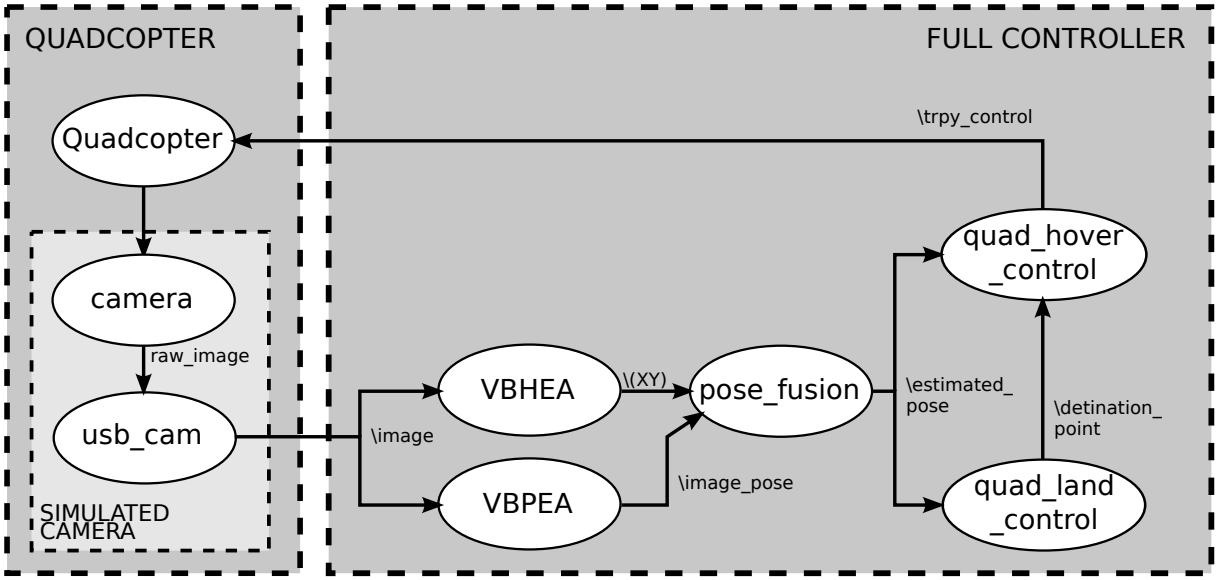


Figure 3.13: Full architecture gathering all algorithms

The architecture sketch may be redrawn as simple block diagram, then simplifying the structure for an easier analysis, in Figure 3.14.

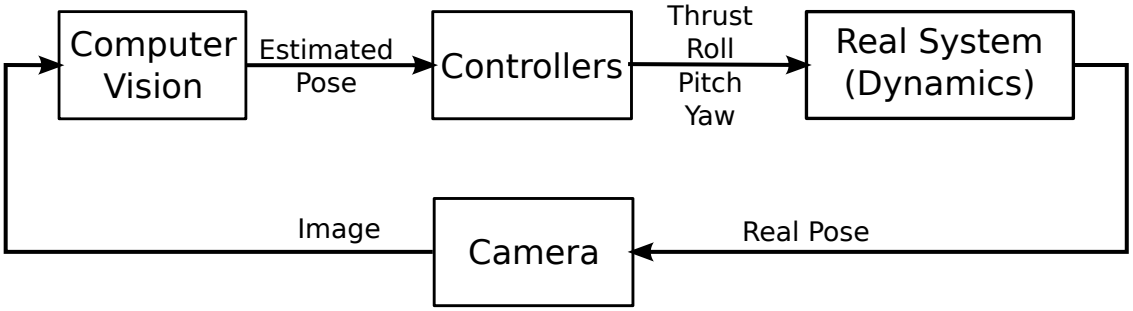


Figure 3.14: Full architecture block diagram

As this architecture is very similar in simulations and in real world, it is simple to port the developed algorithms, tested in the simulator, into the real world.

3.5.2 Simulation

The simulation is performed based on a physics simulator called Gazebo[33]. Gazebo is a multi-robot simulator for indoor or outdoor environments and it is capable of simulating a population of robots, sensors and objects, in a three-dimensional world. It generates both realistic sensor feedback (with noise included) and physically plausible interactions between objects (it includes an accurate simulation of rigid-body physics). Since this simulator operates on ROS it makes it easier to implement all the rest.

The use of this simulator in the thesis implies the modelling of a quadcopter robot, an IMU sensor on the robot, a camera assembled on the bottom of the quadcopter, as well as the interacting world which must have some markers included. The robots and sensors descriptions are made through some Unified Robot Description Format (URDF) files which are gathered within ROS. The world is designed in a 3D software and included in a .world file that the Gazebo will read. The graphical view of this simulation can be observed in Figure 3.15.

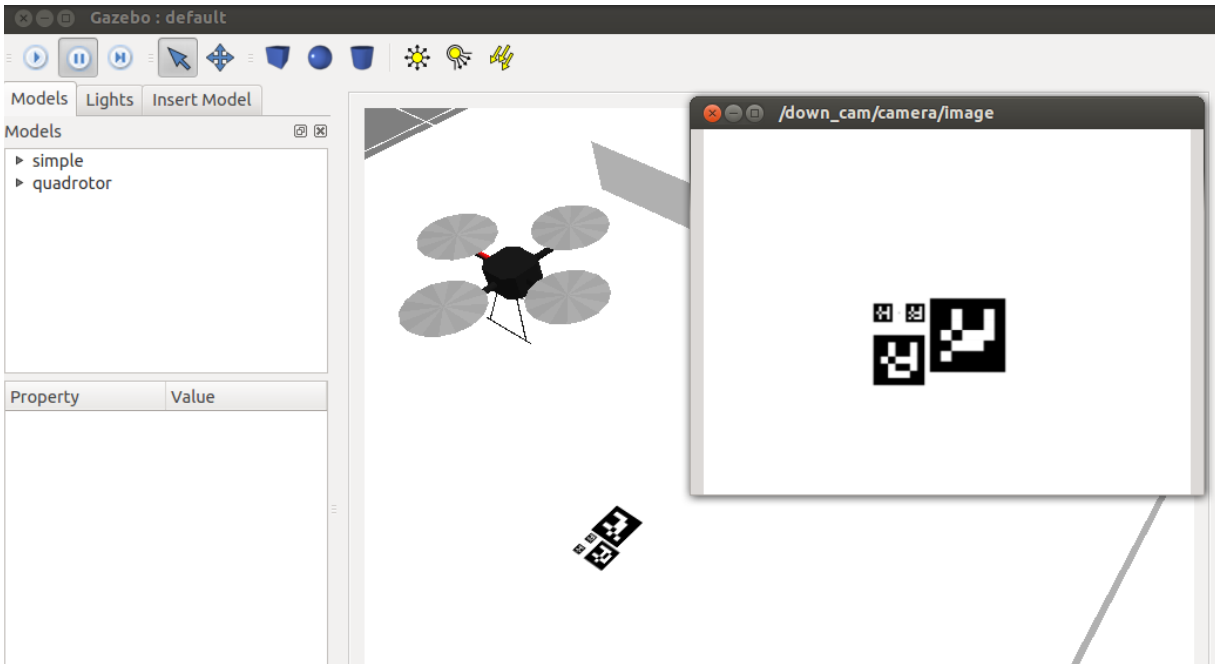


Figure 3.15: Gazebo simulating a markers board and a quadcopter with an onboard camera

The process of simulating assumes a computer with ROS running the Gazebo. It must be loaded the world into Gazebo and configured some parameters like gravity and light conditions. After the simulator is ready, the robot model is loaded into the world. By doing this, the robot starts to behave like in real world, i.e. it has the noisy behaviour (drift) and can be controlled via trpy variables. The quadcopter node includes the simulated camera, which starts immediately to broadcast the virtual simulated image.

After having a world simulation with a robot model in it, the processing units are initialized: the VBPEA, the VBHEA and the Controller. The VBPEA an VBHEA read the topic published by the quadcopter's camera and process it, outputting the estimated pose of the robot. The Controller reads information about the pose given by the VBPEA and VBHEA, and outputs the trpy variables to control the quadcopter. For applying the PID controller, this node also reads the *destination_point* topic.

With all this nodes working together, the simulating system architecture is the same as in Figure 3.13, where the QUADCOPTER block is the Gazebo, with the quadcopter and the simulated camera running within it.

In order to check the behaviour of the controller and the evaluate the performance of the system, the Gazebo has also an extra topic which is publishing the ground truth pose of the simulated quadcopter model, so the user is able to always check the accuracy of the estimators.

4

Experimental Setup

Contents

4.1 Introduction

The experimental setup was developed in two separate frameworks. The first one was a development exclusively on simulation. The hardware in Section 4.2 was modeled in software and then in a simulator, as it is explained in Section 3.5.2. The second framework consists on the real quadcopter, equipped with a Pandaboard and a camera, and a set of markers. Both frameworks were developed as similar as possible, to maximize the compatibility of the developed methods between both frameworks.

4.2 Hardware

- UAVision Quadcopter UX-4001 mini

  This quadcopter (in Figure 4.1) was developed by UAVision[4] and it is equipped with the Paparazzi autopilot ([34]). It is also equipped with some essential sensors (gyros, accelerometers, magnetometer) and some other sensors not used in this thesis (GPS, Zigbee, WiFi). The Paparazzi autopilot is responsible for stabilizing the quadcopter and performs a low level control of all the four rotors, so it is possible to control the desired thrust, yaw rotation, roll and pitch. This allows a better control interface for the end user.

Figure 4.1: UAVision Quadcopter UX-4001 mini

- Camera

  The camera used was a Logitech QuickCam Messenger (in Figure 4.2), but the principal idea is that any calibrated camera may be used. This camera is just an ordinary webcam, with 640×480 resolution, ability to film at 15fps and it has a low angle of view ($\sim 45^o$). The connection is done via USB and can be connected to any computer, including the Pandaboard.

