Visual Tracking and Control of a Quadrocopter

Martin Ericsson

LUND UNIVERSITY

Department of Automatic Control

MSc Thesis
ISRN LUTFD2/TFRT--5965--SE
ISSN 0280-5316

Department of Automatic Control
Lund University
Box 118
SE-221 00 LUND
Sweden

Printed in Sweden by Media-Tryck
Lund 2015

Contents

0.1 Acknowledgements

Foremost, I thank my dear Yasemin for her endless support, encouragement and patience. I also would like to show gratitude towards Vladimerous Vladimerou for making this thesis possible and for all valuable help I received during the first months of this work. Magnus Linderoth for the ideas he provided whenever my own creativity was lacking. David Anisi for making the summer I spent at ABB, Oslo a great experience allowing me to work in a professional environment alongside people with knowledge by far exceeding mine. And of course Anders Robertsson for his seemingly infinite amount of work hours and late night problem solving. Karl-Erik Årzén for helping me putting together the final pieces of this report.

Abstract

The main goal of this master thesis project was to take a manually controlled consumer quality quadrocopter and build the foundations which allows it to be autonomously controlled. This was achieved by introducing a exterior frame of reference through the use of a webcam coupled with image analysis algorithms. The position and rotation of the quadrocopter was identified, and several control structures were implemented and tested, which allowed the quadrocopter to be commanded to move to positions in space. Both the onboard ultrasound sensor, and an altitude estimation through image analysis were used to control the altitude of the quadrocopter. Position control in x, y and orientation (yaw rotation) completely relied on data extracted and analysed from the video stream. Control of velocity along a predefined trajectory was also successfully implemented, which enables future development of an obstacle avoiding path planner. Lastly, the master thesis also covers work carried out at ABB's Strategic R&D department for oil, gas and petrochemicals i Oslo, Norway. Here the focus was on using a quadrocopter to track and follow the motion of an industrial robot by analysing the video stream of the onboard camera.

Chapter 1

Introduction

The purpose of this master thesis has been to investigate how vision based feedback can be used to give an already well stabilized quadrocopter an absolute or relative frame of reference, whereas two separate solutions have been implemented. The first solution is by utilizing a ceiling mounted camera which through the means of image processing techniques gives the position and rotation of the quadrocopter. The second solution is by using a forward facing onboard camera of the quadrocopter to visually track an industrial robot to give an estimate of its position relative the quadrocopter. This information is used to automatically control the quadrocopter to follow the motion of the robot.

1.1 Visual feedback from external camera

The purpose of this part of the master thesis is to improve the flying capabilities of a quadrocopter by using vision based feedback. The quadrocopter itself is equipped with two cameras, one frontfacing and one directed toward the ground. Additionally, one webcamera is mounted on the ceiling of the room. Glued to the quadrocopter (see Figure 1.1) is an image of a symbol with known size and characteristics that make it easy to identify. This enables the calculation of the position of the quadrocopter in the room, given that it is within the view of the camera. This information has been used to automatically move the quadrocopter in its 4 degrees of freedom, namely acceleration forward/backwards, acceleration left/right, altitude velocity and yaw rotation velocity. The quadrocopter used in this thesis is the consumer oriented Parrot AR.Drone [3].

1.2 Autonomous Tracking of Industrial Robots

During the summer of 2011, as a summer student project at ABB Strategic R&D for oil, gas and petrochemicals, a system for autonomous tracking of

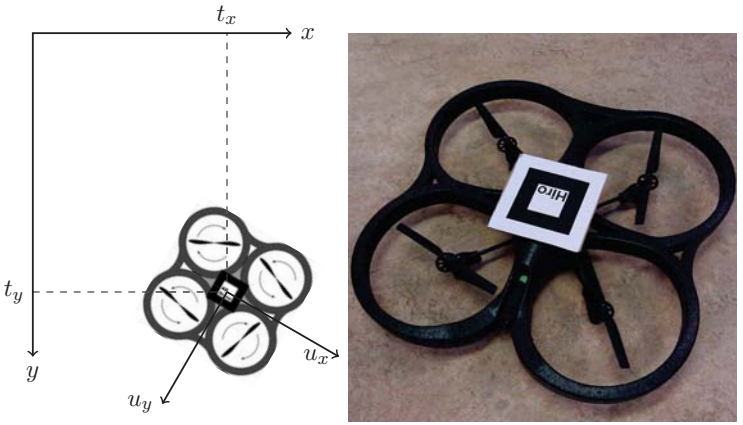

Figure 1.1: Left: The AR.Drone as viewed from the camera, Right: A photo of the Parrot AR.Drone equipped with marker

an industrial robot with a quadrocopter has been developed.

New oil and gas fields have in recent years been increasingly difficult to find. Most accessible fields have already been explored and to keep the supply high, the industry need to explore its opportunities in, from a safety perspective, harsh environments. Along with this development in industry, comes a demand for automation and the use of robotics. In fields with unsatisfactory health, safety and environment (HSE), the use of robotics is not only desired, but necessary to enable exploration. This requires a high level on the automation of the plants but also in the cases where a human remote operator must interact, on the human machine interface (HMI) to give the operator feedback to execute a task. The goal is to reach a level of situational awareness to the extent that the robot is the ears, eyes and hands of the remote operator. To overcome this obstacle traditionally several stationary cameras have been used. In harsh environments it is often difficult to find room for enough cameras and the limited point of views are overcome by putting cameras in the hands of industrial robots. While this also does provide an advantage compared to stationary cameras with regards to mobility, it is expensive. A camera equipped quadrocopter is a much more suited tool for this task since it is comparably cost efficient, light weight and much more mobile.

Also in this part of the thesis, has the Parrot AR.Drone [3] been used. It has two onboard cameras, one pointing down toward the ground and one in the forward direction. All control action has been based on analysis of the video stream from the forward facing camera which sends a new image 15 times per second. The quadrocopter has four degrees of freedom, out of which, three have been used to position and orient the quadrocopter. The

goal is to always point toward the robot and also be positioned orthogonally to the robot. The control signals used to achieve this is acceleration in the directions right/left (roll) and forward/backward (pitch) as well as rotational velocity (yaw rate). The fourth and unused degree of freedom is altitude which is set to a constant. The setup has also been shown as a demonstration of future technologies at an Open House event at ABB, Oslo. During the limited time I spent at ABB, focus was on getting the basic functionality to work and producing presentable results had a low priority. Therefore no results from this part of the thesis is presented.

1.3 Related Work

In recent years, the development of small remotely controlled quadrocopters has made an impact in the scientific community as a platform for research on Unmanned Autonomous Vehicles (UAVs). While the quadrocopter as a concept was invented as early as in the 1920's, and some less successful prototypes were made, it is not until the last 10 years that we have seen well functioning implementations of quadrocopters. The biggest problem with the early incarnations of quadrocopters, was the difficulty for a human to operate them. The success of the design in recent years is in the form of unmanned small radio controlled vehicles, where the stabilization of the vehicle is handled by efficient control algorithms instead of by a human. It is mainly the development of fast processors, long lasting batteries, and cheaper brushless motors that has enabled the realization of these remotely controlled quadrocopters.

The scientific community has taken a large interest in quadrocopters utilizing them as UAVs. Much of the research is focused on enabling the quadrocopter to navigate autonomously. This field can be divided into two areas, one which utilizes an external frame of reference, for example Vicon motion capture [10] used by the GRASP Lab of University of Pennsylvania [6]. The other area is navigation based on input from sensors mounted on the quadrocopter itself. By using an external frame of reference, you impose a significant restriction in the working area of the quadrocopter. For example, GRASP Lab, University of Pennsylvania, uses an indoor lab environment of 5x5x5 m in their research. While this is a very small space, the combination of an external tracking system and fast computers enables some truly spectacular moves [13]. Another example of the usage of a fast external reference system is the article on Quadrocopter Ball Juggling, carried out at ETH Zürich [14]. A group of methods and algorithms commonly used when navigating with no external frame of reference and no previous information about the environment is called "Simultaneous Localization and Mapping" (SLAM). This field typically utilizes a camera or range camera to create a model of its surroundings in real-time [11].

1.4 Outline of Report

The report consists of the following chapters. Below is a small description of the contents of each chapter.

- **Introduction:** This chapter contains a description of the two parts of this master thesis. It also includes a section about related work on the subject, and the outline of the report.
- **Modeling:** This chapter describes the dynamics of a general quadrocopter, and how it is commonly controlled.
- **System Setup:** An overall description of the control system developed in this master thesis. It is a broader overview, not getting into the details of the algorithms.
- **Method:** Is a detailed description of the control and image processing algorithms that are necessary to close the loop. This is divided into the two parts of this master thesis, namely the work conceived at Department of Automatic Control, LTH, and at ABB Oslo.
- **Results:** Describes the results of the work at the Department of Automatic Control, LTH.
- **Conclusion:** Some concluding words about the thesis work, and a chapter with suggestions about possible improvements.
- **Appendix:** Here you'll find the source code of thesis.

Chapter 2

Modeling

2.1 The dynamics of a quadrocopter

A quadrocopter has, as the name implies, four rotors, all of them providing lift force in the same direction. Two rotors in opposing corners provide lift force by rotating clockwise (CW) and the other two by rotating counter-clockwise (CCW). This cancels out the moment that otherwise would cause the quadrocopter to spin out of control. Let Ω_i be the total rotor speed of each motor, and Ω_H the rotor speed which stabilizes the altitude of the vehicle. Different control actions can be achieved by adding and subtracting to the speed of certain rotors. A positive rotation around the yaw axis is achieved by adding to the CW rotating rotors a $\Delta_A > 0$ and subtracting a $\Delta_B > 0$ from the CCW rotating rotors, while maintaining the same lift force. This will change the torque of the vehicle around the yaw axis, and it will start to rotate increasingly. To achieve a change in pitch angle, a $\Delta_A > 0$ is added to the speed of rotors 1 and 2, and rotation speed of rotor 3 and 4 is decreased by a $\Delta_B > 0$, see Figure 2.1. A change in roll angle is achieved in the same manner, however by adding rotation speed of rotor 1 and 4 while decreasing the speed of rotor 2 and 3. Throttle is acceleration in the direction of the rotors, and is achieved by increasing the rotor speed of all motors. The combinations for manipulating throttle, roll, pitch and yaw for a general quadrocopter are summarized in Table 2.1. If the motors are linear the approximation, $\Delta_A \approx \Delta_B$ can be made. In this thesis the variable ϕ represent the pitch angle, θ the roll angle where $\theta = 0$ and $\phi = 0$ is the angle when the quadrocopter is parallell to the ground. The yaw angle must be defined in a coordinate system and is often referred to as ψ.

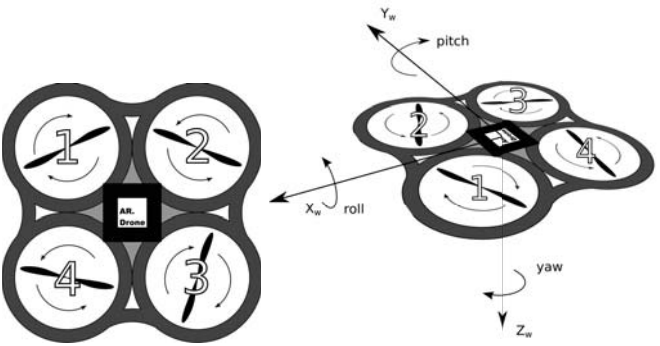

Figure 2.1: Left: Rotation directions of each rotor, Right: Description of roll, pitch and yaw rotations for a general quadrocopter

Maneuver	Ω_1	Ω_2	Ω_3	Ω_4
throttle	$\Omega_H + \Delta_A$	$\Omega_H + \Delta_A$	$\Omega_H + \Delta_A$	$\Omega_H + \Delta_A$
pitch	$\Omega_H + \Delta_A$	$\Omega_H + \Delta_A$	$\Omega_H - \Delta_B$	$\Omega_H - \Delta_B$
roll	$\Omega_H + \Delta_A$	$\Omega_H - \Delta_B$	$\Omega_H - \Delta_B$	$\Omega_H + \Delta_A$
yaw	$\Omega_H + \Delta_A$	$\Omega_H - \Delta_B$	$\Omega_H + \Delta_A$	$\Omega_H - \Delta_B$

Table 2.1: Rotor speed deviations resulting in a change of throttle, pitch, roll and yaw. Here $\Delta_A > 0$ and $\Delta_B > 0$ represent constant deviation of rotor speed from the altitude stabilizing rotor speed, Ω_H.

Chapter 3

The System Setup

3.1 The AR.Drone

The quadrocopter used in this thesis is called AR.Drone and is developed and built by the french company Parrot [9].

Hardware specification

The AR.Drone is equipped with a 6 degrees of freedom MEMS[1] IMU[2] providing measurement of pitch, yaw, and roll movements. An ultrasound sensor pointing downward gives an estimation of the altitude. There is also one camera pointed forward mainly for the human operator and gaming, and a camera pointing downward which is used to stabilize the motion in the xy-plane. It is powered by a 11.1V LiPo battery at 1000mAh which gives an approximate fly time of 12min. All onboard calculations are handled by an ARM9 @468 MHz with 128MB RAM on a Linux/BusyBox system. It is also equipped with a USB port which at the moment can only be used to update firmware but might allow expansions such as a GPS or 3G-modem in the future.

Software specification

Stabilization of the AR.Drone is handled internally, which means that there is no direct access to the motors. All motion is executed by changing reference values for yaw rate, roll and pitch angles along with rate of climb. These reference values are controlled internally by the onboard system of the AR.Drone. The onboard cameras uses a update frequency of 30 Hz which is used by the onboard controller, however this video stream is also sent over Wi-Fi to the operating computer at 15 Hz. All sensor values and cam-

[1] Microelectromechanical systems
[2] Inertial Measurement System

era streams are available to the operating computer but at a lower update frequency than what is used onboard.

It can be controlled by any ad-hoc compatible Wi-Fi device, since the AR.Drone itself sets up an ad-hoc network. The controlling device communicates with the AR.Drone by sending AT commands in UDP-packets on port 5556. Likewise, sensor values and other variables calculated onboard the AR.Drone are received on port 5554 at 30 Hz. A third port (UDP 5555) is used to receive a video stream at 15 Hz from either of the two cameras, but not both at once.

