Technical University of Denmark

Søren Krogh Andersen (s123369)

Self-contained Autonomous Quadrotor

Bachelor project, June 2015

DTU Electrical Engineering
Department of Electrical Engineering

SØREN KROGH ANDERSEN (S123369)

Self-contained Autonomous Quadrotor

Bachelor's Project, June 2015

Supervisor:

Søren Hansen, Professor at the Electrical Engineering Department of DTU

DTU - Technical University of Denmark, Kgs. Lyngby - 2015

Self-contained Autonomous Quadrotor

This report was prepared by:
Søren Krogh Andersen (s123369)

Adviser:
Søren Hansen, Professor at the Electrical Engineering Department of DTU

DTU Electrical Engineering
Automation and Control
Technical University of Denmark
Elektrovej, Building 326
2800 Kgs. Lyngby
Denmark
Tel: +45 4525 3576

studieadministration@elektro.dtu.dk

Project period: February 2015- June 2015

ECTS: 20

Education: BSc

Field: Electrical Engineering

Class: Open

Remarks: This report is submitted as partial fulfillment of the requirements for graduation in the above education at the Technical University of Denmark.

Table of Contents

List of Figures

Abstract

The aim of the project is to construct a small, autonomous quadrotor capable of navigation using only on-board sensors and processors. The quadrotor is constructed with indoor flight i mind and the controller focuses on slow and precise movements.

To achieve autonomy visual odometry based on features detected on the ground is implemented. The odometry is used to estimate the state of the quadrotor using an *Extended Kalman Filter*. The state is propagated using an inertial measurement unit (*IMU*). Features on the ground are tracked between images using optical flow and used to estimate egomotion, which in turn is used to update the state estimate.

A controller for manual control is also implemented, this enables an operator to take over control in case of a fault in the localization.

The final quadrotor is capable of hovering autonomously inside a length $45\,\mathrm{cm} \times$ width $45\,\mathrm{cm}$ area, without leaving. Can be commanded to fly in predefined patterns. Can be controlled manually with minimal effort due to attitude and altitude controller.

Preface

This is the final report for the bachelor's project conducted as the culmination of the Bachelor in Electrotechnology study at The Technical University of Denmark. The project period is 2nd February 2015 to 26th June 2015. The project was done in collaboration with the *Automation and Control* group at *DTU* under the supervision of Søren Hansen, Professor at the Electrical Engineering Department of DTU.

Thanks to *Prevas A/S* for letting me borrow their 3D printer and laboratory for construction of the hardware.

Søren Krogh Andersen (s123369)

CHAPTER 1

Introduction

1.1 Project Introduction

In recent years multirotors have become very popular, both in the industry and in hobby communities. The big market for multirotors and their mechanical simplicity has also driven the price very low, which in turn makes them even more attractive. While autonomous multirotors that fly outdoors with the aid of a *GPS* are not uncommon, this project aims to design and construct an autonomous quadrotor for indoor flight. This requires an alternative to *GPS* for localization, as *GPS* is not very accurate indoors.

Although the design specifications are made with the *DTU Robocup* [1] in mind, such a drone could be use for many other things. A similar drone could be used to transport urgent samples or medicine on a hospital, or for inventory keeping in warehouses.

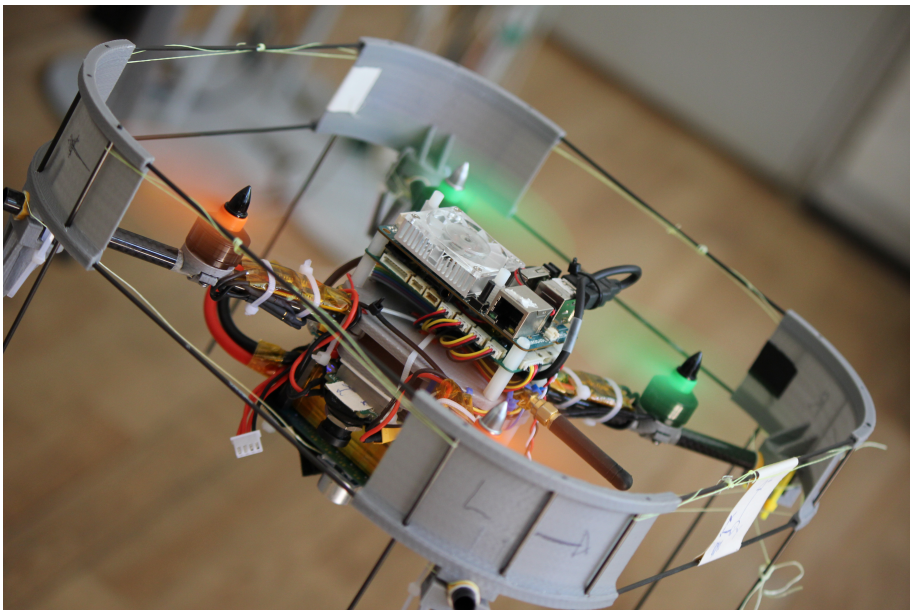

Figure 1.1: The final quadrotor in mid-flight.

To be able to compete in *DTU Robocup* the quadrotor has to be small and self-contained. The focus on the controller is slow and precise movements. To be capable of passing through the "gates" a small $30\,\mathrm{cm} \times 35\,\mathrm{cm}$) quadrotor is constructed. the quadrotor has a caged around the rotor to protect spectators.

[1] DTU Robocup `http://www.robocup.dtu.dk/` is an annual competition held at the Technical University of Denmark, where autonomous robots on an indoors course must overcome different obstacles and drive through "gates" to earn points.

1.2 Previous Work

Prior the the start of the project the mechanical and electrical hardware was constructed and an attitude controller was devised. This was done to evaluate the possibility of constructing an autonomous quadrotor, as this project deals with.

1.3 Problem Statement

The aim of the project is to construct a quadrotor drone capable of autonomous flight. The quadrotor must do so without aid from any sensors or computational aids not located on the quadrotor.

As the term "autonomy" is quite broad, the scope is limited to stabilizing the drone when hovering and flying at slow speeds.

The requirements of the problem are aimed towards a platform that can participate in the annual competition *DTU Robocup*[2] in which autonomous robots have to follow a line and pass through "goals" marked by an upright steel frame of width $45\,\mathrm{cm} \times$ height $50\,\mathrm{cm}$.

Requrements

- **Self contained**. All processing power and sensors must be on-board.
- **Autonomous**. The drone shall be capable of flight without human iteration.
- **Safe**
  - Must have a **guard around propellers** to protect the environment form the spinning blades.
  - **Manual controller** that can be switched on and take over in case of failure in the autonomous system.
  - Must be equipped with a **dead man's switch**. This makes sure that termination of operation is always a possibility.

Sucess Criteria

- Hover inside an area of length $45\,\mathrm{cm} \times$ width $45\,\mathrm{cm}$ for at least $30\,\mathrm{s}$ without leaving the area.
- Fly through an opening of width $45\,\mathrm{cm} \times$ height $50\,\mathrm{cm}$ without hitting the sides.
- Manual controller where inputs to the drone are: **Altitude rate**, **yaw rate**, **pitch**, **roll**. This will allow even an inexperienced operator to steer the quadrotor.

1.4 Problem sub-parts

Since this is a rather big task, the problem will be divided into smaller sub-part. These are:

Hardware Construction The main part of the electronics and mechanical hardware for the project was constructed before the project started and a description of it can be fund in the implementation section.

[2] http://www.robocup.dtu.dk/

Visual Odometry To be capable of functioning autonomously the quadrotor must be capable of locating itself. This can be done in multiple ways. Visual odometry was chosen as cameras are light and inexpensive, furthermore they work in indoor environments where *GPS* signals are unavailable. Visual odometry will have to be implemented on a Linux computer with sufficient performance.

Altitude and Attude Controller An altitude and an attitude controller will be implemented on a low level micro controller. This enables manual control to work even in the case that something on the high level controller fails.

Position controller Position control (and in the future mission planning) will be done on the same Linux computer as the odometry is running on.

1.5 Report Structure

First some basic theory about state estimation and general computer vision techniques will be presented. Two approaches to visual odometry will be discussed the *MSCKF* and *GPFVO*. While only one of them, *GPFVO*, is used in the final implementation, there are some similarities between the two. A large chapter about implementation follows. This chapter is split into three major parts: Description of the used hardware, software implementation of *GPFVO*, and design and implementation of a controller. Finally results are presented and discussed.

1.6 Notation

Scalars are denoted by italic, non-bold letters: Vectors are italic, bold. Matrices are bold, non-italic.

When vectors are mentioned they will have a preceding superscript denoting the base they are expressed in and a subscript denoting what they are.

Rotation matrices and quaternions will have both a preceding super- and subscript denoting which frame the rotate from and to. Some examples are:

$$^{G}\boldsymbol{p}_{I}: \quad \text{The location of the Inertia frame in global coordinates} \tag{1.1}$$

$$^{G}\boldsymbol{a}_{m} = {}^{G}_{I}\boldsymbol{q}\,{}^{I}\boldsymbol{a}_{m}: \quad \text{Transformation of measured acceleration from inertia to global frame} \tag{1.2}$$

$$^{G}\boldsymbol{a}_{m} = {}^{G}_{I}\mathbf{R}\,{}^{I}\boldsymbol{a}_{m}: \quad \text{Same transformation with a matrix} \tag{1.3}$$

$$^{G}C\boldsymbol{q} = {}^{G}I\boldsymbol{q} \otimes {}^{I}C\boldsymbol{q}: \quad \text{Chained transformation} \tag{1.4}$$

1.6.1 Quaternions

Unit quaternions can be used to represent spacial rotation in a way that avoids the "gimbal lock" associated with Euler angles. More that one definition of quaternions exists. We use the notation:

$$\boldsymbol{q} = \boldsymbol{i}q_1 + \boldsymbol{j}q_2 + \boldsymbol{k}q_3 + q_4; \tag{1.5}$$

$$\boldsymbol{i}^2 = -1 \qquad \boldsymbol{j}^2 = -1 \qquad \boldsymbol{k}^2 = -1 \tag{1.6}$$

$$-\boldsymbol{ij} = \boldsymbol{ji} = \boldsymbol{k} \qquad -\boldsymbol{jk} = \boldsymbol{kj} = \boldsymbol{i} \qquad -\boldsymbol{ki} = \boldsymbol{ik} = \boldsymbol{j} \tag{1.7}$$

Which allow quaternion rotations to be chained as following:

$${}^{A}_{C}\boldsymbol{q} = {}^{A}_{B}\boldsymbol{q} \otimes {}^{B}_{C}\boldsymbol{q} \tag{1.8}$$

The quaternion multiplication of $\boldsymbol{q}$ and $\boldsymbol{p}$ is:

$$\boldsymbol{q} \otimes \boldsymbol{p} = \begin{bmatrix} q_4 p_1 + q_3 p_2 - q_2 p_3 + q_1 p_4 \\ -q_3 p_1 + q_4 p_2 + q_1 p_3 + q_2 p_4 \\ q_2 p_1 - q_2 p_2 + q_4 p_3 + q_3 p_4 \\ -q_1 p_1 - q_2 p_2 - q_3 p_3 + q_4 p_4 \end{bmatrix} \tag{1.9}$$

When used to rotate a vector the quaternion ${}^A_B\boldsymbol{q}$ is analoug to the matrix ${}^A_B\mathbf{R}$:

$${}^A_B\boldsymbol{q} \sim {}^A_B\mathbf{R} = \begin{bmatrix} q_1^2 - q_2^2 - q_3^2 + q_4^2 & 2(q_1 q_2 + q_3 q_4) & 2(q_1 q_3 - q_2 q_4) \\ 2(q_1 q_2 - q_3 q_4) & -q_1^2 + q_2^2 - q_3^2 + q_4^2 & 2(q_2 q_3 + q_1 q_4) \\ 2(q_1 q_3 + q_2 q_4) & 2(q_2 q_3 - q_1 q_4) & -q_1^2 - q_2^2 + q_3^2 + q_4^2 \end{bmatrix} \tag{1.10}$$

1.6.2 Coordinate Frames

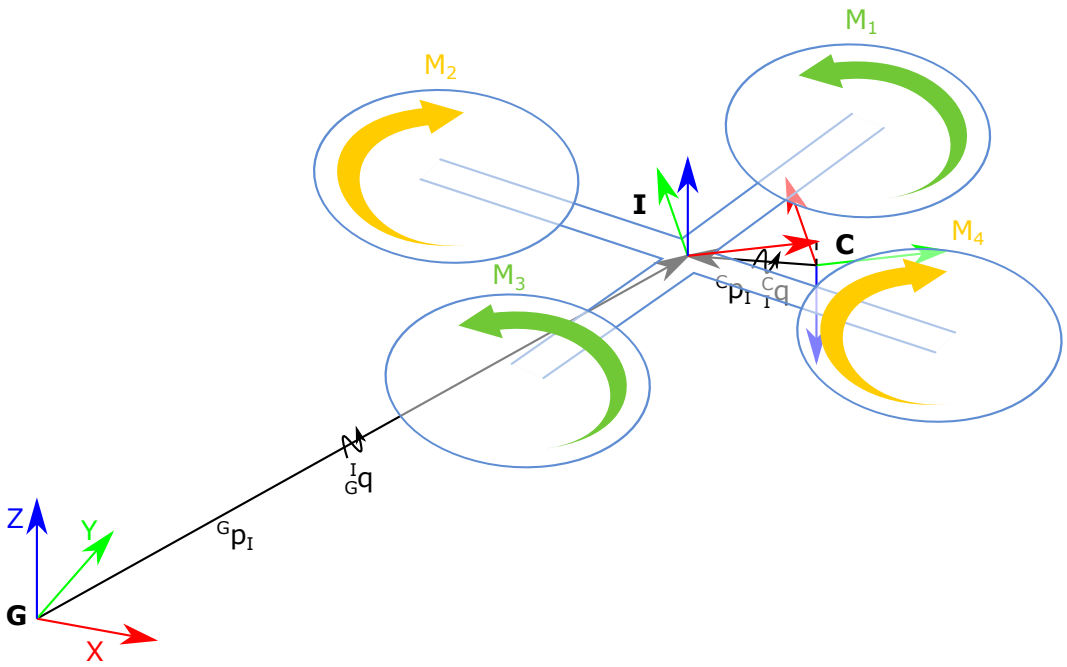

Figure 1.2: Illustration of most common coordinate frames.

Throughout the report multiple coordinate frames will be referenced. The most common are illustrated in figure 1.2 they are:

- G: The global frame is fixed in the world with the z-axis pointing up.
- I: The inertia frame is centered on the *IMU* and also assumed to be the center of the quadrotor. The inertia frame follows the quadrotor, so that the z-axis points in the direction of the motor thrust and the frame does not move in relation to the quadrotor.
- C: The camera frame is centered on the center of the image sensor in the camera, with the z-axis along the view direction of the camera.

CHAPTER 2

Probabilistic State Estimation

When a robot has to localize itself, or otherwise quantify information about itself or the world around it, it must do so using various sensors. Since these sensors never measure the true state of the world or the robot, the sensor data is usually processed in some manner to find the most likely state. Simply stated probabilistic state estimation is the process of estimating the probability of all or some possible states.

In this chapter we will solely concentrate on a type of probabilistic estimator called the Kalman filter, a process in which the estimate of the state probabilities are represented with correlated normal distributions.

This chapter is based on the derivations in Shelley (2014) and Thrun et al. (2005).

2.1 Normal Distributions

The probabilistic density function (*pdf*) of a one-dimensional normal distribution is given by:

$$p(X = x) = (2\pi\sigma^2)^{-\frac{1}{2}} \exp\left\{-\frac{1}{2}\frac{(x-\mu)^2}{\sigma^2}\right\} \tag{2.1}$$

And has the well known bell-shape seen in figure 2.1. When specifying a random variable, with a *pdf* of a normal distribution, this is often abbreviated $x \sim \mathcal{N}(\mu, \sigma^2)$, which specifies the random variable, its mean, and the variance.

In the remaining part of this report the probability $p(X = x)$ that a random value X has the value x will be abbreviated $p(x)$.

In the multi-dimensional case, where x is a vector, the *pdf* becomes:

$$p(\boldsymbol{x}) = \det(2\pi\boldsymbol{\Sigma})^{-\frac{1}{2}} \exp\left\{-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^T\boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu})\right\} \tag{2.2}$$

where $\boldsymbol{\Sigma}$ is a positive semidefinite symmetric matrix called the *covariance matrix*. The diagonal of $\boldsymbol{\Sigma}$ is simply the variance of each element of $\boldsymbol{x}$: $(\sigma^2_{x_1}, \sigma^2_{x_2}, \ldots, \sigma^2_{x_n})$, the non-diagonal elements of $\boldsymbol{\Sigma}$ describe the correlation between two elements of $\boldsymbol{x}$, x_a and x_b: $\rho(x_a, x_b)\sigma_{x_a}\sigma_{x_b}$, where $\rho(x_a, x_b)$ is the correlation. An example of a two-dimensional normal distribution can be seen in figure 2.2

A great advantage of normal distributions, is that they remain normal after being put through many operators, such as:

- Marginalization (finding the distribution of part of the vector $\boldsymbol{x}$, when the other part are unknown)
- Conditioning (finding the distribution of part of the vector $\boldsymbol{x}$, when the other part are known)
- Linear transformation

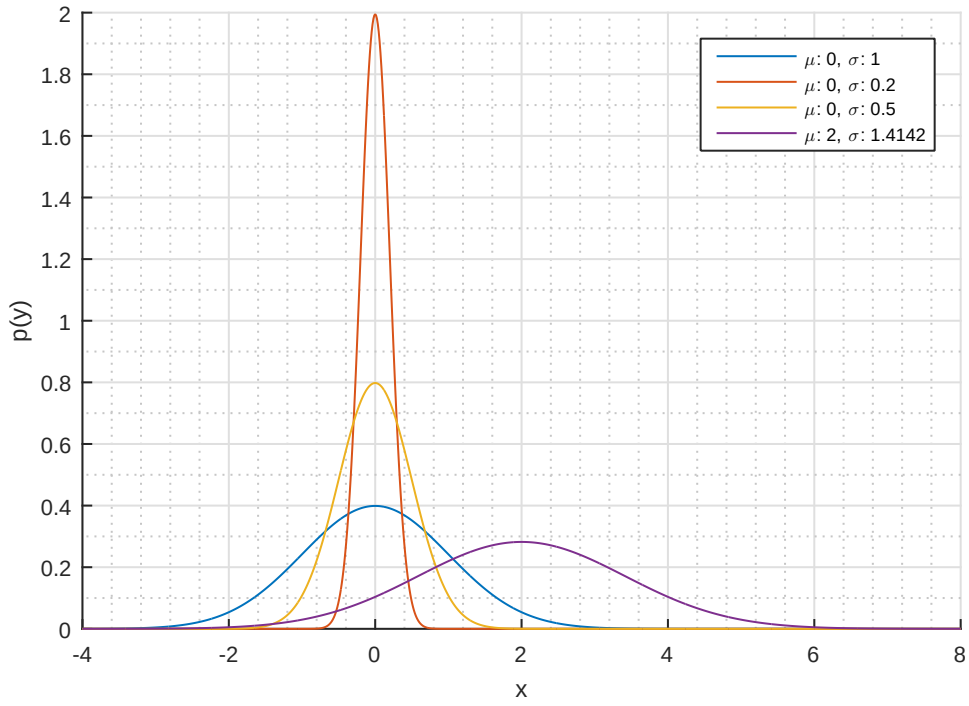

Figure 2.1: The *pdf* of some different normal distributions

- Intersection with another normal distribution[1]

Since the full shape of the normal distribution is contained in the mean vector and covariance matrix, these can effectively be calculated in closed form.

Marginalization It may be the case that we know the joint distribution of several variables: $p(\boldsymbol{x}, \boldsymbol{y})$, but are only interested in $\boldsymbol{x}$. $\boldsymbol{y}$ can then be marginalized out by integrating along it:

$$p(\boldsymbol{x}) = \int p(\boldsymbol{x}, \boldsymbol{y})\, d\boldsymbol{y} \tag{2.3}$$

$$= \mathcal{N}(\boldsymbol{\mu}_x, \boldsymbol{\Sigma}_{xx}) \tag{2.4}$$

This intuitively make sense, as we for each value of $\boldsymbol{x}$ sum up the probability density of all possible outcomes of $\boldsymbol{y}$ at that value.

Conditioning In many cases we might also want to calculate the probability of $\boldsymbol{x}$ given a known $\boldsymbol{y} = \boldsymbol{y_0}$. This can be done:

$$p(\boldsymbol{x}|_{\boldsymbol{y}=\boldsymbol{y}_0}) = \mathcal{N}\left(\boldsymbol{\mu}_x + \boldsymbol{\Sigma}_{xy}\boldsymbol{\Sigma}_{yy}^{-1}(\boldsymbol{y}_0 - \boldsymbol{\mu}_y), \boldsymbol{\Sigma}_{xx} - \boldsymbol{\Sigma}_{xy}\boldsymbol{\Sigma}_{yy}^{-1}\boldsymbol{\Sigma}_{xy}\right) \tag{2.5}$$

Note how the mean shift by the difference between $\boldsymbol{y}_0$ and $\boldsymbol{\mu}$ multiplied by the correlation between $\boldsymbol{x}$ and $\boldsymbol{y}$. Likewise the covariance of $\boldsymbol{x}$ is decreased in scale with the correlation. That is, the more correlated $\boldsymbol{x}$ and $\boldsymbol{y}$ are, the more we change our best guess of $\boldsymbol{x}$ and the more certain we become of this guess.

[1] This is not true for general intersection, but as we will work only with uncorrelated or linearly correlated random variables, this holds true.

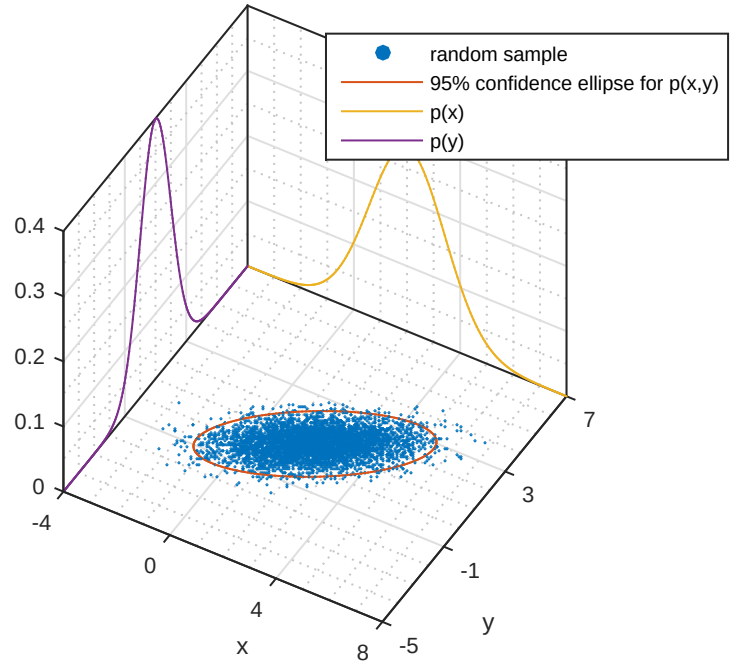

Figure 2.2: 5000 samples of the random variable $\left[\begin{smallmatrix} x \\ y \end{smallmatrix}\right] \sim \mathcal{N}\left(\left[\begin{smallmatrix} 2 \\ 1 \end{smallmatrix}\right], \left[\begin{smallmatrix} 2 & 1 \\ 1 & 1 \end{smallmatrix}\right]\right)$

Summation If we add together two independent, normally distributed variables, their sum is also normal distributed. That is if $\boldsymbol{x} \sim \mathcal{N}(\boldsymbol{\mu}_x, \boldsymbol{\Sigma}_x)$ and $\boldsymbol{y} \sim \mathcal{N}(\boldsymbol{\mu}_y, \boldsymbol{\Sigma}_y)$ are independent (and of the same dimension) then

$$p(\boldsymbol{x} + \boldsymbol{y}) = \mathcal{N}(\boldsymbol{\mu}_x + \boldsymbol{\mu}_y, \boldsymbol{\Sigma}_x + \boldsymbol{\Sigma}_y) \tag{2.6}$$

Intersection If we know the *pdf* of $\boldsymbol{x}$ and $\boldsymbol{y}$ is a linear combination of the elements of $\boldsymbol{x}$ plus some independent, normal distributed noise, that is: $\boldsymbol{y} = \mathbf{A}\boldsymbol{x} + \boldsymbol{b}$ where $\mathbf{A}$ is constant and $\boldsymbol{b} \sim \mathcal{N}(\mathbf{0}, \mathbf{Q})$ then:

$$p\left(\begin{bmatrix} \boldsymbol{x} \\ \boldsymbol{y} \end{bmatrix}\right) = \mathcal{N}\left(\begin{bmatrix} \boldsymbol{\mu}_x \\ \boldsymbol{\mu}_y \end{bmatrix}, \begin{bmatrix} \boldsymbol{\Sigma}_{xx} & \boldsymbol{\Sigma}_{xy} \\ \boldsymbol{\Sigma}_{xy} & \boldsymbol{\Sigma}_{yy} \end{bmatrix}\right) \tag{2.7}$$

$$= \mathcal{N}\left(\begin{bmatrix} \boldsymbol{\mu}_x \\ \mathbf{A}\boldsymbol{\mu}_x + \mathbf{0} \end{bmatrix}, \begin{bmatrix} \boldsymbol{\Sigma}_{xx} & \boldsymbol{\Sigma}_{xx}\mathbf{A}^T \\ \mathbf{A}\boldsymbol{\Sigma}_{xx} & \mathbf{A}\boldsymbol{\Sigma}_{xx}\mathbf{A}^T + \mathbf{Q} \end{bmatrix}\right) \tag{2.8}$$

We will see in the next section, how the simplicity of this is useful.