Figure 4.2: Logitech QuickCam Messenger, http://logitech.com

- Markers board

  For the image detection block, a markers board was made, designed specifically for this purpose. The board has four markers, a big, a middle size and two small markers, displayed on a A4 sheet paper. The main objective of having multiple sized markers is that the camera is able to see at least one marker from a big variety of distances, from 5cm (landed quadcopter) until 3m. The board has two small markers (close to the reference point) because it helps the camera to see more than one marker, even if the quadcopter is very low. The scheme of the sheet can be seen in Figure 4.3 and the full A4 markers board is in Appendix A.2.

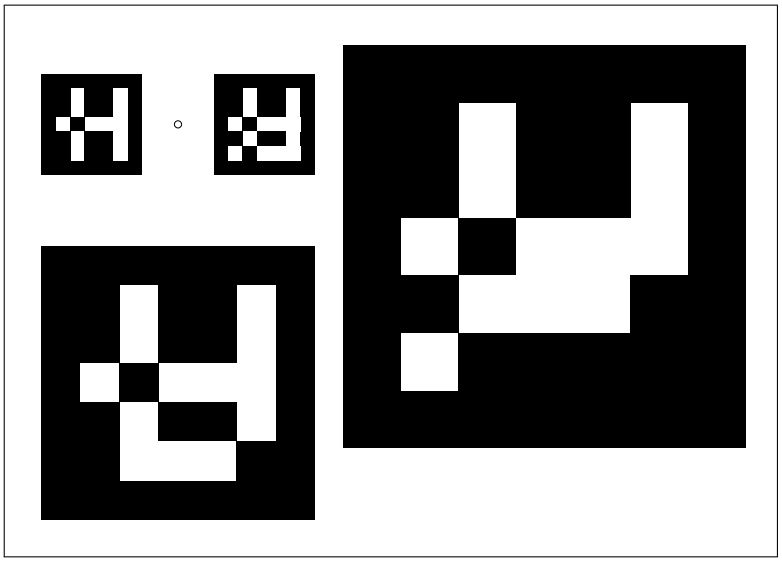


Figure 4.3: A4 markers sheet
small_size=39mm, middle_size=107mm, big_size=158mm
The little dot between the small markers represents the chosen reference center

- Pandaboard

  The Pandaboard (in Figure 4.4) is a small computer weighting only 81.5 grams, equipped with a Dual-core ARM Cortex-A9 at 1.2 GHz, 1 GB low power DDR2 RAM, and ready for Full HD (1080p) multi-standard video encode/decode. The board reads an SD Card (which is the hard drive), has 2 USB, 1 HDMI, 1 Ethernet, wireless connection, etc... It is also able to run an Ubuntu OS and the ROS too. The full list of software dependencies installed on the Pandaboard is described in Appendix A.1.

  All this properties make the Pandaboard a good choice to perform the onboard control of the quadcopter.

4.3 Software

The development of this thesis is done within the Operative System Ubuntu 12.04. It is a stable and Long Term Support of the Linux OS's. Thereby, this Ubuntu distribution is also compatible with the Pandaboard, with the ARM distribution OMAP4.

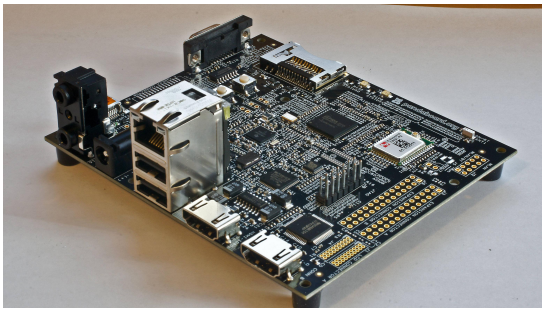


Figure 4.4: Pandaboard, [35]

On top of the Operative System is running the Robot Operating System (ROS), which can be considered as a managing software, which manages all the processes (nodes), threads and communication between nodes. Most of the software development is written in C++ code, compiled for using inside ROS.

This thesis takes advantage of multiple libraries and programs publicly available. There were used some essential software like Gazebo, usb_cam package, Hector packages (part of quadcopter model), ArUco library (part of VBPEA and VBHEA), Paparazzi libraries (communications for the Paparazzi board), OpenCV and some other libraries on C++.

As stated before, a list of installed software is available in Appendix A.1.

4.4 Experiments

4.4.1 Vision-Based Pose Estimation Algorithm

In order to determine how the number of markers detected affects the output, a test bench was assembled. It consists on a real camera aligned with the marker(s), by making the camera XY-frame parallel to the markers XY-frame, and by keeping a steady (X, Y) coordinates on the camera, while varying the distance from the marker(s), i.e., measuring the Z coordinate while keeping $X = Y = 0$.

In this experiment, the camera was fixed on a moving platform, with a mark aligned with the camera image sensor. The marker(s) board was fixed and a ruler was also fixed and its beginning edge attached to the marker(s) board. By using the platform's mark (camera image sensor) to fix the position on the ruler, it was possible to perform the test with the required accuracy. The draw of the test bench in represented in Figure 4.5.

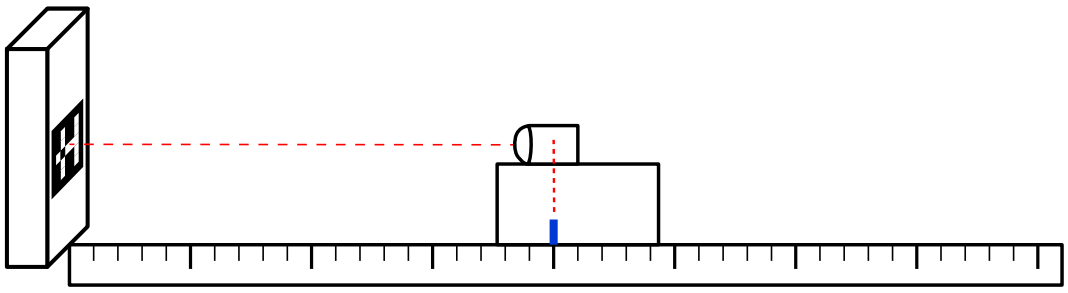
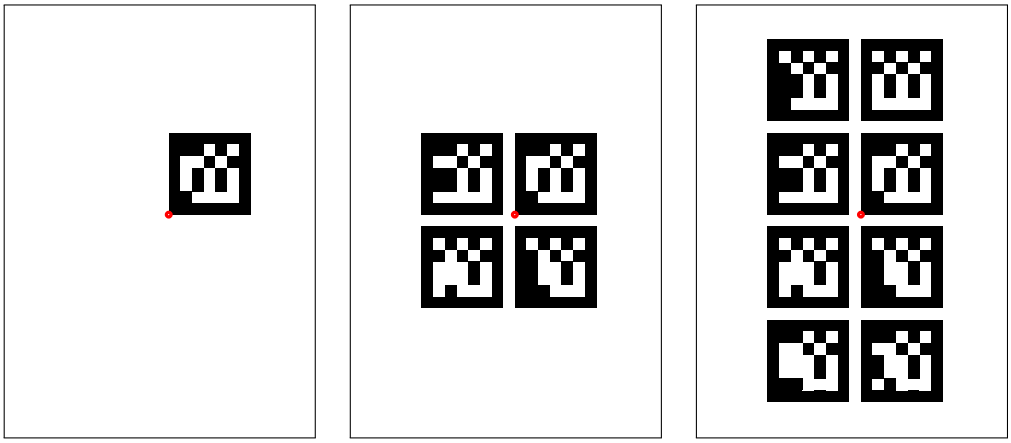


Figure 4.5: Test bench for the VBPEA experiment

This set of tests was performed using real data from a real camera. The distance of the camera to the marker(s) is measured with a ruler and the output of the VBPEA for each distance (from 0.40 to 1.45 m, spaced by 0.10 m) is analysed. This procedure is repeated with one, four and eight markers, all having the same fixed side size (6 cm), as suggested in Figure 4.6. Because of the ruler smaller scale, there is an implicit error related to the readings, which is equal to 1 mm.



(a) 1 marker board **(b)** 4 markers board **(c)** 8 markers board

Figure 4.6: Different boards used for testing vision-based markers detection
The red dots represent the chosen reference point

The results of this experiment are analysed in Section 5.2.

4.4.2 Comparing VBPEA with a Laser Range Finder

In this experiment, a real quadcopter with an attached Laser Range Finder (LRF) was used. In order to compare the VBPEA estimations with the data provided by the LRF, a downwards camera was also on board of the quadcopter. The LRF was measuring the height of the quadcopter, the same variable provided by the VBPEA we want to test.

This trial was run using a real LRF [6], on the real quadcopter with a downwards camera, operated manually with a remote controller. The quadcopter was driven over the one marker on the ground, in order to detect it by the camera.

The results of this experiment are analysed in Section 5.3.

4.4.3 Vision-Based Horizontal Estimation Algorithm

As this is one of the crucial algorithms, its quality must be verified. This set of tests consist on fixing the real camera at a fixed Z coordinate and varying the X or Y coordinate (one of the horizontal coordinates), while maintaining the other coordinate equal to zero.

As suggested in Figure 4.7, the camera was fixed on a moving platform, with a mark aligned with the camera image center. The markers board was fixed on the ground and a ruler was fixed on a horizontal structure, which was above the markers board. By using the platform's mark (camera image center) to

[6] Hokuyu's URG-04LX-UG01, `http://www.hokuyo-aut.jp/02sensor/07scanner/urg_04lx_ug01.html`, consulted on October 2013

fix the position on the ruler, it was possible to perform the test with the required accuracy. Because of the ruler smaller scale, there is an implicit error related to the readings, which is equal to 1 mm.