How the AR.Drone is controlled internally is for the most part unknown. What is known is that the vertical camera is used to estimate the velocity in the xy-plane by using optical flow [8] estimation. Optical flow is also used to keep the yaw rotation stationary, however in a room with uniform color on the floor, yaw rotation has a tendency to drift slightly. There are also function calls available to set proportional and/or integral gains, which implies that these are controlled by PI-controllers. Table 3.1 lists available gains and corresponding control variables. However, it is not known exactly how the errors for each variable is calculated nor how they mix into motor speeds.

variable	*controller*	
pitch/roll angular rate	P	K_p
yaw angular rate	PI	K_p K_i
euler angle[1]	PI	K_p K_i
altitude	PI	K_p K_i
vertical speed	PI	K_p K_i
hovering	PI	K_p K_i

Table 3.1: Tunable gains on the onboard controller

The quadrocopter is controlled by using a wrapper for an AT command with the C prototype

```
ardrone_at_set_progress_cmd(int32_t enable, float32_t phi,
float32_t theta, float32_t gaz, float32_t yaw).
```

The arguments are interpreted by the onboard controller as reference values. When a command is to be sent, the first argument, “enable”, must be set to 1, and the remaining floats are variables mapped from -1 to 1 which controls the pitch, roll and climb and angular velocity of the quadrocopter. The arguments phi and theta correspond to pitch and roll angles of the quadrocopter, whereas gaz is climb/descent velocity and yaw is angular

[1]This is, by Parrot’s convention, referring to pitch and roll angles, not yaw.

velocity. All of the variables are internally scaled with a predefined constant e.g., sending a command with $phi \neq 0$ means onboard to set $\phi_{ref} = phi \cdot$ euler_angle_max. The constant euler_angle_max can at most be set to 0.52 rad $= 30^\circ$ which equals a maximum lateral acceleration of $9.82 \cdot \tan 30$ m/s^2 $= 5.67$ m/s^2 while maintaining altitude and disregarding air resistance. The input variables to ardrone_at_set_progress_cmd() are used as reference values to the onboard controller developed by Parrot, and are used as control commands in this thesis. This means that the controls developed will be of a cascaded structure, i.e., an outer loop to Parrot's onboard control. The input reference values to the onboard controller will for the rest of this thesis be regarded as control signals with names according to Table 3.2.

input value		*control signal*	*physical meaning*
float32_t phi	$\sim$	u_y	pitch angle
float32_t theta	$\sim$	u_x	roll angle
float32_t gaz	$\sim$	u_z	climb/descent velocity
float32_t yaw	$\sim$	u_{yaw}	angular velocity

Table 3.2: The meaning of the input variables

A Software Development Kit is provided with the AR.Drone along with a demonstration program with tools for plotting and accessing sensor values. This has served as a foundation for the control algorithms developed in this thesis.

Chapter 4

Method - Visual feedback from external camera

4.1 Background

This is the first part of the thesis, which was conducted at The Department of Automatic Control at Lund University. In this part a camera has been mounted in the ceiling of a room pointing toward the quadrocopter which has a black and white symbol mounted on top. The video stream is analysed and an estimate of the position of the quadrocopter is calculated. By using this information, control signals are calculated and sent to the quadrocopter which enables it to fly to different points in space automatically.

4.2 The Setup

The setup, which can be seen in Figure 4.1, consists of the AR.Drone, one or two computers and a webcam. The AR.Drone sets up an ad-hoc wireless network to which *Computer B* connects. That is all that is necessary for manual control of the AR.Drone. Autonomous flight additionally requires a webcam connected to *Computer A* which sends signals to *Computer B* over a TCP-socket. *Computer A* handles all image analysis of the webcam stream and only sends the extracted data e.g., rotation and the drone's position offset relative to the camera center. Recognition of up to two markers has been implemented where one marker represents the AR.Drone and the other one a setpoint held in a hand or mounted on another vehicle. During the development the tests have been conducted on computers with the processors Pentium 4@3.0 GHz (*Computer A*) and a Pentium M@1.6 GHz (*Computer B*). The programming language C has been chosen for both image processing and for the control algorithms. This choice is motivated by

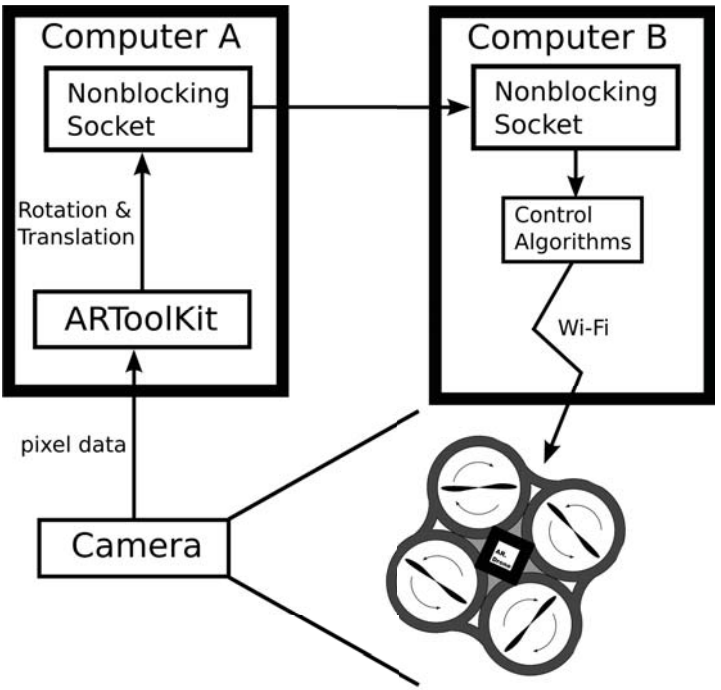

Figure 4.1: Sketch of the system

the fact that both the SDK[1] for AR.Drone as well as ARToolKit [4] are implemented in C. ARToolKit is a framework for detection of binary images, i.e. an image consisting only of the colors black and white, in a video stream. The methods included in ARToolKit simplifies the calculation of the position and orientation of the binary image in the camera frame. ARToolKit is open source and not in any way related to Parrot AR.Drone. The programs have been tested and confirmed to work well on Ubuntu for the control algorithm code (*Computer A*) and Ubuntu and Fedora for the image processing (*Computer A*), see Figure 4.1. The webcam used is a logitech c250 which has a resolution of 640x480 at 30fps but any webcam capable of 30 Hz which also complies with v4l2 [1] standard should work. The implementation has been divided into three different modes namely, *fly to a point in space*, *follow a mobile setpoint* and *follow a trajectory in the xy-plane*. The mobile setpoint tracking also allows the initiation of the built in landing algorithm, when the error is small enough. The camera has been mounted in the ceiling of a room, directed toward the floor. In order to ensure optimal performance of

[1]Software Development Kit

the algorithm, the image plane from the camera must be as closely as possible aligned with the quadrocopters floating plane, i.e., the normal vector of the image plane must be parallel to the gravity vector. For visualization of logged data, and prototyping different parts of the algorithms, Matlab [7] has been used.

4.3 Realtime implementation

The control loop is updated at 30 Hz since this is the frequency at which the video stream and image analysis operates. Below is a simplified sketch in pseudocode of how ARToolKit operates:

```
//visible1 & visible2 are booleans which states whether
//a marker is visible patt_trans1 & patt_trans2 contains
//the translation and rotation of each pattern

waituntil=1/30;
while(1){
   start=gettimeofday();
   image=getvideoframe();
   pattern1=find(pattern1);
   pattern2=find(pattern2);
   patt_trans1 = gettrans(pattern1);
   patt_trans2 = gettrans(pattern2);
   writetosocket(patt_trans1,patt_trans2,visible1,visible2);
   end=gettimeofday();
   sleep(waituntil-end+start);
}
```

The socket is a non-blocking TCP-socket, meaning that the receiving end isn't put in a wait state in case there is no data to receive. A boolean for each marker is also set which states whether the marker is visible or not. If a marker is not visible, the most recently calculated orientation data will be sent. Unfortunately it has been difficult to find a way to guarantee that the received frame really is new. There is a risk that two consecutive frames are identical if the camera for some reason temporarily slows down.

4.4 Image Processing

As mentioned earlier, the system used for image analysis and marker detection is called ARToolKit [4]. It is a C library developed mainly for augmented reality applications. However, its marker detection is good and is well suited for this project. ARToolKit provides some example markers but it can also be trained on any marker you choose which fulfills the following

criterias: The marker must have a thick black border around a white square, see Figure 4.2. What is inside the white square is a question of design, but it should consist of only black and white and must not interfere with the border. For best performance, the imagery inside the white square should keep a low level of detail and must also not be rotationally symmetric. It is also good to make a marker in a non-reflective material, since reflections in the black parts is likely to aggravate marker detection. The image analysis is sensitive to ambient lighting conditions, and the lighting condition must be consistent all over the image plane. Therefore, the best scenario is a room without windows with multiple soft light sources. By default the cameras are initiated with automatic exposure activated, which in low light condition leads to long exposure times and motion blur. Therefore an application called v4l2ucp has been used to set these parameters manually to ensure short and fixed exposure time. An exposure time shorter than 30^{-1} s is necessary to be able to run the loop at 30 Hz.

Figure 4.2: Example of a marker

The marker detection is a process of several steps. Below is a brief description of the steps taken from image to identification of marker, marker position and rotation. This describes the inner workings of ARToolKit, these image processing algorithms haven't been developed in this thesis work.

4.4.1 Grayscaling and thresholding

First the image is preprocessed which consist of mixing it to greyscale as well as thresholding it. In a color image, each pixel is usually represented by three 8-bit channels for the colors red, green and blue which combined can represent $(2^8)^3 = 16777216$ colors. Conversion to a grayscale pixel $\hat{p}_i$ for a pixel $p_i = \{R_i, G_i, B_i\}$ is done by constructing a linear combination of these channel values $\hat{p}_i = \alpha_1 R_i + \alpha_2 G_i + \alpha_3 B_i$. The simplest choice of weights is $\alpha_k = 3^{-1}$, for all k. The thresholding operation is a function that takes a pixel value and returns 255 if the value is above a certain threshold or 0 if it is below the threshold. It can be described by the Heaviside step function for a threshold γ as $p_{new} = 255 \cdot \theta(p - \gamma)$, the result is a binary image which is necessary to find the markers, which are binary as well. Basically, it makes bright pixels even whiter and dark pixels even darker.

4.4.2 Segmentation and contour and edge detection

Furthermore, an algorithm called segmentation detects large areas of connected pixels with the same color and labels these sets of pixels. Different features are extracted from each such set, such as area, center of mass and color. A contour detection is executed to find edges in the image, that is, points where the difference in intensity between two nearby pixels is large. In this process corners are also identified.

4.4.3 Line fitting

Using the previously extracted data about corners and edges, lines can be fitted over the whole image. Lines that are connected to form any projective transformation of a square are identified and the n best candidates will be regarded as markers.

4.4.4 Marker identification

The patterns within the squares are normalized (transformed back to squares). The feature set inside the pattern is compared to that of a set of template patterns, to find the best matches. The markers position t_x, t_y, t_z have now been identified along with its orientation described by a rotation matrix R as can be viewed in Figure 4.3. The Figure does not show the z-axis, which is orthogonal to x and y and directed into the picture.

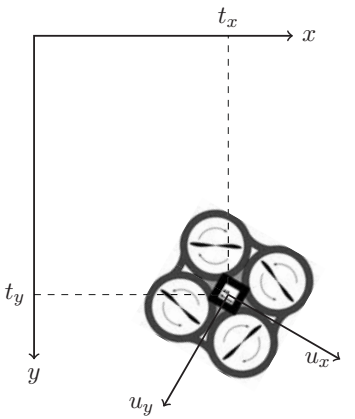

Figure 4.3: The camera coordinate system

4.5 The visual feedback

At each iteration the video processing outputs whether a marker is detected, and its pose and position in the coordinate system given by the camera (See Figure 4.3), i.e.,

$$\begin{bmatrix} R \mid & t \end{bmatrix} = \left[\begin{array}{ccc|c} r_{1,1} & r_{1,2} & r_{1,3} & t_x \\ r_{2,1} & r_{2,2} & r_{2,3} & t_y \\ r_{3,1} & r_{3,2} & r_{3,3} & t_z \end{array}\right].$$

R is below expressed in terms of angles around the camera base vectors. Here ψ represents the rotation of the marker around the cameras z-axis and ϕ and θ describe the rotation of the marker in the x and y respectively. Also, introducing the shorthand notation $s_x = \sin x$ and $c_x = \cos x$, for all $x \in \theta, \psi, \phi$, the rotation matrix becomes:

$$R = \begin{bmatrix} c_\theta c_\psi & -c_\phi s_\psi + s_\phi s_\theta c_\psi & s_\phi s_\psi + c_\phi s_\theta c_\psi \\ c_\theta s_\psi & c_\phi c_\psi + s_\phi s_\theta s_\psi & -s_\phi c_\psi + c_\phi s_\theta s_\psi \\ -s_\theta & s_\phi c_\theta & c_\phi c_\theta \end{bmatrix}$$

Of special interest is the *yaw*-angle ψ since by measuring *yaw*, and by using the position information t_x and t_y, a combination of pitch and roll angles can be calculated which drives the quadrocopter to points in x and y. The yaw angle ψ is retrieved by calculating $\psi = \text{atan2}(r_{2,1}, r_{1,1})$. A new rotation matrix for yaw which is not influenced by roll and pitch is constructed, namely

$$R_{2\times 2} = \begin{bmatrix} \cos\psi & -\sin\psi \\ \sin\psi & \cos\psi \end{bmatrix} \tag{4.1}$$

Since the camera is always steady with respect to the world, the coordinate system given by the camera is chosen as the main coordinate system. For a general setpoint in the xy-plane

$$\mathbf{x_{ref}} = \begin{bmatrix} x_{ref} \\ y_{ref} \end{bmatrix}$$

the error in this coordinate system is

$$\hat{e} = \mathbf{x_{ref}} - \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x_{ref} - x \\ y_{ref} - y \end{bmatrix}.$$

To get this error in the drone's coordinate system, it must be rotated and get a change of sign in the y-axis.

$$e = R_{2\times 2} \cdot \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \cdot \begin{bmatrix} x_{ref} - x \\ y_{ref} - y \end{bmatrix}$$

Additionally, a modified error has also been defined which allows scaling of the influence from the setpoint. This error is

$$e_\gamma = R_{2\times 2} \cdot \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \cdot \begin{bmatrix} \gamma x_{ref} - x \\ \gamma y_{ref} - y \end{bmatrix}, 0 \leq \gamma \leq 1 \tag{4.2}$$

and it has been used in situations where it is undesired to differentiate the change of setpoint. Since the marker has a predefined size, its distance from the camera can also be calculated. This is the output z from ARToolKit and it can be used to control altitude without any transformation. The altitude error is $e_{\gamma z} = -(\gamma_z z_{ref} - z)$ where the change of sign makes sense since positive acceleration in u_z decreases z.

$$e_{\gamma x} = \hat{x} \cdot e_\gamma$$

$$e_{\gamma y} = \hat{y} \cdot e_\gamma$$

$$e_{\gamma z} = z - \gamma_z z_{ref}$$

4.6 Modeling

In Figure 4.5, the local coordinate system of quadrocopter can be seen. Not present in the illustration is the altitude direction, z, which is directed as the negative normal to earth, e.g., x, y and z are orthogonal. By the use of trigonometry, roll and pitch angles are converted to lateral acceleration through the equations:

$$\ddot{y} = 9.82 \cdot \tan\phi \approx 9.82 \cdot \phi \text{ where } \phi \in [-\frac{\pi}{6}, \frac{\pi}{6}]$$

$$\ddot{x} = 9.82 \cdot \tan\theta \approx 9.82 \cdot \theta \text{ where } \theta \in [-\frac{\pi}{6}, \frac{\pi}{6}]$$

The angles θ and ϕ are close to linear in this region, see Figure 4.4, therefore the approximation $\tan\phi = \phi$ has been made.