2.2 The Kalman Filter

The Kalman was originally invented as a efficient technique for implementing Bayes filters for linear systems with Gaussian noise, but is also often (mis)used on non-linear systems and in systems where the noise is not exactly Gaussian.

When using the Kalman filter we assume that the state of the system (ex. pose of a robot) changes linearly according to some motion model. An example could be a quadrotor, with all rotors facing up. Assuming that the input to the motors are directly converted to thrust, we ca use basic physics to calculate a

model that defines the linear relationship between the current *pdf* of the state $\boldsymbol{x}_t \sim \mathcal{N}(\boldsymbol{\mu}_t, \boldsymbol{\Sigma}_t)$ and the next $\boldsymbol{x}_{t+1}$ such that: $\boldsymbol{x}_{t+1} = \mathbf{A}\boldsymbol{x}_t$ then through intersection $\boldsymbol{x}_{t+1} \sim \mathcal{N}(\mathbf{A}\boldsymbol{\mu}_t, \mathbf{A}\boldsymbol{\Sigma}_t\mathbf{A}^T)$.

Usually some information about inputs to the system, or measurements of it is also available. If this input $\boldsymbol{u}$ is independent of the state and affects the next step it should also be included. For the quadrotor example, the state could be vertical position and velocity. The input could then be the motor thrust or measurements from an on-board accelerometer. Both of these are independent of the state, but provide information about, how it changes.

Lastly we will have some noise on both the inputs and the state $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{Q})$. For the input this could be due to noisy measurements or the thrust command could be different from the actual thrust due to wind or shaky motors. This noise will also cover process noise, the origin of the process noise could originate from model imperfections or discretization and is therefore not strictly covered by the assumptions of being normal distributed. This leads us to the equation:

$$\boldsymbol{x}_t \sim \mathcal{N}(\boldsymbol{\mu}_t, \boldsymbol{\Sigma}_t) \tag{2.9}$$

$$\boldsymbol{\epsilon}_t \sim \mathcal{N}(\mathbf{0}, \mathbf{Q}) \tag{2.10}$$

$$\boldsymbol{x}_{t+1} = \mathbf{A}\boldsymbol{x}_t + \mathbf{B}\boldsymbol{u}_t + \boldsymbol{\epsilon}_t \tag{2.11}$$

The *pdf* of $_{t+1}$, $p(\boldsymbol{x}_t)$ can be found from $p(\boldsymbol{x}_t)$ by first using intersection to find $p\left(\left[\begin{smallmatrix}\boldsymbol{x}_t \\ \boldsymbol{x}_{t+1}\end{smallmatrix}\right]\right)$ and then marginalizing out $\boldsymbol{x}_1$:

$$p\left(\begin{bmatrix}\boldsymbol{x}_t \\ \boldsymbol{x}_{t+1}\end{bmatrix}\right) = \mathcal{N}\left(\begin{bmatrix}\boldsymbol{\mu}_t \\ \mathbf{A}\mu_t + \mathbf{B}\boldsymbol{u}\end{bmatrix}, \begin{bmatrix}\boldsymbol{\Sigma}_t & \boldsymbol{\Sigma}_t\mathbf{A}^T \\ \mathbf{A}\boldsymbol{\Sigma}_t & \mathbf{A}\boldsymbol{\Sigma}_t\mathbf{A}^T + \mathbf{Q}\end{bmatrix}\right) \tag{2.12}$$

$$\begin{aligned} p(\boldsymbol{x}_{t+1}) &= \mathcal{N}(\mathbf{A}\mu_t + \mathbf{B}\boldsymbol{u}, \mathbf{A}\boldsymbol{\Sigma}_t\mathbf{A}^T + \mathbf{Q}) \\ &= \mathcal{N}(\boldsymbol{\mu}_{t+1}, \boldsymbol{\Sigma}_{t+1}) \end{aligned} \tag{2.13}$$

$$\boldsymbol{\mu}_{t+1} = \mathbf{A}\mu_t + \mathbf{B}\boldsymbol{u} \tag{2.14}$$

$$\boldsymbol{\Sigma}_{t+1} = \mathbf{A}\boldsymbol{\Sigma}_t\mathbf{A}^T + \mathbf{Q} \tag{2.15}$$

as we can see, the shape of the *pdf* for the next step can be calculated using only the mean and covariance of the current step, and some cheap matrix operations. This is what is known as the Kalman propagation step.

We also need some measurements that are a linear combination of the state. These are needed as the system and input noise ($\mathbf{Q}$) would otherwise have the covariance, and the estimation uncertainty, increase indefinitely. These measurements $\boldsymbol{z}$ are a linear combination of the states and include some noise and can be described:

$$\boldsymbol{\delta}_t \sim \mathcal{N}(\mathbf{0}, \mathbf{R}) \tag{2.16}$$

$$\boldsymbol{z}_t = \mathbf{C}\boldsymbol{x}_t + \boldsymbol{\delta}_t \tag{2.17}$$

By intersection the measurement with the state we can find the joint *pdf*:

$$p\left(\begin{bmatrix}\boldsymbol{x}_t \\ \boldsymbol{z}_t\end{bmatrix}\right) = \mathcal{N}\left(\begin{bmatrix}\boldsymbol{\mu}_t \\ \mathbf{C}\boldsymbol{\mu}_t\end{bmatrix}, \begin{bmatrix}\boldsymbol{\Sigma}_t & \boldsymbol{\Sigma}_t\mathbf{C}^T \\ \mathbf{C}\boldsymbol{\Sigma}_t & \mathbf{C}\boldsymbol{\Sigma}_t\mathbf{C}^T + \mathbf{R}\end{bmatrix}\right) \tag{2.18}$$

and by conditioning with the actual measurement $\boldsymbol{z}_0$ we can find the updated estimates. From (2.5) we have:

$$p(\boldsymbol{x}_t|_{\boldsymbol{z}_t=\boldsymbol{z}_0}) = \mathcal{N}(\boldsymbol{\mu}_{x_t} + \boldsymbol{\Sigma}_{x_t z_t}\boldsymbol{\Sigma}_{z_t z_t}^{-1}(\boldsymbol{z}_0 - \boldsymbol{z}_t), \boldsymbol{\Sigma}_{x_t x_t} - \boldsymbol{\Sigma}_{x_t z_t}\boldsymbol{\Sigma}_{z_t z_t}^{-1}\boldsymbol{\Sigma}_{x_t z_t}) \tag{2.19}$$

By using the information in (2.18):

$$\boldsymbol{z}_t = \mathbf{C}\boldsymbol{x}_t \tag{2.20}$$
$$\boldsymbol{\mu}_{x_t} = \boldsymbol{\mu}_t \tag{2.21}$$
$$\boldsymbol{\Sigma}_{z_t z_t} = \mathbf{C}\boldsymbol{\Sigma}_t\mathbf{C}^T + \mathbf{R} \tag{2.22}$$
$$\boldsymbol{\Sigma}_{x_t z_t} = \boldsymbol{\Sigma}_t\mathbf{C}^T \tag{2.23}$$
$$\mathbf{K}_t = \boldsymbol{\Sigma}_{x_t z_t}\boldsymbol{\Sigma}_{z_t z_t}^{-1} \tag{2.24}$$
$$= \boldsymbol{\Sigma}_t\mathbf{C}^T(\mathbf{C}\boldsymbol{\Sigma}_t\mathbf{C}^T + \mathbf{R})^{-1} \tag{2.25}$$

We can write (2.19) as:

$$\begin{aligned} p(\boldsymbol{x}_t|_{\boldsymbol{z}_t=\boldsymbol{z}_0}) &= \mathcal{N}(\boldsymbol{\mu}_{x_t} + \mathbf{K}_t(\boldsymbol{z}_0 - \boldsymbol{z}_t), \boldsymbol{\Sigma}_{x_t x_t} - \mathbf{K}_t\boldsymbol{\Sigma}_{x_t z_t}) \\ &= \mathcal{N}(\boldsymbol{\mu}_t + \mathbf{K}_t(\boldsymbol{z}_0 - \mathbf{C}\boldsymbol{x}_t), \boldsymbol{\Sigma}_t\mathbf{C}^T - \mathbf{K}_t\boldsymbol{\Sigma}_t\mathbf{C}^T) \end{aligned} \tag{2.26}$$
$$\mathbf{K}_t = \boldsymbol{\Sigma}_t\mathbf{C}^T(\mathbf{C}\boldsymbol{\Sigma}_t\mathbf{C}^T + \mathbf{R})^{-1} \tag{2.27}$$
$$\boldsymbol{\mu}_t|_{\boldsymbol{z}_t=\boldsymbol{z}_0} = \boldsymbol{\mu}_t + \mathbf{K}_t(\boldsymbol{z}_0 - \mathbf{C}\boldsymbol{x}_t) \tag{2.28}$$
$$\boldsymbol{\Sigma}_t|_{\boldsymbol{z}_t=\boldsymbol{z}_0} = \boldsymbol{\Sigma}_t\mathbf{C}^T - \mathbf{K}_t\boldsymbol{\Sigma}_t\mathbf{C}^T \tag{2.29}$$

Which is known as the Kalman update step.

2.2.1 Summation and Common Notation

When using the Kalmain filter we are often only interested in the most likely of the possible states, that is the mode. Luckily for normal distributions the mean and mode are the same, since the *pdf* is symmetric. The most likely state is called the *state estimate* and is usually denoted with a hat, in our case this would be $\hat{\boldsymbol{x}}_{t|t-1}$. Where the subscript is read "the state estimate at time t using all measurements from time 0 through $t-1$". Note also that $\boldsymbol{z}_t$ is the actual measurement at time t and not the statistical distribution of it. This gives us the equations:

- Initial Estimate

$$\boldsymbol{x}_0 \sim \mathcal{N}(\hat{\boldsymbol{x}}_{0|0}, \boldsymbol{\Sigma}_{0|0})$$

- Prediction Step

$$\begin{aligned} &\text{Given: } \boldsymbol{x}_{t+1} = \mathbf{A}\boldsymbol{x}_t + \mathbf{B}\boldsymbol{u}_t + \boldsymbol{\epsilon}_t \text{ where } \boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{Q}) \\ &\hat{\boldsymbol{x}}_{t+1|t} = \mathbf{A}\hat{\boldsymbol{x}}_{t|t} + \mathbf{B}\boldsymbol{u}_t \\ &\boldsymbol{\Sigma}_{t+1|t} = \mathbf{A}\boldsymbol{\Sigma}_{t|t}\mathbf{A}_t^T + \mathbf{Q}_t \end{aligned}$$

- Update Step

$$\begin{aligned} &\text{Given: } \boldsymbol{z}_t = \mathbf{C}\boldsymbol{x}_t + \boldsymbol{\delta}_t \text{ where } \boldsymbol{\delta} \sim \mathcal{N}(\mathbf{0}, \mathbf{R}) \\ &\hat{\boldsymbol{x}}_{t|t} = \hat{\boldsymbol{x}}_{t|t-1} + \mathbf{K}_t(\boldsymbol{z}_t - \mathbf{C}\hat{\boldsymbol{x}}_{t|t-1}) \\ &\boldsymbol{\Sigma}_{t|t} = \boldsymbol{\Sigma}_{t|t-1} - \mathbf{K}_t\mathbf{C}\boldsymbol{\Sigma}_{t|t-1} \\ &\mathbf{K}_t = \boldsymbol{\Sigma}_{t|t-1}\mathbf{C}_t^T(\mathbf{C}\boldsymbol{\Sigma}_{t|t-1}\mathbf{C}^T + \mathbf{R})^{-1} \end{aligned}$$

2.3 Kalman Filter Example, Altitude Estimation

Assume that we are interested in finding the altitude above ground for a quadrotor. The quadrotor is equipped with an accelerometer measuring the z-acceleration at $400\,\mathrm{Hz}$ and an ultrasonic distance sensor, measuring the distance to the ggroundat $20\,\mathrm{Hz}$.

If we chose the state representation of the quadrotor to include the altitude and the z-velocity we can use the measured acceleration to propagate the state estimate, since the state estimate is independent of the acceleration. Furthermore we can use the altitude measurement for the update step, as this measurement is a linear combination of the state. In reality the accelerometer will also have a bias that drifts a bit over time, we will include this bias in our state as well:

$$\hat{\boldsymbol{x}}_t = \begin{bmatrix} \hat{p}_t \\ \hat{v}_t \\ \hat{b_{at}} \end{bmatrix} \tag{2.30}$$

where $hatp_t$ is the altitude estimate, $\hat{v}_t$ is the velocity estimate and $\hat{b_{at}}$ is the bias estimate defined so that $a = a_m - b_a$ where a is the actual acceleration and a_m is the measured acceleration.

We will now look at what happens to the estimates from one time step to another.

$$b_{at+1} = b_{at} \tag{2.31}$$

$$v_{t+1} = v_t + \int_t^{t+1} a_{m\tau} - b_{a\tau}\, d\tau \approx v_t - b_{at}\Delta t + a_{mt}\Delta t \tag{2.32}$$

$$p_{t+1} = p_t + \int_t^{t+1} v_\tau\, d\tau \approx p_t + v_t\Delta t - \tfrac{1}{2}b_{at}\Delta t^2 + \tfrac{1}{2}a_{mt}\Delta t^2 \tag{2.33}$$

Recalling the formulation of the propagation step we rewrite this:

$$\boldsymbol{x}_{t+1} = \mathbf{A}\boldsymbol{x}_t + \mathbf{B}a_{mt} + \boldsymbol{\epsilon}_t \tag{2.34}$$

$$\mathbf{A} = \begin{bmatrix} 1 & \Delta t & -\frac{1}{2}\Delta t^2 \\ 0 & 1 & -\Delta t \\ 0 & 0 & 1 \end{bmatrix} \tag{2.35}$$

$$\mathbf{B} = \begin{bmatrix} \frac{1}{2}\Delta t^2 \\ \Delta t \\ 0 \end{bmatrix} \tag{2.36}$$

$$\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{Q}) \tag{2.37}$$

The contribution of the noise from the measurement a_m to the covariance of $\boldsymbol{\epsilon}$ is found by applying a linear transformation of $\mathbf{B}$ to the variance σ_a^2 of the noise on the accelerometer measurement:

$$\mathbf{Q}_a = \mathbf{B}\sigma_a^2\mathbf{B}^T = \begin{bmatrix} \frac{1}{4}\Delta t^4 & \frac{1}{2}\Delta t^3 & 0 \\ \frac{1}{2}\Delta t^3 & \Delta t^2 & 0 \\ 0 & 0 & 0 \end{bmatrix} \sigma_a^2 \tag{2.38}$$

The bias of the accelerometer can be modeled by a random walk process, where a normal distributed noise $\mathcal{N}(0, \sigma_{wa})$ is integrated to give the random walk. This noise could be treated as an extra input r_t with noise to it and a matrix $\mathbf{G}$ relating it to $\hat{\boldsymbol{x}}_{t+1}$ such that:

$$\hat{\boldsymbol{x}}_{t+1} = \mathbf{A}\hat{\boldsymbol{x}}_t + \mathbf{B}a_{mt} + \mathbf{G}r_t + \boldsymbol{\epsilon}_t \tag{2.39}$$

but since $r_t = 0$, $\mathbf{G}$ is omitted in this equation. We skip the actual calculation of $\mathbf{G}$ as the method is similar to that of the calculation of $\mathbf{B}$ and trivial.

$$\mathbf{G} = \begin{bmatrix} -\frac{1}{3}\Delta t^3 \\ -\frac{1}{2}\Delta t^2 \\ \Delta t \end{bmatrix} \tag{2.40}$$

We can now calculate the contribution to the covariance of $\boldsymbol{\epsilon}_t$:

$$\mathbf{Q}_{wa} = \mathbf{G}\sigma_{wa}^2\mathbf{G}^T = \begin{bmatrix} \frac{1}{9}\Delta t^6 & \frac{1}{6}\Delta t^5 & -\frac{1}{3}\Delta t^4 \\ \frac{1}{6}\Delta t^5 & \frac{1}{4}\Delta t^4 & -\frac{1}{2}\Delta t^2 \\ -\frac{1}{3}\Delta t^4 & -\frac{1}{2}\Delta t^3 & \Delta t^2 \end{bmatrix} \sigma_{wa}^2 \tag{2.41}$$

And $\mathbf{Q}$

$$\mathbf{Q} = \mathbf{Q}_a + \mathbf{Q}_{wa} = \begin{bmatrix} \frac{1}{4}\Delta t^4 & \frac{1}{2}\Delta t^3 & 0 \\ \frac{1}{2}\Delta t^3 & \Delta t^2 & 0 \\ 0 & 0 & 0 \end{bmatrix} \sigma_a^2 + \begin{bmatrix} \frac{1}{9}\Delta t^6 & \frac{1}{6}\Delta t^5 & -\frac{1}{3}\Delta t^4 \\ \frac{1}{6}\Delta t^5 & \frac{1}{4}\Delta t^4 & -\frac{1}{2}\Delta t^2 \\ -\frac{1}{3}\Delta t^4 & -\frac{1}{2}\Delta t^3 & \Delta t^2 \end{bmatrix} \sigma_{wa}^2 \tag{2.42}$$

The distance sensor measures the altitude directly and is corrupted by a zero-mean normal distributed noise with variance σ_h^2:

$$z_t = \mathbf{C}\boldsymbol{x}_t + \delta_t \tag{2.43}$$

$$\delta_t \sim \mathcal{N}(0, \sigma_h^2) \tag{2.44}$$

$$\mathbf{C} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \tag{2.45}$$

By using *Matlab* and choosing some values for σ_a, σ_{aw}, σ_h and initial conditions for $\hat{\boldsymbol{x}}$ and $\boldsymbol{\Sigma}$ we can simulate the behavior of this example filter. We chose the following values and initial conditions:

$$\sigma_a = 0.1 \tag{2.46}$$

$$\sigma_{wa} = 0.1 \tag{2.47}$$

$$\sigma_h = 0.015 \tag{2.48}$$

$$\hat{\boldsymbol{x}}_0 = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \tag{2.49}$$

$$\boldsymbol{\Sigma} = \begin{bmatrix} 1^2 & 0 & 0 \\ 0 & 1^2 & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{2.50}$$

And simulate a sinusoidal motion with an amplitude of $10\,\mathrm{cm}$ and a frequency of $0.5\,\mathrm{Hz}$. In figure 2.3 we see the estimated altitude. Even though the measurements of the actual altitude are infrequent (compared to the accelerometer) and noisy, the estimate follows the real altitude nicely. Also note that we get the velocity estimate, figure 2.4, "for free".

2.3.1 Outlier Detection

When working with ultrasonic distance sensors it is not uncommon that some measurements will be complete outliers. This can happen if the echo for some reason does not return, or if wind is blown into the

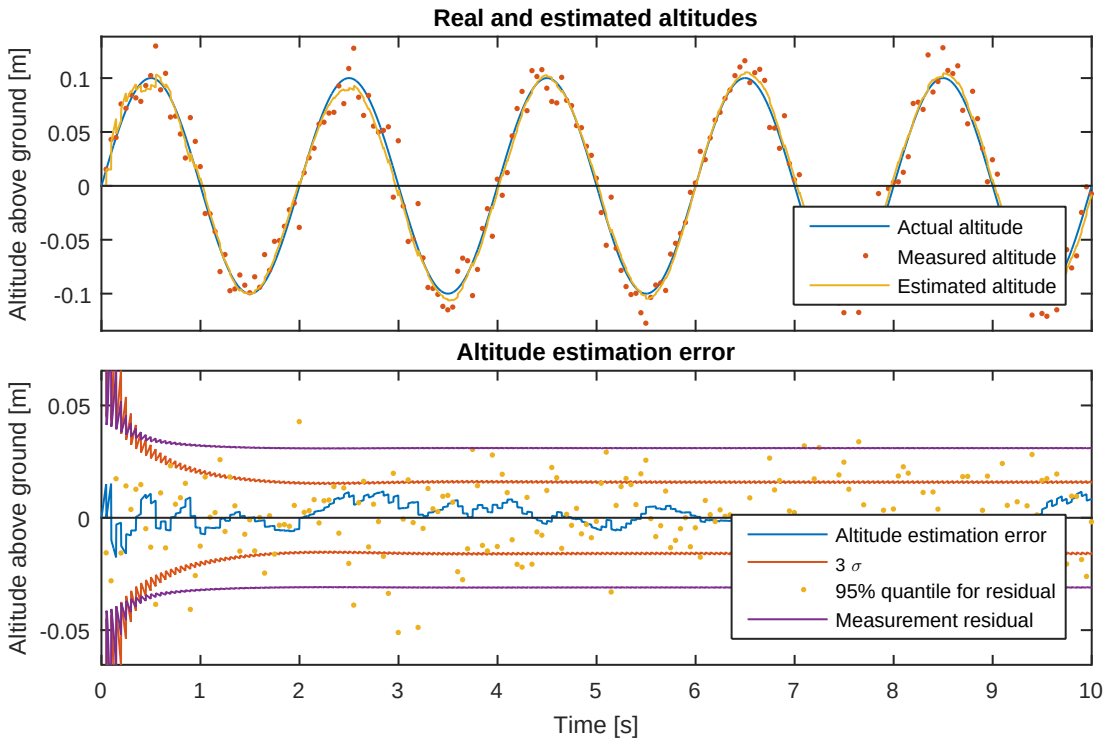

Figure 2.3: Estimated altitude of a quadrotor flying in a sinusoidal pattern

sensor. As the sensor is mounted on a quadrotor these outliers are quite common, as the rotors generate wind tat can go into the sensor.

As the Kalman filter assumes that the noise is normal distributed outliers leads to filter inaccuracies as the assumptions are not met. In figure 2.6 we see the effect on the filter if outliers are not detected.

One advantage of the Kalman filter is that besides the state estimate, we also have an estimate of the variances, thus it is possible to calculate the probability of a measurement being an outlier. If the measurement is an inlier, it can be described by:

$$z_{ti} = \mathbf{C}\boldsymbol{x}_t + \delta_t z_{ti} \sim \mathcal{N}(\mathbf{C}\hat{\boldsymbol{x}}, \mathbf{C}\boldsymbol{\Sigma}_t\mathbf{C}^T + \sigma_h^2) \tag{2.51}$$

By subtracting the estimated state on both sides we get the residual $r_t = z_t - \mathbf{C}\hat{\boldsymbol{x}}$ of an inlier:

$$r_{ti} \sim \mathcal{N}(0, \mathbf{C}\boldsymbol{\Sigma}_t\mathbf{C}^T + \sigma_h^2) \tag{2.52}$$

We can use the χ^2 test to check if it is likely that a measurement with this residual is an inlier, that is we test the likelihood of the residual r_t coming from the distribution r_{ti}. We calculate γ (for vector case see note[2]):

$$\gamma = \frac{r_t^2}{\mathbf{C}\boldsymbol{\Sigma}_t\mathbf{C}^T + \sigma_h^2} \tag{2.53}$$

and compare this to the χ^2 quartile function $chi_2inv(p, k)$ with k degrees of freedom and given probability p. In our case we have one measurement, so the degree of freedom is 1 and we chose a significance level of 95% (that is we want to reject a maximum of 5% inliers). A measurement is then considered an inlier if

$$\gamma < chi_2inv(0.95, 1) \tag{2.54}$$

[2]If we had multiple measurements, so that z_t was a vector this equation would be $\gamma = \boldsymbol{r}_t^T(\mathbf{C}\boldsymbol{\Sigma}_t\mathbf{C} + \mathbf{R})^{-1}\boldsymbol{r}_t$

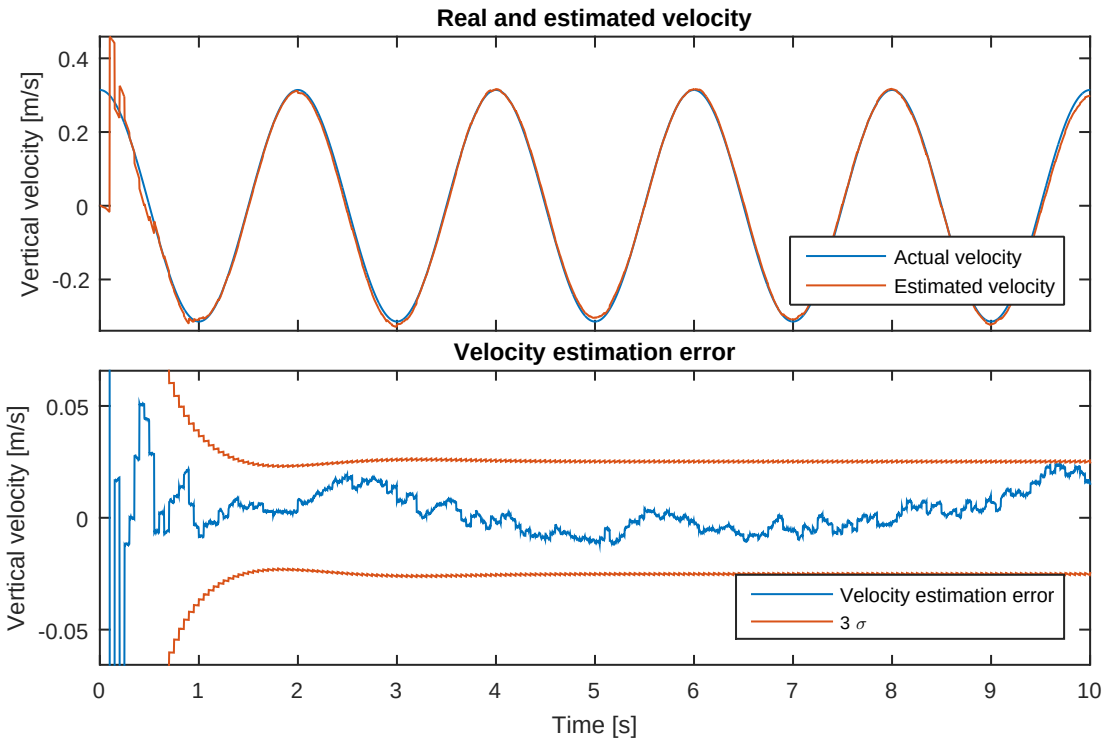

Figure 2.4: Estimated vertical velocity of a quadrotor flying in a sinusoidal pattern

A simulated test with outliers and outlier detection can be seen in figure 2.7

2.3.2 Ground Level Shift Detection

As the ultrasonic distance sensor measure the distance to the ground the filter will not function optimally when flying over obstacles, as this changes the altitude above ground, but does not actually move the quadrotor.