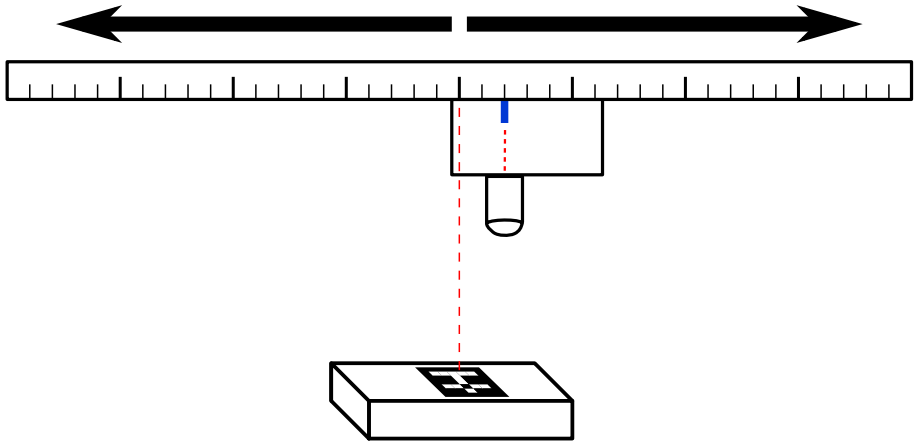
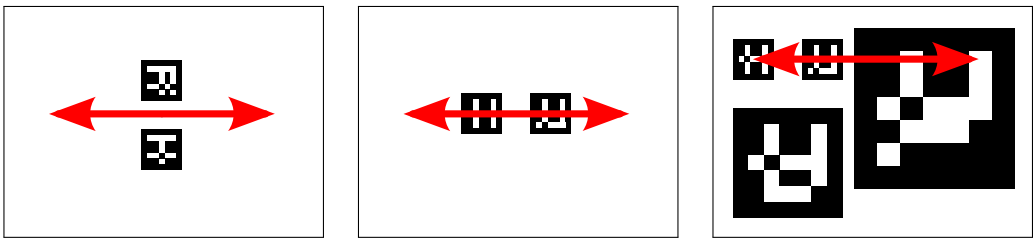


Figure 4.7: Test bench for the VBPEA experiment

Three different trials were conducted, with different markers boards.

- Trial 1
  Using two fixed, 4 cm sided markers, aligned vertically (in 2D), as in Figure 4.8(a). The camera was displaced horizontally (in 2D), from -0.25 to +0.25 m, spaced by 0.05 m.
- Trial 2
  Using two fixed, 4 cm sided markers, aligned horizontally (in 2D), as in Figure 4.8(b). The camera was displaced horizontally (in 2D), from -0.25 to +0.25 m, spaced by 0.05 m.
- Trial 3
  Using the full markers board in Appendix A.2, aligned horizontally (in 2D), as in Figure 4.8(c). The camera was displaced horizontally (in 2D), from -0.15 to +0.25 m, assuring that all markers were visible during the whole trial, spaced by 0.05 m.



(a) Trial 1 **(b)** Trial 2 **(c)** Trial 3

Figure 4.8: Different boards used for testing VBHEA detection
The red arrows represent the camera movement, hovering the markers boards

The results of this experiment are analysed in Section 5.4.

4.4.4 Visual-Based Horizontal Position Estimation

To demonstrate the problem of finding the (X, Y) coordinates, stated in Section 2.3, a test was made in the simulator. This test consists on simulating the quadcopter with a downwards camera and a

markers board (as in Section 3.5.2), running both VBPEA and VBHEA, and comparing both outputs, as well as the ground truth information, possible in the simulation environment.

This simulation data was collected during a hovering test. The quadcopter's (X, Y) trajectories over time were analysed through three different methods: the simulator ground truth, the VBPEA and the VBHEA. From what was previously stated, we are expecting bad estimations provided by the VBPEA and better estimations from the VBHEA.

The results of this experiment are analysed in Section 5.5.

4.4.5 Inbuilt quadcopter controllers

The controller developed in Section 3.4.2 uses the approximation that the autopilot controller is perfect. In order to analyse the autopilot controller present within the simulator, this tests were conducted.

This test consists on introducing a step input and observing the controller's output. For classifying roll and pitch, only the roll controller was observed and similarity for pitch is assumed.

Three step values were tested: 0.5 and 0.1 rad for testing the response over time; and a 0.01 rad step, used to test for the small variations (noise) inherent to this particular quadcopter model.

The results of this experiment are analysed in Section 5.6.

4.4.6 Case 1 - Steady Hovering

In order to test how the quadcopter behaves when hovering is intended, the Gazebo was used. For the simulation, the following steps were followed by the presented order:

- Start Gazebo;
- include world model in the Gazebo;
- spawn the quadcopter, with a downwards camera included;
- start VBPEA;
- start VBHEA;
- start hovering control.

This first experiment was intended to study how the quadcopter manages to hover a steady markers board. Because the simulated quadcopter has a noisy behaviour included, i.e. it starts to move if no order is given, a delay was introduced before launching the hovering controller, in order to check if the quadcopter is able to stabilize if it does not start in the destination point. In terms of the image captured, the hovering controller was started when the markers board was near the image limits.

With this test, it was possible to analyse that when the quadcopter is able to see the markers within the image, it is also possible to control it, even when it does not start in the equilibrium point.

Although the quadcopter is hovering the target, its positions changes along time. This behaviour was expected and it is dued to the added noise present in the quadcopter's model of the rotors. This is useful

for understanding the real world behaviour, as it is expected that the real quadcopter is not perfectly steady on hovering.

4.4.7 Case 2 - Landing on a Steady Target

This experiment is the final experimentation, with all developed algorithms running. This one is similar to the previous experiment from Section 4.4.6, with one more process running. For the simulation, the following steps were followed by the presented order:

- Start Gazebo;
- include world model in the Gazebo;
- spawn the quadcopter, with a downwards camera included;
- start VBPEA;
- start VBHEA;
- start hovering control;
- start landing control

As it was developed, this experiment is based on sequential hovering processes, as the objective is to hover the target, but with different altitudes along time. After stabilizing the first time, the quadcopter starts to smoothly descend its altitude.

5

Experimental Results

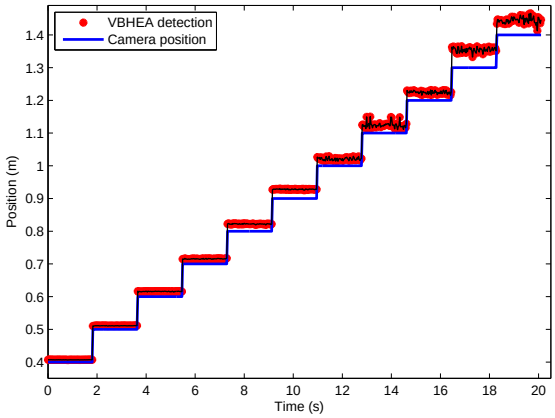
Contents

5.1 Introduction

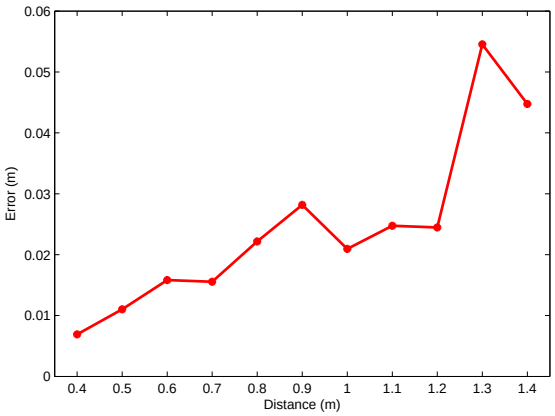
In this chapter the results of some tests are presented. The tests are intended to confirm the theoretical results obtained previously, thus proving if the methods described in this thesis are valid for implementation.

5.2 Vision-Based Pose Estimation Algorithm

This set of tests was performed using real data from a real camera. The distance of the camera to the marker(s) is measured from 0.40 to 1.45 m, spaced by 0.10 m. This procedure is repeated with one, four and eight markers, as described previously in Section 4.4.1.

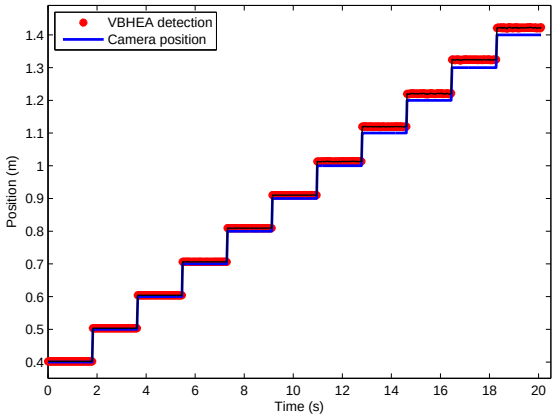


(a) Comparison between the distance of the camera and the distance perceived by the VBPEA

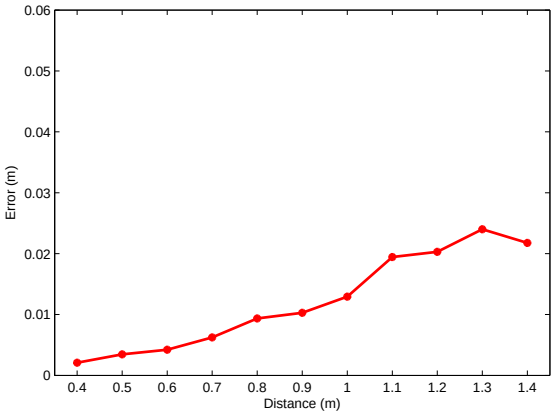


(b) Absolute value of the error for the VBPEA detection

Figure 5.1: Trial for testing the VBPEA detection, using one marker



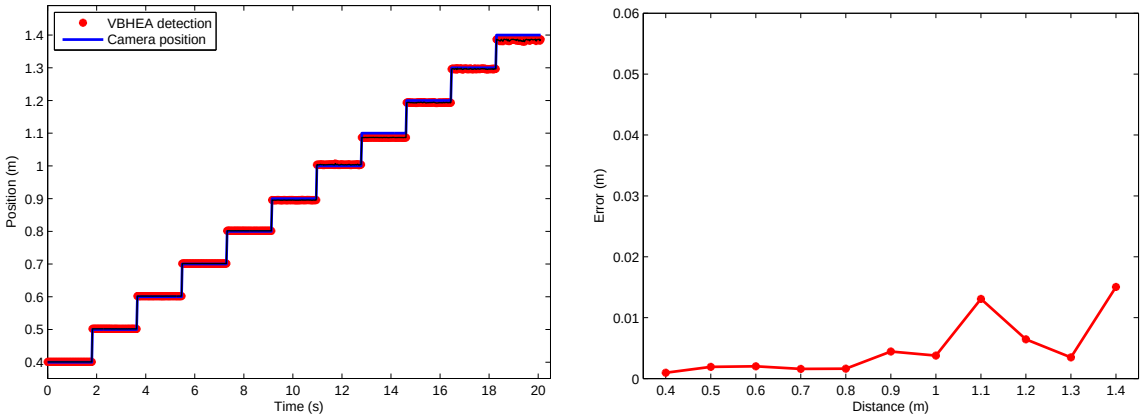
(a) Comparison between the distance of the camera and the distance perceived by the VBPEA



(b) Absolute value of the error for the VBPEA detection

Figure 5.2: Trial for testing the VBPEA detection, using four markers

The analysis of the results in Figure 5.1 to 5.3 demonstrate that the number of detected markers relates with the error on the VBPEA output; the more detected markers, the less is the error. The Figure 5.4 shows the comparison between the detected error in each trial, where it is clear that the estimation is better when more markers are used.