Since we have chosen $\frac{\pi}{6}$ to be the maximum tilt angle; *euler_angle_max* (which was earlier mentioned in Section 3.1), we have $\phi = \frac{\pi}{6} u_y$ and $\theta = \frac{\pi}{6} u_x$. The relation becomes,

$$\ddot{x} \approx 9.82 \cdot \frac{\pi}{6} u_x \text{ where } u_x \in [-1, 1] \tag{4.3}$$

$$\ddot{y} \approx 9.82 \cdot \frac{\pi}{6} u_y \text{ where } u_y \in [-1, 1] \tag{4.4}$$

Control over yaw has also been implemented due to the onboard controller's tendency to drift, this controller will be referred to as $u_{yaw}(t)$. The maximum yaw rate has been defined to $100°s^{-1} \approx 1.74$rad/s.

$$\dot{\psi} = 1.74 \cdot u_{yaw} \text{ where } u_{yaw} \in [-1, 1] \tag{4.5}$$

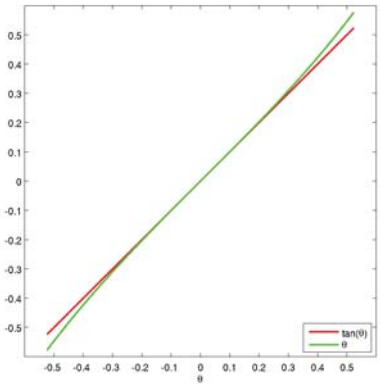

Figure 4.4: $\tan\theta$ and θ compared in the range $[-\frac{\pi}{6}, \frac{\pi}{6}]$

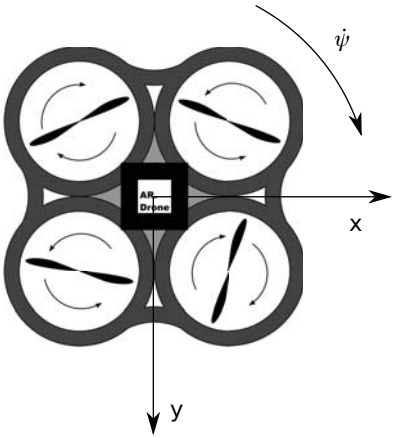

Figure 4.5: Positive directions for the quadrocopter

4.7 Control algorithms

In this thesis, a couple of control schemes have been developed to enable position control in 3D space. For a simpler implementation, it has been assumed that the three directions $(\hat{x}, \hat{y}, \hat{z})$ are decoupled. This means for example that the position, in the x-direction does not change when we apply a control signal u_y in the y-direction. Tests have shown that in reality x and y really are decoupled, however, x with z and y with z are not. This is revealed when a thrust is applied in y, the quadrocopter at first descends, but gradually regains its desired altitude due to the built in controller. A controller has been implemented which closes the loop on the altitude of the quadrocopter based on the onboard ultrasound measurement, using altitude velocity as control signal. Altitude control has further been developed to instead of using ultrasound, base the measurement on the visual feedback. Furthermore for x/y, a vision based position controller has been implemented, and also a trajectory controller in the x/y-plane, with velocity control along a predefined path. All of these controllers work regardless of the quadrocopters yaw rotation, however, control of yaw has also been implemented for completeness.

4.7.1 Altitude control with ultrasound

The altitude is controlled by setting an altitude velocity. This is an intuitive way for a human to control but for autonomous flight, the possibility to track setpoints in space is a necessity. This controller has been implemented as warm up exercise, therefore it is not improved any further. The main goal of this thesis is the vision based control which naturally includes vision based altitude control to which this controller is a good complement. Let the distance to the ground, as measured by ultrasound, be z and the desired distance be z_{ref}. The error is then defined as $e_z(t) = z_{ref}(t) - z(t)$. A proportional controller can then be designed as

$$u_z(t) = K_z \cdot e_z(t),$$

which means that (for $K > 0$), $u_z(t) > 0$ when you are below the setpoint and $u_z(t) < 0$ when you are above the setpoint. The quadrocopter expects the control signal to be constrained by $-1 \leq u_z(t) \leq 1$, therefore it must also be saturated within those boundaries. The resulting control scheme is

$$u_z(t) = sat(K_z \cdot e_z(t)).$$

4.7.2 Positioning in the xy-plane

For positioning in the xy-plane, two PD-controllers have been implemented. The PD-controller is an extension of the previously mentioned proportional

controller. The main difference is the introduction of a derivative part, i.e., the estimated derivative also affects the control signal. The general structure for PD control is $u(t) = K_p e(t) + K_d \dot{e}(t)$. Since the setpoint often changes with large increments, it is not suitable to differentiate it. Therefore the second error has been modified according to equation 4.2 to

$$u(t) = K_p e(t) + K_d \dot{e}_\gamma(t) \tag{4.6}$$

where γ is a design variable which controls the influence of the setpoint. Transforming to frequency domain and replacing $K_d s$ with a filter $\frac{NK_d s}{N+K_d s}$ and using the Tustin approximation of the derivative, $s = \frac{2}{h}\frac{z-1}{z+1}$, gives a new expression for the equation. That is $U(z) = K_p E(z) + \frac{NK_d \frac{2}{h}(z-1)}{N(z+1)+K_d \frac{2}{h}(z-1)} E_\gamma(z)$ where N is a design parameter which suppresses high frequency noise which otherwise appear when differentiating a discrete signal. For low frequencies the expression $\lim_{s\to 0} \frac{NK_d s}{N+K_d s} = 0$ and for high frequency it is limited to N; $\lim_{s\to\infty} \frac{NK_d s}{N+K_d s} = N$. After some simplification this is

$$\begin{aligned}(N(z+1) + K_d\frac{2}{h}(z-1))U(z) = K_p(N(z+1) \\ + K_d\frac{2}{h}(z-1))E(z) + (NK_d\frac{2}{h}(z-1))E_\gamma(z)\end{aligned} \tag{4.7}$$

and setting

$$\begin{bmatrix} A = (N + K_d\frac{2}{h}) \\ B = (N - K_d\frac{2}{h}) \\ C = K_d N\frac{2}{h} \end{bmatrix}$$

gives a simple expression to calculate the control signal in each iteration:

$$u(n+1) = K_p e(n+1) + \frac{B}{A}(K_p e(n) - u(n)) + \frac{C}{A}(e_\gamma(n+1) - e_\gamma(n)) \tag{4.8}$$

4.7.3 Altitude control with Visual Feedback

An attempt to control the altitude by using the distance z from the marker to the camera, has also been made. In Section 4.5 the error

$$e_{\gamma z} = z - \gamma_z z_{ref}$$

has been defined, this is what we base this feedback loop on. As has been previously mentioned, manipulating u_z controls the altitude velocity of the quadrocopter. The control signal has been calculated also using the same PD-control structure as in Subsection 4.7.2. By using Equation 4.8, with these variables, the control signal is:

$$\begin{bmatrix} A = (N + K_{dz}\frac{2}{h}) \\ B = (N - K_{dz}\frac{2}{h}) \\ C = K_{dz}N\frac{2}{h} \end{bmatrix}$$

$$u(n+1) = K_{pz}e(n+1) + \frac{B}{A}(K_{pz}e(n) - u(n)) + \frac{C}{A}(e_{1z}(n+1) - e_{1z}(n))$$

4.7.4 Yaw control

The AR.Drone has an onboard control of yaw. It controls the rotational velocity about the yaw-axis, based on information received from the bottom camera and the inertial measurement unit. This control implementation has shown some tendencies to drift. Therefore an outer control loop has been implemented based on the information received from the external camera. The rotation about the yaw axis is as previously mentioned (See Equation 4.1) represented in the rotation matrix $R_{2\times 2} = \begin{bmatrix} r_{1,1} & r_{1,2} \\ r_{2,1} & r_{2,2} \end{bmatrix} = \begin{bmatrix} \cos\psi & -\sin\psi \\ \sin\psi & \cos\psi \end{bmatrix}$ where $-\pi < \psi < \pi$. The intuitive way to use ψ would be to calculate the yaw error as $e_{yaw} = \psi_{ref} - \psi$, however this would be troublesome if the desired angle ψ_{ref} is close to the discontinuity in $\psi = \begin{cases} \pi \\ -\pi \end{cases}$. To be able to choose a reference yaw angle arbitrarily this must be avoided. The approach chosen has instead been to define the error as a rotation of the coordinate system, such that the reference yaw angle is placed in $\psi = 0$. More precisely, rotate the rotation matrix with a rotation with the reference angle, that is, construct a matrix $\hat{R}_{2\times 2}$, such that,

$$\hat{R}_{2\times 2} = \begin{bmatrix} \cos\psi & -\sin\psi \\ \sin\psi & \cos\psi \end{bmatrix} \cdot \begin{bmatrix} \cos\psi_{ref} & -\sin\psi_{ref} \\ \sin\psi_{ref} & \cos\psi_{ref} \end{bmatrix},$$

then extract the error as $e = \text{atan2}(\hat{r}_{2,1}, -\hat{r}_{1,1})$. This way, the reference angle will always be half a rotation from the discontinuity, and the risk of crossing it for example due to a disturbance is minimal. A proportional controller will always drive the angle towards zero away from the discontinuity. The resulting control signal is $u_{yaw}(t) = K_p \cdot e(t)$.

4.7.5 Trajectory

For autonomous flight it is often desired to be able to set up a route to follow. The simplest approach to do this would be to define the route as a list of points to fly to in a certain order, changing setpoints when the error is smaller than a predefined distance, e.g., $\sqrt{x^2 + y^2 + z^2} < \epsilon$. A disadvantage with this approach is that it is difficult to predict how fast the vehicle will travel, following a straight line of points would with a small

derivative gain make it accelerate for a long time, whereas with a large derivative gain it would brake at each setpoint. It would be difficult to find a parameter that would give a consistent result along the trajectory. Instead a different [12] approach has been chosen. The trajectory has been defined as a set of points $p(n), n \in [0, N-1]$, along with a desired velocity between each of these points. One controller minimizes the orthogonal distance to the line segment between $p(n+1)$ and $p(n)$ (see Figure 4.6) and another controller keeps the reference velocity along the line segment. As soon as the quadrocopter crosses the dashed line in the Figure 4.6, n is increased with one and a new line segment is defined. For $p(n) = \begin{bmatrix} x_{ref}(n) & y_{ref}(n) \end{bmatrix}^T$ the direction along the current line segment is $\hat{\mathbf{t}} = \frac{(p(n+1)-p(n))}{|(p(n+1)-p(n))|}$, and its normal vector is $\hat{\mathbf{n}} = \begin{bmatrix} \hat{t}_y & -\hat{t}_x \end{bmatrix}^T$. With this convention, v_{ref} is a quantity along the current line segment $\mathbf{v_{ref}} = v_{ref} \cdot \hat{\mathbf{t}}$ and the velocity error is

$$e_{\gamma a} = (\gamma v_{ref} - \begin{bmatrix} \dot{x} \\ \dot{y} \end{bmatrix}^T \cdot \hat{\mathbf{t}}), \tag{4.9}$$

where the subscript a implies *"along the line"*. The velocity $\begin{bmatrix} \dot{x} & \dot{y} \end{bmatrix}$ is approximated with a backward difference operation. The crossing error, denoted e_c is the orthogonal distance from the AR.Drone to the line segment,

$$e_{\gamma c} = (\gamma p(n+1)^T - \begin{bmatrix} x \\ y \end{bmatrix}^T) \cdot \hat{\mathbf{n}} = (\gamma \begin{bmatrix} x_{ref}(n+1) \\ y_{ref}(n+1) \end{bmatrix}^T - \begin{bmatrix} x \\ y \end{bmatrix}^T) \cdot \hat{\mathbf{n}} \tag{4.10}$$

The errors defined in Equations 4.9 and 4.10 are put into Equation 4.6 to get the control signals:

$$u_a(t) = K_p e_a(t) + K_d \dot{e}_{\gamma a}(t) \tag{4.11}$$

and

$$u_c(t) = K_p e_c(t) + K_d \dot{e}_{\gamma c}(t) \tag{4.12}$$

where $e_c(t) := e_{1c}(t)$ and $e_a(t) := e_{1a}(t)$. Represented as vectors in the direction in which they act gives the $\mathbf{u}_c(t) = u_c(t) \cdot \hat{\mathbf{n}}$ and $\mathbf{u}_a(t) = u_a(t) \cdot \hat{\mathbf{t}}$. They are discretized the same way as in Equation 4.8 so the same controller can be reused. The only thing left to do is expressing the controller in the quadrocopters coordinate system, that is setting

$$\begin{bmatrix} u_x \\ u_y \end{bmatrix} = R_{2\times 2} \cdot \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \cdot \begin{bmatrix} \mathbf{u}_c(t) \\ \mathbf{u}_a(t) \end{bmatrix} \tag{4.13}$$

4.8 Parameter tuning

The controllers have been tuned by hand. To find a good combination of K_p and K_d for each controller, a step response has been evaluated with $K_d = 0$. K_p has been increased until a slight overshoot is achieved, then the overshoot has been decreased by increasing K_d to a reasonable amount.