With outlier detection turned on this leads to all measurements being ignored until the variance of the altitude estimate becomes large enough. This can take quite a while, allowing the velocity and bias estimate to drift.

On the other hand with outlier detection turned off, the filter will not function as intended, and the cause of the measurement residual will be perceived as an error in velocity and bias estimate. The effect of this can be seen in figure 2.8

A solution to this problem is to count consecutive outliers. If this amount gets too large, it is unlikely that they actually are outliers and more likely that the ground level changed. We define an amount of maximum consecutive outliers and if the current amount of consecutive outliers is larger we set the variance of the altitude estimate to a predefined large value. This forces the next altitude measurement to set the altitude estimate, without affecting velocity and bias estimate. A simulation with both outliers and step detection can be seen in figure 2.9

It is noteworthy that when applying thistechniquee, the velocity estimate stays correct, due to the accelerometer measurements. As does the temporary changes in estimated altitude. The result of this is, that if the estimates are later used in a controller, the controller will stay stable during this step (but will of cause respond delayed to the step).

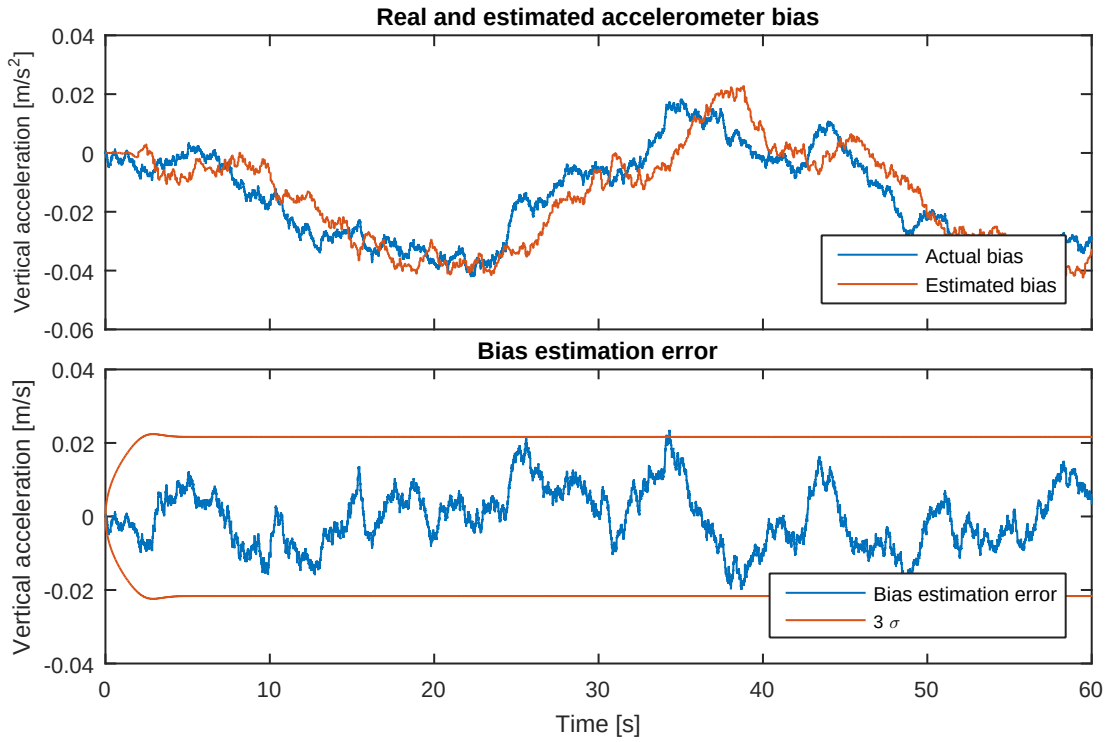

Figure 2.5: Estimated accelerometer bias of a quadrotor flying in a sinusoidal pattern

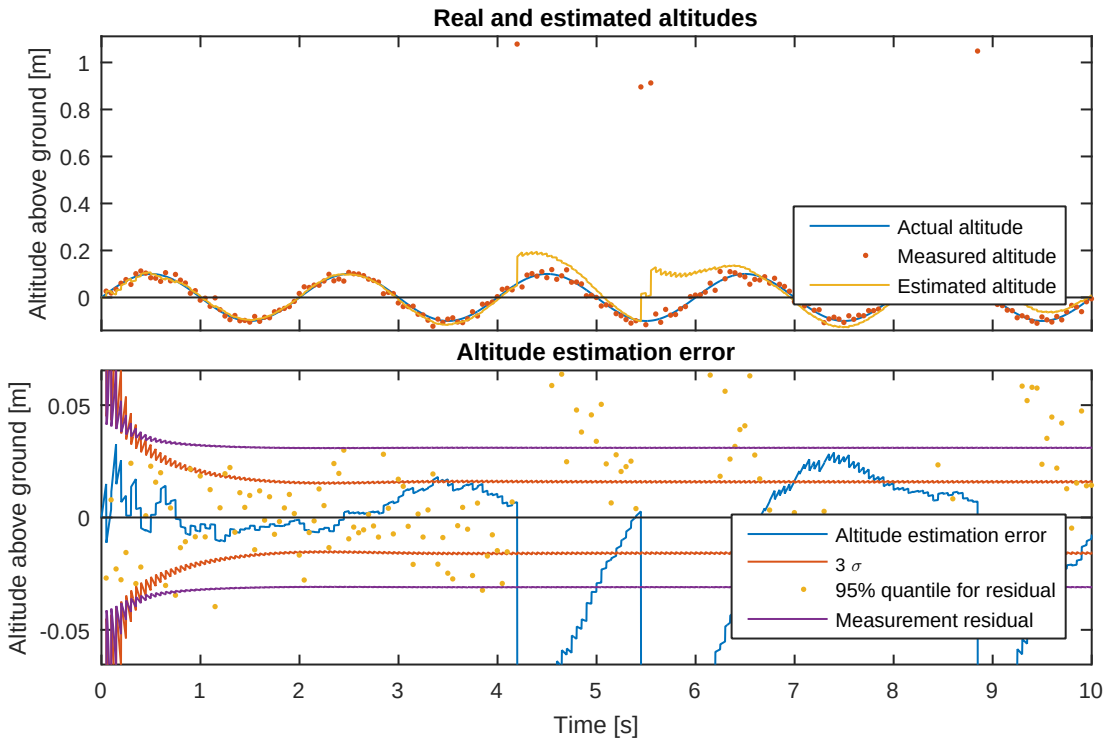

Figure 2.6: Estimated altitude of a quadrotor flying in a sinusoidal pattern. The altitude sensor is corrupted by spare $1\,\mathrm{m}$ peaks.

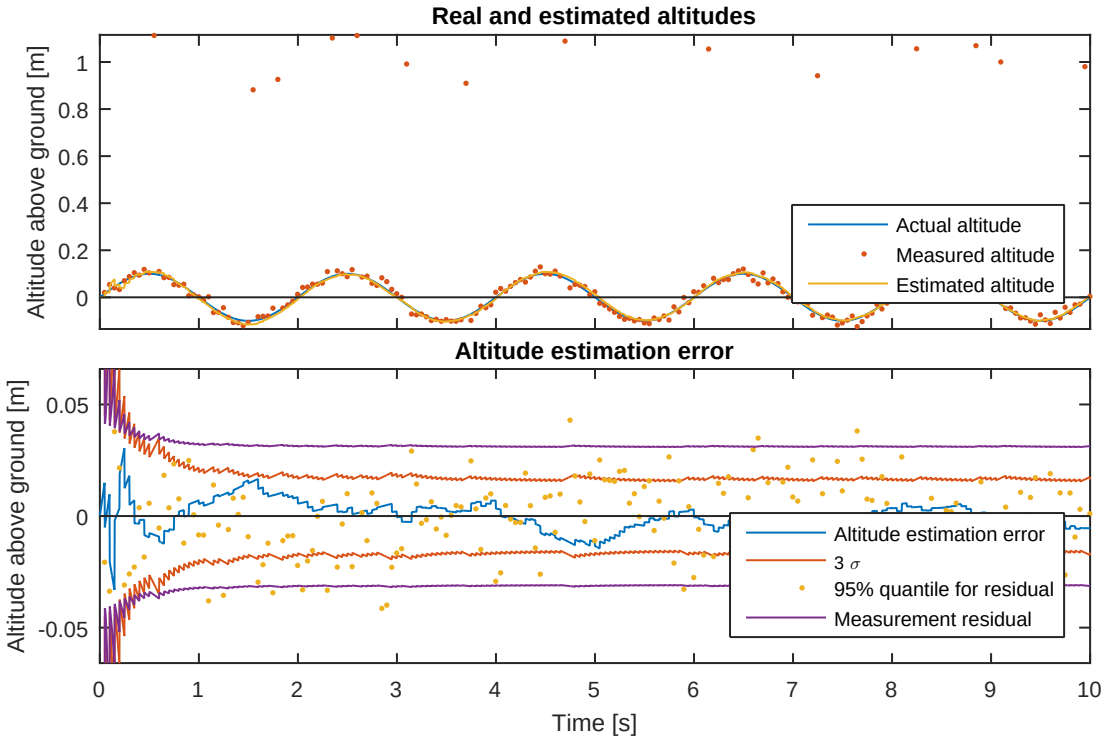

Figure 2.7: Estimated altitude of a quadrotor flying in a sinusoidal pattern. The altitude sensor is corrupted by spare $1\,\mathrm{m}$ peaks. Outliers are detected and rejected. Note that some inliers are also rejected, but since the amount is small this is a better option than not discarding outliers.

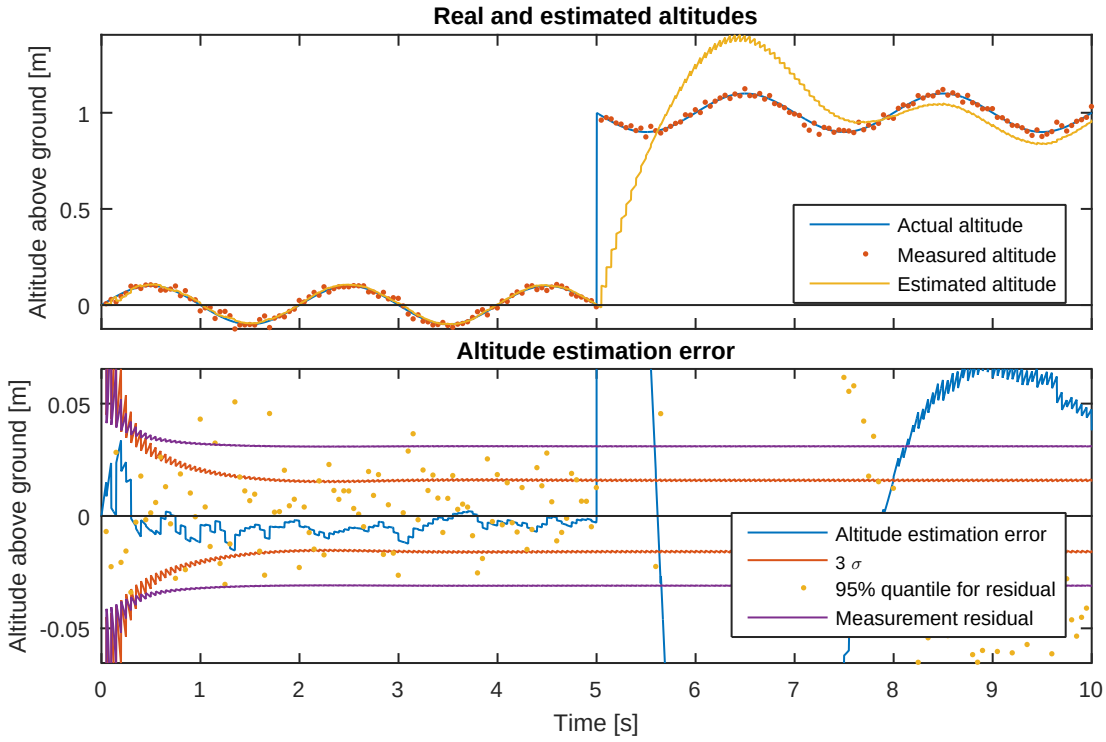

Figure 2.8: Estimated altitude of a quadrotor flying in a sinusoidal pattern. At time $t = 5\,\mathrm{s}$ the quadrotor flies over an edge, where the ground is located $1\,\mathrm{m}$ lower.

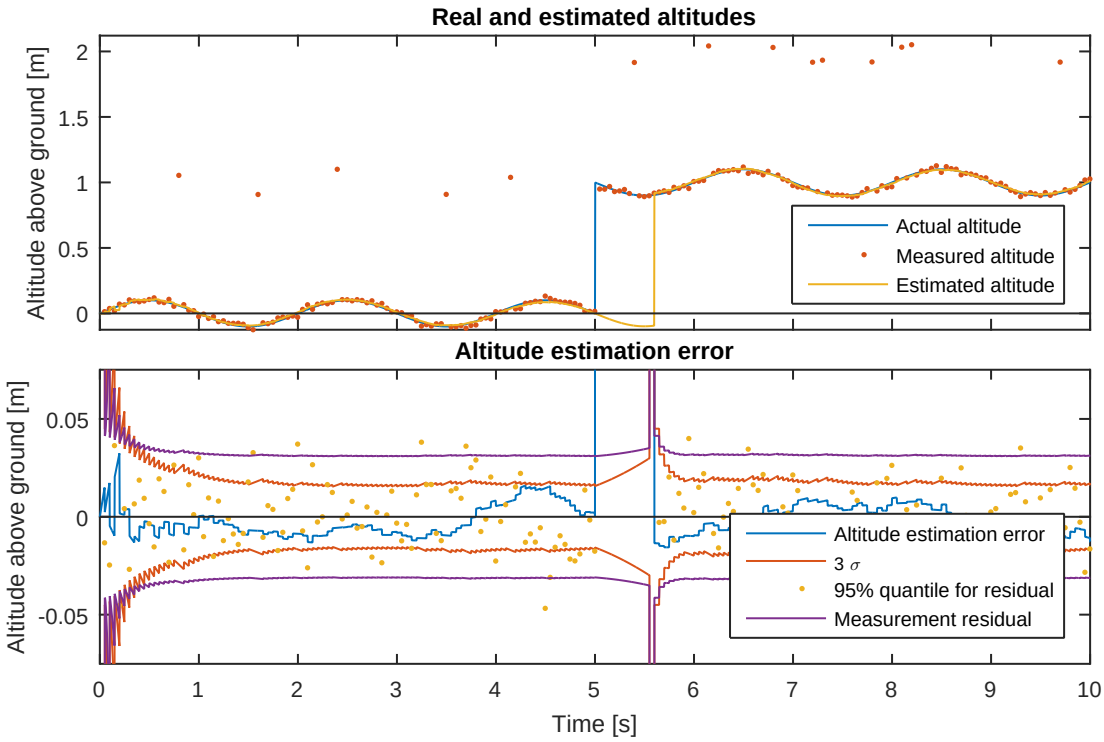

Figure 2.9: Estimated altitude of a quadrotor flying in a sinusoidal pattern. At time $t = 5\,\mathrm{s}$ the quadrotor flies over an edge, where the ground is located $1\,\mathrm{m}$ lower. Note how after some time this step is detected and the state estimate corrected

CHAPTER 3

Computer Vision Basics

In this chapter we will briefly go over some computer vision basics used in the project, such as camera basics, feature detection and tracking, and image transformation.

3.1 Camera Model

3.1.1 Pinhole Camera Model

A pinhole camera is a simple device for focusing light. The camera consists of a black box with a small hole in the center of one of the sides and a canvas on the opposite wall. The pinhole ensures that for each point on the canvas only photons from one direction can reach it. In figure 3.1 we see an illustration of a pinhole camera.

The simplicity of the pinhole model also leads to a very simple transformation between camera coordinates and image coordinates. As there is no reason to deal with the $180°$ rotation associated with the image plane, we will concentrate on the projection onto the virtual image plane. The conversion between camera coordinates and coordinates on the virtual image plane is illustrated in figure 3.2. Here a point in the camera frame has coordinates: ${}^{C}\boldsymbol{P}_i = \begin{bmatrix} X_i \\ Y_i \\ Z_i \end{bmatrix}$, in the virtual image plane the point has the coordinates ${}^{Img}\boldsymbol{p}_i = \begin{bmatrix} x_i \\ y_i \end{bmatrix}$. We see that the relation between these points are:

$$x_i = f\frac{X_i}{Z_i} \tag{3.1}$$

$$y_i = f\frac{Y_i}{Z_i} \tag{3.2}$$

For images on a computer origo is usually located in the upper left corner with positive x direction towards right and positive y direction downwards, for that reason it is common that we define the coordinate system of the camera so the z axis points from the focal point to the virtual image plane.

Since the unit of f is usually given in meters and the units we want to work with are pixels and origo of an image is located in the top left corner and not in the center of the image we reformulate this to:

$$x_i = \frac{f}{d_x}\frac{X_i}{Z_i} + o_x \tag{3.3}$$

$$y_i = \frac{f}{d_y}\frac{Y_i}{Z_i} + o_y \tag{3.4}$$

where $(o_x, o_y)^T$ is the location of the camera origo in the image coordinate system and d_x, d_y are the distance between each pixel in the x and y direction.

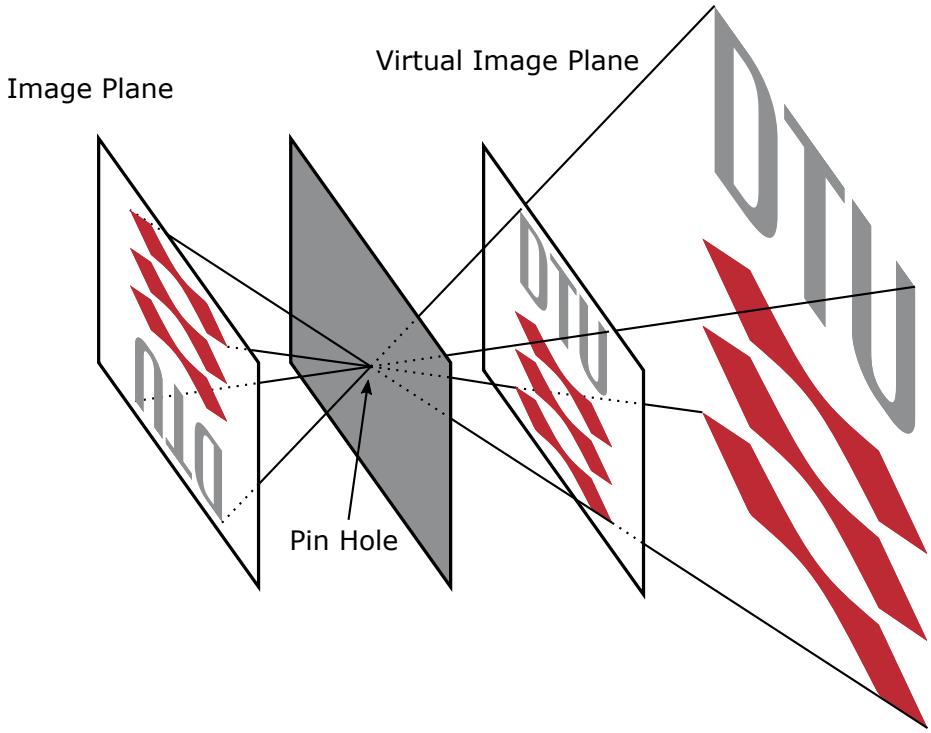

Figure 3.1: An illustration of a pinhole camera taking a photo of the *DTU* logo. The pinhole makes sure that for each point on the image plane only photons from one direction can hit.

This gives us the projection function:

$$\boldsymbol{p} = \boldsymbol{h}\left(\begin{bmatrix} X \\ Y \\ Z \end{bmatrix}\right) = \begin{bmatrix} o_x \\ o_y \end{bmatrix} + \begin{bmatrix} f_x & 0 \\ 0 & f_y \end{bmatrix} \begin{bmatrix} \frac{X}{Z} \\ \frac{Y}{Z} \end{bmatrix} \tag{3.5}$$

3.1.2 Lens Distortion

While the pinhole camera is simple to describe and useful for describing the process of image projection, we do not commonly use pinhole cameras. The small opening in a pinhole camera does not allow for much light to enter. Using a lens to focus the light overcomes this problem, but has it's own.

A common problem of especially cheap lenses are that they distort the image slightly. One way of approximating the lens distortion is by fitting a simple polynomic function to the distortion.

We split the distortion in radial and tangential components d_r and $\boldsymbol{d}_t$. For the radial distortion we only include 1$^{\text{st}}$ and 2$^{\text{nd}}$ moment, however it is not uncommon to include the 3$^{\text{rd}}$ moment as well. This gives us the projection function:

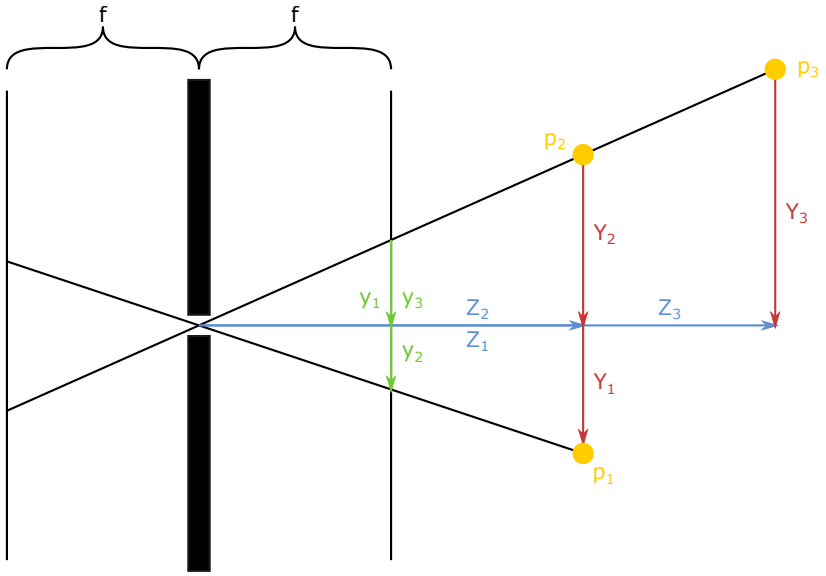

Figure 3.2: Projection of points onto the virtual image plane. Note how more that on point in 3D space can map onto the same point in the image.

$$\boldsymbol{h}\left(\begin{bmatrix} X \\ Y \\ Z \end{bmatrix}\right) = \begin{bmatrix} o_x \\ o_y \end{bmatrix} + \begin{bmatrix} f_x & 0 \\ 0 & f_y \end{bmatrix} \left(d_r \begin{bmatrix} u \\ v \end{bmatrix} + \boldsymbol{d}_t \right) \tag{3.6}$$

$$d_r = 1 + k_1 r + k_2^2 \tag{3.7}$$

$$\boldsymbol{d}_t = \begin{bmatrix} 2uvt_1 + (r + 2u^2)t_2 \\ 2uvt_2 + (r + 2v^2)t_1 \end{bmatrix} \tag{3.8}$$

$$u = \frac{X}{Z}, \quad v = \frac{Y}{Z}, \quad r = u^2 + v^2 \tag{3.9}$$

This is a commonly used model and programs such as *Matlab* can estimate the required values from a set of calibration images.

Correcting Lens Distortion

If we want to calculate the direction of an incoming beam of light that hits a pixed we have to correct for the lens distortion. The direction of the beam can be parametrized by u and v as $u = \frac{X}{Z}$ and $v = \frac{Y}{Z}$. The problem we are now facing is to invert the function:

$$\boldsymbol{h}'\left(\begin{bmatrix} u \\ v \end{bmatrix}\right) = \begin{bmatrix} o_x \\ o_y \end{bmatrix} + \begin{bmatrix} f_x & 0 \\ 0 & f_y \end{bmatrix} \left(d_r \begin{bmatrix} u \\ v \end{bmatrix} + \boldsymbol{d}_t \right) \tag{3.10}$$

$$d_r = 1 + k_1 r + k_2^2 \tag{3.11}$$

$$\boldsymbol{d}_t = \begin{bmatrix} 2uvt_1 + (r + 2u^2)t_2 \\ 2uvt_2 + (r + 2v^2)t_1 \end{bmatrix} \tag{3.12}$$

Note that:

$$\boldsymbol{h}'\left(\begin{bmatrix}u\\v\end{bmatrix}\right) = \boldsymbol{h}\left(\begin{bmatrix}u\\v\\1\end{bmatrix}\right) \tag{3.13}$$

This is useful during implementation, since this makes it possible to reuse some functions.