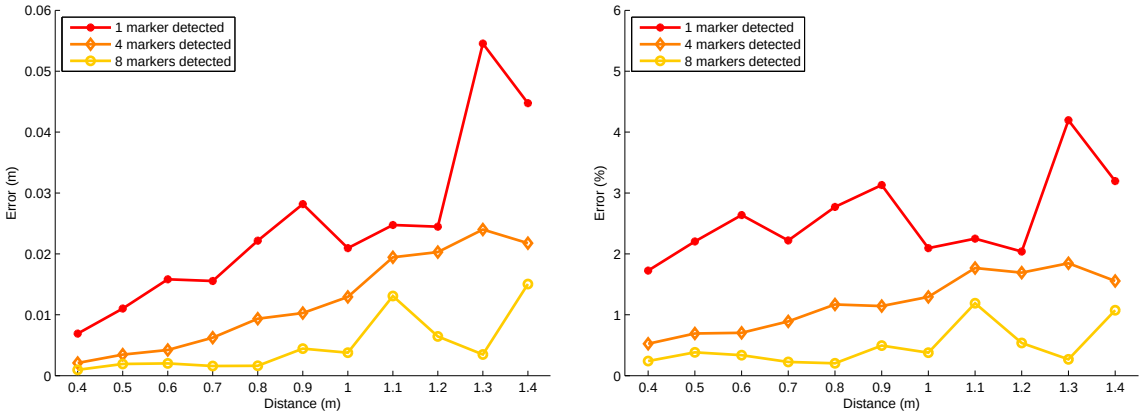


(a) Comparison between the distance of the camera and the distance perceived by the VBPEA

(b) Absolute value of the error for the VBPEA detection

Figure 5.3: Trial for testing the VBPEA detection, using eight markers

This results confirm that the VBPEA performs very well. Even with just one marker, the Z position error is less than 5% of the the camera Z coordinate (in Figure 5.4), which is enough for the intended application.



(a) Comparison between absolute error detection

(b) Comparison between error detection relative to the distance of the camera

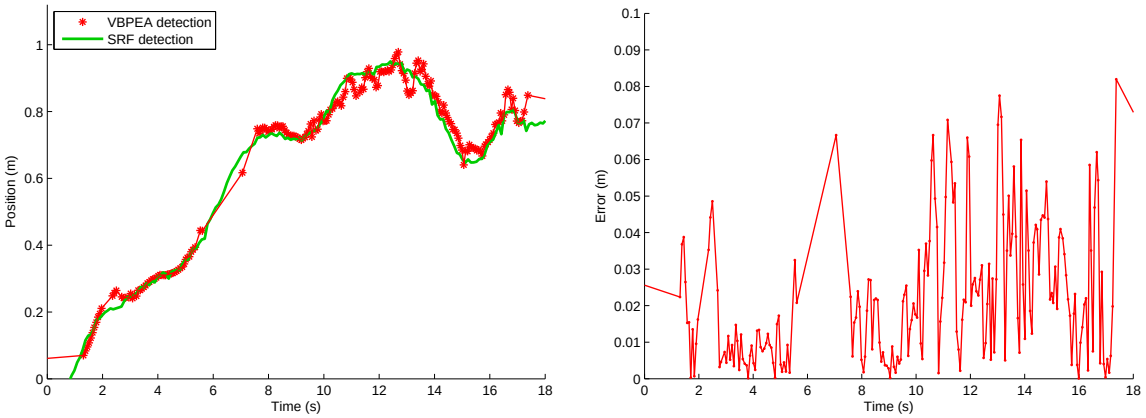
Figure 5.4: Comparison between error detection, using one, four or eight markers

5.3 Comparing VBPEA with a Laser Range Finder

In order to compare the VBPEA with another sensor performing a similar task, some tests using a LRF on board the quadcopter were conducted, and the results are expressed in Figure 5.5. Because the quadcopter was commanded manually, it was difficult to keep the marker within the image and sometimes, as it occurred at 6-7 s, the VBPEA was not able to provide any data.

Although the real data is noisier than the previous tests in Section 5.2 (as expected), the obtained VBPEA output is still acceptable for the intended purpose, with errors smaller than 10 cm. This error corresponds to less than 10% of the detected height, as in Figure 5.6.

This results allow to conclude that the VBPEA is able for using within a real application.



(a) Comparison between the distance perceived by the LRF and the VBPEA

(b) Absolute value of the difference between the LRF and the VBPEA detections

Figure 5.5: Comparison between LRF and VBPEA

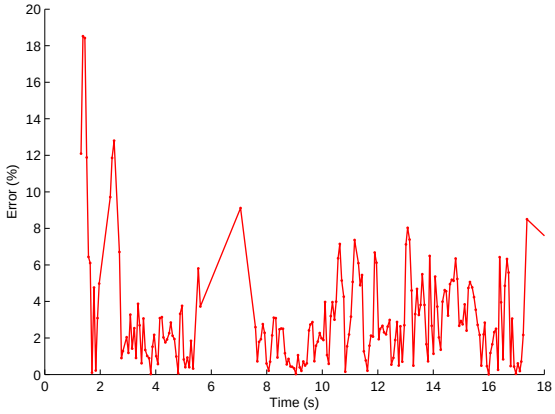
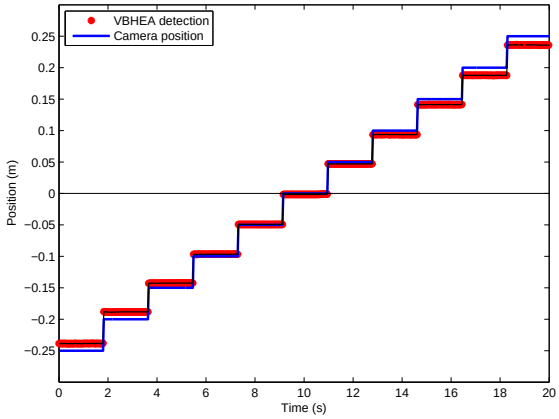


Figure 5.6: Difference in percentage of VBPEA and LRF, relative to the height

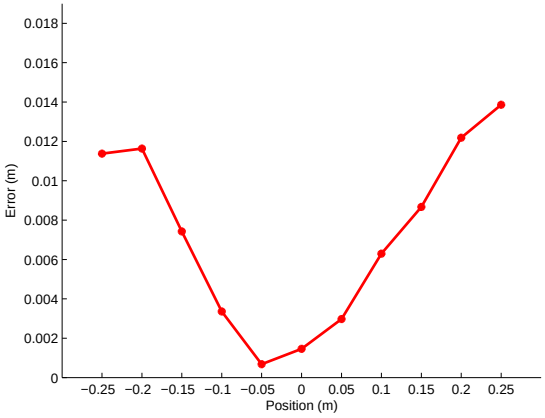
5.4 Vision-Based Horizontal Estimation Algorithm

In this experiment, real data from a real camera were used. Three different trials were conducted, with different markers boards.

- **Trial 1** – Using two fixed, 4 cm sided markers, aligned vertically (in 2D).



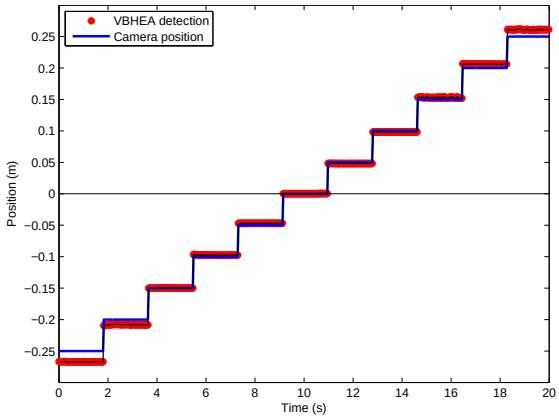
(a) Comparison between the horizontal displacement of the camera and the displacement perceived by the VBHEA



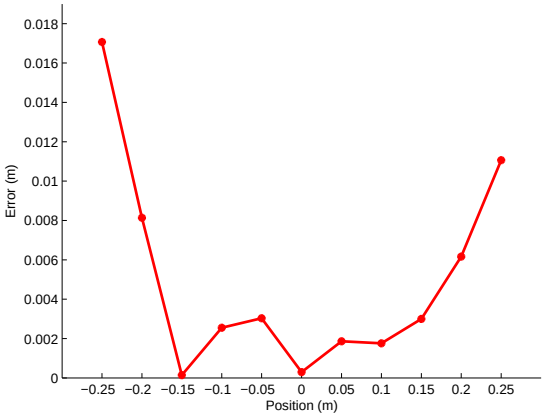
(b) Absolute value of the error for the VBHEA detection

Figure 5.7: Test for the VBPEA detection, trial 1

- **Trial 2** – Using two fixed, 4 cm sided markers, aligned horizontally (in 2D).