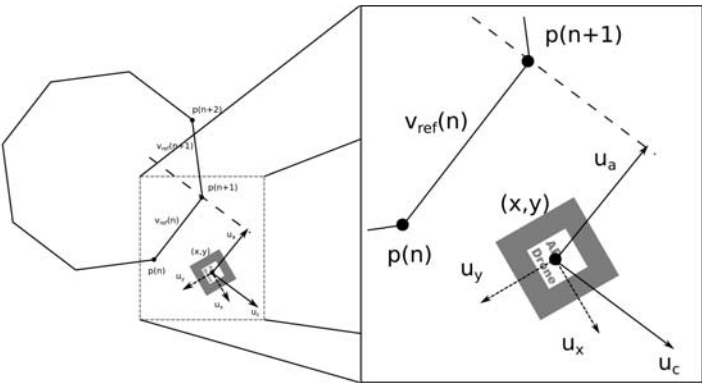

Figure 4.6: Illustration of an octagon shaped trajectory and its associated variables

4.9 Landing

Landing is handled by the internal control loops and can be executed at any time. When tracking a second marker, a landing command is automatically triggered if the euclidean distance between the marker and the quadrocopter is within a certain margin, i.e., if $(E_x^2 + E_y^2) < \alpha^2$, where alpha is the desired minimum distance.

Chapter 5

Method - Autonomous Tracking of Industrial Robots

5.1 Background

This part of the thesis covers the work conducted at ABB - Strategic R&D for Oil, Gas and Petrochemicals, Oslo. The development has been completely separate from the work at The Department of Automatic Control, Lund University. It however, shares common ground in which tools that have been used, namely the Parrot AR.Drone. They are also closely related with regard to the application, improving the autonomous capabilities of a Quadrocopter. The programming language *Microsoft C# 4.0* [5] has been used during the development, with help from the image processing library AForge.NET [2]. The quadrocopter has four degrees of freedom, out of which, three have been used to position and orient the quadrocopter. The goal is to always point toward the industrial robot and also be positioned orthogonally to the robot. The control signals used to achieve this is acceleration in the directions right/left (roll) and forward/backward (pitch) as well as rotational velocity (yaw rate). The fourth degree of freedom, which has not been used, is altitude which is set to a constant. The setup has been shown as a demonstration of future technologies on an Open House event at ABB, Oslo. The next section is a description of each step of the algorithm which ultimately leads to autonomous tracking of an industrial robot.

5.2 Image Processing

5.2.1 Binarization of image

Industrial robots have a tradition of being one-colored and most frequently orange, but often it is a color which stands out in the working environment. Therefore, robot identification through the means of color recognition

has been a natural choice. That is, the pixels most similar to the color of the robot have been identified. Each pixel in each frame have been converted from RGB color space to HSL, a color representation much more suitable for discriminating between different colors. In RGB a color is represented as a blend of the colors red, green and blue, and mathematically for $C_{R,G,B} \in [0,1]$ as $\{C_R, C_G, C_B\}$. For example red is $\{1,0,0\}$ and $\{1,1,0\}$ would be yellow whereas in HSL, each color is characterised by a Hue value, amount of Saturation and Luminance which essentially means how bright a color is. The hue channel is measured in degrees or radians since it is circular, saturation is a value from 0 to 1 and so is also luminance. Similar to how we represent the RGB-channels, we can represent a color in HSL as $\{C_H, C_S, C_L\}$ for $C_H \in [0,360]$ and $C_{S,L} \in [0,1]$. Let us assume we have measured the mean value of the color of the object we regard as the robot to be $\{I_H, I_S, I_L\} = \{E(R_H), E(R_S), E(R_L)\}$, then the color of any pixel in the image can be compared to this color, by calculating the smallest difference in each channel. For the hue channel the arithmetic mean value does not make sense since hue is a circular quantity, instead the mean value is for a set of angles α_i calculated as $E(\alpha_i) = \arctan \frac{E(\sin(\alpha_i))}{E(\cos(\alpha_i))}$. The smallest difference in each channel is calculated as,

$$\{\varepsilon_H, \varepsilon_S, \varepsilon_L\} = \{|\min(I_H - p_H, p_H - I_H + 360)|, |I_S - p_S|, |I_L - p_L|\},$$

where I_H, I_S, I_L is the chosen ideal color. If the sum of the errors $\varepsilon = \frac{\varepsilon_H}{360} + \varepsilon_S + \varepsilon_L$, is small enough the candidate color is considered to be close enough to the robot's color. However, in this particular case, we are not interested in differentiating dark from bright orange that much since the robot does have dark orange parts. It therefore makes sense to put a weight on each channel error to be able to adjust the error according to our problem. So an appropriately normed weighting factor w is introduced for each channel, and we also introduce the euclidean error and the resulting error is instead $\varepsilon = \sqrt{(\frac{w_H \varepsilon_H}{360})^2 + (w_S \varepsilon_S)^2 + (w_L \varepsilon_L)^2}$. We also decide a threshold T such that if $\varepsilon < T$ the pixel is regarded as the same color as the robot. The resulting image contains only one channel pixels with 1bit information in each. Care has to be taken to adjust the parameters for the ideal color, the weights and the threshold, since the quadrocopter adjusts hue, brightness and contrast adaptively onboard, so there is no way to find parameters that always work. Besides, the robot is not perfectly orange colored, there are tools and wires in the way which makes the identification process even more difficult.

5.2.2 Blob detection

The binary image is retrieved and a routine called BlobCounter from the image processing library AForge.NET is used to find the largest area of

connected equal-colored pixels. This pixel cloud is initially assumed to be the robot. For the identification to work well, it is important that nothing else but the robot is orange in the field of view of the camera, therefore precaution has been taken to visually separate similar colors.

5.2.3 Convex hull of the blob

A convex hull of a blob is defined as the minimal convex set containing the blob. An algorithm from AForge.NET called HullFinder is used to extract the convex hull of the blob, which is represented as a set of corner points on the convex hull, see Figure 5.3. On the very first received frame, by looking at the center of gravity of the blob we can decide on which side of the robot the quadrocopter is positioned. On a robot with its tooltip to the right in the image, the center of gravity will deviate slightly to the left. We will assume for the sake of describing the algorithm, that this is the position the quadrocopter is at initially. By using the information in the convex hull, we can get a good understanding of the robot's pose. We have defined and extracted three feature points, namely the position of the base plate p_1, the position of the tooltip p_3 and the other end of the robot arm p_2, see Figure 5.2. These points have been chosen for several reasons, namely p_3 since it is the point of operation, p_2 has been chosen because it combined with p_3 gives a description of the length of the arm which is a good measure of distance to the robot. Feature point p_1 has been added because it together with p_2 and p_3 gives information about the angle of joint 3. In the next section we will describe how these points have been identified.

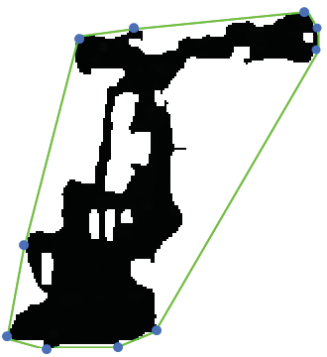

Figure 5.1: Binary image and its convex hull

5.2.4 Extraction of robot feature points

The extraction of feature points is performed as follows. For the baseplate, we look at the convex hull, and choose the point in the convex hull closest

to the lower left corner. This point, becomes the center point of a circle with radius $r = 0.25\sqrt{A}$ where A is the area of the blob in number of pixels. The radius is chosen as the square root of the blobs area, to make sure the algorithm gives consistent result regardless of how large the robot is in the image (*i.e.*, how close to the robot the quadrocopter is). The feature point p_1 is defined as the mean value of all hull points inside the circle. To find feature point p_3, we use the same method, however, with a different center for the circle. In this case we look only in the upper half of the image and search for the point furthest to the right in this half. This point is, as in the case for feature point p_1, used to create a mean value of all points within a $0.25\sqrt{A}$ radius.

Now we have for most poses and orientations, the baseplate and tooltip of the robot. Feature point p_2, or the back of the arm, is decided by once again looking at all remaining convex hull points and find the point p_2, which best fulfills the relation $\frac{\|\overrightarrow{p_1p_2}\|}{\|\overrightarrow{p_3p_2}\|} = Q$ which means the point p_2 which gives the smallest error $\epsilon = \|\overrightarrow{p_1p_2}\| - \|\overrightarrow{p_3p_2}\| \cdot Q$. The point p_2 must also be on the correct side of the line $\overrightarrow{p_1p_3}$, *i.e.*, it must be above and to the left of the line for a robot with its arm pointing to the right as in Figure 5.2, otherwise it will not be considered. A suitable value for Q, for an ABB industrial robot model IRB4400, is 1.35. Even if the center points are extracted perfectly each time, the real quotient will differ a lot from 1.35, mainly since this model assumes the point p_2 is exactly in joint 3 on the robot, but due to the parallellogram construction of the IRB 4400, it is not. Still, with this simplification, the model works remarkably well and only rarely gives erroneous point estimations. The point calculated is, just as with feature point p_1 and p_3, used to create a mean value of the hull points within a radius of $r = 0.25\sqrt{A}$.

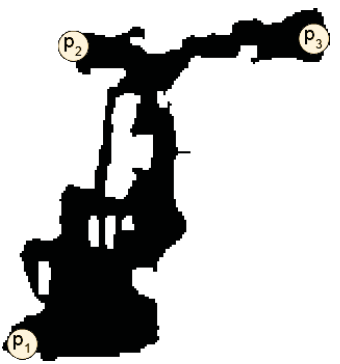

Figure 5.2: Binary image with feature points

5.2.5 Isolation of robot arm

Since we now have extracted feature points p_2 and p_3, we can be certain that the arm is somewhere between these points. The arm is a part of the blob which should have more or less the same shape regardless of the robot's pose. It can of course be rotated in different angles, however, there is always a viewing angle at which it will look more or less the same. By measuring the center of gravity of this arm divided by its full length, we get an understanding of how we are positioned relative to the arm. For example, if we look at the arm slightly from left, the left part of the arm will be larger than if we look at it from the right. There are other parts on the robot which also have a fixed shape regardless of pose, but the arm has the advantage that it is easily distinguished since we already have access to its end points. Another advantage is that it is the longest part of the robot which means that we get good resolution on this measure. Keep in mind though that it is the length of its projection on the x-axis which is important, so an arm in horizontal orientation has much more resolution than one in vertical orientation. The rest of this section will refer to the x-component of the center of gravity even when this is not stated explicitly.

The arm is isolated by measuring the angle with which the arm is rotated relative to the image's horizontal axis. This angle is calculated from the feature points p_2 and p_3. The blob is rotated and the result is an image of the robot with the arm completely horizontal. The next step is to crop the blob in a rectangle around the points p_2 and p_3. The width of the rectangle is the distance between points p_2 and p_3 with some margin, and the height of the rectangle should be adjusted to include only the arm and nothing more. A close estimate to the arm's height is $\frac{1}{2}\sqrt{A}$ since as mentioned before, including $\sqrt{A}$ makes the extraction of the arm scale-invariant. Once the image has been cropped, the arm is isolated. Doing a blob detection on this image returns the only available blob in it; the arm, which is used for further analysis. A method to calculate the center of gravity of this blob is already available in AForge.NET, however, it is truncated to integer pixel resolution which is not good enough. The center of gravity of the blob is also very sensitive to occlusion of parts of the arm. If wires and boxes are mounted on the arm with a color different from that of the robot, the center of gravity will deviate significantly. Therefore, rather than calculating the center of gravity of the blob, the center of gravity of the convex hull of the blob is calculated. The convex hull of the arm can be seen in Figure 5.3. The convex hull is less sensitive to occlusions and by writing the algorithm which extracts the center of gravity ourself, we can also get subpixel resolution. The center of gravity is calculated by dividing the convex hull into triangles, calculating the center of gravity of each triangle, and adding them according

to

$$C = \frac{1}{\sum m_k} \sum_{k=1}^{N} m_k p_k,$$

where m_k is the area of the kth triangle, and p_k is the center of gravity of each triangle.

Figure 5.3: Robot arm with convex hull

The center of gravity has been mapped from -0.5 to 0.5 which means that if the arm were to be completely uniform, a center of gravity of zero would mean that we are exactly orthogonal to the arm. However, it is not uniform, so an offset has been measured and implemented to make sure that being orthogonal to the robot also gives a center of gravity of zero.

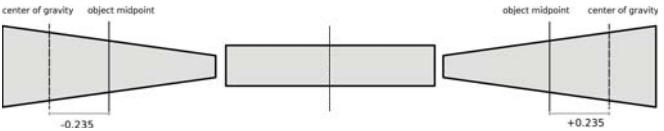

Figure 5.4: The convex hull with center of gravity, from three angles.

5.2.6 The Graphical User Interface

In parallell to the algorithm development, a graphical user interface (GUI) has been built for testing purposes but also as a possible interface for a remote operator. The GUI, which can be seen in Figure 5.5, presents buttons to activate the automatic control, debug information about the image processing and a zoomed view of the tooltip. Especially the zoomed view of the tooltip provides the remote operator with important information about the ongoing task.

5.3 Control Algorithms

5.3.1 Outlier detection

At times, the algorithm confuses the robot with other orange-like objects in the image. Sometimes the areas are connected and adds to the robot blob, and sometimes they are disjunct and a completely different object is chosen if it is larger. No matter how good detection you have, there is always the possibility of a misdetection, therefore it is good to have an

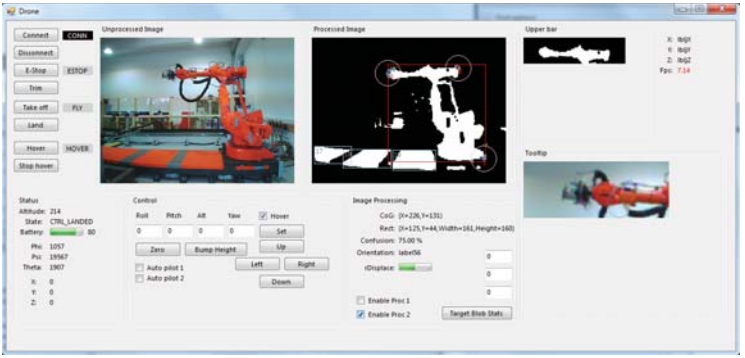

Figure 5.5: Graphical user interface

idea of when this has happened. The values from the last iteration is kept for some variables, namely center of gravity in x and y of the robot blob, the height and width of the blob, and the quotient between height and width referred to as "portraitness". By comparing present values with the former and comparing them each to a constant of maximum allowed change, erroneous robot identification can be spotted. Or more simply put, if any of the variables changes too much during an iteration, the result from that iteration is ignored. If this happens, no control signals are sent and the algorithm just waits until the error is within tolerance again. If this does not happen for 100 iterations the mission is aborted and the quadrocopter is instructed to land.