This is not possible to do in closed form, but we can use an method called the *Gauss–Newton algorithm*.This is an iterative algorithm that tries to minimize the sum of squared residuals.

If we wish to find the direction of pixel $\boldsymbol{P} = \left[\begin{smallmatrix}x\\y\end{smallmatrix}\right]$ we start out with an initial guess of $\boldsymbol{\beta}_0 = \left[\begin{smallmatrix}u_0\\v_0\end{smallmatrix}\right]$. For our initial guess we assume that there is no lens distortion, thus:

$$P = \begin{bmatrix}x\\y\end{bmatrix} = \begin{bmatrix}o_x\\o_y\end{bmatrix} + \begin{bmatrix}f_x & 0\\0 & f_y\end{bmatrix}\begin{bmatrix}u_0\\v_0\end{bmatrix} \tag{3.14}$$

$$\Updownarrow$$

$$\boldsymbol{\beta}_0 = \begin{bmatrix}u_0\\v_0\end{bmatrix} = \begin{bmatrix}\frac{1}{f_x} & 0\\0 & \frac{1}{f_y}\end{bmatrix}\left(\begin{bmatrix}x\\y\end{bmatrix} - \begin{bmatrix}o_x\\o_y\end{bmatrix}\right) \tag{3.15}$$

The residual can be calculated:

$$\boldsymbol{r}_i(\boldsymbol{\beta}) = \boldsymbol{P} - \boldsymbol{h}(\boldsymbol{\beta}_i) \tag{3.16}$$

And we update our estimate:

$$\boldsymbol{\beta}_{i+1} = \boldsymbol{\beta}_i + \left(\mathbf{J}_{h'}^T\mathbf{J}_{h'}\right)^{-1}\mathbf{J}_{h'}^T\boldsymbol{r}(\boldsymbol{\beta}_i) \tag{3.17}$$

where the Jacobian $\mathbf{J}_{h'}$ is

$$\mathbf{J}_{h'} = \left.\frac{\partial \boldsymbol{h}'(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}}\right|_{\boldsymbol{\beta}_i} \tag{3.18}$$

One can chose to stop after the error becomes small enough, or after a fixed number of iterations.

3.2 Feature Tracking

When a computer is presented with an image and has to analyse it, we are presented with a problem; the image simply has too much information in it. A small image of 640pxpx (*VGA* resolution) has around $300,000$pixels, doing anything complicated with that amount of data on a computer would take far to long for real-time applications. One solution could be to scale the image down, but in doing so we risk to blur details in the image that might be useful. Another solution is to find regions, or points in the image, where "interesting stuff" is going on. A white wall does not give much information about the environment, but a dark spot on the wall could be traced for information about egomotion.

A large amount of algorithms for efficiently tracking what is known af features ("interesting" areas of an image) exist. We will concentrate on corner detectors, more specifically *Harris & Stephens Corner Detector* Harris and Stephens (1988) and *Shi-Tomasi Corner Detector* Shi and Tomasi (1994).

3.2.1 Harris & Stephens Corner Detector

Known in *OpenCV*[1] as *Harris Corner Detector* is a algorithm for finding points in an image where the image gradient changes a lot in both directions.

[1] Open Computer Vision is a popular open source computer vision library for C, C++, Phyton, Java

In the algorithm, for each pixel we estimate the *Structure tensor*, a 2×2 matrix that describes the predominant directions of the gradient in a neighborhood of pixels. This matrix is estimated as:

$$\mathbf{M} = \sum_{x,y} w(x,y) \begin{bmatrix} I_x I_x & I_x I_y \\ I_y I_x & I_y I_y \end{bmatrix} \tag{3.19}$$

where $\begin{bmatrix} I_x \\ I_y \end{bmatrix}$ is an estimate of the gradient (can be found with numerical integration) and $w(x,y)$ is a window function used for averaging, usually rectangular or Gaussian.

The eigenvalues λ_1, λ_2 of $\mathbf{M}$ are found.

- $\lambda_1 \approx 0, \quad \lambda_2 \approx 0$: If both are small, there is litte variation in intensity at that point at all.
- $\lambda_1 >> \lambda_2 \vee \lambda_2 >> \lambda_1$: If one of the eigenvalues are significantly larger than the other, the pixel is part of an edge.
- $\lambda_1 >> 0, \quad \lambda_2 >> 0$: If both eigenvalues are large it means that the gradient changes a lot in all directions and we have a corner.

Each pixel is scored as following:

$$R = \lambda_1 \lambda_2 - k(\lambda_1 + \lambda_2)^2 \tag{3.20}$$

where k is a tuning parameter usually in the range $[0.04; 0.15]$.

Either all points with a score over a set value are considered interesting, or the best n are chosen as being "interesting"

A problem with this algorithm is that points with large values of R usually group together in regions, where one of the pixel has the largest value of R. The *OpenCV* implementation deals with this problem by checking the neighborhood of each found feature and check if a pixel with a larger R value exists in its vicinity. If this is the case, the pixel with the lowest R is discarded.

An example of some features found with the *Harris Corner Detector* can be seen in figure 3.3

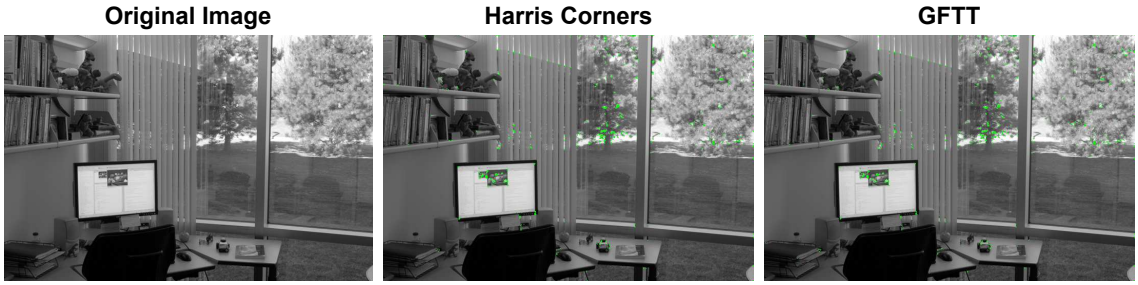

Figure 3.3: An example image of feature detection. In the two rightmost images the 0.5% of pixels with the largest R score are highlighted in green. Note that they lump together, which is why the non-maxima suppression is needed

Note that the *Harris Corner Detector* is not scale invariant. A blurred corner will have a low R, but if we zoom out, the change in intensity per pixel gets larger, thus the R value increases. On the other hand if we zoom out, some features might disappear because they get lost in spacial quantization.

3.2.2 Shi-Tomasi Corner Detector, Good Features to Track

In 1994, J. Shi and C. Tomasi made a modification to the *Harris Corner Detector* in their paper *Good Features To Track (GFTT)*. Instead of rating the features by:

$$R = \lambda_1\lambda_2 - k(\lambda_1 + \lambda_2)^2$$

they propossed to rate the features based on the minimal value of λ_1 and λ_2:

$$R = \min(\lambda_1, \lambda_2) \tag{3.21}$$

As with the *Harris Corner Detector* it is required to suppress features close to each other with a non maximal R.

The *OpenCV* implementation of *GFTT* allows the user to control the minimal distance between points. This both suppresses features with non maximal R and helps spread the features evenly over the image (you might have an image with very strong features in one area, setting the threshold for R so high that no other features get accepted)

3.2.3 Tracking Features Between Images

If we want to use our features for egomotion estimation, we need to keep track of which features in one image corresponds to which features in another image. This can be done in multiple ways.

Feature Description and Matching

One solution is to extract a small patch of the image around each feature. This patch can then be compared to features in the image we are matching against. Often some metric is derived from the patches to describe them, reducing the number of computations in the comparison.

This method, however, scales badly with the number of features. For each feature we want to mach, we have to search for the candidate that is most similar to it. Usually this involves calculating a score for each feature pair, and then sort the list of scores. If this is done with brute force the complexity rises with roughly the square of the number of features (depending on the used sorting algorithm).

Another problem is that it is not uncommon for features to be mismatched. This happens when both images have multiple patches that look alike. Thus we have to find and match more features than we need, so we can detect and throw away mismatches (outliers).

Optical Flow for Feature Tracking

Another option is to use optical flow to estimate the movement of a feature. When doing so, one needs only detect features in one image and then their movement from one image to the next will be estimated by optical flow. This can in some places be faster than finding features in both images and matching them with brute force. The amount of outliers due to mistracking is also reduced, thus a smaller amount of features need to be detected in the first place.

OpenCV includes an algorithm for efficiently estimating the optical flow of a few selected features (that is, it does not estimate the optical flow of all pixels, but only those provided to it), this is based on the *Lucas–Kanade Method* Lucas and Kanade (1981). This is an iterative algorithm, but stop conditions can be set to ensure bounded computation time, at the expense of lost accuracy.

3.3 Estimating Geometric Transformation

We can use the estimated motion of a set of features in one image to another, to estimate the motion of the camera. In this section we will show how a simple transformation can be estimated.

If given a set of n points in frame A: ${}^A\boldsymbol{p}_1, {}^A\boldsymbol{p}_2, \ldots, {}^A\boldsymbol{p}_n$ where each point is given by: ${}^A\boldsymbol{p}_i = \left[\begin{smallmatrix} {}^A x_i \\ {}^B y_i \end{smallmatrix}\right]$ a matrix $\mathbf{H}$ exists that transforms the points from frame A to frame B:

$$\begin{bmatrix} {}^B\boldsymbol{p}_1^T \\ \vdots \\ {}^B\boldsymbol{p}_n^T \end{bmatrix} = \begin{bmatrix} {}^A\boldsymbol{p}_1^T & 1 \\ \vdots & \vdots \\ {}^A\boldsymbol{p}_n^T & 1 \end{bmatrix} \mathbf{H} \tag{3.22}$$

If we let $\mathbf{H}$ correspond to a nonreflective similarity transformation (rotation, scaling, offset) it can be parametrize: $\mathbf{H} = \left[\begin{smallmatrix} h_1 & -h_2 \\ h_2 & h_1 \\ h_3 & h_4 \end{smallmatrix}\right]$. This lets us formulate the transformation:

$$\begin{bmatrix} {}^B x_1 & {}^B y_1 \\ \vdots & \vdots \\ {}^B x_n & {}^B y_n \end{bmatrix} = \begin{bmatrix} {}^A x_1 & {}^A y_1 & 1 \\ \vdots & \vdots & \vdots \\ {}^A x_n & {}^A y_n & 1 \end{bmatrix} \begin{bmatrix} h_1 & -h_2 \\ h_2 & h_1 \\ h_3 & h_4 \end{bmatrix} \tag{3.23}$$

This can be reformulated:

$$\begin{bmatrix} {}^B x_1 \\ {}^B y_1 \\ \vdots \\ {}^B x_n \\ {}^B y_n \end{bmatrix} = \begin{bmatrix} {}^A x_1 & {}^A y_1 & 1 & 0 \\ {}^A y_1 & -{}^A x_1 & 0 & 1 \\ \vdots & \vdots & \vdots & \vdots \\ {}^A x_n & {}^A y_n & 1 & 0 \\ {}^A y_n & -{}^A x_n & 0 & 1 \end{bmatrix} \begin{bmatrix} h_1 \\ h_2 \\ h_3 \\ h_4 \end{bmatrix} \tag{3.24}$$

If instead of knowing $\mathbf{H}$ we have a list of points in one frame and a list of the same points in another frame, we can estimate $\hat{\mathbf{H}}$. We do this by reformulating our transformation once more:

$$\mathbf{0} = \begin{bmatrix} {}^A x_1 & {}^A y_1 & 1 & 0 & -{}^B x_1 \\ {}^A y_1 & -{}^A x_1 & 0 & 1 & -{}^B y_1 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ {}^A x_n & {}^A y_n & 1 & 0 & -{}^B x_n \\ {}^A y_n & -{}^A x_n & 0 & 1 & -{}^B y_n \end{bmatrix} \begin{bmatrix} h_1 \\ h_2 \\ h_3 \\ h_4 \\ 1 \end{bmatrix} \tag{3.25}$$

$$= \begin{bmatrix} {}^A x_1 & {}^A y_1 & 1 & 0 & -{}^B x_1 \\ {}^A y_1 & -{}^A x_1 & 0 & 1 & -{}^B y_1 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ {}^A x_n & {}^A y_n & 1 & 0 & -{}^B x_n \\ {}^A y_n & -{}^A x_n & 0 & 1 & -{}^B y_n \end{bmatrix} \begin{bmatrix} h_1 y_1 \\ h_2 y_2 \\ h_3 y_3 \\ h_4 y_4 \\ y_5 \end{bmatrix} \tag{3.26}$$

This is a set of homogeneous equation of the form $\mathbf{A}\boldsymbol{y} = 0$. Usually we will have an overdefined et of equations, as this is the case when $n > 2$. When the system is overdefined we cannot find a solution to $\boldsymbol{y}$ but will instead find the solution than minimizes the sum of squares: $||\mathbf{A}\boldsymbol{y}||^2$

This is done by doing an *SVD* decomposition Wikipedia (2015). A solution for $\boldsymbol{y}$, $\hat{\boldsymbol{y}}$ that minimizes $||\mathbf{A}\boldsymbol{y}||^2$ is then the rightmost column of $\mathbf{V}$. Since multiple solutions exists for $\hat{\boldsymbol{y}}$ we divide the result by $\hat{y}_5$

thus:

$$\hat{\boldsymbol{Y}} = \frac{\hat{\boldsymbol{y}}}{\hat{y}_5} = \begin{bmatrix} \hat{y}_1/\hat{y}_5 \\ \hat{y}_2/\hat{y}_5 \\ \hat{y}_3/\hat{y}_5 \\ \hat{y}_4/\hat{y}_5 \\ \hat{y}_5/\hat{y}_5 \end{bmatrix} = \begin{bmatrix} \hat{Y}_1 \\ \hat{Y}_2 \\ \hat{Y}_3 \\ \hat{Y}_4 \\ 1 \end{bmatrix} \tag{3.27}$$

We can now stich $\hat{\mathbf{H}}$ together from $\hat{\boldsymbol{Y}}$:

$$\hat{\mathbf{H}} = \begin{bmatrix} \hat{Y}_1 & -\hat{Y}_2 \\ \hat{Y}_2 & \hat{Y}_1 \\ \hat{Y}_3 & \hat{Y}_4 \end{bmatrix} \tag{3.28}$$

CHAPTER 4

Visual Odometry

This chapter will deal with the design of a visual odometry algorithm for tracking egomotion. Two different algorithms were tried out, both having that in common that they rely on finding features in an image stream and using data form an inertial measurement unit, when no images are present.

4.1 Multi-State Constraint Kalman Filter

The *Multi-State Constraint Kalman Filter* is an algorithm for visual odometry presented by *Anastasios I. Mourikis and Stergios I. Roumeliotis* in their paper *A Multi-State Constraint Kalman Filter for Vision-aided Inertial Navigation* Mourikis and Roumeliotis (2006) and later refined in multiple papers and reports Li and Mourikis (2012) adding both rolling shutter compensation Li et al. (2013) and estimating calibration values for camera and *IMU* at run Shelley (2014).

The algorithm is supposed to be lightweight and attempts have been made to make it run in real-time on smartphone devices Li et al. (2013). For that reason, and because the results presented seemed adequate an attempt was made at implementing the algorithm, with the goal of using it as the main means of localisation. This turned out to be difficult, and the results were not satisfactory. We will, however, go through the the details of the algorithm as parts of it was used in the final visual odometry algorithm.

4.1.1 Theory of Operation

The algorithm uses a sliding widow of features in the latest N images and an inertial measurement unit. The state estimate is propagated by measurements from the *IMU*, that is linear acceleration along the axes of the *IMU* and angular rate from the *IMU* is integrated to yield position and attitude. Since integration error, and more significantly bias in measurements will lead to an unbounded error, measurements from the camera is used to update state estimate and also estimate the biases of sensors. In each image features are detected and traced between multiple images, this makes it possible to estimate the spacial location of the features, and from that estimate motion. In figure 4.1 a moving camera frame, capturing features and triangulating their position is seen.

The propagation and updating of the state estimates are done with an *Extended Kalman Filter*, like the *Kalman Filter* this is a computationally inexpensive algorithm that uses normal distributions to approximate the probability of the system state. The addition to the *EKF* is that it also works on nonlinear systems, this is done by linearizing the system around the current state estimate and using the linearized model for prediction and updating.

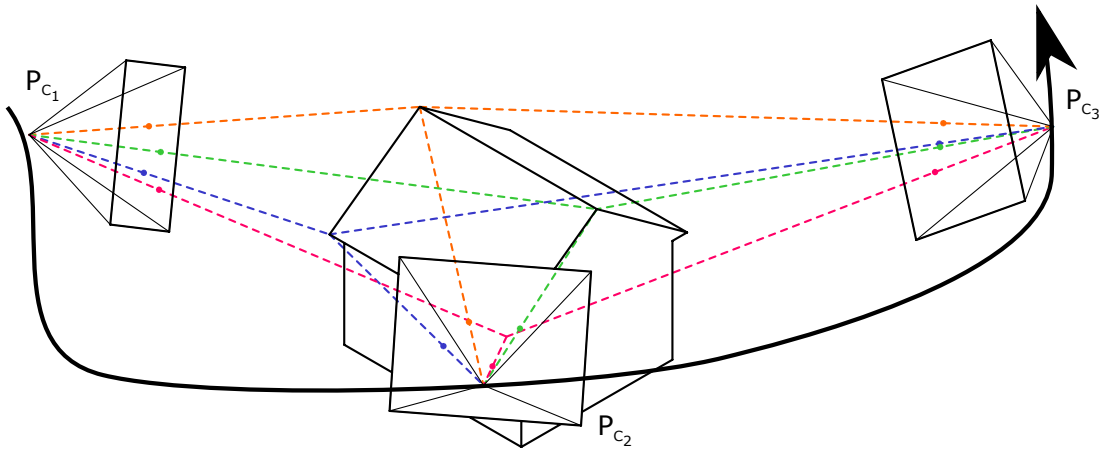

Figure 4.1: Illustration of a camera moving around and capturing features. The dotted lines represents how the spacial location of the features can be fund, using triangulation.

4.1.2 State Representation

Since multiple versions of the *MSCKF* we will now list the state representation used in this project. The state consists of two main parts: The current state, with attitude, position, velocity, bias estimates, and a list of previous states (attitude, velocity, position). We can write the state vector:

$$\boldsymbol{x}_k = \left[{}^I_G\boldsymbol{q}^T, {}^G\boldsymbol{p}_I^T, {}^G\boldsymbol{v}_I^T, \boldsymbol{b}_g^T, \boldsymbol{b}_a^T, \boldsymbol{\pi}_{N-m}^T \dots \boldsymbol{\pi}_{N-1}^T\right]^T \tag{4.1}$$

where $\boldsymbol{\pi}_n$ is the state estimate at the time of image n:

$$\boldsymbol{\pi}_n = \left[{}^I_G\boldsymbol{q}_n^T, {}^G\boldsymbol{p}_{I_n}^T, {}^G\boldsymbol{v}_{I_n}^T\right] \tag{4.2}$$

Error Representation

While the majority of the state estimate errors (that is the difference between the real state and the estimated) can be evaluated with a subtraction: ${}^G\tilde{\boldsymbol{p}}_I = {}^G\boldsymbol{p}_I - {}^G\hat{\boldsymbol{p}}_I$, this is not the case for the attitude. As quaternion representation is used the quaternion must remain unity $\|\boldsymbol{q}\| = 1$. The attitude error can be represented with a quaternion $\delta\boldsymbol{q}$ such that: ${}^I_G\boldsymbol{q} = {}^I_G\hat{\boldsymbol{q}} \otimes \delta\boldsymbol{q}$. Since the error quaternion represents a small rotation this can be reparameterized: $\delta\boldsymbol{q} \approx [\frac{1}{2}\delta\boldsymbol{\theta}^T, 1]$. The error representation is thus:

$$\tilde{\boldsymbol{x}} = [{}^G\delta\boldsymbol{\theta}^T, {}^G\tilde{\boldsymbol{p}}_I^T, {}^G\tilde{\boldsymbol{v}}_I^T, {}^G\tilde{\boldsymbol{b}}_g^T, {}^G\tilde{\boldsymbol{b}}_a^T, \tilde{\boldsymbol{\pi}}_{N-m}^T \dots \tilde{\boldsymbol{\pi}}_{N-1}^T] \tag{4.3}$$

$$\tilde{\boldsymbol{\pi}}_n = [{}^G\delta\boldsymbol{\theta}_n^T, {}^G\tilde{\boldsymbol{p}}_{I_n}^T {}^G\tilde{\boldsymbol{v}}_{I_n}^T] \tag{4.4}$$

Calibration Values

In the implementation of *MSCKF* in Shelley (2014) which was used as the base for our implementation many parameters are estimated in run-time. In our implementation a lot of these are left as predetermined calibration parameters, mainly to simplify the implementation and partially to make debugging easier, as this also makes the state smaller. Calibration variables are:

Camera Calibration

$o_x, \quad o_y$	Principal point	(4.5)
$f_x, \quad f_y$	Focal lengths	(4.6)
$k_1, \quad k_2$	Radial distortion parameters	(4.7)
$t_1, \quad t_2$	Tangential distortion parameters	(4.8)
${}^{C}_{I}\boldsymbol{q}$	Rotation from inertial frame to camera frame	(4.9)
${}^{C}\boldsymbol{p}_I$	Position of origo in inertial frame in camera coordinates	(4.10)

This does not include measurement and system noise variance estimates.

Note that in the Shelley (2014) implantation the matrices $\mathbf{T}_g$, $\mathbf{T}_a$ and $\mathbf{T}_s$ are estimate, while in our implementation $\mathbf{T}_g = \mathbf{T}_a = \mathbf{I}_3$ and $\mathbf{T}_g = \mathbf{0}_3$. $\mathbf{T}_g$ and $\mathbf{T}_a$ describes the alignment of gyroscope and accelerometer axes inside the *IMU* and $\mathbf{T}_g$ is a matrix describing the effect of gravity on the gyroscope.