(a) Comparison between the horizontal displacement of the camera and the displacement perceived by the VBHEA



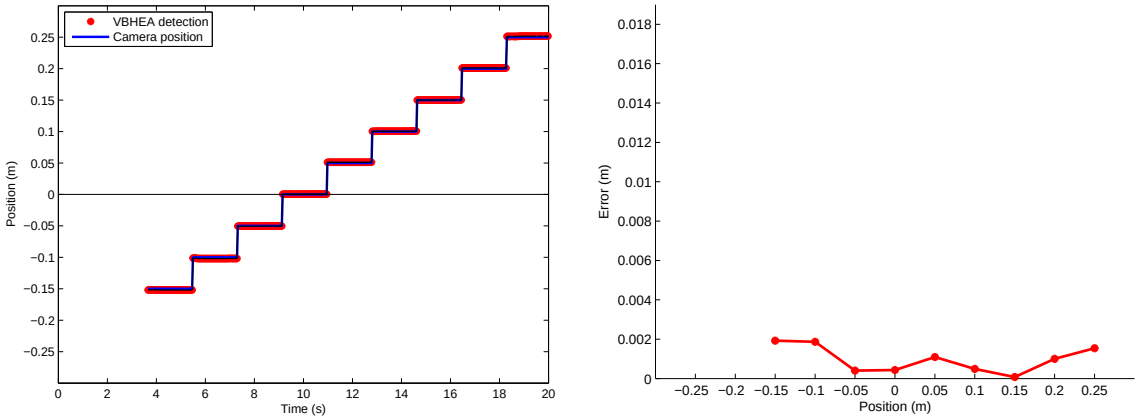
(b) Absolute value of the error for the VBHEA detection

Figure 5.8: Test for the VBPEA detection, trial 2

- **Trial 3** – Using the full markers board in Appendix A.2, aligned horizontally (in 2D).

The Figure 5.10 show some interesting results by comparing the error among the three trials. All trials show some common behaviour, that the error gets bigger as the camera gets further from the center, and when more markers are visible the error gets smaller. These behaviours were already expected, as in the experiment in Section 5.2.

By comparing trial 1 and 2, we can check that the output in trial 2 gets less noisy because the positions of the markers are more restrictive in the camera displacement direction. The results verify the expected behaviour.

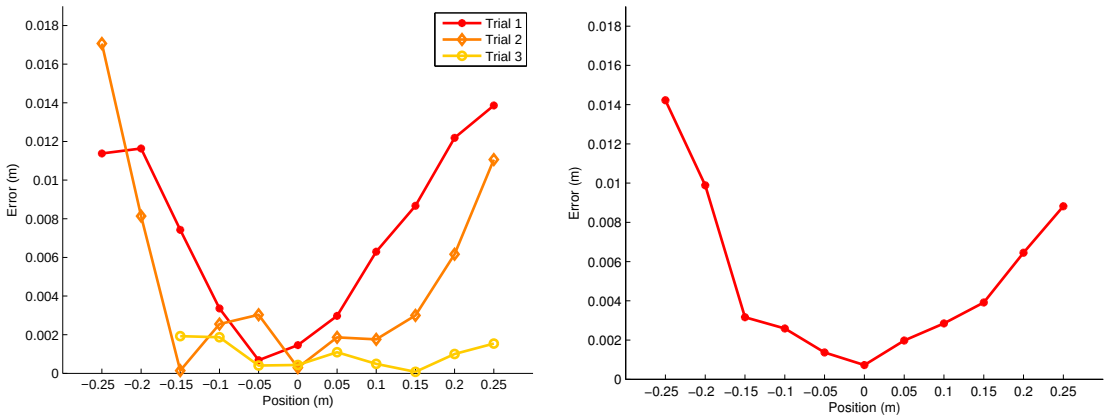


(a) Comparison between the horizontal displacement of the camera and the displacement perceived by the VBHEA

(b) Absolute value of the error for the VBHEA detection

Figure 5.9: Test for the VBPEA detection, trial 3

In conclusion, this algorithm's output is reliable and, within these trials, it has errors above 7%, as in Figure 5.11. More results about the VBHEA are addressed in the next section.



(a) Comparison between absolute errors in the three trials

(b) Average value of the errors of the three trials

Figure 5.10: Comparison between the different trials

5.5 Visual-Based Horizontal Position Estimation

To demonstrate the problem of finding the (X, Y) coordinates, stated in Section 2.3, a test was made in the simulator.

The results observed in Figure 5.12 show that the horizontal coordinates from the VBPEA are very noisy and therefore they are useless for their intended purpose. The results from this method will be ignored from now on. On the other hand, the output from the VBHEA is more accurate.

The absolute error for the VBHEA in Figure 5.13, shows that this method provides data with error lower than 5 cm. This error does not depend only on the distance of the marker but also on the angle of the quadcopter, with the latter being the major responsible for the error. Such fact may be observed during the changes of direction of the quadcopter, i.e. on the peaks of the plot, where the error is bigger.

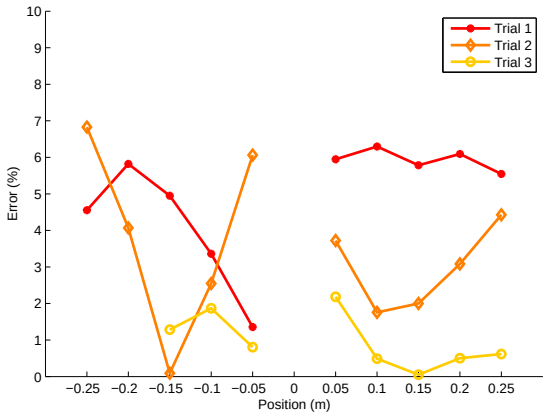
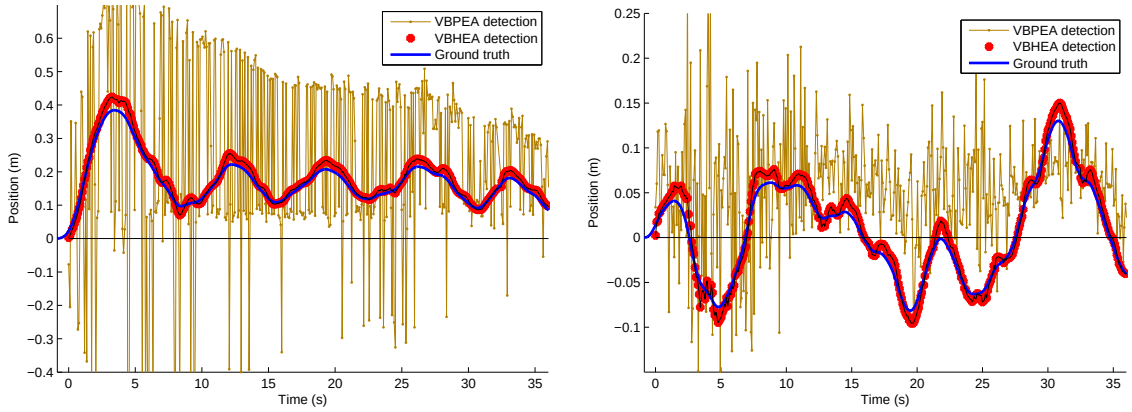


Figure 5.11: Comparison between the relative errors in the different trials

This issue can be minimized if thresholds for the roll and pitch angle are set within the controller. Therefore, even when the quadcopter is not perfectly hovering (horizontal orientation), the information obtained from the VBHEA has a small enough error, thus allowing to use it to control the quadcopter.



(a) Comparing X coordinate

(b) Comparing Y coordinate

Figure 5.12: Comparison between the outputs from the VBPEA, VBHEA and ground truth from the simulator

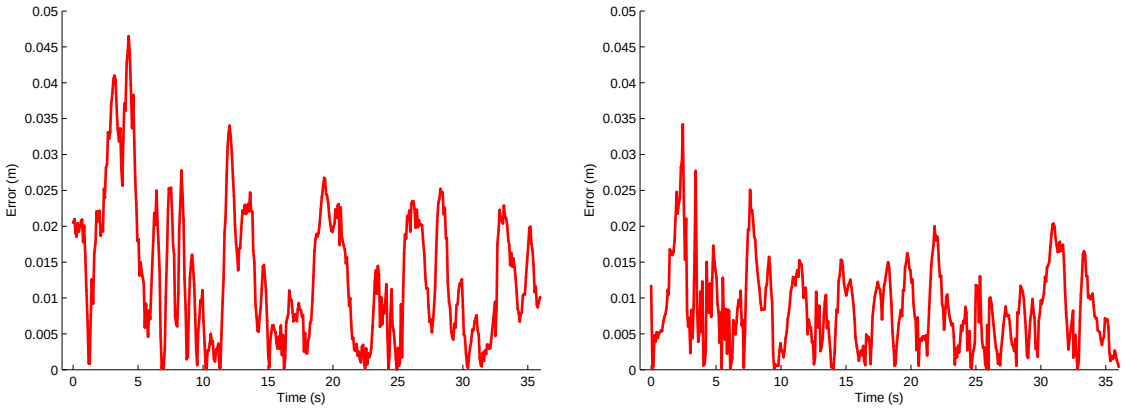
5.6 Inbuilt quadcopter controllers

In order to analyse the autopilot controller present within the simulator, a step input was used in the controller

The controller developed in Section 3.4.2 uses the approximation that the autopilot controller is perfect. However, when testing the simulated quadcopter on Gazebo, the results show that it is not true at all.

By analysing Figure 5.14 is is possible to define some features about the controller.