5.3.2 Coupling between control and data

As have been previously mentioned, control of the quadrocopter is executed by setting reference angles for roll and pitch and a reference angular velocity for yaw and reference altitude velocity. The control objective is to always aim the front camera at the robot, stay orthogonal to the plane spanned by the arms from joint 2 to joint 4 and to stay at a set distance to the robot while keeping a constant altitude. The yaw velocity is used to aim at the robot, or more precisely at a point defined by half-way from the tooltip to the robot center in the x-coordinate. The distance to the robot is linear to the length of the robot arm, which is used to calculate the pitch control signal. This is preferred over using the robot's area as a measure of distance in the image, since its area varies with its position and orientation. Finding an error upon which to control roll angle has been most difficult, the best one suited is the one calculated in Subsection 5.2.5, namely the center of gravity of the convex hull of the robot arm. Without this control, the quadrocopter will drift along a circle around the robot, and eventually reach an angle to

the robot where the image analysis does not make sense since the arm is indistinguishable from the rest of the robot.

5.3.3 Controllers

For all control action, PI-controllers or P-controllers have been used since they are quick to implement, intuitive to tune and give good enough performance. The object of this project has mainly been finding good enough measures to base control on. A derivative part on the roll controller has been considered and would be the next natural step to improve performance.

Chapter 6

Results

This chapter covers the results from the work conceived at Department of Automatic Control, LTH.

6.1 Positioning in the *xy*-plane

Below is the result after doing a step response test in the x-direction. The reference values have been set by clicking in the video frame. It tracks the position fairly well, however there appears to be a small and negligible stationary error.

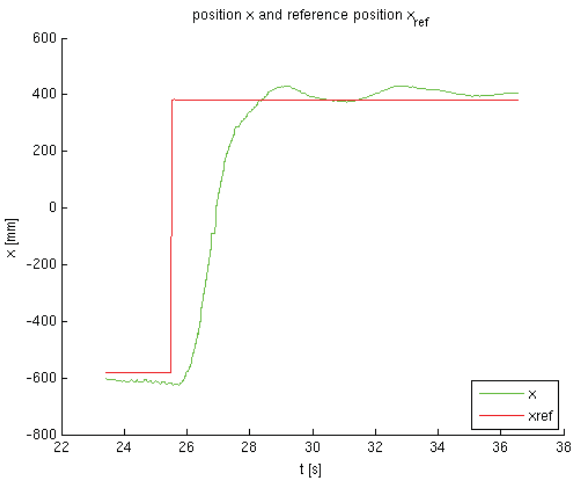

Figure 6.1: Step response in x-direction, when using PD position control.

6.2 Altitude control

6.2.1 Vision and ultrasound feedback compared to just ultrasound feedback

A box with the height 28 cm, see Figure 6.2, has been placed on one of the sides along a square shaped trajectory. The purpose of this test is to see how the altitude control of the quadrocopter reacts to this disturbance, and whether the vision based altitude control implemented in this thesis is able to reject it. The quadrocopter was programmed to follow the trajectory for several laps and the Figures 6.3 and 6.4 show 2-3 of these cycles, that are approximately 20 seconds of length. Figure 6.3 shows the effects when relying only on the onboard ultrasound altitude control. Figure 6.4 shows how the altitude changes when also using an outer loop of vision based altitude control. It is not obvious that the vision based control has any effect rejecting the disturbance, however it can be argued that with the vision based control, the altitude exhibits a more periodic and thus a more predictable behaviour.

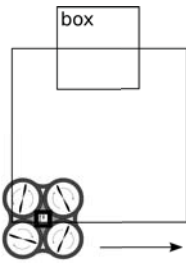

Figure 6.2: The position of the box in the square trajectory.

6.3 Tracking a square trajectory

Figure 6.5 is the results from tracking a square with the desired velocity of 300 mm/s along the trajectory. The direction of flight along the trajectory is clockwise. Figure 6.6 shows the velocity of the quadrocopter in the x component. The reference signals were not available at the time of logging, therefore lines have been plotted at 0.3 and -0.3 to indicate where the maximum and minimum of the reference should be.

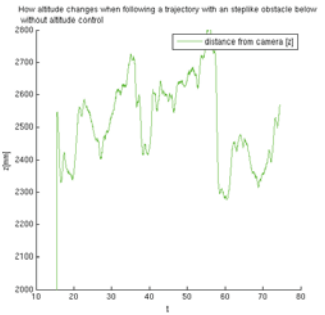

Figure 6.3: Flying over an obstacle without vision based altitude control

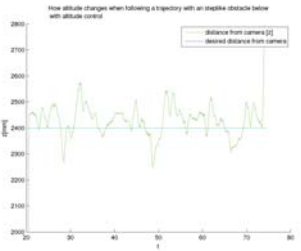

Figure 6.4: Flying over an obstacle using vision based altitude control

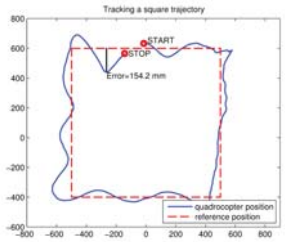

Figure 6.5: Position [mm] when tracking a square trajectory

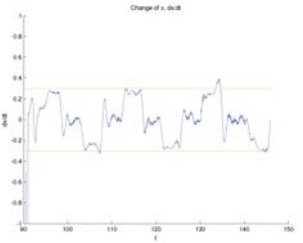

Figure 6.6: The measurement of $\dot{x}$ [mm/s] over time [s] when following a trajectory with the shape of a square

6.4 Tracking a circular trajectory

Figure 6.7 shows the performance when following a circular trajectory. The circle is on purpose simplified as a polygon with 15 edges to allow for easier tracking. Due to the nature of the controller, it will not perform well with high resolution trajectories since it has no knowledge about how the trajectory will change, it only acts on the current error. Therefore, when tracking a circle, it never settles on the trajectory, it has a constant error. Adding a feed-forward controller based on the quadrocopter inertia, would likely improve this. Another approach might be to have a integral part in the controller, making sure it isn't reset on every new line segment would help when tracking a circle.

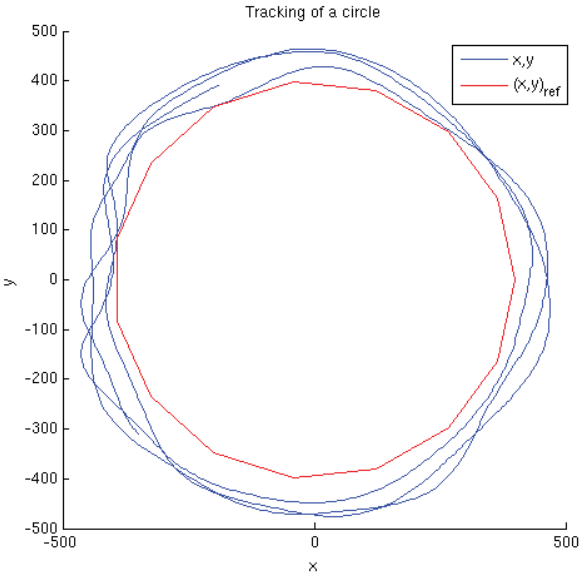

Figure 6.7: A circular trajectory with radius 500 mm, discretized into 15 line segments. The x and y axes are measured in [mm].

Chapter 7

Conclusion

One of the main drawbacks with this quadrocopter is its tendency to react to uneveness on the ground due to its ultrasound based design. Especially when flying over an object like a box or table, it will immediately ascend towards the ceiling to a new point, approximately the height of the obstacle. When the object is no longer below the quadrocopter, it will descend the same distance, at best. Most of the time it does not ascend as much as it descends, and when flying over an object over and over again, the altitude will change significantly from its initial state. In Figure 6.3 this effect can be seen. It is questionable if the implemented vision based altitude controller rejects the disturbance of the box. It is possible that the controller had a too slow response time, and maybe the effect of it would be more visible if it had more time to settle in. The position and speed controller in xy, however, show satisfactory results. The quadrocopter follows its path with a maximum deviation of 154.2 mm when tracking one of the test paths.

7.1 Future Work

The work in this thesis merely scratches the surface on vision based Quadrocopter control. A possible improvement on the control design is introducing a LQG-controller. Another natural improvement on the control structure, is feedforward control and iterative learning control. On the vision part, an interesting problem would be to have several cameras covering a larger space, with the camera views partly overlapping. It would also be an challenging task to develop an obstacle avoiding path planner.

Chapter 8

Appendix

8.1 Source Code

This Section contains the source code for the work carried out at The Department of Automatic Control, LTH. It is split into the parts executed by Computer A which is an implementation of ARToolKit and Computer B which is compiled within the AR.Drone SDK version 1.5, (See Figure 4.1).