4.1.3 Propagation

Propagation is done by numerical integration of accelerometer and gyro measurements from the *IMU*. We will only state the equations used for propagation and not their derivation as this can be found in Shelley (2014)

State Propagation

The state propagation is based on numerical integration of the continuous differential equations:

$$
\begin{aligned}
{}^{I}_{G}\dot{\boldsymbol{q}}(t) &= \frac{1}{2}\boldsymbol{\Omega}({}^{I}\boldsymbol{\omega}(t)){}^{I}_{G}\boldsymbol{q}(t) && (4.11)\\
\boldsymbol{\Omega}(\boldsymbol{\omega}) &= \begin{bmatrix} -\lfloor\boldsymbol{\omega}\times\rfloor & \boldsymbol{\omega} \\ -\boldsymbol{\omega}^T & 0 \end{bmatrix} && (4.12)\\
{}^{G}\dot{\boldsymbol{p}}(t) &= {}^{I}\boldsymbol{v}(t) && (4.13)\\
{}^{G}\dot{\boldsymbol{v}}(t) &= {}^{G}\boldsymbol{a}(t) && (4.14)\\
{}^{G}\dot{\boldsymbol{b}_g}(t) &= {}^{G}\boldsymbol{n}_{w_g}(t) && (4.15)\\
{}^{G}\dot{\boldsymbol{b}_a}(t) &= {}^{G}\boldsymbol{n}_{w_a}(t) && (4.16)\\
& && (4.17)
\end{aligned}
$$

Rotational Dynamics The measured angular rate is ${}^{I}\boldsymbol{\omega}_m$. The angular rate is integrated with the *fourth order Runge-Kutta method* and the resulting quaternion is normalized to ensure unity though the accumulation

of rounding errors that would otherwise occur.

$$
\begin{aligned}
{}^{I}\hat{\boldsymbol{\omega}} &= {}^{I}\boldsymbol{\omega}_m - \hat{\boldsymbol{b}}_g && (4.18)\\
{}^{I_{l+1}}_{I_l}\hat{\boldsymbol{q}} &= \boldsymbol{q}_0 + \frac{\Delta t}{6}(\boldsymbol{k}_1 + 2\boldsymbol{k}_2 + 2\boldsymbol{k}_3 + \boldsymbol{k}_4) && (4.19)\\
\boldsymbol{q}_0 &= [0,\ 0,\ 0,\ 1]^T && (4.20)\\
\boldsymbol{k}_1 &= \frac{1}{2}\boldsymbol{\Omega}(\hat{\boldsymbol{\omega}}_l)\boldsymbol{q}_0 && (4.21)\\
\boldsymbol{k}_2 &= \frac{1}{2}\boldsymbol{\Omega}\left(\frac{\hat{\boldsymbol{\omega}}_l + \hat{\boldsymbol{\omega}}_{l+1}}{2}\right)\left(\boldsymbol{q}_0 + \frac{\Delta t}{2}\boldsymbol{k}_1\right) && (4.22)\\
\boldsymbol{k}_3 &= \frac{1}{2}\boldsymbol{\Omega}\left(\frac{\hat{\boldsymbol{\omega}}_l + \hat{\boldsymbol{\omega}}_{l+1}}{2}\right)\left(\boldsymbol{q}_0 + \frac{\Delta t}{2}\boldsymbol{k}_2\right) && (4.23)\\
\boldsymbol{k}_4 &= \frac{1}{2}\boldsymbol{\Omega}(\hat{\boldsymbol{\omega}}_l)(\boldsymbol{q}_0 + \Delta t\boldsymbol{k}_3) && (4.24)\\
{}^{I_{l+1}}_{G}\hat{\boldsymbol{q}} &= \frac{{}^{I_{l+1}}_{I_l}\hat{\boldsymbol{q}} \otimes {}^{I_l}_{G}\hat{\boldsymbol{q}}}{\|{}^{I_{l+1}}_{I_l}\hat{\boldsymbol{q}} \otimes {}^{I_l}_{G}\hat{\boldsymbol{q}}\|} && (4.25)
\end{aligned}
$$

Translational Dynamics The measured acceleration ${}^{I}\boldsymbol{a}_m$ is converted to global frame and the gravity vector is subtracted to get an estimate of the actual acceleration. This acceleration is then integrated to yield velocity and position estimates:

$$
\begin{aligned}
{}^{I}\hat{\boldsymbol{a}} &= {}^{I}\boldsymbol{a}_m - \hat{\boldsymbol{b}}_a && (4.26)\\
{}^{G}\hat{\boldsymbol{a}} &= {}^{G}_{I}\hat{\boldsymbol{q}}\,{}^{I}\hat{\boldsymbol{a}} + {}^{G}\boldsymbol{g} && (4.27)\\
{}^{G}\boldsymbol{g} &= [0,\ 0,\ -g]^T && (4.28)\\
{}^{G}\hat{\boldsymbol{v}}_{l+1} &= {}^{G}\hat{\boldsymbol{v}}_l + {}^{G}_{I_l}\hat{\boldsymbol{q}}\hat{\boldsymbol{s}}_l + {}^{G}\boldsymbol{g}\Delta t && (4.29)\\
{}^{G}\hat{\boldsymbol{p}}_{l+1} &= {}^{G}\hat{\boldsymbol{p}}_l + {}^{G}\hat{\boldsymbol{v}}_l\Delta t + {}^{G}_{I_l}\hat{\boldsymbol{q}}\hat{\boldsymbol{y}}_l + \frac{1}{2}{}^{G}\boldsymbol{g}\Delta t^2 && (4.30)\\
\hat{\boldsymbol{s}} &= \frac{\Delta t}{2}\left({}^{I_l}_{I_{l+1}}\hat{\boldsymbol{q}}(\boldsymbol{a}_m(t_{l+1}) - \hat{\boldsymbol{b}}_a) + (\boldsymbol{a}_m(t_l) - \hat{\boldsymbol{b}}_a)\right) && (4.31)\\
\hat{\boldsymbol{y}} &= \frac{\Delta t}{2}\hat{\boldsymbol{s}} && (4.32)
\end{aligned}
$$

Error Propagation and Noise Matrix During propagation the noise on the measurements makes the uncertainty of the state estimate increase, but this is only for the current state estimate, since we are not changing the estimate of the previous states ($\hat{\boldsymbol{\pi}}_n$) the covariance of these are also left untouched. This gives us the covariance update:

$$
\boldsymbol{\Sigma}_{l+1} = \begin{bmatrix} \boldsymbol{\Phi}_{I_l} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{I} \end{bmatrix} + \begin{bmatrix} \mathbf{Q}_{dl} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} \end{bmatrix} \tag{4.33}
$$

where Φ_{I_l} is the error propagation matrix. For derivation and equations for entries in the error propagation matrix see Shelley (2014)

The noise matrix $\mathbf{Q}_t$ is given by:

$$Q_d = \frac{\Delta t}{2}\mathbf{\Phi}_{I_l}\mathbf{G}_c\mathbf{Q}_c\mathbf{G}_c^T\mathbf{\Phi}_{I_l}^T \tag{4.34}$$

$$\mathbf{Q}_c = \begin{bmatrix} \sigma_{g_c}^2\mathbf{I}_3 & \mathbf{0}_3 & \mathbf{0}_3 & \mathbf{0}_3 \\ \mathbf{0}_3 & \sigma_{a_c}^2\mathbf{I}_3 & \mathbf{0}_3 & \mathbf{0}_3 \\ \mathbf{0}_3 & \mathbf{0}_3 & \sigma_{wg_c}^2\mathbf{I}_3 & \mathbf{0}_3 \\ \mathbf{0}_3 & \mathbf{0}_3 & \mathbf{0}_3 & \sigma_{wa_c}^2\mathbf{I}_3 \end{bmatrix} \tag{4.35}$$

$$\mathbf{G}_c = \begin{bmatrix} -\mathbf{I}_3 & \mathbf{0}_3 & \mathbf{0}_3 & \mathbf{0}_3 \\ \mathbf{0}_3 & \mathbf{0}_3 & \mathbf{0}_3 & \mathbf{0}_3 \\ \mathbf{0}_3 & -{}^{I_l}_{G}\hat{\mathbf{R}}^T & \mathbf{0}_3 & \mathbf{0}_3 \\ \mathbf{0}_3 & \mathbf{0}_3 & \mathbf{I}_3 & \mathbf{0}_3 \\ \mathbf{0}_3 & \mathbf{0}_3 & \mathbf{0}_3 & \mathbf{I}_3 \end{bmatrix} \tag{4.36}$$

Where $\mathbf{Q}_c$ is the continuous noise of the measurements and $\mathbf{G}_c$ maps the noise to the error state.

4.1.4 Update

Heigth measurement

In the original *MSCKF* algorithm an absolute measurement of the altitude is not available, but since we have access to an ultrasonic distance sensor, used for altitude measurements, this sensor is used for updated as well. The altitude measurements are much more frequent than camera measurements, and updates from altitude measurements are performed independently from camera measurements.

The altitude measurement has the measurement Jacobian:

$$\mathbf{H}_a = [\underbrace{0,\ 0,\ 0}_{\text{attitude}},\ \underbrace{0,\ 0,\ 1}_{\text{position}},\ \underbrace{0,\ 0,\ 0}_{\text{velocity}},\ \underbrace{0 \ldots 0}_{\text{rest}}] \tag{4.37}$$

The technique described in section 2.3.1 is used for outlier detection.

Camera Measurement

The camera update step can be split into multiple parts

State Augmentation First the current state (attitude, position, velocity) is appended to the state vector ant the covariance matrix is augmented accordingly:

$$\mathbf{\Sigma} \leftarrow \begin{bmatrix} \mathbf{\Sigma} & \mathbf{\Sigma}\mathbf{J}_\pi^T \\ \mathbf{J}_\pi\mathbf{\Sigma} & \mathbf{J}_\pi\mathbf{\Sigma}\boldsymbol{J}_\pi^T \end{bmatrix} \tag{4.38}$$

$$\mathbf{J}_\pi = \begin{bmatrix} \mathbf{I}_3 & \mathbf{0}_3 & \mathbf{0}_3 & \mathbf{0}_3 & \ldots \\ \mathbf{0}_3 & \mathbf{I}_3 & \mathbf{0}_3 & \mathbf{0}_3 & \ldots \\ \mathbf{0}_3 & \mathbf{0}_3 & \mathbf{I}_3 & \mathbf{0}_3 & \ldots \end{bmatrix} \tag{4.39}$$

Feature Triangulation The estimates of previous positions and feature pairs from these are used to estimate the spacial location of each feature as illustrated in figure 4.1. A feature is used for the update if it is "lost", that is if the feature was not detected in the current image. To ensure bounded computation time, all features older than some defined number of frames is considered lost. Each feature is then reprojected onto a virtual camera, using the supplied camera distortion model and the residual between the measured feature position and the reprojected feature position is calculated. The residual of feature i in camera frame

j is $\boldsymbol{r}_{i,j} = \boldsymbol{z}_{i,j} - \hat{\boldsymbol{z}}_{i,j}$. For each feature the Jacobian with respect to the state $\mathbf{H}_{xi,j}$ and with respect to the feature position $\mathbf{H}_{f\,i,j}$ is also calculated. The residuals of each feature is stacked to get r_i, likewise for the Jacobians:

$$\boldsymbol{r}_i = [\boldsymbol{r}_{i,I_0}^T \dots \boldsymbol{r}_{i,I_n}^T]^T \tag{4.40}$$

$$\approx \mathbf{H}_{xi}\tilde{\boldsymbol{x}} + \mathbf{H}_{f\,i}{}^G\tilde{\boldsymbol{p}}_{f_i} + \boldsymbol{n}_i \tag{4.41}$$

Feature Error Marginalization Since the residual both carries information about the state error and the error in the estimate of the feature position ${}^G\tilde{\boldsymbol{p}}_f$, the feature error is marginalized out. This is done by multiplying with a matrix $\mathbf{A}_i$ that is the left nullspace of $\mathbf{H}_{f\,i}$:

$$\mathbf{A}_i^T\boldsymbol{r}_i \approx \mathbf{A}_i^T\mathbf{H}_{xi}\tilde{\boldsymbol{x}} + \mathbf{A}_i^T\mathbf{H}_{f\,i}{}^G\tilde{\boldsymbol{p}}_{f_i} + \mathbf{A}_i^T\boldsymbol{n}_i \tag{4.42}$$

$$\boldsymbol{r}_i^0 \approx \mathbf{H}_i^0\tilde{\boldsymbol{x}} + \boldsymbol{n}_i^0 \tag{4.43}$$

$$\mathbf{A}_i^T\mathbf{H}_{f\,i}{}^G\tilde{\boldsymbol{p}}_{f_i} = 0 \tag{4.44}$$

Outlier Detection For each feature a χ^2*-test* is conducted as described in section 2.3.1. For all feature that are not outliers the residual and the corresponding Jacobian is added together to form the final residual vector $\boldsymbol{r}^0$ and Jacobian matrix $\mathbf{H}^0$.

EKF Update An update of the state estimate is then performed, using the marginalized residual vector and Jacobian matrix.

Removing Unused States For all states in the sliding window in which the tracing of all features is lost, the state is removed from the state and covariance matrix.

4.1.5 Constraining Features on the Ground

Since the algorithm relies on estimating the spacial location of the features, this requires that the camera moves significantly, otherwise the depth of the feature along the z-axis of the camera cannot be estimated accurately. This is a problem, if the goal is to use the algorithm for localization of a quadrotor, as this would make hovering steadily impossible.

One solution is to make the camera face the ground and assume that the ground is flat. By assuming that all features are on the ground, we get an estimate of their distance to the camera, even when the camera is not moving.

4.1.6 Results

The algorithm was implemented in *Matlab* to evaluate it's performance and to check if it actually works. When this was confirmed, a *C++* implementation was done on the on-board computer of the quadrotor. The algorithm was, however, not capable of providing satisfactory results. The most likely explanation is, that it simply ran too slow on the embedded platform.

4.2 Ground Projected Features Visual Odometry

As the *MSCKF* algorithm did not perform satisfactory a new algorithm was designed. This was done with the main goal of being computationally inexpensive.

4.2.1 Theory of Operation

As with the *MSCKF* the *GPFVO* algorithm works by approximating the *pdf* of the state with normal distributions and using an *EKF* to estimate the state. Accelerometer and gyroscope measurements are used to propagate the state estimate, while features detected in camera images are used to update the state estimate.

Features are detected and tracked between the current and the previous image. For both images the features are projected onto the ground, using the estimated current state, and estimated state at the time of the previous image. After projecting the features onto the ground, the geometric transformation between the two sets of features is estimated and used as a measurement in the update.

4.2.2 State Representation and Propagation

The state is represented in a similar way as in the *MSCKF* with the exception that only the state at the time of the previous image is kept in the fame *FIFO*:

$$\boldsymbol{x}_k = \left[{}^{I}_{G}\boldsymbol{q}^T, {}^{G}\boldsymbol{p}_I^T, {}^{G}\boldsymbol{v}_I^T, \boldsymbol{b}_g^T, \boldsymbol{b}_a^T, {}^{I}_{G}\boldsymbol{q}_p^T, {}^{G}\boldsymbol{p}_{I_p}^T, {}^{G}\boldsymbol{v}_{I_p}^T \right]^T \tag{4.45}$$

Likewise the state propagation is the same as described in section 4.1.3.

4.2.3 Update

Height Measurement

The state estimate is updated based on altitude measurements in the same way as described in section 4.1.4

Camera Measurement

When a new image is recorded features form the previous image are tracked into the current using optical flow.

Undistorting Features The features in the previous image Im_p and the current Im are undistorted, according to the supplied camera model. This is done to find the direction of the feature in the camera frame

$$\boldsymbol{\theta}_f = \begin{bmatrix} \frac{{}^{C}X}{{}^{C}Z} \\ \frac{{}^{C}Y}{{}^{C}Z} \end{bmatrix} \tag{4.46}$$

where

$${}^{C}p_{f_i} = \begin{bmatrix} {}^{C}X \\ {}^{C}Y \\ {}^{C}Z \end{bmatrix} \tag{4.47}$$

The undistorted feature can be found using technique described in section 3.1.2.

Projecting Features on Ground The features are projected onto the ground, in a coordinate system with origo directly under the inertial frame. The features can be projected by first calculation the position of the camera at the time the feature was captured:

$${}^{G'}\boldsymbol{p}_I = \left[\begin{smallmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{smallmatrix}\right] {}^{G}\boldsymbol{p}_I \tag{4.48}$$

$${}^{C}_{G}\boldsymbol{q} = {}^{C}_{I}\boldsymbol{q} \otimes {}^{I}_{G}\boldsymbol{q} \tag{4.49}$$

$${}^{G}_{C}\boldsymbol{q} = {}^{C}_{G}\boldsymbol{q}^* \tag{4.50}$$

$${}^{G'}\boldsymbol{p}_C = {}^{G'}\boldsymbol{p}_I - {}^{G}_{C}\boldsymbol{q}{}^{C}\boldsymbol{p}_I \tag{4.51}$$

The direction of the feature is given by ${}^{C}\boldsymbol{\theta}_f = \left[\begin{smallmatrix}\boldsymbol{\theta}_f \\ 1\end{smallmatrix}\right]$, in global coordinates this becomes:

$$ {}^{G}\boldsymbol{\theta} = {}^{G}_{C}\boldsymbol{q}\,{}^{C}\boldsymbol{\theta}_f \tag{4.52} $$

The features position can now be found by solving the equation

$$ {}^{G'}\boldsymbol{p}_C + t\,{}^{G}\boldsymbol{\theta}_f = \begin{bmatrix}0\\0\\0\end{bmatrix} \tag{4.53} $$

yeldig the feature position ${}^{G'}\boldsymbol{p}_f$

This is done for both features in the current state I and for features in the state at the time of the previous image I_p.

Estimating Geometric Transformation The geometric transformation between the points ${}^{G'}\boldsymbol{p}_{f_i}$ and ${}^{G'_p}\boldsymbol{p}_{fp_i}$ can now be estimated with the method given in section 3.3. Outliers are detected by using *Random sample consensus (RANSAC)* to iterate over the estimated transformation and reject outliers.

The transformation matrix $\mathbf{A}$ is estimated such that

$$ \begin{bmatrix} {}^{G'}\boldsymbol{p}^T_{f_1} & 1 \\ \vdots & \vdots \\ {}^{G'}\boldsymbol{p}^T_{f_n} & 1 \end{bmatrix} \mathbf{A} = \begin{bmatrix} {}^{G'_p}\boldsymbol{p}^T_{fp_1} \\ \vdots \\ {}^{G'_p}\boldsymbol{p}^T_{fp_n} \end{bmatrix} \tag{4.54} $$

In figure 4.2 it is illustrated how this matrix corresponds to a movement of the camera.

The transformation matrix $\mathbf{A} = \left[\begin{smallmatrix} a_1 & -a_2 \\ a_2 & a_1 \\ a_3 & a_4 \end{smallmatrix}\right]$ this can be split into a transformation, scaling and rotation component. The translation component is $\boldsymbol{t}_m = \left[\begin{smallmatrix} a_3 \\ a_4 \end{smallmatrix}\right]$, the rotation component is $\theta_m = \arctan 2(a_2, a_1)$, and the scale component is $s_m = \sqrt{a_1^2 + a_2^2}$. The scale component is not used.

EKF Update The measured movement is compared to the estimated movement. The estimated translation can be calculated: $\boldsymbol{t}_e = \left[\begin{smallmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{smallmatrix}\right] \left({}^{G}\boldsymbol{p}_I - {}^{G}\boldsymbol{p}_{I_p}\right)$ The estimated rotation can be calculated by calculating the rotation between the x axis in the previous inertial frame and the current:

$$ {}^{I}\boldsymbol{x} = \begin{bmatrix}1\\0\\0\end{bmatrix} \tag{4.55} $$

$$ {}^{I_p}\boldsymbol{x} = \begin{bmatrix}1\\0\\0\end{bmatrix} \tag{4.56} $$

$$ \boldsymbol{a} = \begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\end{bmatrix} {}^{G}_{I}\boldsymbol{q}\,{}^{I}\boldsymbol{x} \tag{4.57} $$

$$ \boldsymbol{b} = \begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\end{bmatrix} {}^{G}_{I_p}\boldsymbol{q}\,{}^{I_p}\boldsymbol{x} \tag{4.58} $$

$$ \theta_e = \arctan 2(\det(\boldsymbol{a}, \boldsymbol{b}), \boldsymbol{a} \cdot \boldsymbol{b}) \tag{4.59} $$

The measurement residual can now be constructed:

$$ \boldsymbol{r} = \begin{bmatrix} \theta_m - \theta_e \\ \boldsymbol{t}_m - \boldsymbol{t}_e \end{bmatrix} \tag{4.60} $$

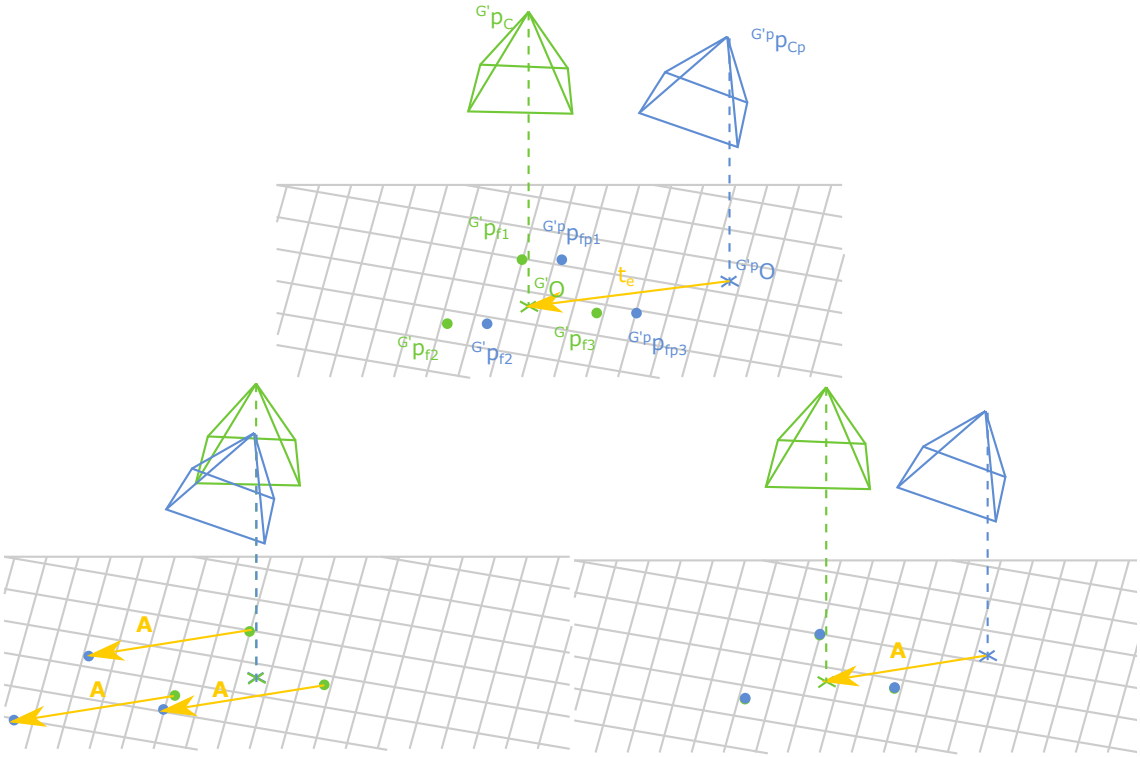

Figure 4.2: *(Top):* The previous state estimate in blue and the current state estimate in green as estimated after propagation. *(Bot. Left):* The movement of the features as seen form the camera is estimated. *(Bot. Right):* This corresponds to a movement of the camera, that is the same, as that of the features from the current t the previous image

The measurement has the Jacobian:

$$\mathbf{H} = \begin{bmatrix} \mathbf{H}_\theta \\ \mathbf{H}_x \\ \mathbf{H}_y \end{bmatrix} \tag{4.61}$$

$$\mathbf{H}_\theta = [\underbrace{0,\ 0,\ 1}_{\text{attitude}},\ \underbrace{0,\ 0,\ 0}_{\text{position}},\ \underbrace{0,\ 0,\ 0}_{\text{velocity}},\ \underbrace{0,\ 0,\ 0,\ 0,\ 0,\ 0}_{\text{bias est.}},\ \underbrace{0,\ 0,\ -1}_{\text{prev. att.}},\ \underbrace{0,\ 0,\ 0}_{\text{prev. pos.}},\ \underbrace{0,\ 0,\ 0}_{\text{prev. vel}}] \tag{4.62}$$

$$\mathbf{H}_x = [\underbrace{0,\ 0,\ 0}_{\text{attitude}},\ \underbrace{1,\ 0,\ 0}_{\text{position}},\ \underbrace{0,\ 0,\ 0}_{\text{velocity}},\ \underbrace{0,\ 0,\ 0,\ 0,\ 0,\ 0}_{\text{bias est.}},\ \underbrace{0,\ 0,\ 0}_{\text{prev. att.}},\ \underbrace{-1,\ 0,\ 0}_{\text{prev. pos.}},\ \underbrace{0,\ 0,\ 0}_{\text{prev. vel}}] \tag{4.63}$$

$$\mathbf{H}_y = [\underbrace{0,\ 0,\ 0}_{\text{attitude}},\ \underbrace{0,\ 1,\ 0}_{\text{position}},\ \underbrace{0,\ 0,\ 0}_{\text{velocity}},\ \underbrace{0,\ 0,\ 0,\ 0,\ 0,\ 0}_{\text{bias est.}},\ \underbrace{0,\ 0,\ 0}_{\text{prev. att.}},\ \underbrace{0,\ -1,\ 0}_{\text{prev. pos.}},\ \underbrace{0,\ 0,\ 0}_{\text{prev. vel}}] \tag{4.64}$$

CHAPTER 5

Implementation

This section will deal with the implementation of the visual odometry and the controller, further also the mechanical construction and electronic components will be described.

5.1 Hardware Description

Most of the hardware for this project is custom made, and while the frame and flight controller PCB was constructed prior to this project a short description will be given. The design was made with size, as well as safety in mind.

A block diagram of the electric wiring and the components on the quadrotor in use is illustrated in figure 5.1

Figure 5.1: Block diagram illustrating wiring between elements on the quadrotor. Only modules in use in the project are illustrated (UARTs, JTAG, CAN bus, I2C bus, and Barometric sensor omitted)

The major components of the quadrotor is:

- 4 **EMAX MT1806-2280**[1] brushless DC motors. Two clockwise rotating and two counter clockwise rotating.
- 4 **EMAX Simon Series 12A**[2] brushless DC motor drivers.
- 4 **Gemfan 5030**[3] propellers, nylon glass fibre mix. Two CW and two CCW.
- **Parallax PING))**[4] Ultrasonic Distance Sensor.
- **SONY PlayStation 3 Eye (PS3Eye)**[5] High framerate, uncompressed VGA and QVGA webcam.
- **Hardkernel oDroid-U3 v0.5**[6] Miniature Linux computer. Based on quad-core 1.7 GHz *Samsung Exynos4412 Prime Cortex-A9* processor with 2 Gb RAM.

[1] http://www.emaxmodel.com/emax-multicopter-motor-mt1806.html

[2] http://www.emaxmodel.com/emax-simon-series-12a-for-multirotor.html, on the contrary to what the name suggests, these are not based on the open source driver by *Simon K*, but *BLHeli*: https://github.com/bitdump/BLHeli

[3] http://www.gemfanhobby.com/Product/Main/En/2c6668b7-39ca-4e98-a7e0-a48300fc66c5

[4] https://www.parallax.com/product/28015

[5] http://uk.playstation.com/ps3/news/articles/detail/item85177/Introducing-PlayStation-Eye/

[6] http://www.hardkernel.com/main/products/prdt_info.php?g_code=G138745696275

- **Custom Flight Controller** based on *Texas Instruments Tiva series*[7] processor (*ARM Cortex-M4F*) with an *Invensense MPU9250 IMU*[8]
- **Texas Instruments CC1101**[9] based 433 MHz based radio transceiver.

5.1.1 Mechanical Design

The quadrotor frame is the mechanical hardware holding all the parts together. The frame is shaped as an *X* made from $8.8\,\mathrm{mm}\emptyset$ carbon fiber tubes held together by a 3D-printed center piece from *ABS*-plastic. The total weight of the quadrotor with battery is $\approx 650\,\mathrm{g}$

The frame is surrounded by a guard of 3D-printed quarter-pipes connected by $2\,\mathrm{mm}$ carbon fiber rods. The guard protects the quadrotor if it should crash, either into objects or the ground. The guard also protects the environment from the quadrotor, as there is only direct access to the propellers from the top and bottom.

At the center of each arm in the *X* a motor and propeller is located, on the arm the motor driver is located. The driver is located so the air stream form the propeller helps to cool it down. In the center of the *X* a tower with battery, power supply, flight controller, Linux computer and sensors is located.