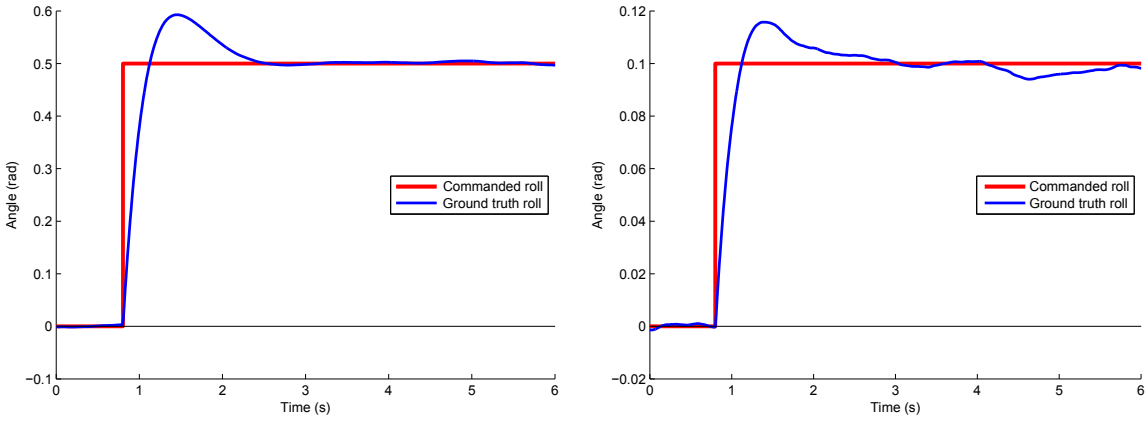
- The rising time is ~250 ms. This value is not very low, considering the kind of system it is applied, but it is much faster than the settling time.
- The settling time is ~1800 ms. This value is very high and reduces the performance of the system response. On the other hand, it is useful to have a lower performance on simulation because if the



(a) Absolute error in X coordinate

(b) Absolute error in Y coordinate

Figure 5.13: Absolute error between the output from the VBHEA and the ground truth from the simulator



(a) Response to a step = 0.5 rad

(b) Response to a step = 0.1 rad

Figure 5.14: Step response of the simulated quadcopter autopilot, for the roll movement

developed controllers work in a noisy simulation environment, they will probably also work on real systems.

- The overshoot is ~20%. This overshoot value is normal and it helps on setting the linear velocity of the quadcopter faster.
- The steady state value is very close to the desired value, having some noise introduced in order to have a simulation more similar to the real system.

The roll and pitch commanded by the hovering controller vary within the interval $[-0.1, 0.1]\ rad$. For easily analyse what happens to the roll and pitch of the quadcopter, the Figure 5.15 shows the response to a 0.01 rad step. In this figure, it is possible to observe the maximum error on the roll, which is ~0.006 rad, which is an acceptable value, corresponding to less than 10% of the maximum commanded roll. Note that the scale for the angle axis is expressed in $10^{-3} rad$.

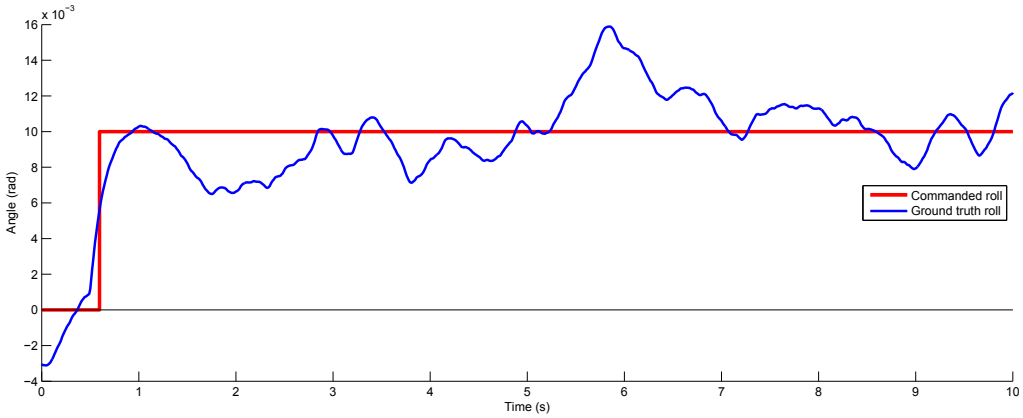


Figure 5.15: Response of the simulated quadcopter autopilot, for a roll step=0.01 rad

5.7 Case 1 - Steady Hovering

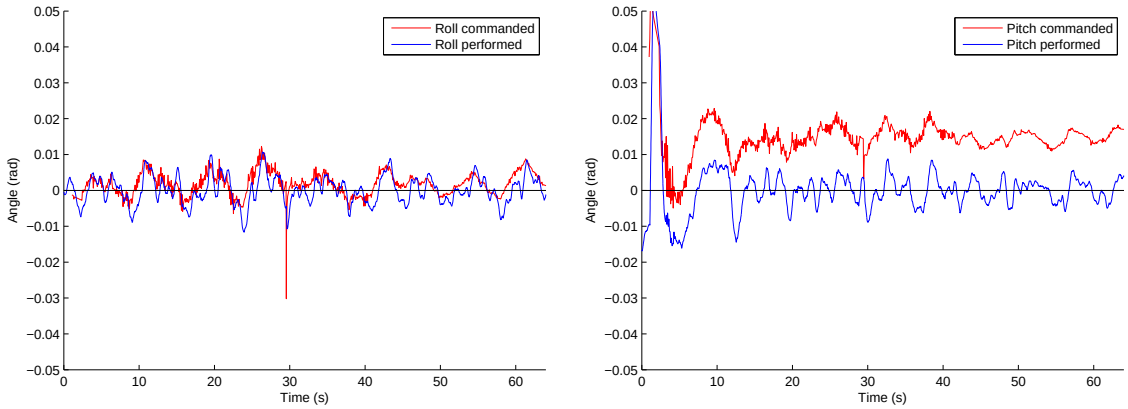
This study case showed positive results. The quadcopter was able to hover the target without loose the markers from the image. The Figure 5.16 shows some interesting results.

It is notorious a constant drift within the pitch that is being continuously corrected by hovering controller. This simulated drift aims on creating more realistic data, as this kind of systematic error exists in real sensors. Since the quadcopter was hovering, this is a good result; the developed controller is able to successfully compensate the existent drift.

The differences between the input and the output variables were expected, from the results in Section 5.6. Although the output does not replicate de intended values, it still follows the general commands, thus being able to control the quadcopter.

5.8 Case 2 - Landing on a Steady Target

This study case tested the landing process of a simulated quadcopter and it showed some good results. The visualization of this simulation is available at [36]. The quadcopter landed on the target and



(a) Comparing roll

(b) Comparing pitch

Figure 5.16: Comparing the commanded roll and pitch with the real output

the reference point was visible within the image. As the camera has a low angle of view (~45º), in order to being possible to see the reference marker, the camera must be within an area with less than 5 cm radius. As the reference point was visible within the image, this means the quadcopter landed on the right place, with a 5 cm error.

The quadcopter landed smoothly and it took less than one minute to do it. In Figure 5.17 it is possible to see the vertical trajectory followed by the quadcopter.

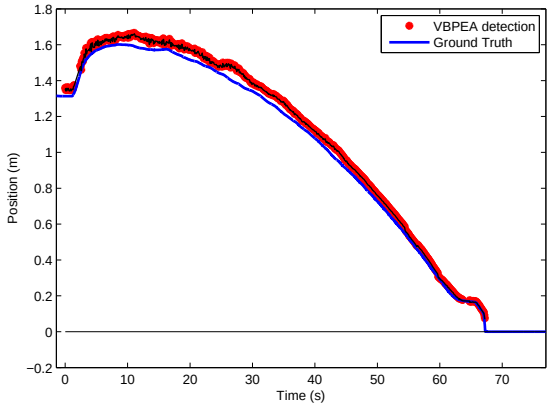


Figure 5.17: Height of the quadcopter during the landing process

A movie about this experiment can be viewed in [36].

6

Conclusions and Future Work

Contents

6.1 Conclusions

In this thesis, a method for autonomous landing a quadcopter was developed. It was demonstrated that it is possible to implement these methods in a non calibrated and unknown environment, as long as it is possible to use a markers board within the scenario. It was also proved that it is possible to perform an autonomous landing with onboard sensors only.

An algorithm for full autonomous hovering and landing was developed. For the quadcopter's pose estimation, a Vision-Based Pose Estimation Algorithm, based on markers observation, was implemented and improved, by considering the fixed orientation of the on board camera. This method demonstrated dependable results, good for the controlling algorithms.

In order to prove the concepts in this thesis, a virtual simulation environment was also developed, using ROS tools. This implementation not only allowed to test these methods but it will also be useful for further investigations within the fields of UAVs robotics. The simulator also includes a model for a quadcopter. In the simulator, a successful landing was taken, thus producing the desired achievement.

In conclusion, the following items were developed:

- Visual-based pose estimation algorithm development and implementation;
- Identification of the required software for onboard controlling, for future development;
- Development of a simulator with a controllable quadcopter;
- Autonomous hovering algorithm;
- Autonomous landing algorithm.

Although the autonomous landing was not performed on a real quadcopter, the developed algorithms and implementations are able to solve the initial addressed problem, which is the development of a quadcopter autonomous landing algorithm (in this case, a simulated quadcopter).

6.2 Future Work

There are several possible future developments within this work.

The most obvious future development is the implementation on a real quadcopter. The communications bridge between the Pandaboard and the Paparazzi must be considered.

The VBPEA and the VBHEA could become one single algorithm in order to improve the performance of the pose estimation.

It is possible to improve the simulator by adding a moving station with a markers board, to help further development on following and landing on moving targets.

For the pose estimation algorithms, the information from the IMU could be added, possibly resulting in a more accurate pose estimation and improving noise reduction. For the (X, Y) estimation from the VBHEA, a model including the small angle would also improve the estimations.

Using the image, a method to track the changes within the image could be developed, thus estimating the camera motion. This could be useful for the hovering application. This method is also known as optical flow.