8.1.1 Computer A

```
#ifdef _WIN32
#include <windows.h>
#endif
#include <stdio.h>
#include <stdlib.h>
#ifndef __APPLE__
#include <GL/gl.h>
#include <GL/glut.h>
#else
#include <OpenGL/gl.h>
#include <GLUT/glut.h>
#endif

#include <AR/gsub.h>
#include <AR/video.h>
#include <AR/param.h>
#include <AR/ar.h>
#include <time.h>
#include <stdio.h>
#include <string.h>
#include "goserver.h"
```

```
#include "netapi.h"
#define DEFAULTSERVERIP "130.235.83.248"
#define PACKETLENGTH 120
#define PRINT 0
#define YMAX 480
#define XMAX 640
#define LOOPCONTROL 1
//
// Camera configuration.
//

char      *vconf = "v4l2src device=/dev/video0 !
   capsfilter caps=video/x-raw-rgb,width=640,height
   =480,bpp=24,fps=30 ! ffmpegcolorspace ! identity
   name=artoolkit ! fakesink"; //use for logitech
   webcam
char      *vconf1 = "v4l2src device=/dev/video1 !
   capsfilter caps=video/x-raw-rgb,width=640,height
   =480,bpp=24,fps=30 ! ffmpegcolorspace ! identity
   name=artoolkit ! fakesink"; //use for logitech
   webcam
char      *vconf2 = "v4l2src device=/dev/video2 !
   capsfilter caps=video/x-raw-rgb,width=640,height
   =480,bpp=24,fps=30 ! ffmpegcolorspace ! identity
   name=artoolkit ! fakesink"; //use for logitech
   webcam


int              xsize, ysize;
int              thresh = 120;
int              count = 0;
int              mode =1;
int   ucount = 0;

char            *cparam_name    = "Data/camera_para.
   dat";       //use for logitech webcam
ARParam          cparam;

char            *patt_name1      = "Data/patt.hiro";
char            *patt_name2      = "Data/patt.kanji";
int   patt_id1;
int              patt_id2;
int     sock;
FILE*     handle;
```

```
double          patt_width1     = 100.0;  //width of
    pattern in mm
double    patt_width2 = 100.0;  //width of pattern
   in mm
double          patt_center[2] = {0.0, 0.0};
double          patt_center2[2] = {0.0, -400.0};
double          patt_trans1[3][4];
double          patt_trans1past[3][4];
double          patt_trans2[3][4];
double          patt_trans2past[3][4];
double    xref;
double    yref;
double          zref;
struct timeval pst;
struct timeval curr;
int dt;
int frequency; //desired frequency<=1000, 30hz
int waitatleast;
int servernconnected;
int visible1;
int visible2;
////////////////////
int mouseDown = 0;
float xmouse = 0.0f;
float ymouse = 0.0f;
////////////////////
static void   init(char*);
static void   cleanup(void);
static void   keyEvent( unsigned char key, int x,
   int y);
static void   mouseEvent( int button, int state, int
    x, int y);
static void   mainLoop(void);
static void   draw( double M[3][4]);
static void   drawred(double M[3][4]);
static void   dot(double M[3][4]);

static int    gettimedifference(struct timeval,
   struct timeval);
static void   printtrans(double M[3][4]);


int main(int argc, char **argv)
{
```

```
  glutInit(&argc, argv);
  frequency =30;
  waitatleast=(int) 1000000/frequency; //
     microseconds

  gettimeofday(&pst, NULL);
  gettimeofday(&curr,NULL);

  if (argv[1]==NULL){
    init(DEFAULTSERVERIP);
  } else {
    init(argv[1]);
  }
  char filename[20];
  sprintf(filename,"logfile%d.txt", pst.tv_usec);
  handle=fopen(filename,"wb");

  fprintf(handle,"timestamp; patt1x; patt1y; patt1z;
      patt2x; patt2y; patt2z; rot00; rot01; rot10;
     rot11; patt1visible; patt2visible \n");
  arVideoCapStart();
  argMainLoop( mouseEvent, keyEvent, mainLoop );
  return (0);
}

static int gettimedifference(struct timeval first,
    struct timeval last){
  int diff = last.tv_sec*1000000 - first.tv_sec
     *1000000 + last.tv_usec-first.tv_usec;
  return diff;
}

static void mouseEvent(int button, int state, int x,
    int y){
if (button == GLUT_LEFT_BUTTON && state == GLUT_DOWN
   )
{
mouseDown = 1;

xmouse = x - XMAX/2;
ymouse = y - YMAX/2;
}
else
mouseDown = 0;
```

```
}

static void    keyEvent( unsigned char key, int x,
   int y)
{
  /* quit if the ESC key is pressed */
  if( key == 0x1b ) {
    printf("*** %f (frame/sec)\n", (double)count/
       arUtilTimer());
    cleanup();
    exit(0);
  }
  if( key == 'c' ) {
    printf("*** %f (frame/sec)\n", (double)count/
       arUtilTimer());
    count = 0;

    mode = 1 - mode;
    if( mode ) printf("Continuous mode: Using
       arGetTransMatCont.\n");
    else       printf("One shot mode: Using
       arGetTransMat.\n");
  }
}

/* main loop */
static void mainLoop(void)
{
  static int contF1 = 0;
  static int contF2 = 0;
  ARUint8         *dataPtr;
  ARMarkerInfo    *marker_info;
  int             marker_num;
  int             j, k;

  //write message and send (loop)
  gettimeofday(&pst,NULL);
  /* grab a vide frame */
  if( (dataPtr = (ARUint8 *)arVideoGetImage()) ==
     NULL ) {
    arUtilSleep(2);
    printf ("NULL\n");
    return;
  }
```

```
  if( count == 0 ) arUtilTimerReset();
  count++;
  if (mouseDown==1){
    //printf("x=%f, y=%f \n",xmouse,ymouse);
    xref=1.25*3*xmouse*patt_trans1[2][3]/2880; //
       measured scaling
    yref=1.25*3*ymouse*patt_trans1[2][3]/2880;
    zref=patt_trans1[2][3];
    printf("xref=%f, yref=%f, zref=%f, xmouse=%f,
       ymouse=%f \n",xref,yref,zref ,xmouse,ymouse);
  }

  argDrawMode2D();
  //argDrawMode     = AR_DRAW_BY_GL_DRAW_PIXELS; //
     image unaffected by camera parameters
  argDispImage( dataPtr, 0,0 );
  /* detect the markers in the video frame */
  if( arDetectMarker(dataPtr, thresh, &marker_info,
     &marker_num) < 0 ) {
    cleanup();
    exit(0);
  }
  arVideoCapNext(); //probably not necessary

if(PRINT){
  for( j = 0; j < marker_num;j++ ) {
    printf("marker confidence is %f and id is :%d\n"
       ,marker_info[j].cf,marker_info[j].id);
  }
}

  /* check for object visibility */
  k = -1;
  for( j = 0; j < marker_num; j++ ) {
    if( patt_id1 == marker_info[j].id ) {
      if( k == -1 ) k = j;
      else if( marker_info[k].cf < marker_info[j].cf
          ) k = j;
    }
  }
  if( k == -1 ) {
    visible1=0;
    contF1=0;
    drawred(patt_trans1);
```

```
  } else {
    visible1=1;
    if (mode == 0|| contF1 == 0){
      arGetTransMat(&marker_info[k], patt_center,
         patt_width1, patt_trans1);
      //printf("arGetTransMat\n");
    }
    else {
      arGetTransMatCont(&marker_info[k], patt_trans1
         , patt_center, patt_width1, patt_trans1);
      //printf("arGetTransMatCont\n");
    }
  }


  k = -1;
  for( j = 0; j < marker_num; j++ ) {
    if( patt_id2 == marker_info[j].id ) {
      if( k == -1 ) k = j;
      else if( marker_info[k].cf < marker_info[j].cf
          ) k = j;
    }
  }
  if( k == -1 ) {
    visible2=0;
    contF2=0;
    draw(patt_trans2);
  }else {
    visible2=1;
    if (mode == 0|| contF2 == 0){
      arGetTransMat(&marker_info[k], patt_center2,
         patt_width2, patt_trans2);
      //printf("arGetTransMat\n");
    }
    else {
      arGetTransMatCont(&marker_info[k], patt_trans2
         , patt_center2, patt_width2, patt_trans2);
      //printf("arGetTransMatCont\n");
    }
  }
  if (PRINT){
  printf("%d markers | visible 1:%d 2:%d | ",
     marker_num,visible1,visible2);
}
```

```
  patt_trans1[0][3],
patt_trans1[1][3],
patt_trans1[2][3],
patt_trans2[0][3],
patt_trans2[1][3],
patt_trans2[2][3],
patt_trans1[0][0],
patt_trans1[0][1],
patt_trans1[1][0],
patt_trans1[1][1],
contF1 = visible1;
contF2 = visible2;

/* send message over tcp */

char message[PACKETLENGTH]; //11+ 9 for each float

if(xref!=0 && yref!=0){ //send mouse coords
   instead of marker2
  sprintf( message,
      "%010d %010.4f %010.4f %010.4f %010.4f
         %010.4f %010.4f %07.4f %07.4f %07.4f
         %07.4f %1d %1d",
     (int) ((double) 1000*arUtilTimer()),
     patt_trans1[0][3],patt_trans1[1][3],
        patt_trans1[2][3],
     xref,yref,zref,
     patt_trans1[0][0],patt_trans1[0][1],
     patt_trans1[1][0],patt_trans1[1][1],
     visible1,visible2);//wa,wb,wc);


fprintf(handle,
  "%010d %010.4f %010.4f %010.4f %010.4f %010.4f
     %010.4f %07.4f %07.4f %07.4f %07.4f %1d %1d \
     n",
  (int) ((double) 1000*arUtilTimer()),
  patt_trans1[0][3],patt_trans1[1][3],patt_trans1
     [2][3],
    xref,yref,zref,
  patt_trans1[0][0],patt_trans1[0][1],
  patt_trans1[1][0],patt_trans1[1][1],
  visible1,visible2);
```

```
} else {
sprintf( message,
     "%010d %010.4f %010.4f %010.4f %010.4f %010.4
        f %010.4f %07.4f %07.4f %07.4f %07.4f %1d
        %1d",
     (int) ((double) 1000*arUtilTimer()),
     patt_trans1[0][3],patt_trans1[1][3],
        patt_trans1[2][3],
     patt_trans2[0][3],patt_trans2[1][3],
        patt_trans2[2][3],
     patt_trans1[0][0],patt_trans1[0][1],
     patt_trans1[1][0],patt_trans1[1][1],
     visible1,visible2);//wa,wb,wc);


fprintf(handle,
  "%010d %010.4f %010.4f %010.4f %010.4f %010.4f
     %010.4f %07.4f %07.4f %07.4f %07.4f %1d %1d \
     n",
  (int) ((double) 1000*arUtilTimer()),
  patt_trans1[0][3],patt_trans1[1][3],patt_trans1
     [2][3],
  patt_trans2[0][3],patt_trans2[1][3],patt_trans2
     [2][3],
  patt_trans1[0][0],patt_trans1[0][1],
  patt_trans1[1][0],patt_trans1[1][1],
  visible1,visible2);
}
int size = sizeof(message);
if(!servernconnected){
  int i = write(sock,message,sizeof(message));
  //int i = write(sock,o1,sizeof(o1));

}
//    printf("Sent %d bytes to tcp socket %d\n", i
   , sock);

if (visible1) draw(patt_trans1);
if (visible2) drawred(patt_trans2);
double ctr[3][4];

ctr[0][0]=1;
ctr[0][1]=0;
ctr[0][2]=0;
```

```
  ctr[0][3]=0;
  ctr[1][0]=0;
  ctr[1][1]=1;
  ctr[1][2]=0;
  ctr[1][3]=0;
  ctr[2][0]=0;
  ctr[2][1]=0;
  ctr[2][2]=1;
  ctr[2][3]=700;
  dot(ctr);

  printf("tx:%f ty:%f tz:%f \n",patt_trans1[0][3],
     patt_trans1[1][3],patt_trans1[2][3]);

  argSwapBuffers();
  gettimeofday(&curr,NULL);
  dt =gettimedifference(pst, curr); //returns an int
      representing time between pst and curr in ms
  if(LOOPCONTROL&&dt<waitatleast){ //waitatleast is
     an int which represents how long each cycle
     shall take at most
    usleep(waitatleast-dt);
  }
  //printf(" | elapsed time is %d , should be larger
      than %d ,slept %d| \n",dt,waitatleast,
     waitatleast-dt);
}

static void printtrans(double M[3][4]){
  /*printf("************Calib_pat
     ********************\n");
    printf("r11:%f r12:%f r13:%f \n",M[0][0],M
       [0][1],M[0][2]);
    printf("r21:%f r22:%f r23:%f \n",M[1][0],M
       [1][1],M[1][2]);
    printf("r31:%f r32:%f r33:%f \n",M[2][0],M
       [2][1],M[2][2]);
    printf
       ("****************************************\
       n");
  */
printf("tx:%f ty:%f tz:%f \n",M[0][3],M[1][3],M
   [2][3]);
```

```
}

static void init( char* serverip )
{
  ARParam  wparam;
  //start TCP socket
  sock = tcpsock();
  servernconnected=sockconn(sock,serverip,12345);
  if (servernconnected==0){
    printf("success\n");
  } else {
    printf("must start server!!\nmessages won't be
       sent\n");
  }
  /* open the video path */
  struct stat st;
  if(stat("/dev/video0",&st) == 0){
    printf(" /video0 is present\n");
    if( arVideoOpen( vconf ) < 0 ) {
      exit(0);
    }
  } else if(stat("/dev/video1",&st) == 0){
    printf(" /video1 is present\n");
    printf("trying video1 instead of video0\n");
    if(arVideoOpen(vconf1) < 0) {
      exit(0);
    }
  } else if(stat("/dev/video2",&st) == 0){
  printf(" /video1 is present\n");
    printf("trying video2 instead of video0\n");
    if(arVideoOpen(vconf2) < 0) {
      exit(0);
    }
  }

  /* find the size of the window */
  if( arVideoInqSize(&xsize, &ysize) < 0 ) exit(0);
  printf("Image size (x,y) = (%d,%d)\n", xsize,
     ysize);

  /* set the initial camera parameters */
  if( arParamLoad(cparam_name, 1, &wparam) < 0 ) {
    printf("Camera parameter load error !!\n");
    exit(0);
```

```
  }
  arParamChangeSize( &wparam, xsize, ysize, &cparam
     );
  arInitCparam( &cparam );
  printf("*** Camera Parameter ***\n");
  arParamDisp( &cparam );

  if( (patt_id1=arLoadPatt(patt_name1)) < 0 ) {
    printf("pattern1 load error !!\n");
    exit(0);
  }
  if ((patt_id2=arLoadPatt(patt_name2)) < 0){
      printf("pattern2 load error !!\n");
      exit(0);
  }

  /* open the graphics window */
  argInit( &cparam, 1.0, 0, 0, 0, 0 );
}

/* cleanup function called when program exits */
static void cleanup(void)
{
  arVideoCapStop();
  arVideoClose();
  argCleanup();
  //exit
  fclose(handle);
  close(sock);
}

static void drawred( double M[3][4]){
  double    gl_para[16];
  GLfloat   mat_ambient[]     = {0.0, 0.0, 0.0,
     1.0};
  GLfloat   mat_flash[]       = {0.0, 0.0, 0.0,
     1.0};
  GLfloat   mat_flash_shiny[] = {50.0};
  GLfloat   light_position[]  =
     {100.0,-200.0,200.0,0.0};
  GLfloat   ambi[]            = {0.1, 0.1, 0.1,
     0.1};
  GLfloat   lightZeroColor[]  = {0.6, 0.1, 0.3,
     0.1};
```

```
  argDrawMode3D();
  argDraw3dCamera( 0, 0 );
  glClearDepth( 1.0 );
  glClear(GL_DEPTH_BUFFER_BIT);
  glEnable(GL_DEPTH_TEST);
  glDepthFunc(GL_LEQUAL);

  /* load the camera transformation matrix */
  argConvGlpara(M, gl_para);
  glMatrixMode(GL_MODELVIEW);
  glLoadMatrixd( gl_para );

  glEnable(GL_LIGHTING);
  glEnable(GL_LIGHT0);
  glLightfv(GL_LIGHT0, GL_POSITION, light_position);
  glLightfv(GL_LIGHT0, GL_AMBIENT, ambi);
  glLightfv(GL_LIGHT0, GL_DIFFUSE, lightZeroColor);
  glMaterialfv(GL_FRONT, GL_SPECULAR, mat_flash);
  glMaterialfv(GL_FRONT, GL_SHININESS,
     mat_flash_shiny);
  glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient);
  glMatrixMode(GL_MODELVIEW);
  glTranslatef( 150.0, -150.0, 0.0 );
  //  glutSolidCube(50.0);
  glutSolidTorus(40,80, 10, 10);
  //glutSolidTetrahedron();
  glTranslatef( -300.0, 0.0, 0.0 );
  glutSolidTorus(40,80, 10, 10);
  glTranslatef( 0.0, 300.0, 0.0 );
  glutSolidTorus(40,80, 10, 10);
  glTranslatef( 300.0, 0.0, 0.0 );
  glutSolidTorus(40,80, 10, 10);
  //  glutSolidIcosahedron();
  //  glutSolidSphere(50.0,20,20);
  glDisable( GL_LIGHTING );

  glDisable( GL_DEPTH_TEST );
}

static void draw(double M[3][4])
{
  double    gl_para[16];
```

```
GLfloat   mat_ambient[]     = {0.0, 0.0, 1.0,
   1.0};
GLfloat   mat_flash[]       = {0.0, 0.0, 1.0,
   1.0};
GLfloat   mat_flash_shiny[] = {50.0};
GLfloat   light_position[]  =
   {100.0,-200.0,200.0,0.0};
GLfloat   ambi[]            = {0.1, 0.1, 0.1,
   0.1};
GLfloat   lightZeroColor[]  = {0.9, 0.9, 0.9,
   0.1};

argDrawMode3D();
argDraw3dCamera( 0, 0 );
glClearDepth( 1.0 );
glClear(GL_DEPTH_BUFFER_BIT);
glEnable(GL_DEPTH_TEST);
glDepthFunc(GL_LEQUAL);

/* load the camera transformation matrix */
argConvGlpara(M, gl_para);
glMatrixMode(GL_MODELVIEW);
glLoadMatrixd( gl_para );
glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);
glLightfv(GL_LIGHT0, GL_POSITION, light_position);
glLightfv(GL_LIGHT0, GL_AMBIENT, ambi);
glLightfv(GL_LIGHT0, GL_DIFFUSE, lightZeroColor);
glMaterialfv(GL_FRONT, GL_SPECULAR, mat_flash);
glMaterialfv(GL_FRONT, GL_SHININESS,
   mat_flash_shiny);
glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient);
glMatrixMode(GL_MODELVIEW);
glTranslatef( 150.0, -150.0, 0.0 );
//  glutSolidCube(50.0);
glutSolidTorus(40,80, 10, 10);
//glutSolidTetrahedron();
glTranslatef( -300.0, 0.0, 0.0 );
glutSolidTorus(40,80, 10, 10);
glTranslatef( 0.0, 300.0, 0.0 );
glutSolidTorus(40,80, 10, 10);
glTranslatef( 300.0, 0.0, 0.0 );
glutSolidTorus(40,80, 10, 10);
//  glutSolidIcosahedron();
```

```
  //  glutSolidSphere(50.0,20,20);

  glDisable( GL_LIGHTING );
  glDisable( GL_DEPTH_TEST );
}


static void dot(double M[3][4])
{
  double    gl_para[16];
  GLfloat   mat_ambient[]     = {0.0, 0.0, 1.0,
     1.0};
  GLfloat   mat_flash[]       = {0.0, 0.0, 1.0,
     1.0};
  GLfloat   mat_flash_shiny[] = {50.0};
  GLfloat   light_position[]  =
     {100.0,-200.0,200.0,0.0};
  GLfloat   ambi[]            = {0.1, 0.1, 0.1,
     0.1};
  GLfloat   lightZeroColor[]  = {0.9, 0.9, 0.9,
     0.1};

  argDrawMode2D();
  argDraw3dCamera( 0, 0 );
  glClearDepth( 1.0 );
  glClear(GL_DEPTH_BUFFER_BIT);
  glEnable(GL_DEPTH_TEST);
  glDepthFunc(GL_LEQUAL);

  /* load the camera transformation matrix */
  argConvGlpara(M, gl_para);
  glMatrixMode(GL_MODELVIEW);
  glLoadMatrixd( gl_para );
  glEnable(GL_LIGHTING);
  glEnable(GL_LIGHT0);
  glLightfv(GL_LIGHT0, GL_POSITION, light_position);
  glLightfv(GL_LIGHT0, GL_AMBIENT, ambi);
  glLightfv(GL_LIGHT0, GL_DIFFUSE, lightZeroColor);
  glMaterialfv(GL_FRONT, GL_SPECULAR, mat_flash);
  glMaterialfv(GL_FRONT, GL_SHININESS,
     mat_flash_shiny);
  glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient);
  glMatrixMode(GL_MODELVIEW);
  glutSolidCube(20.0);
```

```
  glDisable( GL_LIGHTING );
  glDisable( GL_DEPTH_TEST );
}
```