A picture of the quadrotor can be seen in figure 5.2

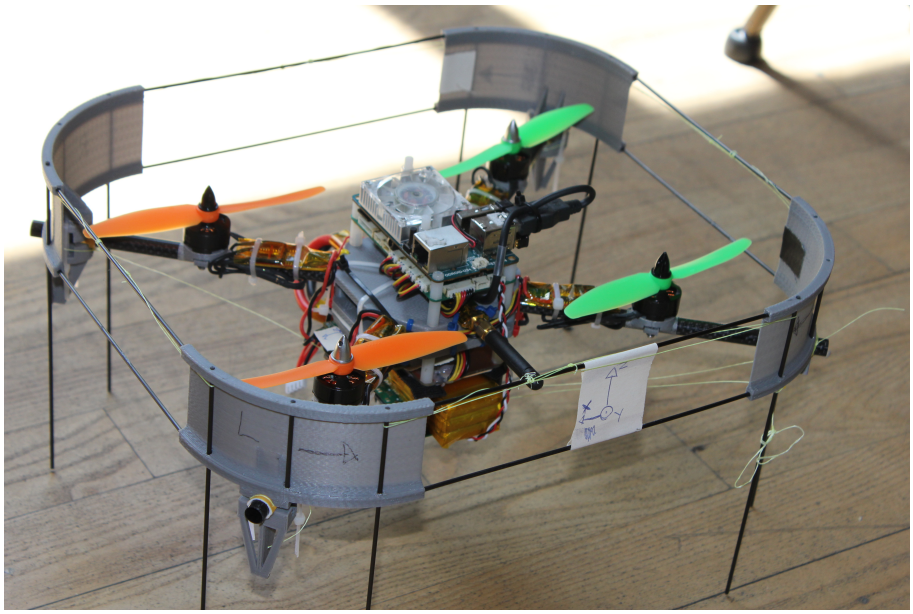

Figure 5.2: The mechanical construction of the quadrotor. In the center is a"tower" with *oDroid*, flight controller, power supply and battery. The antenna for the 433 MHz radio can be seen sticking out to the side.

The frame is slightly rectangular, this allows for the frame to be slim enough to pass through narrow passages with as much free space on each side, while sill allowing space for electronics and battery to fit in between the propellers.

Motors and propellers are the *EMAX MT1806* motors and *Gemfan 5030*. The propellers are $5'' \approx 12.7\,\mathrm{cm}$ in diameter and each motor/propeller can provide $\approx 350\,\mathrm{g} \approx 3.4\,\mathrm{N}$ of thrust, for at total of $\approx 1.4\,\mathrm{kg} \approx 13.7\,\mathrm{N}$. This allows the quadrotor to have enough thrust to both lift itself and have enough surplus thrust to maneuver.

[7] http://www.ti.com/product/tm4c123gh6pm
[8] http://www.invensense.com/products/motion-tracking/9-axis/mpu-9250/
[9] http://www.ti.com/product/cc1101

5.1.2 Electronics

The electronic hardware can be categorized in four categories:

1. Power electronics: Motors, motor drivers, battery, and power supply.
2. Sensors.
3. Low level flight controller: Altitude and attitude, sensor interface, radio interface.
4. High level controller: Visual odometry, position controller, WIFI telemetry.

Flight Controller

The quadrotor has a flight controller that takes care of all low-level sensor data gathering and outputting commands to the 4 motors. The flight controller is in constant contact with a remote over $433\,\mathrm{MHz}$ radio. The flight controller sends important data, such as battery level back to the remote. The remote ccontinuously sends a "keep alive" signal back to the flight controller. This signal works as a dead man's switch, if it disappears for any reason for more than $0.5\,\mathrm{s}$ the motors will shut off and the quadrotor will emergency crash land.

Since the flight controller has all the necessary sensors to estimate altitude and attitude, these are estimated on the flight controller and the quadrotor can be controlled manually over the remote in case the Linux computer stops working.

The flight controller can also send and receive data to the high level controller over a fast *SPI* interface. The user can control from the remote which inputs should be used for the controller: The ones from the remote joysticks or the ones from the high level controller. This enables the user to take manual control over the quadrotor, in case of a malfunction in the high-level controller. A diagram of the main elements of the flight controller software can be seen in figure 5.3

Figure 5.3: Block diagram illustrating wiring between elements on quadrotor. Only modules in use in the project are illustrated (UARTs, JTAG, CAN bus, I2C bus, and Barometric sensor omitted)

Remote

The remote is based on a *Texas Instruments Tiva C Launchpad* with the same processor as the flight controller. The processor is connected to two analog joysticks providing input for pitch, yaw, roll and alititude set-point. The remote also has a small *OLED* display with 128×64 pixels for displaying data. When the remote has a connection to the flight controller the display is showing in order:

- **U:** Voltage on the battery. Used to decide when the battery is empty.
- **I:** Current draw from battery. Currently this displays $0\,\mathrm{A}$ as the current sensor is not connected.
- **z:** Altitude estimate on flight controller.
- **R:** Recieved signal strength indicator (RSSI). Estimated signal strenth in $\mathrm{dB\,mW}$. The antennas used has some few dead spots, this can be used to avoid hitting those.
- **Q:** Quality of received signal. Smaller is better. This is a relative indication of the difficulty in demodulating the signal. This gives an idea about how much noise or other signals there are in the same band.

The remote is powered by a USB micro cable and can transmit all data that it receives over telemetry to a PC by *USB Bulk Transfer* or *USB Serial Port @ 115200baud*. If connected by serial port similar data to the data in the display will be shown in the terminal. The transmit rate decreases when connected to the terminal so the user has to initiate the terminal by sending any character to the remote first.

In figure 5.4 the remote can be seen with the display on.

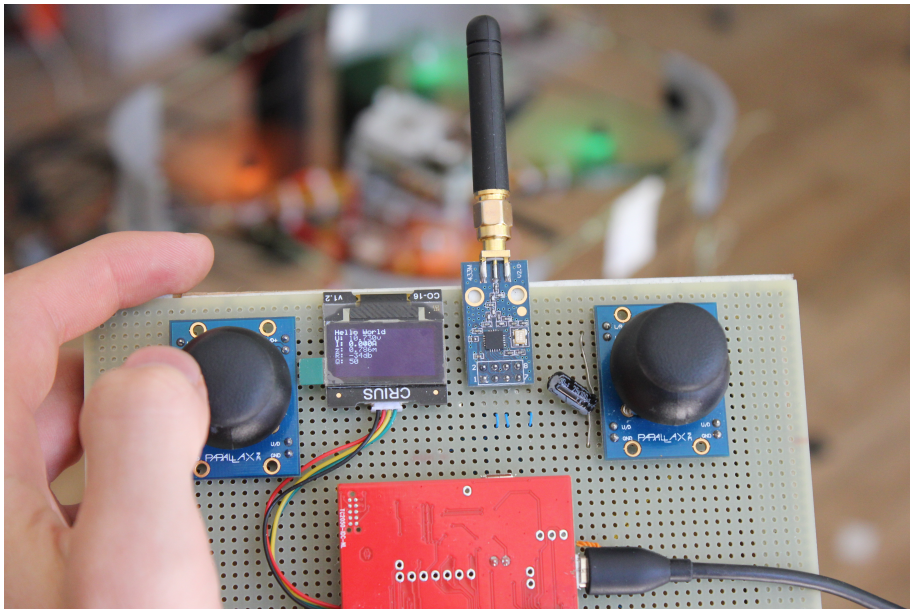

Figure 5.4: The remote used for controlling the quadrotor manually. The quadrotor can be forced to shut off all rotors by pulling the left joystick south. On the display it can be seen that the battery voltage is a bit low ($10.7\,\mathrm{V}$) and that the quadrotor is in the air with an altitude of $0.78\,\mathrm{m}$. The current sensor is not connected, hence the measured current is $0\,\mathrm{A}$.

Linux Computer

The *oDroid-U3 V0.5* is an embedded platform from South Korean based *Hardkernel*. It is based on the *Samsung Exynos4412 Prime*, a processor commonly found in modern smartphones. The board aims at the same market as the well known *Raspberry Pi*, educational and hobby use, but has less IO and more processing power. The *oDroid-U3 V0.5* can use both SD card and a flash module for file storage. We chose an SD card, since this made system backup very easy.

This board was chosen for multiple reasons: Reativly cheap ($\$69.00$ from Korea or $79.95€$ from Germany), light ($\approx 50\,\mathrm{g}$), small ($83\,\mathrm{mm} \times 48\,\mathrm{mm} \times 30\,\mathrm{mm}$), powerful (quad-core $1.7\,\mathrm{GHz}$), SPI interface (makes communication with flight controller fast an reliable).

OS setup The official *Official Lubuntu 14.04 LTS* distribution was chosen, since this distribution has the mosT stable driver support for the hardware.

Some changes had to be made for the system to function properly, including:

- **Add start-up script to set date and time** to some date this year, since there is no backup battery to keep time when powered off. This enables the device to connect to the university network and synchronise the date over the Internet, even after being powered off.
- **Increasing Kernel page cache size**. This helps hide write latency to the SD card.

- **Creating ram-disk**. Allows to sore log-files to RAM, preventing I/O-stalls.
- **Rewrite *PS3Eye* driver**. Minor changes were made to the camera driver to turn off an image filter running on the camera, that increased the perceived sharpness but also adding noise. (Spacial high-boost filter)

5.2 Ground Projected Visual Odometry, Implementation

The visual odometry algorithm is implemented on the embedded *oDroid Linux* computer. This is done in *C++*, matrix and vector algebra is done with the *Eigen* library and computer vision elements are implemented with the *OpenCV* library.

This section will mainly deal with implementation techniques used to speed up processing time, making th algorithm run in real time, at a reasonable framerate.

5.2.1 Fetaure Tracking

Feature tracing is implemented by first detecting interesting points in the previous image and then using optical flow to track those features into the current image. To reduce computation cost, features that were successfully tracked from the previous image to the current are reused as features to be tracked when a new image arrives.

Interesting points are detected with the *OpenCV GFTT* algorithm. The *OpenCV* implementation of *Lucas-Kanade*-optical flow is used for feature tracking. As *LK* is an iterative algorithms, measures were taken to limit the number of iterations to ensure bounded computation time. A disadvantage of *LK* is that the computation time increases, when features move more. This is a problem, as this will decrease the rate at which images are processed, which in turn increases the amount features move by.

Speeding Up Feature Detection

To speed up the process the feature detection method is changed a bit. The speed at which *GFTT* runs is largely dependent on the size of the image in which features are detected. Because features are tracked by optical flow they can be reused, this allows us to detect few new features for each image, while maintaining a large amount of features to track.

We have chosen to detect new features only in a sub-part of the old image, and let this "window" move around. This ensures that features are well spread and reduces computation time, as *GFTT* works on a smaller area.

A constant N_OPTIMAL $= 50$ determines the most optimal number of features (chosen by user). When less features exists, more are added from a window of size ROI_Y_SIZE $\times$ ROI_Y_SIZE . The windowsize is chosen so that each side length is $\frac{1}{4}$ of the original image

To ensure that features don't lump together a certain amount of the traced features are discarded before finding new features. The algorithm used is:

```
Remove 1/3 random features that were previously tracked
Pick random position for window
Find (N_OPTIMAL-n_current)/2 new features
```

The position of the previous features in the current image can be estimated from the state estimation data, but this is not currently used for feature tracking. In figure 5.5 we see some tracked features (marked with green circles), their movement since the previous image (marked with green lines) and their estimated position (red circles).

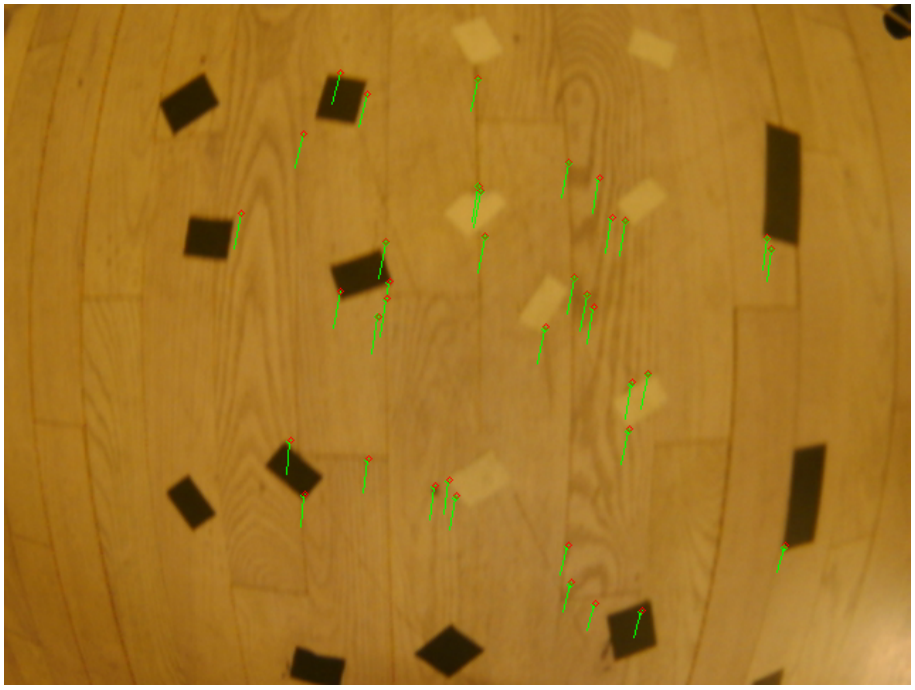

Figure 5.5: Features tracked during quadrotor movement. Features in current image marked by green circles. Feature movement since previous image marked by green lines. Estimated feature position marked by red circles

5.2.2 Threads

The implementation uses multiple threads, this is done for two reasons. To speed up the execution of the algorithm, this is especially effective because the embedded computer is a quad-core, meaning that some of the threads can run in parallel. The other reason is because parts of the code needs to be capable of interrupting the processor to get real-time response. It also makes coding easier, when waiting for sensor inputs (other tasks can execute, while the program wait).

When reading the documentation on the thread structure figure 5.6 can be referred to for a visual representation of the structure of the threads.

Sensor Inputs

Two realtime threads exists for capturing data from the camera and the *IMU*

Camera Server The *PS3Eye* camera has an internal buffer of five images. If the program that uses the camera does not pull new images from the camera fast enough this buffer will fill up, and no new images will be captured until the program pulls an image from the buffer. This makes it very difficult to time-stamp the images, which in turn makes it difficult to synchronize the image stream with the *IMU*.

To solve this problem, a thread that continuously pulls new images from the camera and places them in a temporary buffer i implemented. When the visual odometry algorithm is ready for a new image, the buffer is copied. This ensures that the most recent image is always the one used.

Running the camera server in a real-time thread enables accurate time-stamping, as this setup disallows any other thread to interrupt the process form image capture to time-stamping.

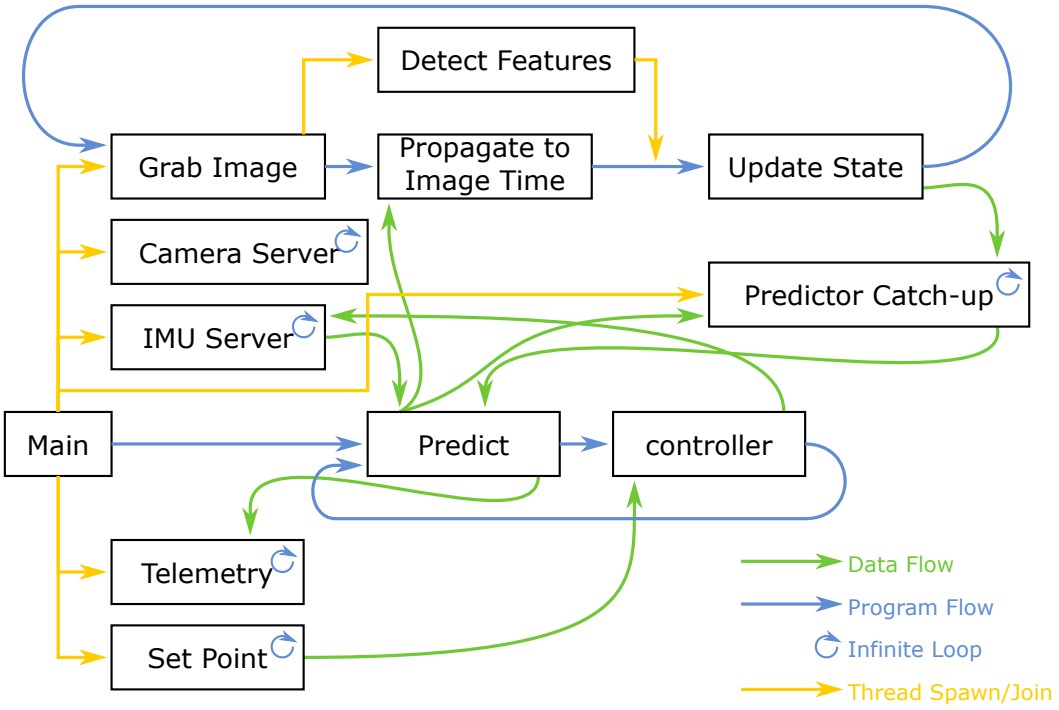

Figure 5.6: An overview of the different threads running in the implementation of the visual odometry and the data being passed between them

IMU Server Data from the *IMU* is captured on the flight controller and down-sampled as described in section 5.1.2. The *IMU* data is transferred to the *oDroid* over an *SPI* interface, while attitude and altitude commands are transferred from the *oDroid* to the flight controller is set back simultaneously.

Like the camera server the *IMU* server runs in a real-time thread. An interrupt request pin on the flight controller is connected to a hardware interrupt enabled *GPIO* pin on the *oDroid*. This allows the *oDroid* to act as a *SPI* master, wile letting the flight controller decide when transfers should be initiated.

When a transfer is initiated the interrupt request is cleared and the transfer is time-stamped, the the actual transfer begins. After receiving the *IMU* data, the data and time-stamp is put into a queue. This is done to make sure, that no measurements are missed by the main loop.

Program I/O

Two threads takes care of user inputs and outputs.

Telemetry This thread sens data over a *UDP* connection to ex. *Matlab*. When starting the program, the thread will listen for any message to the correct port (55000) and once a message has been received it will continuously transmit the messages passed to the thread back to the address that connected.

This allows for sparse data to be sent and plotted on a remote computer for debugging and visualization.

Set Point This thread is not strictly part of the visual odometry, but takes user inputs from the terminal (can be controlled with shell scripts as well) and passes them to the controller. This allows for set-points to be set, while the program is running, allowing the quadrotor to fly in predetermined patterns.

State Estimation

GPFVO The *GPFVO* is run in a separate thread. The thread waits for a new camera image, once the image is received two things happens: A new thread is immediately spawned, tracking features from the previous to the new image with optical flow. The state estimate is propagated up to the time of the new image using buffered *IMU* data, if altitude measurements are available, the state is updated using those as well. Once both tasks complete, the state estimate is updated with the camera image.

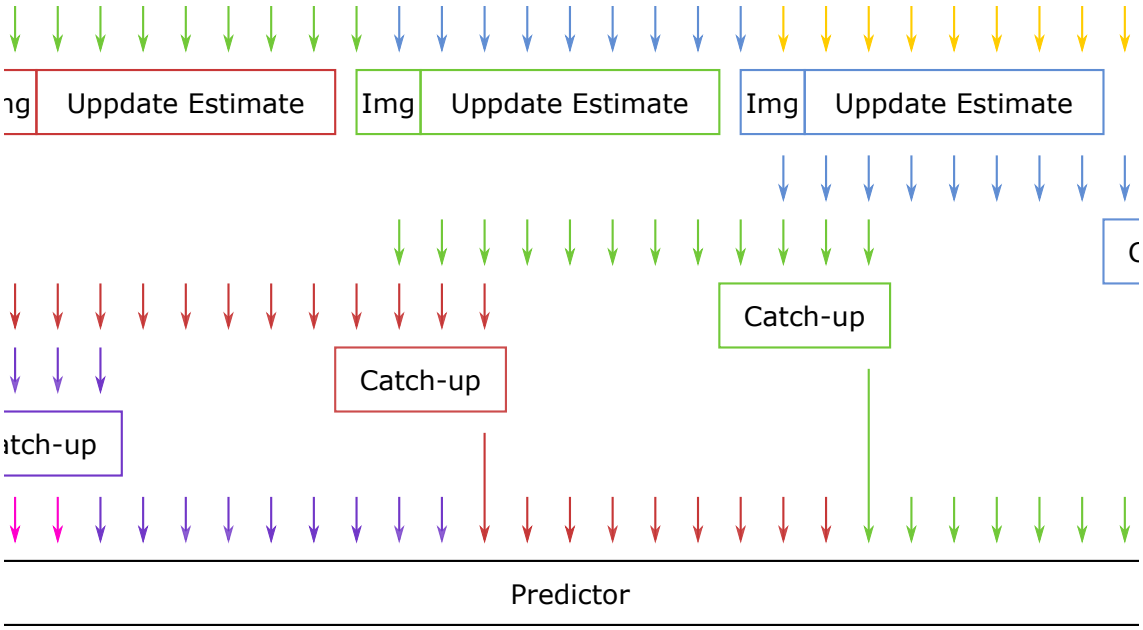

Figure 5.7: Visualization of the data flow for the state predictor. Each color represents *IMU* measurements and state estimate that is updated with a different image.

Predictor The state estimate from the *GPFVO* thread is not used in the controller directly. While the image was being captured and processed and the state being propagated, the drone moves. This means that our state estimate is delayed, compared to the actual state. To overcome this problem, the state is propagated using *IMU* measurements.

When the main thread receives *IMU* samples from the queue created by the *IMU* server, multiple things happens with the data: The sample is immediately used to propagate the most recent prediction. The sample is passed to a queue for processing in the estimator thread. If the so called “prediction catch-up” thread i running, the sample is added to the prediction catch-up queue as well.

The prediction catch-up thread is a thread used to close the temporal gap between a new estimate and the current prediction. Once a new estimate is available from the estimator thread, the *IMU* queue used for estimation is copied to the predictor catch-up thread and all new *IMU* samples are added to the queue as well. Once the predictor catch-up queue is empty, the predicted state in the main thread is replaced with this updated prediction.

The data flow is visualised in figure 5.7 where *IMU* data is visualized as colors arrows. Each color represents *IMU* data used for propagation or prediction for a state estimate updated with newer images. As long as the predictor catch-up thread runs fast enough to catch up before a new update is complete, the delay from an image is captured, to the time it is used to update the predicted state stays between two and three images.

In figure 5.8 the effect of the predictor is illustrated. The quadrotor was moved in sinusoidal patters by hand and the position estimate and predicted position was logged. Note that while the time of the updated estimate and the predicted estimate is synchronised, the data is processed at different times. An estimation error was introduced to show the delayed response to updates in the predictor.

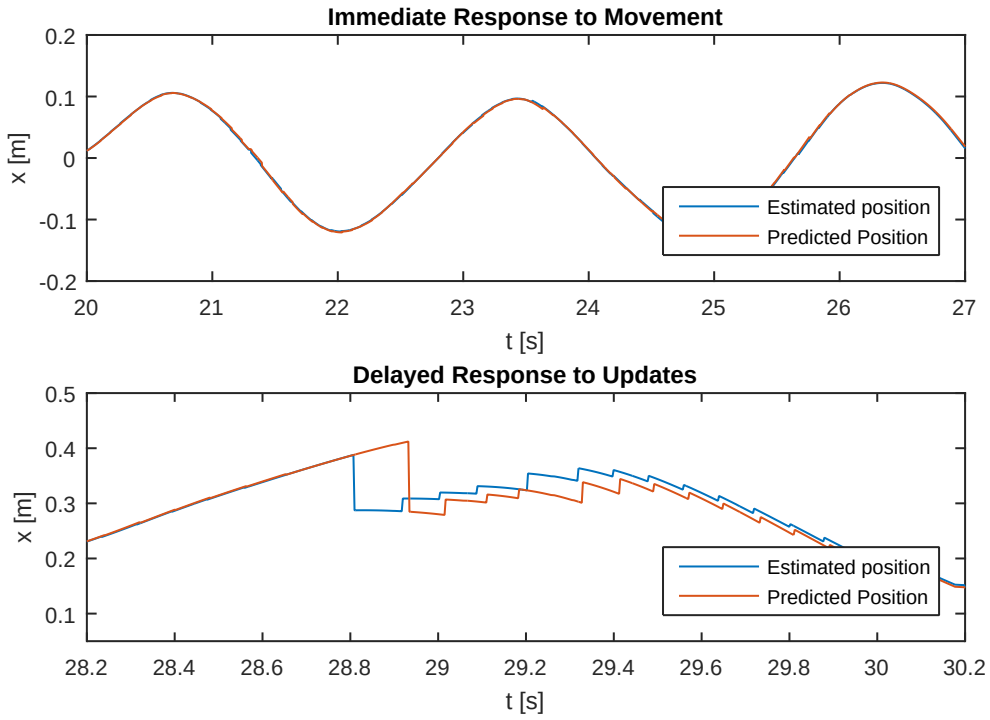

Figure 5.8: *(Top):* Under normal conditions the predictor effectively hides delays in the estimated state. *(Bottom):* When updates are applied to the state estimate, this is done in the past and thus the predictor must start over from the new updated estimate. This results in a delayed response to camera measurements.

In figure 5.8 we can count the number of images processed per second. This is roughly 12fps.

Controller To avoid having to setup variable parsing between two programs, the odometry and controller is implemented in the same program.