Bibliography

[1] R. Mahony and V. Kumar. Aerial robotics and the quadrotor [from the guest editors]. *Robotics Automation Magazine, IEEE*, 19(3):19–19, September 2012. ISSN 1070-9932. doi: 10.1109/MRA.2012.2208151.

[2] Institute for Systems and Robotics, Instituto Superior Técnico. The Rescue Project, 2013. URL `http://rescue.isr.ist.utl.pt/`. Consulted on October 2013.

[3] Academia da Força Aérea. PITVANT - Programa de Investigação e Tecnologia em Veículos Aéreos Autónomos Não-Tripulados. *Cadernos do IDN*, (4):9–24, July 2009.

[4] UAVision, 2013. URL `http://uavision.com/`. Consulted on October 2013.

[5] MikroKopter, 2013. URL `http://www.mikrokopter.de/`. Consulted on October 2013.

[6] E-Volo. Volocopter, 2013. URL `http://www.e-volo.com/`. Consulted on October 2013.

[7] Wikipedia. List of Unmanned Aerial Vehicles, March 2010. URL `http://en.wikipedia.org/wiki/List_of_unmanned_aerial_vehicles`. Consulted on October 2013.

[8] R. Mahony, V. Kumar, and P. Corke. Multirotor aerial vehicles: Modeling, estimation, and control of quadrotor. *Robotics Automation Magazine, IEEE*, 19(3):20–32, September 2012. ISSN 1070-9932. doi: 10.1109/MRA.2012.2206474.

[9] Hyon Lim, Jaemann Park, Daewon Lee, and H. J. Kim. Build your own quadrotor: Open-source projects on unmanned aerial vehicles. *Robotics Automation Magazine, IEEE*, 19(3):33–45, September 2012. ISSN 1070-9932. doi: 10.1109/MRA.2012.2205629.

[10] T. Tomic, K. Schmid, P. Lutz, A. Domel, M. Kassecker, E. Mair, I.L. Grixa, F. Ruess, M. Suppa, and D. Burschka. Toward a fully autonomous uav: Research platform for indoor and outdoor urban search and rescue. *Robotics Automation Magazine, IEEE*, 19(3):46–56, September 2012. ISSN 1070-9932. doi: 10.1109/MRA.2012.2206473.

[11] A. Franchi, C. Secchi, M. Ryll, H. H. Bulthoff, and P. R. Giordano. Shared control: Balancing autonomy and human assistance with a group of quadrotor uavs. *Robotics Automation Magazine, IEEE*, 19(3):57–68, September 2012. ISSN 1070-9932. doi: 10.1109/MRA.2012.2205625.

[12] I. Palunko, P. Cruz, and R. Fierro. Agile load transportation: Safe and efficient load manipulation with aerial robots. *Robotics Automation Magazine, IEEE*, 19(3):69–79, September 2012. ISSN 1070-9932. doi: 10.1109/MRA.2012.2205617.

[13] AD King. Inertial Navigation -– Forty Years of Evolution. *GEC review*, 13(3):140–149, 1998.

[14] R. D'Andrea, A. Schollig, M. Sherback, and S. Lupashin. A simple learning strategy for high-speed quadrocopter multi-flips. In *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, pages 1642–1648, 2010. doi: 10.1109/ROBOT.2010.5509452.

[15] ©Vicon Motion Systems Ltd. UK, 2013. URL `http://www.vicon.com/`. Consulted on October 2013.

[16] Daniel Mellinger, Michael Shomin, and Vijay Kumar. Control of quadrotors for robust perching and landing. In *Proc. Int. Powered Lift Conf*, pages 119–126, 2010.

[17] Inkyu Sa and Peter Corke. Estimation and Control for an Open-Source Quadcopter. In *Proceedings of the Australasian Conference on Robotics and Automation 2011*, 2011.

[18] Daniel Mellinger, Nathan Michael, and Vijay Kumar. Trajectory Generation and Control for Precise Aggressive Maneuvers with Quadrotors. *The International Journal of Robotics Research*, 31(5): 664–674, 2012.

[19] Johannes Friis, Ebbe Nielsen, Rasmus Foldager Andersen, Jesper Boending, Anders Jochumsen, and A Friis. Autonomous Landing on a Moving Platform. *Control Engineering, 8th Semester Project, Aalborg University, Denmark*, 2009.

[20] P Robuffo Giordano, H Deusch, J Lächele, and HH Bülthoff. Visual-vestibular feedback for enhanced situational awareness in teleoperation of UAVs. In *Proc. of the AHS 66th Annual Forum and Technology Display*, 2010.

[21] Raffaello D'Andrea. The Astounding Athletic Power of Quadcopters. `http://www.ted.com/talks/raffaello_d_andrea_the_astounding_athletic_power_of_quadcopters.html`, June 2013. Consulted on October 2013.

[22] Aplicaciones de la Visión Artificial from Universidad de Córdoba. ArUco: a minimal library for Augmented Reality applications based on OpenCV, April 2013. URL `http://www.uco.es/investiga/grupos/ava/node/26`. Consulted on October 2013.

[23] Human Interface Technology Laboratory (HIT Lab) at the University of Washington, HIT Lab at the University of Canterbury, New Zealand, and ARToolworks, Inc, Seattle. ARToolKit, 2013. URL `http://www.hitl.washington.edu/artoolkit/`. Consulted on October 2013.

[24] Hirokazu Kato and Mark Billinghurst. Marker tracking and hmd calibration for a video-based augmented reality conferencing system. In *Augmented Reality, 1999.(IWAR'99) Proceedings. 2nd IEEE and ACM International Workshop on*, pages 85–94. IEEE, 1999.

[25] Tommaso Bresciani. *Modelling , Identification and Control of a Quadrotor Helicopter*. PhD thesis, Lund University, 2008.

[26] ROS - Robotic Operative System, 2013. URL `http://www.ros.org`. Consulted on October 2013.

[27] Jack B Kuipers. *Quaternions and Rotation Sequences*. Princeton university press, Princeton, 1999.

[28] Don Koks. *Explorations in Mathematical Physics: The Concepts Behind an Elegant Language*. Springer, 2006.

[29] Henrique Ribeiro Delgado da Silva. Controlo de Formações em Veículos Aéreos não Tripulados. Master's thesis, Instituto Superior Técnico, June 2012.

[30] ROS.ORG. camera_calibration ROS Package, 2013. URL `http://www.ros.org/wiki/camera_calibration`. Consulted on October 2013.

[31] Francesc Moreno-Noguer, Vincent Lepetit, and Pascal Fua. Accurate Non-Iterative $O(n)$ Solution to the PnP Problem. In *Computer Vision, 2007. ICCV 2007. IEEE 11th International Conference on*, pages 1–8. IEEE, 2007.

[32] Satoshi Suzuki and Keiichi Abe. Topological Structural Analysis of Digitized Binary Images by Border Following. *Computer Vision, Graphics, and Image Processing*, 30(1):32–46, 1985. URL `http://dblp.uni-trier.de/db/journals/cvgip/cvgip30.html#SuzukiA85`. Consulted on October 2013.

[33] Gazebo, 2011. URL `http://gazebosim.org/`. Consulted on October 2013.

[34] Paparazzi Project. URL `http://paparazzi.enac.fr/wiki/Autopilots`. Consulted on October 2013.

[35] Pandaboard. URL `http://pandaboard.org/`. Consulted on October 2013.

[36] Tiago Gomes Carreira. Master Thesis Extra Files. Master's thesis, Instituto Superior Técnico, October 2013. URL `http://web.ist.utl.pt/tiago.carreira/thesis`.

[37] Yi Ma, Stefano Soatto, Jana Kosecka, and S. Shankar Sastry. *An Invitation to 3-D Vision: From Images to Geometric Models*. SpringerVerlag, 2004. ISBN 978-1-4419-1846-8. doi: 10.1007/978-0-387-21779-6.

[38] S Bouabdallah. *Design and Control of Quadrotors with Application to Autonomous Flying*. PhD thesis, Ecole Polytechnique Federale de Lausanne, 2007.

[39] Richard Szeliski. *Computer Vision: Algorithms and Applications*. Springer, 2011. URL `http://szeliski.org/Book/`. Consulted on October 2013.

Appendices

A.1 Pandaboard Software Installation

Everything was tested on a PandaBoard ES rev B2, with a 8GB, class6 microSD card (with adapter).
This instructions were made within the thesis. May 2013
http://rm.isr.ist.utl.pt/projects/quads/wiki/PandaSoftware

A.1.1 Ubuntu

1. Pre-Requisites
   - 8Gb SD card
   - USB keyboard and mouse
   - 5V/4A power supply
   - HDMI cable (used a hdmi/vga converter)
2. Ubuntu Image
   - download the Texas Instruments OMAP4 (Hard-Float) preinstalled desktop from http://cdimage.ubuntu.com/releases/12.04/release/ (http://cdimage.ubuntu.com/releases/12.04/release/ubuntu-12.04-preinstalled-desktop-armhf+omap4.img.gz)
   - Note: The server version is faster, but needs extra tricks.
3. Preparing SD Card
   - Format SD card
   - Make sure it stays unmounted and corresponds to /dev/sdb (use Disk Utility)
   - Move to folder where you downloaded the Ubuntu Image file.
   - Type the following in terminal

```
zcat ./ubuntu-12.04-preinstalled-desktop-armhf+omap4.img.gz |sudo
   dd bs=4M of=/dev/sdb
sudo sync
```

   - In order to use the full space, use GParted to resize the 2GB partition tu use all available space. (use option Round to nothing, if available)
4. Bootup the Pandaboard
   - Connect periferals to your pandaboard (hdmi display, usb keyboard/mouse, ethernet cable).
   - Insert SD card
   - Connect power supply (5V)
   - If everything is OK, there should be a familiar Ubuntu Installation screen.

- Follow the instructions on the screen. Should be piece of cake (takes a while to complete).