8.1.2 Computer B

```
#include <gtk/gtk.h>
#include <gtk/gtkcontainer.h>
#include <stdio.h>
#include <gdk-pixbuf/gdk-pixbuf.h>
#include <sys/time.h>
#include <VP_Os/vp_os_malloc.h>
#include <VP_Os/vp_os_print.h>
#include <ardrone_api.h>
#include <VP_Api/vp_api_thread_helper.h>
#include <string.h>
#include <VP_Os/vp_os_malloc.h>
#include <VP_Os/vp_os_print.h>
#include <config.h>
#include "common/common.h"
#include "ihm/ihm.h"
#include "ihm/ihm_vision.h"
#include "ihm/ihm_stages_o_gtk.h"
#include "ihm/view_drone_attitude.h"
#include "navdata_client/navdata_ihm.h"
#include "mocha.h"
#include <math.h>
#include "/home/martin/EXJOBB/OpenCV-2.1.0/include/
   opencv/cv.h"
#include "/home/martin/EXJOBB/OpenCV-2.1.0/include/
   opencv/highgui.h"
#include "goserver.h"
#include "netapi.h"
#include "/home/martin/EXJOBB/OpenCV-2.1.0/include/
   opencv/cxtypes.h"
#include "/home/martin/EXJOBB/OpenCV-2.1.0/include/
   opencv/cvcompat.h"

extern double ndata[10];
extern int32_t internalstate;
extern int ac_on;
double translx,transly,translz;
int Tcpts[MSIZE];
float cs_b,cs_c, cs_d, cs_e, e;
```

```
double Ex[MSIZE];
double Ey[MSIZE];
double Ez[MSIZE];
double Eyaw[MSIZE];
double Exbeta[MSIZE];
double Eybeta[MSIZE];
double Ezbeta[MSIZE];
double Eyawbeta[MSIZE];

double Tx[MSIZE];
double Ty[MSIZE];
double Tz[MSIZE];
double Spx[MSIZE];
double Spy[MSIZE];
double Spz[MSIZE];
double Spyaw[MSIZE];
double R00[MSIZE];
double R01[MSIZE];
double R10[MSIZE];
double R11[MSIZE];
double Ux[MSIZE];
double Uy[MSIZE];
double Uz[MSIZE];
double Uyaw[MSIZE];
int visible1;
int visible2;

 void savejpg(GdkPixbuf* pbuf, int j){
   int success;
   char buff[104];

   sprintf(buff, "/home/martin/EXJOBB/
      ARDrone_SDK_1_5_Version_20101004/Examples/
      Linux/Build/Release/record/mocha%05d.jpg", j);
   printf("%s\n",buff);
   success = gdk_pixbuf_save (pbuf, buff, "jpeg",
      NULL, "quality", "25", NULL);
   if(success){

     printf("image successfully saved");
   } else {
     printf("image not saved");
```

```
  }


}

void getmarkerposition(uint8_t *data, int32_t width
   , int32_t height, int32_t rowstride){

  typedef struct points{
    int x;
    int y;
  } points;
  points mrkrpos;
  mrkrpos.x=0;
  mrkrpos.y=0;

}

void testfunction(uint8_t *data, int32_t width,
   int32_t height, int32_t rowstride){

  extern double ndata[10];

  printf("%f \n",ndata[4]);
  int k;

  char imdata[height*width*3];

  for (k=0;k<height*width*3;k++){
    imdata[k]=(char) data[k];
  }

  IplImage* currframe = cvCreateImage(cvSize(width,
     height), IPL_DEPTH_8U, 3);
  IplImage* dst = cvCreateImage(cvSize(width,height
     ), IPL_DEPTH_8U, 3);

  currframe->imageData=imdata;
  cvCvtColor(currframe, dst, CV_BGR2RGB);

  struct CvPoint2D32f center;
  center.x=width/2;
  center.y=width/2;
  CvMat *tMat = cvCreateMat(2, 3, CV_32FC1);
```

```
    cv2DRotationMatrix(center, -ndata[4], 1.0, tMat);

    cvWarpAffine(dst, currframe, tMat,
       CV_INTER_LINEAR + CV_WARP_FILL_OUTLIERS,
       cvScalarAll(0));
    cvReleaseMat(&tMat);
    cvShowImage("mainWin", currframe);
    return;
  }

  void printstate(){

    extern int keyb_start;
    extern double altitude, altitude_ref;
    extern double translx,setpointx,transly,setpointy
       ,setpointz,translz;
    printf("
       |-----------------------------------------
-----------------------------|\n");
    printf("|start |AC |altitude |altitude_ref|
       translx|setpointx|transly|setpointy|\n");
    printf("|%1d     |%1d  | %8.3f| %8.3f   |%8.3f
       |%8.3f|%8.3f|%8.3f |\n",
!keyb_start,ac_on, altitude, altitude_ref, translx,
    setpointx,transly,setpointy);
    printf("
       |-----------------------------------------
-----------------------------|\n");
    printf("\033[4A");

  }




  int gettimedifference(struct timeval first, struct
     timeval last){

    int diff = last.tv_sec*1000000 - first.tv_sec
       *1000000 + last.tv_usec-first.tv_usec;
    return diff;
  }
```

```
float flimit(float p,float lo, float hi){
  if(p!=p){
    exit(0);
    printf("nan");
    return 0;//not a nan (not a not a number)
  }
  if (p<lo) return lo;
  else if (p>hi) return hi;
  else return p;
}

int land(){

  while(ac_on && (internalstate&32)==32){
    ardrone_tool_set_ui_pad_start(0);
  }
  return 1;
}

struct FloatPoint {
 float x;
 float y;
};

struct traj {
  float nextSPx;
  float nextSPy;
  float thisSPx;
  float thisSPy;
  float v;
  float length;
  float tangX;
  float tangY;
  float normX;
  float normY;
  float vref;
  float ea[MSIZE];
  float ec[MSIZE];
  int count;
};

float PDtust(float Kp,float Kd,int tcpdt, float N,
   float Upst, float Enow, float Epst ){
  float A=(2*Kd/tcpdt+N);
```

```
  float B=(2*Kd/tcpdt-N);
  float C=(Kp*N+2*Kd*Kp/tcpdt+2*N*Kd/tcpdt);
  float D=(Kp*N-2*Kd*Kp/tcpdt-2*N*Kd/tcpdt);
  float Unow = B*Upst/A+D*Epst/A+C*Enow/A;
  return Unow;
}
float PDtust2(float Kp,float Kd,int tcpdt, float N,
   float Upst, float Enow, float Epst, float
   Ebetanow,float Ebetapst){
  float A=tcpdt*N+2*Kd;
  float B=tcpdt*N-2*Kd;
  float C=2*N*Kd;
  float Unow=B/A*(Kp*Epst-Upst) +Kp*Enow + C/A*(
     Ebetanow-Ebetapst);
  return Unow;
}

int paintmap(struct traj t,float Tx,float Ty){
  float mapheight=600;
  float mapwidth=600;
  struct CvPoint p0;
  struct CvPoint pT;
  struct CvPoint pSt;
  struct CvPoint pSn;
  if( abs(t.nextSPx)>2*mapwidth || abs(t.nextSPy)
     >2*mapheight ) { //check if points are within
     boundaries
    return 0;
  }


  p0 = cvPoint(mapwidth/2, mapheight/2);
  pT = cvPoint(mapwidth/2+Tx/4, mapheight/2+Ty/4);
  pSn = cvPoint(mapwidth/2+t.nextSPx/4, mapheight
     /2+t.nextSPy/4);
  pSt  = cvPoint(mapwidth/2+t.thisSPx/4, mapheight
     /2+t.thisSPy/4);
  struct CvPoint pU;
  pU = cvPoint((int) mapwidth*(0.5+cs_b/2),(int)
     mapheight*(0.5+cs_c/2));

  IplImage* mycircle = cvCreateImage(cvSize(
     mapwidth,mapheight), IPL_DEPTH_8U, 3);
```

```
  cvLine(mycircle,p0,pU,CV_RGB( 34, 200, 150 )
     ,1,8,0);
  cvCircle(mycircle, pT,  3,CV_RGB( 90,  100, 200 )
      , 2, 8, 0);
  cvCircle(mycircle, pSn, 3,CV_RGB( 90,   50, 100 )
      , 2, 8, 0);
  cvCircle(mycircle, pSt, 3,CV_RGB( 150, 200, 100 )
      , 2, 8, 0);
  cvShowImage("helpWin", mycircle);
  cvReleaseImage(&mycircle);
}

struct FloatPoint p1;

static float line(int p){
  return (float) 20*((p%28)-14);
}


static struct FloatPoint circle(int p,float radius,
   int N, float shift){
   struct FloatPoint fp;
   fp.x = radius*cos(p*2*M_PI/N)+shift;//+30;
   fp.y = radius*sin(p*2*M_PI/N)+shift;//+30;
  return fp;
}
static float xsquare(int p){
  float points[4];
  points[0]=-500;
  points[1]=-500;
  points[2]=500;
  points[3]=500;
  return points[p%4];
}

static float ysquare(int p){
  float points[4];
  points[0]=-500;
  points[1]=500;
  points[2]=500;
  points[3]=-500;
  return points[p%4];
}
```

```
struct traj trajectory(struct traj t, float
   velocity_ref){

  if(SQUARE){
    t.nextSPx = xsquare(t.count+1);
    t.nextSPy = ysquare(t.count+1);
    t.thisSPx = xsquare(t.count);
    t.thisSPy = ysquare(t.count);
  }
  if(CIRCLE){
    t.nextSPx = circle(t.count+1,400,16,0).x; //(,
       radius,number of nodes,shift)
    t.nextSPy = circle(t.count+1,400,16,0).y;
    t.thisSPx = circle(t.count,400,16,0).x;
    t.thisSPy = circle(t.count,400,16,0).y;
  }
  if(LINE){
    t.nextSPx=line(t.count+1);
    t.nextSPy=line(t.count+1);
    t.thisSPx=line(t.count);
    t.thisSPy=line(t.count);
  }

  t.tangX = (t.nextSPx - t.thisSPx);  //400*cos(idx
     *2*pi/300)+200;
  t.tangY = (t.nextSPy - t.thisSPy);  //400*sin(idx
     *2*pi/300)+600;
  t.length = sqrt(pow(t.tangX,2)+pow(t.tangY,2));
  t.tangX = t.tangX/t.length;//vector in tangent of
      trajectory
  t.tangY = t.tangY/t.length;//normed
  t.normX = -t.tangY;//rotated 90 clockwise
  t.normY = t.tangX;
  t.vref = velocity_ref/1000;//0.1mm/ms=0.1m/s
  printf("count is %d\n",t.count);

  return t;
}

void controlmain()
{
  int frequency=30; //desired frequency<=1000,
     probably 50-60hz
  int waitatleast=1000000/frequency; //microseconds
```

```
double alt_sat=1;

struct timeval pst;
struct timeval curr;
int dt;
cs_b=0; //left/right (x)
cs_c=0; //forward/backward (y)
cs_d=0; //altitude (z)
cs_e=0; //rotation
extern double altitude;
extern double setpointx;
extern double setpointy;
extern double setpointz;
extern double altitude_ref;
extern double r00;
extern double r01;
extern double r10;
extern double r11;
int iter=MSIZE;

int received;
int tcpdt=30;
int glseq=0; //global sequence number used in
   control commands

//all past data stored in arrays of size MSIZE
   (5), values are
//stored and accessed in a circular manner

//gains
const float Kpx=0.14;
const float Kpy=0.14;
const float Kpz=0.4;
const float Kpyaw = 0.35;
//
const float Kdx=120.0;
const float Kdy=120.0;
const float Kdz=6.0;
//
const float Kpa=120.0;
const float Kpc=0.25;
//
```

```
const float Kda=0;
const float Kdc=190;
//
const float Nc=3;
const float Na=4.5;
//
float Axy=1;
float Bxy=1;
float Cxy=1;
float Dxy=1;
float Az=1;
float Bz=1;
float Cz=1;
float Dz=1;
float beta=BETA; //zero for no derivative of
   setpoint

const float Nxy=6;
const float Nz=7;
int idx[MSIZE];
register int i;
gettimeofday(&pst, NULL);
gettimeofday(&curr,NULL);

for(i=0;i<MSIZE;i++){
  Ux[i]=0;
  Uy[i]=0;
  Uz[i]=0;
}

struct traj t1;
t1.count=0;
t1=trajectory(t1,VREF);
for(;!gs_quit;){
  received = get_tcp_dta(&coms_cons[coms_cons_idx
     [0]],iter);
  iter+=received;
  printf("%d samples received\n",received);
  //set indexes idx[1] corresponds to -1, and so
     on
  for (i=0;i<MSIZE;i++){
    idx[i]=(iter-i)%MSIZE;
  }
  tcpdt=Tcpts[idx[0]]-Tcpts[idx[1]];
```

```
if (tcpdt==0){
  tcpdt=30;
}
printf("time elapsed since last sample %d\n",
   tcpdt);
altitude=ndata[3];
switch(1){
  case TRAJECTORY:

    //t1=trajectory(t1);//updates setpoint if
       necessary
    if(t1.tangX*(Tx[idx[0]] - t1.thisSPx) + t1.
       tangY*(Ty[idx[0]] - t1.thisSPy) > t1.
       length){ //scalar product
      t1.count=t1.count+1;
    }
    t1=trajectory(t1,VREF);


    //velocity calculated with central
       difference
    t1.v  = (t1.tangX*(Tx[idx[0]]-Tx[idx[2]]) +
        t1.tangY*(Ty[idx[0]]-Ty[idx[2]]))/2/
       tcpdt; //the projection of velocity on t
       -direction.

    t1.ea[idx[0]] = t1.vref-t1.v; //error along
       , velocity error
    t1.ec[idx[0]] = t1.normX*(t1.nextSPx-Tx[idx
       [0]]) + t1.normY*(t1.nextSPy-Ty[idx[0]])
       ; //error cross, position error


    //here Ex isn't error but control signal in
        the path coordinate system
    Ex[idx[0]]= PDtust(Kpc,Kdc,tcpdt,Nc,Ex[idx
       [1]], t1.ec[idx[0]], t1.ec[idx[1]] );//
       filtered control signal
    Ey[idx[0]]= PDtust(Kpa,Kda,tcpdt,Na,Ey[idx
       [1]], t1.ea[idx[0]], t1.ea[idx[1]] );


    if(0&&CIRCLE) Ex[idx[0]]+=6000*atan(pow(t1.
       vref,2)/(9.82*400))/3.14;
```

```
    //rotate to camera system
    float v1 = (Ex[idx[0]]*t1.normX+Ey[idx[0]]*
       t1.tangX);
    float v2 = (Ex[idx[0]]*t1.normY+Ey[idx[0]]*
       t1.tangY);
    //rotate and scale to quadrotor system
    Ux[idx[0]] = (float) (R00[idx[0]]*v1+R01[
       idx[0]]*v2)/1000.0f; //webcam control.
    Uy[idx[0]] = (float) -(R10[idx[0]]*v1+R11[
       idx[0]]*v2)/1000.0f; // the coordinate
       systems upside down with respect to
       eachother, therefore a minus sign is
       necessary

    /*
     Ez[idx[0]]= (float) (Tz[idx[0]]-setpointz)
        /800;
     Ez[idx[1]]= (float) (Tz[idx[1]]-setpointz)
        /800;
     Uz[idx[0]]=PDtust(Kpz,Kdz,tcpdt,Nz, Uz[idx
        [1]],Ez[idx[0]],Ez[idx[1]]);
     */
    break;
  default:

    if (FOLLOW){

      //write control signals now
      Ex[idx[0]]=  (float)  (R00[idx[0]]*(Spx
         [0]-Tx[ idx[0] ])+R01[idx[0]]*(Spy[0]-
         Ty[ idx[0] ]))/1000; //webcam control.
      Ey[idx[0]]=  (float) -(R10[idx[0]]*(Spx
         [0]-Tx[ idx[0] ])+R11[idx[0]]*(Spy[0]-
         Ty[ idx[0] ]))/1000; // the coordinate
          systems upside down with respect to
         eachother, therefore a minus sign is
         necessary
      Ex[idx[1]]=  (float)  (R00[idx[1]]*(Spx
         [1]-Tx[ idx[1] ])+R01[idx[1]]*(Spy[1]-
         Ty[ idx[1] ]))/1000;
      Ey[idx[1]]=  (float) -(R10[idx[1]]*(Spx
         [1]-Tx[ idx[1] ])+R11[idx[1]]*(Spy[1]-
         Ty[ idx[1] ]))/1000;
```

```
  Exbeta[idx[0]]=  (float)  (R00[idx[0]]*(
     beta*Spx[0]-Tx[ idx[0] ])+R01[idx
     [0]]*(beta*Spy[0]-Ty[ idx[0] ]))/1000;
      //webcam control.
  Eybeta[idx[0]]=  (float) -(R10[idx[0]]*(
     beta*Spx[0]-Tx[ idx[0] ])+R11[idx
     [0]]*(beta*Spy[0]-Ty[ idx[0] ]))/1000;
      // the coordinate systems upside down
      with respect to eachother, therefore
     a minus sign is necessary
  Exbeta[idx[1]]=  (float)  (R00[idx[1]]*(
     beta*Spx[1]-Tx[ idx[1] ])+R01[idx
     [1]]*(beta*Spy[1]-Ty[ idx[1] ]))/1000;
  Eybeta[idx[1]]=  (float) -(R10[idx[1]]*(
     beta*Spx[1]-Tx[ idx[1] ])+R11[idx
     [1]]*(beta*Spy[1]-Ty[ idx[1] ]))/1000;


} else {

  Ex[idx[0]]=  (float)  (R00[idx[0]]*(Spx
     [0]-Tx[ idx[0] ])+R01[idx[0]]*(Spy[0]-
     Ty[ idx[0] ]))/1000; //webcam control.
  Ey[idx[0]]=  (float) -(R10[idx[0]]*(Spx
     [0]-Tx[ idx[0] ])+R11[idx[0]]*(Spy[0]-
     Ty[ idx[0] ]))/1000; // the coordinate
      systems upside down with respect to
     eachother, therefore a minus sign is
     necessary
  Ex[idx[1]]=  (float)  (R00[idx[1]]*(Spx
     [0]-Tx[ idx[1] ])+R01[idx[1]]*(Spy[0]-
     Ty[ idx[1] ]))/1000;
  Ey[idx[1]]=  (float) -(R10[idx[1]]*(Spx
     [0]-Tx[ idx[1] ])+R11[idx[1]]*(Spy[0]-
     Ty[ idx[1] ]))/1000;
  printf("does not follow\n");
}

if(!TUSTINS){

  Axy=Kdx/(tcpdt*Nxy+Kdx);
  Bxy=Kpx+Kdx*Nxy/(tcpdt*Nxy+Kdx);
```

```
        Cxy=-(Kpx*Kdx+Kdx*Nxy)/(tcpdt*Nxy+Kdx);
        //---------------------------//
        Az=Kdz/(tcpdt*Nz+Kdz);
        Bz=Kpz+Kdz*Nz/(tcpdt*Nz+Kdz);
        Cz=-(Kpz*Kdz+Kdz*Nz)/(tcpdt*Nz+Kdz);
        //---------------------------//
        Ux[idx[0]]=Axy*Ux[idx[1]]+Bxy*Ex[idx[0]]+
           Cxy*Ex[idx[1]];
        Uy[idx[0]]=Axy*Uy[idx[1]]+Bxy*Ey[idx[0]]+
           Cxy*Ey[idx[1]];
        Uz[idx[0]]=Az*Uz[idx[1]]+Bz*Ez[idx[0]]+Cz
           *Ez[idx[1]];
        printf("not tustins\n");
        //---------------------------//
      } else {
        if(FOLLOW){
          Ux[idx[0]]=PDtust2(Kpx,130,tcpdt,3,Ux[
             idx[1]],Ex[idx[0]],Ex[idx[1]],
             Exbeta[idx[0]],Exbeta[idx[1]]);
          Uy[idx[0]]=PDtust2(Kpy,130,tcpdt,3,Uy[
             idx[1]],Ey[idx[0]],Ey[idx[1]],
             Eybeta[idx[0]],Eybeta[idx[1]]);
          //printf("tust2x %f tust2y %f\n", Ux[
             idx[0]],Uy[idx[0]]);
        } else {
          Ux[idx[0]]=PDtust(Kpx,Kdx,tcpdt,Nxy,Ux[
             idx[1]],Ex[idx[0]],Ex[idx[1]]);
          Uy[idx[0]]=PDtust(Kpy,Kdy,tcpdt,Nxy,Uy[
             idx[1]],Ey[idx[0]],Ey[idx[1]]);
          //printf("tustx %f tusty %f\n", Ux[idx
             [0]],Uy[idx[0]]);
        }


        printf("tustins\n");

      }
      break;
}
//no tracking in z, keeps setpoint defined by "
   E" regardless of control setting

Ez[idx[0]]= (float) (Tz[idx[0]]-2400)/800;
Ez[idx[1]]= (float) (Tz[idx[1]]-2400)/800;
```

```
      Uz[idx[0]]=PDtust(Kpz,Kdz,tcpdt,Nz, Uz[idx[1]],
         Ez[idx[0]],Ez[idx[1]]);
      printf("ucontrol is %f \n", Uz[idx[0]]);

     //yaw contrl
//     Eyaw[0] = (float) (atan2(R10[idx[0]],-R00[idx
    [0]]));

     //yaw control ref
     float yawref = M_PI;
     printf("yawreference is %f",yawref);
     Eyaw[0]=atan2(R10[idx[0]]*cos(yawref)+R11[idx
        [0]]*sin(yawref),-R00[idx[0]]*cos(yawref)-R01
        [idx[0]]*sin(yawref));
     printf("angle is: %f\n",atan2(R10[idx[0]],-R00[
        idx[0]]));


      cs_b=Ux[idx[0]];
      cs_c=Uy[idx[0]];
      cs_d=Uz[idx[0]];
      cs_e = Kpyaw * Eyaw[0];
      //  printf("cs_d is %f\n",cs_d);
      cs_b = flimit(cs_b,-0.3,0.3);
      cs_c = flimit(cs_c,-0.3,0.3);
      cs_d = flimit(cs_d,-0.2,0.2);
      cs_e = flimit(cs_e,-1,1);//yaw
      if(ac_on){ //&& ((FOLLOW&&visible1&&visible2)
         ||(!FOLLOW && visible1 ) )){
        //control signal being sent
        printf("signal is: %f, %f, %f, %f \n", cs_b,
           cs_c,cs_d,cs_e);
        ardrone_at_set_progress_cmd(visible1,cs_b,
           cs_c,ZCONTROL*cs_d,cs_e);
        //checks how close to landing spot we are and
            lands if close enough
        if(LAND&&(pow(Ex[idx[0]],2)+pow(Ey[idx[0]],2)
           <0.02) ){
          land();
        }
      }
      if(0){
```

```
  printstate();
}
if (1){//print all translations, rotations and
   control signals in memory

  for(i=MSIZE-1;i>=0;i--){
    printf("vars are: %d|%d|%d |%8f %8f %8f |
       %8f %8f %8f |%8f %8f | %8f  | %8f %8f |
       %d %d| %d| %d\n",
            iter,tcpdt,
            Tcpts[idx[i]],
            Ex[idx[i]],Ey[idx[i]],Ez[idx[i]],
            Tx[idx[i]], Ty[idx[i]], Tz[idx[i]],
            t1.nextSPx, t1.nextSPy,
            t1.v,
            cs_b,cs_c,
            visible1,visible2,
            t1.count,ac_on);  //verifies that we
                have access to webcam values.
               those are received in goserver.c
  }
  printf("description: it,tcpdt,time,Ex,Ey,Ez,
     Tx,Ty,Tz,nextspx,nextspy,v,cs_b,cs_c,
     visible1,visible2,count,ac_on\n");

}
if(MAP){

  paintmap(t1,Tx[idx[0]],Ty[idx[0]]);
}
gettimeofday(&curr,NULL);
dt = gettimedifference(pst, curr); //returns an
    int representing time between pst and curr
   in ms
if(dt<waitatleast){ //waitatleast is an int
   which represents how long each cycle shall
   take at most
  usleep(waitatleast-dt);
  // printf("s\n");
}
//    printf(" | elapsed time is %d , should be
    larger than %d ,slept %d| \n",dt,
   waitatleast,waitatleast-dt);
```

```
    gettimeofday(&pst,NULL);

  }
}

DEFINE_THREAD_ROUTINE(mocha,data) //macro says
   thread_mocha()
{
  printf("\n   mocha initialisation\n\n");
  cvNamedWindow("helpWin", CV_WINDOW_AUTOSIZE);
  cvMoveWindow("helpWin", 100, 100);
  init_server();
  printf("\n   Server initialized\n\n");
  connect_server();
  printf("\n   Connected to client\n\n");
  printf("retrieve navdat\n");
  //pcfg = (mobile_config_t *)data;
  /* MOCHA main loop */
  printf("\n Control loop starts\n");
  controlmain();
  close(gs_ctrl_skt);
  close(gs_coms_skt);
  printf("sockets going down\n");
  return 0;
}
```