5.3 Controller

This section will deal with the implementation of a position controller for the quadrotor. In a previous project an attitude controller was implemented, but was only ever tested with the quadrotor in a fixture, allowing it only to rotate around the y-axis. The report *Control of Agile Quadrotor* for the previous project can be found on the supplied CD.

5.3.1 Quadrotor Dynamics

As the name suggests a quadrotor has four propellers, or rotors. The rotors are located in such a way, that they allow control of pitch, yaw, roll and thrust. In each corner a rotor is located, two of the rotors spins clockwise, while two rotors spins counter clockwise. In figure 5.9 the rotor configuration is shown. By increasing the speed of pairs of propeller, the torque and thrust can be controlled as illustrated.

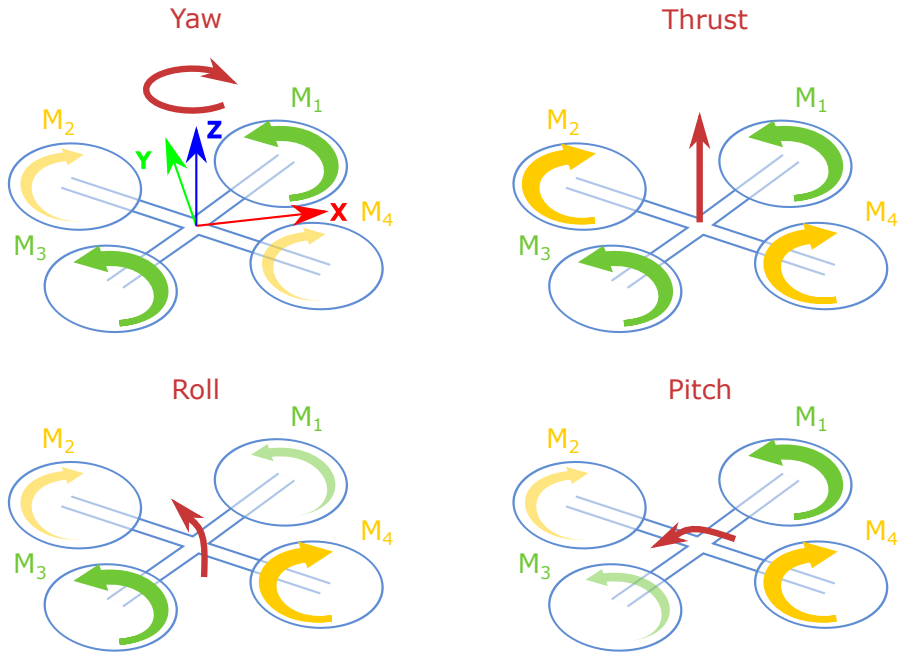

Figure 5.9: By speeding up propellers in pairs the torque and thrust of the quadrotor can be controlled

Each motor generates a specific amount of force at its location and some torque around the axis of rotation in the opposite direction of the rotation (drag). If we know the position and thrust from each rotor, we can calculate the total torque and force on the quadrotor.

We assume that the quadrotor can be modeled as a ridged body, inertia of the propellers are small enough to be neglected when calculating motion and rotors to generate a force and torque that is independent of their movement through the air (in reality the thrust decreases, if the air is already moving in the same direction, as the propeller tries to force it).

Rotational Dynamics

If we assume that the quadrotor center of mass (COM) is located at the base of the inertial frame and the motors are placed as following:

$$ {}^{I}\boldsymbol{p}_{M_1} = \begin{bmatrix} l_x \\ l_y \\ 0 \end{bmatrix} \qquad {}^{I}\boldsymbol{p}_{M_2} = \begin{bmatrix} -l_x \\ l_y \\ 0 \end{bmatrix} \qquad {}^{I}\boldsymbol{p}_{M_3} = \begin{bmatrix} -l_x \\ -l_y \\ 0 \end{bmatrix} \qquad {}^{I}\boldsymbol{p}_{M_4} = \begin{bmatrix} l_x \\ -l_y \\ 0 \end{bmatrix} \tag{5.1} $$

$$ {}^{I}\boldsymbol{p}_{COM} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \tag{5.2} $$

$$ {}^{I}\boldsymbol{F}_{M_1} = \begin{bmatrix} 0 \\ 0 \\ F_{M_1} \end{bmatrix} \qquad {}^{I}\boldsymbol{F}_{M_2} = \begin{bmatrix} 0 \\ 0 \\ F_{M_2} \end{bmatrix} \qquad {}^{I}\boldsymbol{F}_{M_3} = \begin{bmatrix} 0 \\ 0 \\ F_{M_3} \end{bmatrix} \qquad {}^{I}\boldsymbol{F}_{M_4} = \begin{bmatrix} 0 \\ 0 \\ F_{M_4} \end{bmatrix} \tag{5.3} $$

$$ {}^{I}\boldsymbol{\tau}_{M_1} = \begin{bmatrix} 0 \\ 0 \\ \tau_{M_1} \end{bmatrix} \qquad {}^{I}\boldsymbol{\tau}_{M_2} = \begin{bmatrix} 0 \\ 0 \\ \tau_{M_2} \end{bmatrix} \qquad {}^{I}\boldsymbol{\tau}_{M_3} = \begin{bmatrix} 0 \\ 0 \\ \tau_{M_3} \end{bmatrix} \qquad {}^{I}\boldsymbol{\tau}_{M_4} = \begin{bmatrix} 0 \\ 0 \\ \tau_{M_4} \end{bmatrix} \tag{5.4} $$

Where ${}^{I}\boldsymbol{p}_{M_i}$ is the motor position, ${}^{I}\boldsymbol{F}_{M_i}$ is the motor thrust, and ${}^{I}\boldsymbol{\tau}_{M_i}$ is the torque produced by the rotation of the propeller. The total torque can be calculated:

$$ \begin{aligned} {}^{I}\boldsymbol{M} &= {}^{I}\boldsymbol{p}_{M_1} \times {}^{I}\boldsymbol{F}_{M_1} + {}^{I}\boldsymbol{p}_{M_2} \times {}^{I}\boldsymbol{F}_{M_2} + {}^{I}\boldsymbol{p}_{M_3} \times {}^{I}\boldsymbol{F}_{M_3} + {}^{I}\boldsymbol{p}_{M_4} \times {}^{I}\boldsymbol{F}_{M_4} \\ &\quad + {}^{I}\boldsymbol{\tau}_{M_1} + {}^{I}\boldsymbol{\tau}_{M_2} + {}^{I}\boldsymbol{\tau}_{M_3} + {}^{I}\boldsymbol{\tau}_{M_4} \end{aligned} \tag{5.5} $$

$$ = \begin{bmatrix} F_{M_1} l_y + F_{M_2} l_y - F_{M_3} l_y - F_{M_4} l_y \\ -F_{M_1} l_x + F_{M_2} l_x + F_{M_3} l_x - F_{M_4} l_x \\ \tau_{M_1} + \tau_{M_2} + \tau_{M_3} + \tau_{M_4} \end{bmatrix} \tag{5.6} $$

The drag torque and trust force of a rotor is approximately proportional, but because the rotors spins in different directions the drag torque is opposite for M_1, M_3 and M_2, M_4 such that: $\tau_{M_i} \approx G_\tau F_{M_i}$ for motor M_2, M_4 and $\tau_{M_i} \approx -G_\tau F_{M_i}$ for motor M_1, M_3. This allows us to simplify the equation for the total torque:

$$ {}^{I}\boldsymbol{M} \approx \begin{bmatrix} l_y & 0 & 0 \\ 0 & l_x & 0 \\ 0 & 0 & G_\tau \end{bmatrix} \begin{bmatrix} 1 & 1 & -1 & -1 \\ -1 & 1 & 1 & -1 \\ -1 & 1 & -1 & 1 \end{bmatrix} \begin{bmatrix} F_{M_1} \\ F_{M_2} \\ F_{M_3} \\ F_{M_4} \end{bmatrix} \tag{5.7} $$

The torque can be converted to angular acceleration $\dot{\boldsymbol{\omega}}$, this is done by *Euler's rotation equations*:

$$ {}^{I}\boldsymbol{I} \cdot {}^{I}\dot{\boldsymbol{\omega}} + {}^{I}\boldsymbol{\omega} \times ({}^{I}\boldsymbol{I} \cdot {}^{I}\boldsymbol{\omega}) = {}^{I}\boldsymbol{M} \tag{5.8} $$

Rearranging (5.8) we get:

$$ {}^{I}\dot{\boldsymbol{\omega}} = {}^{I}\mathbf{I}^{-1} \cdot \left({}^{I}\boldsymbol{M} - {}^{I}\boldsymbol{\omega} \times ({}^{I}\mathbf{I} \cdot {}^{I}\boldsymbol{\omega}) \right) \tag{5.9} $$

Where the term ${}^{I}\boldsymbol{\omega} \times ({}^{I}\boldsymbol{I} \cdot {}^{I}\boldsymbol{\omega})$ is the gyroscopic force.

In our controller design, we assume that the inertia tensor is diagonal, thus: ${}^{I}\mathbf{I} = \begin{bmatrix} I_x & 0 & 0 \\ 0 & I_y & 0 \\ 0 & 0 & I_x \end{bmatrix}$, furthermore the angular velocity is always small, as the drone is mainly flown in hover. This allows us to neglect the

gyroscopic force, and we have:

$$
\begin{aligned}
{}^{I}\dot{\boldsymbol{\omega}} &= \begin{bmatrix} \frac{1}{I_x} & 0 & 0 \\ 0 & \frac{1}{I_y} & 0 \\ 0 & 0 & \frac{1}{I_x} \end{bmatrix} \boldsymbol{M} && (5.10) \\
&= \begin{bmatrix} \frac{l_y}{I_x} & 0 & 0 \\ 0 & \frac{l_x}{I_y} & 0 \\ 0 & 0 & \frac{G_\tau}{I_z} \end{bmatrix} \begin{bmatrix} 1 & 1 & -1 & -1 \\ -1 & 1 & 1 & -1 \\ -1 & 1 & -1 & 1 \end{bmatrix} \begin{bmatrix} F_{M_1} \\ F_{M_2} \\ F_{M_3} \\ F_{M_4} \end{bmatrix} && (5.11)
\end{aligned}
$$

Translational Dynamics

As a quadrotor only has four controller inputs, but six degrees of freedom (3D pose and orientation) the system is under-actuated. It is, however possible to control the position and heading of the drone, as the total thrust of the quadrotor is along the z-axis in the inertial frame:

$$
\begin{aligned}
{}^{I}\boldsymbol{F} &= {}^{I}F_{M_1} + {}^{I}F_{M_2} + {}^{I}F_{M_3} + {}^{I}F_{M_4} && (5.12) \\
&= \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} (F_{M_1} + F_{M_2} + F_{M_3} + F_{M_4}) && (5.13)
\end{aligned}
$$

When the quadrotor is in the air, this gives the acceleration

$$
{}^{G}\boldsymbol{a}_I = \frac{1}{m} {}^{G}_{I}\boldsymbol{q} {}^{I}\boldsymbol{F} + \begin{bmatrix} 0 \\ 0 \\ -1 \end{bmatrix} g \tag{5.14}
$$

Where m is the mass of the drone and g is the gravitational acceleration.

As it is possible to control the attitude (${}^{G}_{I}\boldsymbol{q}$), it is also possible to control the acceleration and thus the position of the quadrotor. It is however not possible to control attitude and position independently.

Under steady state conditions the thrust of each motor is approximately proportional to the output signal: $F_{M_i} \approx G_F u_i$. We define the output signal u_i to be in the range $[0; 1]$, thus making G_f the maximum thrust a single rotor can produce. The propellers, however, do not spin up instantly. The frequency response of the rotor can be approximated with a first order system: $\dot{F}_{M_1} = (G_F u_i - F_{M_1})/\tau_M$, or we can move the retardation directly to the total angular acceleration and force:

$$
{}^{I}\ddot{\boldsymbol{\omega}} = G_F \begin{bmatrix} \frac{l_y}{I_x} & 0 & 0 \\ 0 & \frac{l_x}{I_y} & 0 \\ 0 & 0 & \frac{G_\tau}{I_z} \end{bmatrix} \begin{bmatrix} -1 & -1 & 1 & 1 \\ 1 & -1 & -1 & 1 \\ -1 & 1 & -1 & 1 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{bmatrix} \frac{1}{\tau_M} - \frac{{}^{I}\dot{\boldsymbol{\omega}}}{\tau_M} \tag{5.15}
$$

$$
{}^{I}\dot{\boldsymbol{F}} = G_F \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} (u_1 + u_2 + u_3 + u_4) \frac{1}{\tau_M} - \frac{{}^{I}\boldsymbol{F}}{\tau_M} \tag{5.16}
$$

5.3.2 Controller Output Allocation

It is possible to reassign the controller inputs, in such a way that they correspond directly to angular acceleration around the x, y, z-axes and thrust. This is done by defining $u_{[1;4]}$ as:

$$\begin{aligned} u_1 &= u_{roll} - u_{pitch} - u_{yaw} + u_z \qquad u_2 = u_{roll} + u_{pitch} + u_{yaw} + u_z \\ u_3 &= -u_{roll} + u_{pitch} - u_{yaw} + u_z \quad u_4 = -u_{roll} - u_{pitch} + u_{yaw} + u_z \end{aligned} \tag{5.17}$$

$$\begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{bmatrix} = \begin{bmatrix} 1 & -1 & -1 & 1 \\ 1 & 1 & 1 & 1 \\ -1 & 1 & -1 & 1 \\ -1 & -1 & 1 & 1 \end{bmatrix} \begin{bmatrix} u_{roll} \\ u_{pitch} \\ u_{yaw} \\ u_z \end{bmatrix} \tag{5.18}$$

With the new controller inputs the angular acceleration and thrust derivative can be expressed:

$${}^I\ddot{\boldsymbol{\omega}} = 4G_F \begin{bmatrix} \frac{l_y}{I_x} & 0 & 0 \\ 0 & \frac{l_x}{I_y} & 0 \\ 0 & 0 & \frac{G_\tau}{I_z} \end{bmatrix} \begin{bmatrix} u_{roll} \\ u_{pitch} \\ u_{yaw} \\ u_z \end{bmatrix} \frac{1}{\tau_M} - \frac{{}^I\dot{\boldsymbol{\omega}}}{\tau_M} \tag{5.19}$$

$${}^I\dot{\boldsymbol{F}} = 4u_z G_F \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \frac{1}{\tau_M} - \frac{{}^I\boldsymbol{F}}{\tau_M} \tag{5.20}$$

5.3.3 State Estimation

The flight controller has an altitude and attitude controller running off of only *IMU* and distance sensor measurements. This enables the flight controller to function as an altitude/attitude controller without the embedded Linux computer.

5.3.4 Attitude Estimation

The attitude is estimated using the *Madgwick IMU* algorithm Madgwick et al. (2010) Madgwick et al. (2011). The algorithm works by integrating gyroscope measurements to fin the attitude. To correct for drift due to a gyro bias the direction of the acceleration vector is used. Since the accelerometer also measures the gravitational pull the acceleration measurement can be used by assuming that the average acceleration is zero, thus that the average of the measured acceleration primarily is from the gravity. As the direction of the measured gravitational acceleration is always aligned to the global z-axis, this can correct for drift errors around the x- and y-axis, but not the z axis.

As the quadrotor is flying indoors the on-board magnetometer is not used for state estimation, as iron in the building alters the magnetic field.

To reduce drifting, mainly around the global z-axis, due to gyroscope bias, the bias is estimated by low-pass filtering gyro measurements before the quadrotor is armed.

5.3.5 Altitude Estimation

The altitude estimator is implemented as described in the *Kalman Filter* example section 2.3 with outlier detection. The acceleration measurement used for propagating the state estimate is acquired by rotating the

measured acceleration by the estimated attitude and subtraction the gravitational acceleration:

$$^{G}\boldsymbol{a} = {}^{G}_{I}\boldsymbol{q}\,{}^{I}\boldsymbol{a}_m - \begin{bmatrix}0\\0\\g\end{bmatrix} \tag{5.21}$$

5.3.6 Attitude and Altitude Controller

The flight controller always uses the local estimates of attitude and altitude as feedback in the controller, this ensures that even if the state estimate form the visual odometry diverges, the drone will remain relatively stable - stable enough for a human to react and take control before a crash.

Roll and Pitch

Error Angle To control the direction of the thrust vector ${}^{I}\boldsymbol{F}$ we define a unit vector in the inertia frame ${}^{I}\boldsymbol{\theta}$ that defines the direction set-point.

We can calculate the error between the current direction of thrust and the set-point as an error angle around the inertia x-axis and the y-axis:

$$^{I}\bar{\boldsymbol{F}} = \frac{^{I}\boldsymbol{F}}{\|^{I}\bar{\boldsymbol{F}}\|} \tag{5.22}$$

$$\begin{aligned}\delta\theta_{roll} &= \arctan 2\left(\det\left(\begin{bmatrix}^{I}\bar{F}_2\\ ^{I}\bar{F}_3\end{bmatrix}, \begin{bmatrix}^{I}\theta_2\\ ^{I}\theta_3\end{bmatrix}\right), \begin{bmatrix}^{I}\bar{F}_2\\ ^{I}\bar{F}_3\end{bmatrix} \cdot \begin{bmatrix}^{I}\theta_2\\ ^{I}\theta_3\end{bmatrix}\right) \\ &= \arctan 2(-{}^{I}\theta_2, {}^{I}\theta_3)\end{aligned} \tag{5.23}$$

$$\begin{aligned}\delta\theta_{pitch} &= \arctan 2\left(\det\left(\begin{bmatrix}^{I}\bar{F}_1\\ ^{I}\bar{F}_3\end{bmatrix}, \begin{bmatrix}^{I}\theta_1\\ ^{I}\theta_3\end{bmatrix}\right), \begin{bmatrix}^{I}\bar{F}_1\\ ^{I}\bar{F}_3\end{bmatrix} \cdot \begin{bmatrix}^{I}\theta_1\\ ^{I}\theta_3\end{bmatrix}\right) \\ &= \arctan 2({}^{I}\theta_1, {}^{I}\theta_3)\end{aligned} \tag{5.24}$$

$$\tag{5.25}$$

As these angles are already in inertial coordinate their derivatives can be approximated by:

$$\delta\dot{\theta}_{roll} \approx {}^{I}\omega_1 \tag{5.26}$$

$$\delta\dddot{\theta}_{roll} \approx 4G_F\frac{l_y}{I_x}u_{roll}\frac{1}{\tau_M} - \frac{^{I}\dot{\boldsymbol{\omega}}}{\tau_M} \tag{5.27}$$

$$\delta\dot{\theta}_{pitch} \approx {}^{I}\omega_2 \tag{5.28}$$

$$\delta\dddot{\theta}_{pitch} \approx 4G_F\frac{l_x}{I_y}u_{pitch}\frac{1}{\tau_M} - \frac{^{I}\dot{\boldsymbol{\omega}}}{\tau_M} \tag{5.29}$$

Controller The attitude controller is implemented as a cascaded controller with feedback from angular velocity (direct measurement) and angular acceleration (estimated from measurements). In each cascaded step the output to the next stage is saturated. This is done for two reasons: One it to make the controller behave in a controlled way (you might want to limit angular velocity to avoid saturating gyro measurements), the other reason being that it increases stability when there is a hard saturation on the output (the rotors can only spin so fast). This method is used to generate smooth trajectories in Wang, Jian and Holzapfel, Florian and Xargay, Enric and Hovakimyan Naira (2013)

In figure 5.10 the system with regulator is seen, ignoring the propeller speedup retardation. By setting

$$k_0 = \omega_\theta^3 \qquad k_1 = 3\omega_\theta^2 \qquad k_2 = 3\omega_\theta \tag{5.30}$$

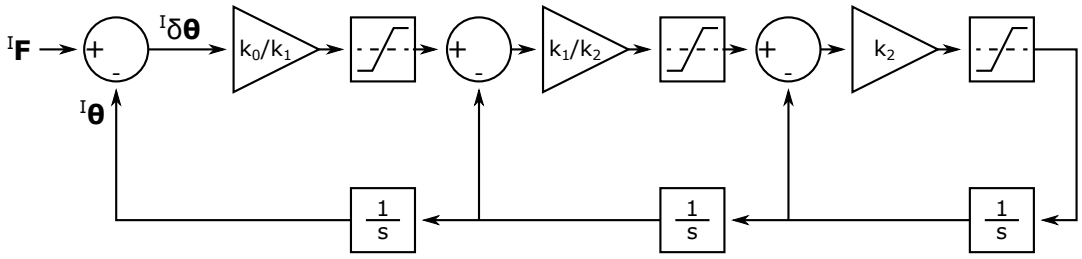

Figure 5.10: Illustration of the the controller and system, when neglecting the rotor dynamics

The system will approximate a critically dampened third-order low-pass filter. The system will have to be tuned manually, but since only one tuning parameter ω is present this is an easy task.

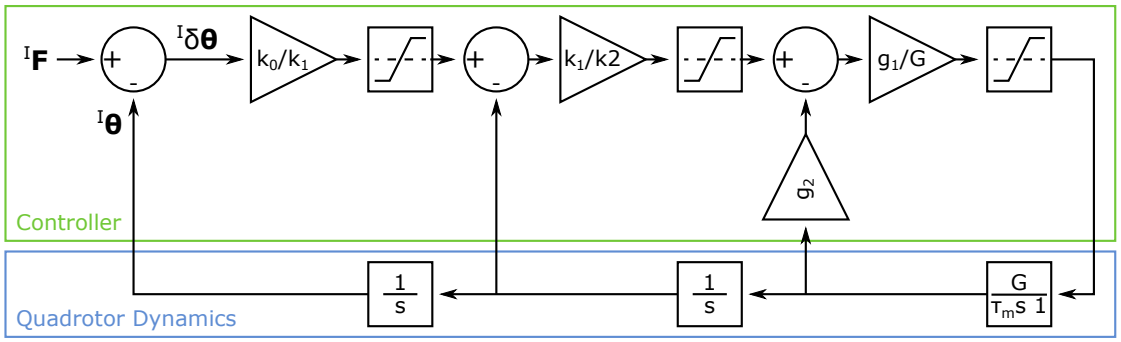

Figure 5.11: Illustration of the the controller and system, when hiding the rotor dynamics. The static gain G is $4G_F \frac{l_x}{I_y}$ for the pitch and $4G_F \frac{l_y}{I_x}$ for roll

By introducing a constant gain to the output and changing the angular acceleration feedback, we can hide the rotor spin up. This is done as seen in figure 5.11. g_1 and g_2 are defined as:

$$g_1 = \max(k_2 \tau_M, 1) \qquad g_2 = 1 - \frac{1}{g_1} \tag{5.31}$$

g_1 is limited to $[0; 1]$ as values above 1 would cause positive feedback from the angular acceleration. In the case that $g_1 = 1$ it simply means that the rotor response is "much faster" than that of the whole system, in that case the dynamics of the rotor are ignored.

Input Re-parametrization The input to the controller is re-parametrized, so that it becomes acceleration along the x- and y-axes of the local coordinate system L. L is defined by having its z-axis parallel to the global z-axis and the projection of the x-axis onto the global x, y plane parallel to the x-axis in the inertia frame I. See figure 5.12.

At hover the length of ${}^G\boldsymbol{F}$ along the global z-axis, and there for also the local z-axis, is mg. If we define

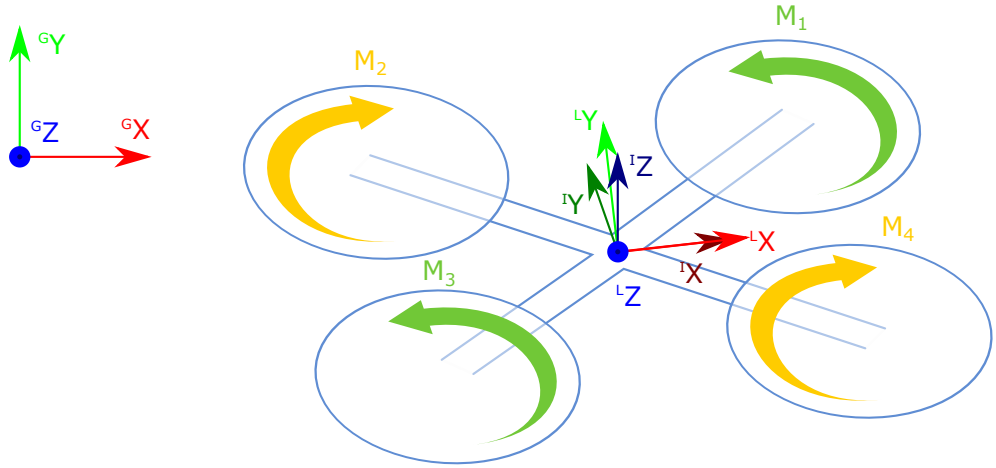

Figure 5.12: Illustration of the orientation of the local frame L compared to the global frame G and the inertia frame I

an input ${}^{L}\boldsymbol{a}_{sp} = \left[\begin{smallmatrix} a_{sp,x} \\ a_{sp,y} \end{smallmatrix}\right]$ as the acceleration set-point, the desired thrust vector, and its direction is:

$$ {}^{L}\boldsymbol{\theta}' = \begin{bmatrix} a_{sp,x} \\ a_{sp,y} \\ g \end{bmatrix} \tag{5.32} $$

$$ {}^{L}\boldsymbol{\theta} = \frac{{}^{L}\boldsymbol{\theta}'}{\|{}^{L}\boldsymbol{\theta}'\|} \tag{5.33} $$

$$ {}^{I}\boldsymbol{\theta} = {}^{I}_{G}\boldsymbol{q} \otimes {}^{G}_{L}\boldsymbol{q}\,{}^{L}\boldsymbol{\theta} \tag{5.34} $$

Altitude

The altitude controller is constructed in a similar way. As the z-axis of the quadrotor is facing primarily in the direction of the global z-axis, the acceleration of the drone in the global z-axis is:

$$ {}^{G}a_{I3} \approx \frac{1}{m}\,{}^{I}F_3 - g \tag{5.35} $$

$$ {}^{I}\dot{F}_3 = 4u_z G_F \frac{1}{\tau_M} - \frac{{}^{I}F_3}{\tau_M} \tag{5.36} $$

$$ u_z = u'_z + \frac{m}{4G_F} g \tag{5.37} $$

By adding a static offset, we can compensate for the effect of the gravitational pull and thus have:

$$ {}^{G}\dddot{p}_{I3} \approx 4G_F \frac{1}{m} u'_z \frac{1}{\tau_M} - \frac{{}^{G}a_{I3}}{\tau_M} \tag{5.38} $$

Note that this equation is equal in construction as that of the roll and pitch equations (5.27), (5.29). The controller is implemented in the same way with $k_{[0;2]}$ as

$$ k_0 = \omega_z^3 \qquad k_1 = 3\omega_z^2 \qquad k_2 = 3\omega_z \tag{5.39} $$

Yaw Rate

For the controller around the z-axis a simple *PID* controller is chosen. Since the yaw angle is not observable, a yaw rate controller is implanted. The yaw rate is measured directly by the gyroscope z-axis.