5. Installing the TI ppa update

   - Configure su password (input username password, and then su password twice):

     ```
     sudo passwd
     ```

   - Add TI OMAP release PPA (if there is a proxy, configure it before)

     ```
     su \# SuperUser mode
     add-apt-repository ppa:tiomap-dev/release
     apt-get update
     apt-get dist-upgrade
     apt-get install ubuntu-omap4-extras
     apt-get dist-upgrade
     exit \# return to user mode
     ```

   - After waiting A LOT of time completing the last step, **reboot the board**

   - Because we have a PandaES 4460 (run in user mode):

     ```
     /usr/bin/alsaucm -c PandaES set \_verb HiFi
     ```

   - Change the bootargs

     ```
     sudo cp /boot/boot.script /boot/boot.script.bak \# making a backup
        file
     sudo gedit /boot/boot.script
     ```

     - Please replace "vram=40M mem=456M@0x80000000 mem=512M@0xA0000000" or any vram or split mem settings with: "mem=1G@0x80000000"
     - Example of my file:

       ```
       fatload mmc 0:1 $0\times80000000$ uImage
       fatload mmc 0:1 $0\times81600000$ uInitrd
       setenv bootargs ro elevator=noop mem=1G@0x80000000 root=UUID=66
          ae1943-9036-498f-b6b5-501b43b8d550 fixrtc quiet splash
       bootm $0\times80000000$ $0\times81600000$
       ```

     - You need to force installation of new bootloaders through command:

       ```
       sudo /usr/sbin/flash-kernel ---update-bootloader
       ```

   - Reboot to check everything is OK

A.1.1.A Some software

- Install the following software tools:

  - *chkconfig
  - *GIT (for OpenCV)

- *SVN (optional)
- *SSH Server (optional)
- *vim (terminal text editor)

```
sudo apt-get install chkconfig git-core subversion openssh-server
  vim
```

- `sudo apt-get install qtcreator \# desktop text editor (good one)`

A.1.2 ROS

Following the guide for installing the experimental version
(http://www.ros.org/wiki/groovy/Installation/UbuntuARM)

1. Setup your sources.list (Ubuntu 12.04)

```
sudo sh -c 'echo "deb http://packages.ros.org/ahendrix-mirror/
  ubuntu precise
main" \textgreater{} /etc/apt/sources.list.d/ros-latest.list'
```

2. Set up your keys

```
wget http://packages.ros.org/ros.key -O - | sudo apt-key add -
```

3. Installation

```
sudo apt-get update
sudo apt-get install ros-groovy-ros ros-groovy-ros-base
```

4. Initialize rosdep

```
sudo rosdep init
rosdep update
```

5. Environment setup

```
echo "source /opt/ros/groovy/setup.bash" \textgreater{}\textgreater
  {} \~{}/.bashrc
source \~{}/.bashrc
```

6. Getting rosinstall

```
sudo apt-get install python-rosinstall
```

7. Reboot and make sure if everything is OK.
8. Setup another (main) Environment

```
rosws init \~{}/groovy\_workspace /opt/ros/groovy
echo "source \~{}/groovy\_workspace/setup.bash" \textgreater{}\
  textgreater{} \~{}/.bashrc
```

A.1.2.A Others

- Dependencies

```
sudo apt-get install ros-groovy-rqt ros-groovy-rqt-common-plugins
sudo apt-get install ros-groovy-image-common ros-groovy-image-view
sudo apt-get install v4l-*
```

- Things installed `sudo apt-get install ros-groovy-ros-tutorials`

- Bunch of things (some are already installed)

```
sudo apt-get install python-pip build-essential python-yaml cmake
  subversion wget python-setuptools mercurial git-core python-yaml
  libapr1-dev libaprutil1-dev libbz2-dev python-dev python-empy python
  -nose libgtest-dev python-paramiko libboost-all-dev liblog4cxx10-dev
   pkg-config libqt4-dev qt4-qmake
```

- usb_cam (a good driver to capture image, in ROS)

  - Download package bosch_drivers

```
roscd
rosws set usb_cam "git://github.com/bosch-ros-pkg/usb_cam.git" --
   git
source setup.bash
rosws update usb_cam
rosdep install usb_cam
```

  - Then, you need to edit usb_cam/src/usb_cam.cpp

    * Find the line where it is used this function `avcodec_decode_video2( .... )`
    * Then you need to find the very previous

      \#if LIBAVCODEC_VERSION_MAJOR > 52

      (less than 10 lines before)
    * and change it into this:

      \#if LIBAVCODEC_VERSION_MAJOR > 50

    * Save and close the file (There are at least 2 `#if LIBAVCODEC_VERSION_MAJOR > 52` so be careful to change THIS line).

  - Then you need to compile all

```
roscd usb\_cam
cmake .
make
```

  - In the end, test it

    * In one terminal

```
roscore
```

* Open another terminal

```
rosrun usb_cam usb_cam_node
# if more than one camera in usb (or something wrong with device
    /dev/video0):
# rosrun usb_cam usb_cam_node _video_device:=/dev/video1 \#
   check first what is the device..
```

(should see something running)

* Open another terminal

```
rosrun image_view image_view image:=/usb_cam/image_raw
```

* You should see now a window with an image from the webcam. (If not, don't ask me).

A.1.3 OpenCV

- Takes forever to do this.

1. Step 1

   - Delete any sotfware dependency

   ```
   sudo apt-get update
   sudo apt-get remove ffmpeg x264 libx264-dev libvpx0
   ```

   - if ' E: Unable to locate <package> ', retry the last line without that <package>

2. Step 2

   - Install libgstreamer

   ```
   sudo apt-get install libgstreamer0.10-0 libgstreamer0.10-dev
     gstreamer0.10-tools gstreamer0.10-plugins-base libgstreamer-
     plugins-base0.10-dev gstreamer0.10-plugins-good gstreamer0.10-
     plugins-ugly gstreamer0.10-plugins-bad gstreamer0.10-ffmpeg
   ```

3. Step 3

   - Install some adicional packages (some are repeated, but never mind)

   ```
   sudo apt-get install build-essential checkinstall git cmake libfaac
     -dev libjack-jackd2-dev libmp3lame-dev libopencore-amrnb-dev
     libopencore-amrwb-dev libsdl1.2-dev libtheora-dev libva-dev
     libvdpau-dev libvorbis-dev libx11-dev libxfixes-dev libxvidcore-
     dev texi2html yasm zlib1g-dev
   ```

4. Step 4

   - Get x264 codecs

```
cd
git clone git://git.videolan.org/x264.git
cd x264
./configure ---enable-static ---enable-pic ---enable-shared
make
sudo make install
cd
```

5. Step 5

   - Get libvpx

   ```
   cd
   git clone http://git.chromium.org/webm/libvpx.git
   cd libvpx
   ./configure ---enable-static ---enable-pic
   make
   sudo make install
   cd
   ```

6. Step 6

   - Download ffmpeg version 0.7.x from ~~http://ffmpeg.org/releases/ (alrealdy in the script)~~ (version >= 0.8.x is not compatible with OpenCV 2.3)

   ```
   cd
   wget http://ffmpeg.org/releases/ffmpeg-0.7.15.tar.gz
   tar -xvf ffmpeg-0.7.15.tar.gz
   cd ffmpeg-0.7.15
   ./configure ---enable-shared ---enable-libvpx ---enable-pic ---
      enable-gpl ---enable-libfaac ---enable-libmp3lame ---enable-
      libopencore-amrnb ---enable-libopencore-amrwb ---enable-
      libtheora ---enable-libvorbis ---enable-libx264 ---enable-
      libxvid ---enable-nonfree ---enable-postproc ---enable-version3
     ---enable-x11grab ---disable-ffserver
   make
   sudo make install
   cd
   ```

7. Step 7

   - Get libjpeg62

   ```
   sudo apt-get install libjpeg62 libjpeg62-dev
   ```

8. Step 8

   - Download v4l-utils-**.**.*.tar.bz2 from ~~http://www.linuxtv.org/downloads/v4l-utils/ (Already in the script)~~

```
cd
wget http://www.linuxtv.org/downloads/v4l-utils/v4l-utils-0.9.5.tar
    .bz2
tar -xvf v4l-utils-0.9.5.tar.bz2
cd v4l-utils-0.9.5
./configure
make
sudo make install
cd
```

9. Step 9

   - Get libgtk

     ```
     sudo apt-get install libgtk2.0-0 libgtk2.0-dev
     ```

10. Step 10 – OpenCV

    - Download file from ~~http://sourceforge.net/projects/opencvlibrary/files/ (should be on the script)~~

      ```
      cd
      wget http://switch.dl.sourceforge.net/project/opencvlibrary/opencv-
          unix/2.4.5/opencv-2.4.5.tar.gz
      tar -xvf opencv-2.4.5.tar.gz
      cd opencv-2.4.5
      ```

    - Then

      ```
      mkdir release
      cd release
      cmake -D CMAKE_BUILD_TYPE=RELEASE -D CMAKE_INSTALL_PREFIX=/usr/
          local -D BUILD_NEW_PYTHON_SUPPORT=ON -D BUILD_EXAMPLES=ON ..
      ```

    - Then check if the output of cmake has: (should have)

      ```
      GTK+ 2.x: YES
      FFMPEG: YES
      GStreamer: YES
      V4L/V4L2: Using libv4l
      ```

    - In the end: (infinite time ++)

      ```
      make
      sudo make install
      cd
      ```

11. Step 11 – Configure Linux with OpenCv

    - First configure .bashrc

```
cd
echo "export LD\_LIBRARY\_PATH=/usr/local/lib" \textgreater{}\
  textgreater{} .bashrc
su
```

- And in the end (in su mode)

```
echo "PKG\_CONFIG\_PATH=\$PKG\_CONFIG\_PATH:/usr/local/lib/
  pkgconfig" \textgreater{}\textgreater{} /etc/bash.bashrc
echo "export PKG\_CONFIG\_PATH" \textgreater{}\textgreater{} /etc/
  bash.bashrc
exit
```

12. Step 12 – Reboot

A.2 Markers Sheet

The full A4 markers sheet is present in the next page.