Bibliography

[1] Video4linux2. http://en.wikipedia.org/wiki/Video4Linux, January 2012.

[2] Aforge.net. http://www.aforgenet.com, April 2015.

[3] Ar.drone. http://ardrone.parrot.com/, April 2015.

[4] Artoolkit. http://www.hitl.washington.edu/artoolkit/, April 2015.

[5] C#. http://msdn.microsoft.com/en-us/vstudio/hh388566, April 2015.

[6] Grasp lab, university of pennsylvania. https://www.grasp.upenn.edu/, April 2015.

[7] The mathworks matlab. http://www.mathworks.com/products/matlab/, April 2015.

[8] Optical flow. http://en.wikipedia.org/wiki/Optical_flow, April 2015.

[9] Parrot. http://www.parrot.com/, April 2015.

[10] Vicon motion capture. http://www.vicon.com/, April 2015.

[11] Nick Dijkshoorn. Simultaneous mapping and localization with the ar.drone. http://www.nickd.nl/dl/thesis_Nick_Dijkshoorn.pdf, July 2012.

[12] Gabriel M. Hoffmann, Steven L. Waslander, and Claire J. Tomlin. Quadrotor helicopter trajectory tracking control. *In 2008 AIAA Guidance, Navigation and Control Conference and Exhibit, Honolulu, Hawaii, USA*, August 2008.

[13] Daniel Mellinger and Vijay Kumar. Minimum snap trajectory generation and control for quadrotors. May 2011.

[14] Mark W. Mueller, Sergei Lupashin, and Raffaello D'Andrea. Quadrocopter ball juggling. http://www.idsc.ethz.ch/people/staff/mueller-m/QuadrocopterBallJuggling.pdf, January 2013.

Lund University
Department of Automatic Control
Box 118
SE-221 00 Lund Sweden

Document name
MASTER´S THESIS

Date of issue
May 2015

Document Number
ISRN LUTFD2/TFRT--5965--SE

Author(s)
Martin Ericsson

Supervisor
Vladimeros Vladimerou, Dept. of Automatic Control, Lund University, Sweden
Magnus Linderoth, Dept. of Automatic Control, Lund University, Sweden
Anders Robertsson, Dept. of Automatic Control, Lund University, Sweden
Karl-Erik Årzén, Dept. of Automatic Control, Lund University, Sweden (examiner)

Sponsoring organization

Title and subtitle
Visual Tracking and Control of a Quadrocopter

Abstract

The main goal of this master thesis project was to take a manually controlled consumer quality quadrocopter and build the foundations which allows it to be autonomously controlled. This was achieved by introducing a exterior frame of reference through the use of a webcam coupled with image analysis algorithms. The position and rotation of the quadrocopter was identified, and several control structures were implemented and tested, which allowed the quadrocopter to be commanded to move to positions in space. Both the onboard ultrasound sensor, and an altitude estimation through image analysis were used to control the altitude of the quadrocopter. Position control in x, y and orientation (yaw rotation) completely relied on data extracted and analysed from the video stream. Control of velocity along a predefined trajectory was also successfully implemented, which enables future development of an obstacle avoiding path planner. Lastly, the master thesis also covers work carried out at ABB's Strategic R&D department for oil, gas and petrochemicals i Oslo, Norway. Here the focus was on using a quadrocopter to track and follow the motion of an industrial robot by analysing the video stream of the onboard camera.

Keywords

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title
0280-5316

ISBN

Language
English

Number of pages
1-69

Recipient's notes

Security classification

http://www.control.lth.se/publications/