As the position of the motors is not very precise a nett torque might be present, even though the motors are spinning at the same speed. To remove a static error in yaw rate an I-term is added. The I-term integrator is saturated to reduce wind-up (wind-up is especially apparent when the drone takes off, as the system model only holds for a quadrotor in the air).

5.3.7 Position Controller

The position controller is implemented on the embedded Linux computer as a *PID* controller. The z set-points is passed directly to the flight controller. The x and y set-points are fed through a *PID* regulator, where the P and D term are evaluated in global coordinates and the I is evaluated in the local frame. The purpose of the I term is to compensate for misalignment in rotors, leading in a static error of the acceleration. The output from the position controller is fed to the flight controller in the same manner as the user would pass commands via the remote.

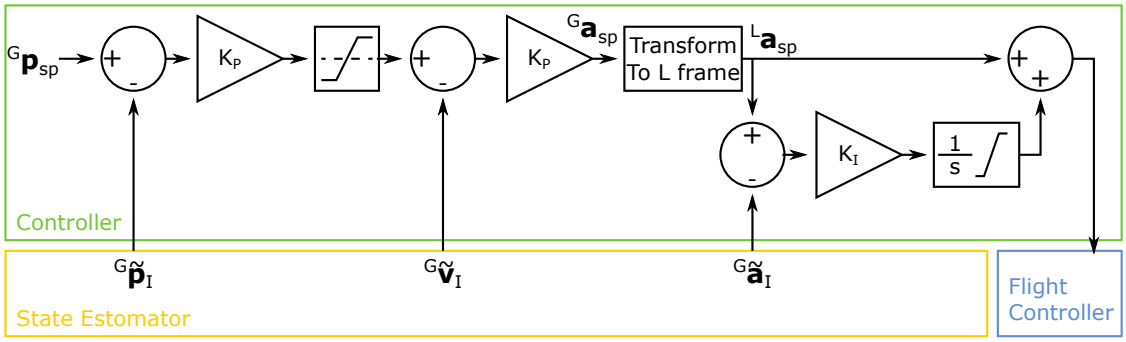

Figure 5.13: Block diagram of the position controller. The desired acceleration is rotated into the local frame L and an I term is added to compensate for motor misalignment.

In figure 5.13 the architecture of the position controller can be seen. Note that for each stage in the controller, the output to the next stage is limited. This is done to make the quadrotor move slowly, no matter the error in set-point and current pose. This is done to minimize camera motion blur.

CHAPTER 6

Results

6.1 Visual Odometry Results

To test the visual odometry the quadrotor was mounted on one of the *SMR*s (*Small Mobile Robot*) at the *DTU Automation and Control* institute. The quadrotor was secured to the front of the *SMR* with the camera $50\,\mathrm{cm}$ in front of the wheel base.

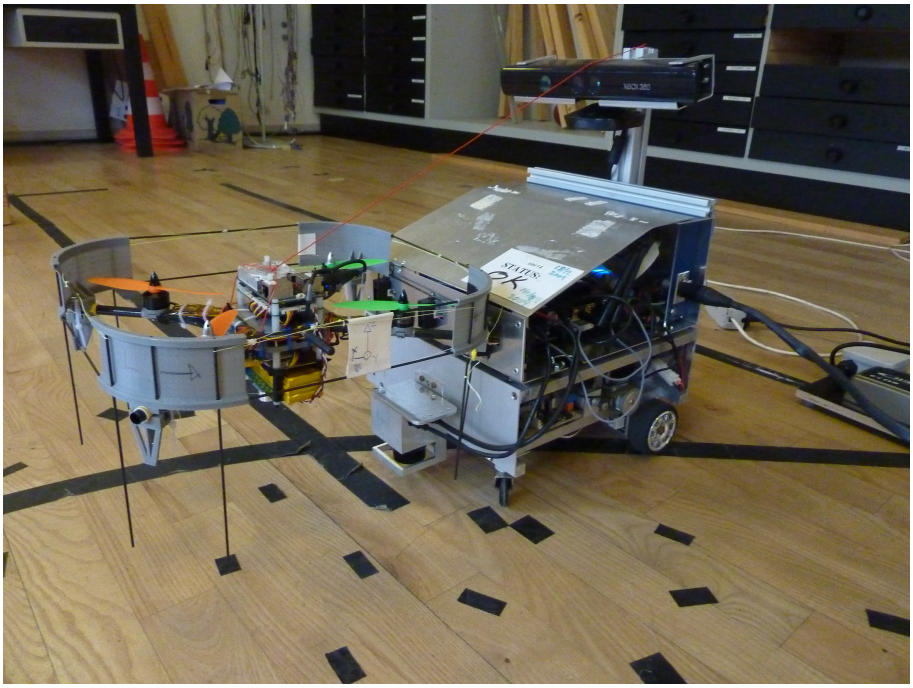

Figure 6.1: The quadrotor is mounted to an *SMR* for testing the odometry.

The *SMR* drives around in a square of $80\,\mathrm{cm} \times 80\,\mathrm{cm}$ with a speed of $10\,\mathrm{cm\,s^{-1}}$. At the corners the *SMR* turns on the spot with a speed at the wheels of $5\,\mathrm{cm\,s^{-1}}$. Because the quadrotor is mounted in front of the wheels the quadrotor does not move in a square. a picture of the quadrotor mounted to the *SMR* can be seen in figure 6.1. To ensure that enough features are detected by the camera, rectangles of tape are put on the floor.

The estimated position of two identical runs are compared to the ground truth in figure 6.2. It is obvious, that the absolute position is not tracked very well. The estimated end position is off by approximately $1\,\mathrm{m}$. This is, however, mainly due to errors in heading estimation, as all linear pieces of the movement are

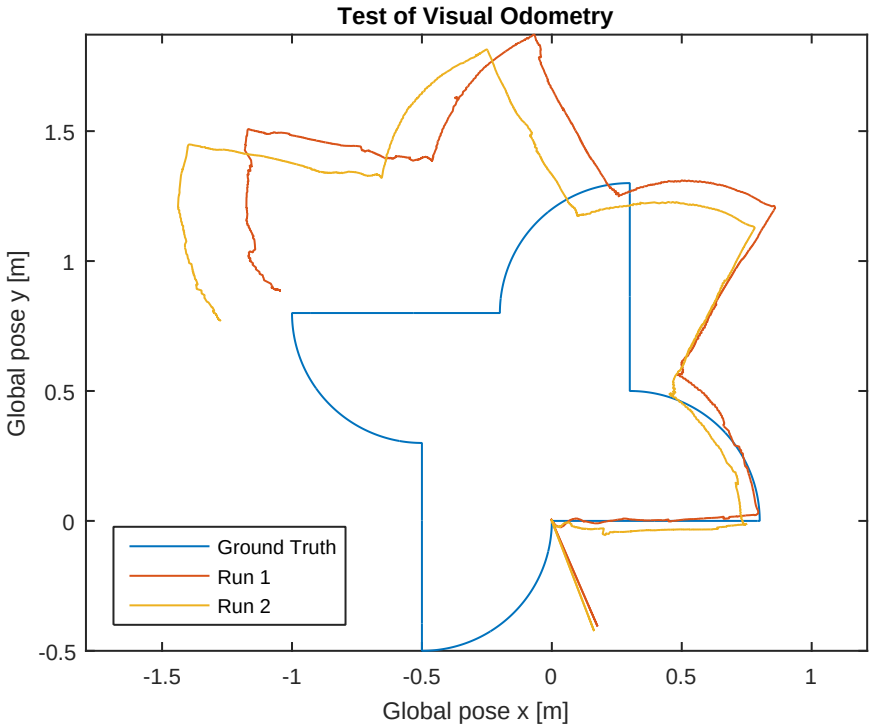

Figure 6.2: The quadrotor is mounted to an *SMR* driving in a square. The actual motion o the quadrotor is illustrated in blue. The path is driven twice and the estimated position of each run is illustrated in red and yellow. The Path start in $(0, 0)$. Note that the deviation in the beginning is the estimate before the first camera image.

estimated to be roughly linear and have roughly the correct length. Note that the irregularity at the beginning is due to a delay before the first camera updates propagate into the state estimate.

6.2 Position Controller Results

The position controller can only be tested with the visual odometry in the loop, as we do not have access to any positioning system.

Te controller will be tested both when flying in patterns without external disturbances and in hover with disturbances.

6.2.1 Linear Pattern

By scripting input to the odometry/controller program the drone can be commanded to fly in predetermined patterns. In this test the quadrotor is commanded to fly forwards by $50\,\mathrm{cm}$ along the global x-axis and back again four times.

In figure 6.3 the estimated 3d trajectory of the quadrotor is illustrated. In figure 6.4 the global position and velocity over time is illustrated. The controller tries to limit the velocity to $10\,\mathrm{cm\,s^{-1}}$, this be seen from the near-linear slope in the position plot, or more clearly in the velocity plot. A video of the test can be found at `https://youtu.be/Z_v9zJ7ZZHU`, or on the attached CD. In the video it can be seen that the quadrotor safely passes trough a yellow “gate”. the “gate” is $45\,\mathrm{cm}$ wide.

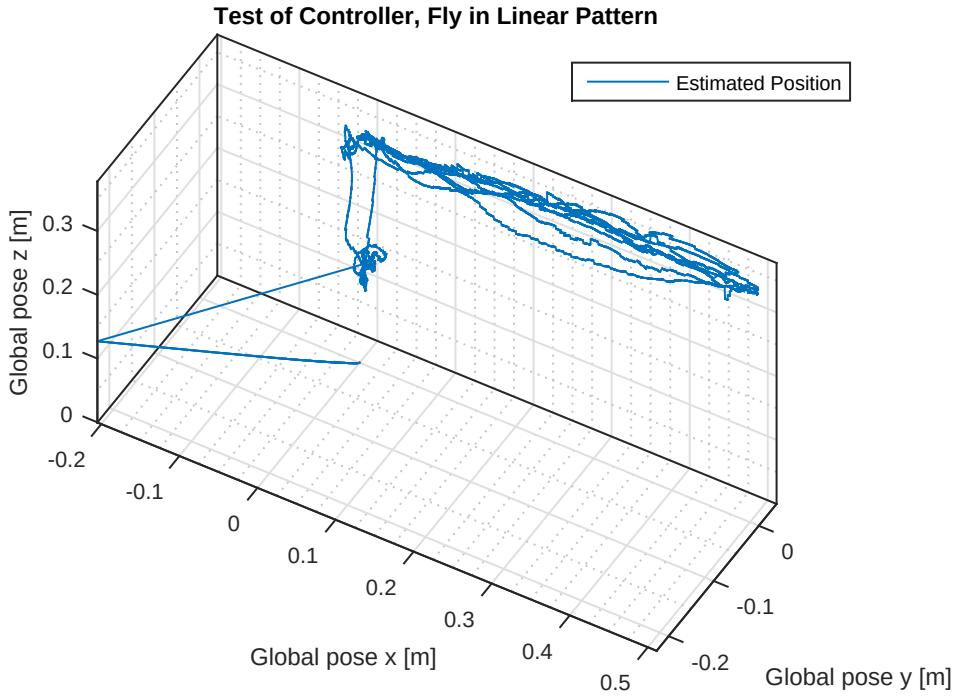

Figure 6.3: The quadrotor is commanded to fly back and forth. This is a 3D plot of the estimated position.

6.2.2 Disturbances

In this test the quadrotor was commanded to stay at an altitude of $40\,\mathrm{cm}$ while it was pushed and lifted. After some time the drone is commanded to ascent to $80\,\mathrm{cm}$ and descend back to $40\,\mathrm{cm}$, then land.

In figure 6.5 the estimated 3d trajectory of the quadrotor is illustrated. In figure 6.6 the global position and velocity over time is illustrated. A video of the test can be found at `https://youtu.be/EyRYwbe2Al8`, or on the CD.

According to the odometry the drone finds back to the set-point after being let go, in reality it drifts a little away from the starting point, due to inaccuracies in the odometry.

6.2.3 Box Test

To test the combined drift of the odometry and controller, the quadrotor was placed in a length $45\,\mathrm{cm} \times$ width $45\,\mathrm{cm}$ box made from *Robocup* "gates". The drone was able to stay within the box, without touching it for more than $30\,\mathrm{s}$. A video of this can be seen at `https://youtu.be/PeOpLe-VL48` or on the CD.

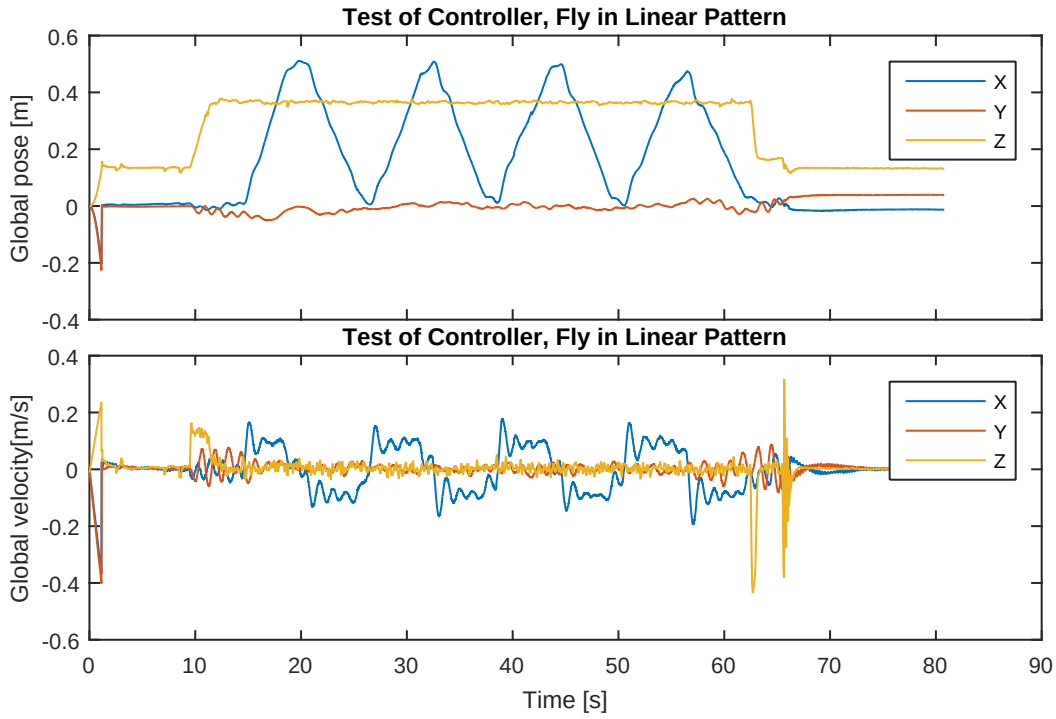

Figure 6.4: The quadrotor is commanded to fly back and forth. Position and velocity in global coordinates is visualized.

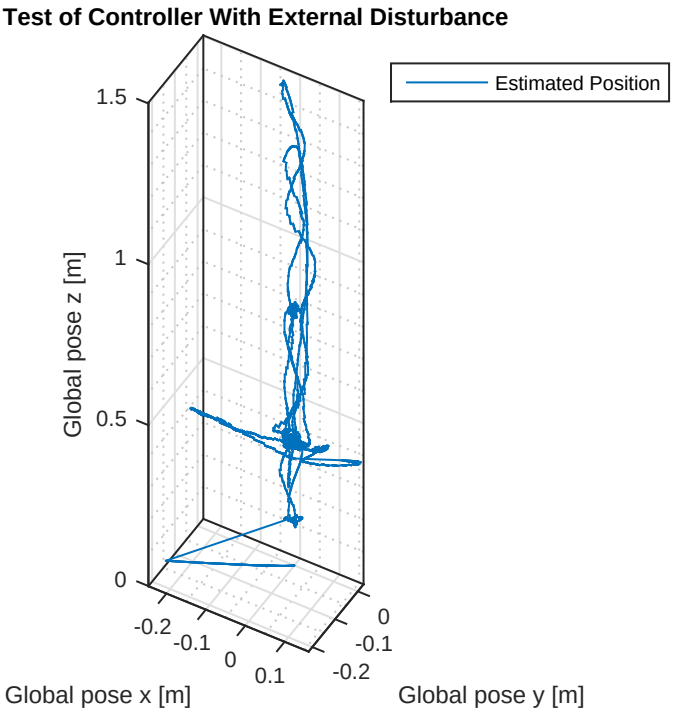

Figure 6.5: The drone is commanded to hover at an altitude of $40\,\mathrm{cm}$, while a user pushes and lifts the drone. Note how (according to the odometry) the drone returns to $(0, 0, 0.4)$

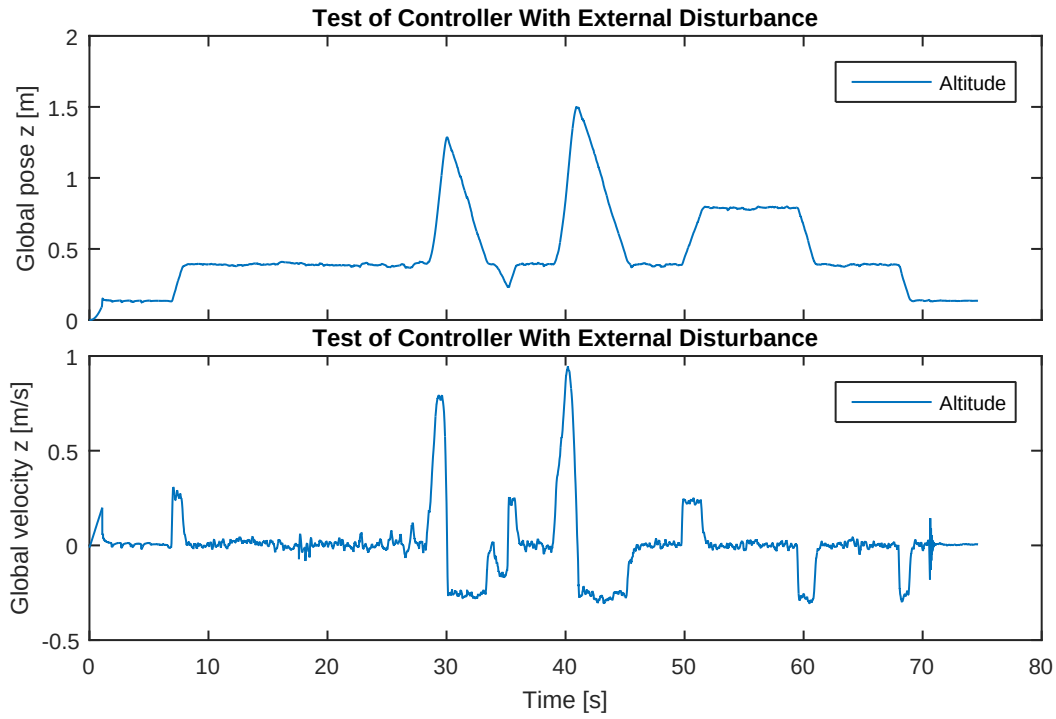

Figure 6.6: The drone is commanded to hover at an altitude of $40\,\mathrm{cm}$, while a user pushes and lifts the drone. At approximately $29\,\mathrm{s}$ and $40\,\mathrm{s}$ the drone is lifted and let go. At $34\,\mathrm{s}$ the quadrotor is pushed down. At $50\,\mathrm{s}$ the drone is commanded to ascent to an altitude of $80\,\mathrm{cm}$, then descend again and land.

6.3 Discussion

6.3.1 Visual Odometry

The visual odometry works well enough for the drone to stabilize in hover and at slow speeds, however there are multiple drawbacks with the odometry.

- The *GPFVO* algorithm does not take into account the uncertainty of the ground projected features due to state estimation errors. This leads to a correlation between the measurement residual and the state that is not taken into account. The algorithm does however work in our case, since the error in the state estimate is small enough not to have an effect.
- In the testing of the visual odometry it was visible, that the heading error is misestimated. In the test the quadrotor was moving sideways at a relatively large speed, this could make it difficult to separate movement from rotation.
- The camera requires very distinct features on the ground. This prohibits deployment in an environment without these features. During testing this was fixed by taping black peaces of tape to the floor.
- Motion blur and slow processing of the images prohibits flight at large speed. It is, however, possible to fly faster, as the drone gets further from the ground.

That being said, the odometry is precise enough to allow the quadrotor to safely stay within a length $45\,\mathrm{cm} \times$ width $45\,\mathrm{cm}$ area for more than $30\,\mathrm{s}$ and also fly trough an opening of width $45\,\mathrm{cm} \times$ height $50\,\mathrm{cm}$ multiple times without human interaction.

6.3.2 Controller

Splitting the controller in two parts, an inner loop running on the flight controller and an outer loop running on the Linux computer has both advantages and disadvantages. While the estimated accelerometer and gyro biases from the odometry cannot be used in the attitude and altitude controller, the split nature allows for, at best a gracious manual take over, or at worst a much smoother emergency landing. The possibility of landing the drone without a violent crash, greatly outweighs the possibility of having a controller that overall might perform slightly better - especially in a development environment.

The position controller does show some oscillation, mainly visible in the velocity. This could be due to the I-term in the controller, or the fact that there is no feedback from the measured acceleration, apart from in the integrator. Another explanation could be that th quadrotor flies close to the ground. This means that turbulence from the propellers will affect the system in a way that is not taken into account.

Manual Control Having a controller that keeps the altitude ad heading of the quadrotor, while allowing the user to input attitude commands greatly simplifies the amount of skill required to fly the quadrotor. The quadrotor can be flown with a single joystick, much similar to a car in a racing game. As the inputs are for attitude, which corresponds to acceleration in the x, y-plane, the user is tasked to be the outer loop of a system that can be approximated by two chained integrators.

One shortcoming of the controller is, that at the moment there is no way to manage a static offset in either attitude estimation or output. This leads the quadrotor to drift off, if the user lets go of the joystick.

CHAPTER 7

Conclusion

During the project an autonomous quadrotor was designed and implemented. The quadrotor satisfies the in section 1.3 defined requirements, that is the quadrotor is:

- **Self contained**. All processing power and sensors are on-board the quadrotor.
- **Autonomous**. The drone is capable of flight without human iteration.
- **Safe**
  - Equipped with a **guard around propellers** to protect the environment form the spinning blades.
  - Capable of **Manual controller** that can be switched on and take over in case of failure in the autonomous system.
  - Equipped with a **dead man's switch**. This makes sure that termination of operation is always a possibility.

Autonomous control is achieved through localization bt visual odometry and cascaded controllers with limited output. The odometry is capable of localising when flying over areas with sufficient features, when flying at slow speeds. The odometry performs well in detecting translational movement, but rotational movement is not tracked very well.

By using visual odometry the autonomous quadrotor is capable of hovering inside an area of inside an area of length $45\,\mathrm{cm} \times$ width $45\,\mathrm{cm}$ for more than $30\,\mathrm{s}$ without leaving the area. The quadrotor is capable of autonomously flying through an opening of width $45\,\mathrm{cm} \times$ height $50\,\mathrm{cm}$ without touching the sides.

A manual controller was implemented, the controller hold the altitude and heading steady, while allowing a user to input attitude commands. This allows even an inexperienced pilot to fly the system.

7.1 Future Work

7.1.1 Improve Odometry

The most significant drawback of the implemented system is that it requires a large amount of distinct features, this reduces the usability.

As the end goal of creating a small scale autonomous quadrotor is participation in *DTU Robocup*, where features on the ground are not guaranteed, it is suggested that work is done to enable the odometry to function in such an environment. In *Robocup* a path the robot can take is marked with a dark or light curve of gaffer tape, a solution could thus be to track the center of this line and thus estimate position.

Another solution could be to tilt the camera up, looking up into the room. While this does produce more features, there is the added problem of estimating feature depth.

7.1.2 Mission Control

It would be useful to have a set of programs to specify missions (path planning, line following etc.). Likewise the system needs a good framework for setting parameters such as controller gain.

Appendices

APPENDIX A

Source code

A.1 CD

All source code, both for the flight controller and the visual odometry, is available on the supplied CD, or on *Campusnet*. The CD and *Campusnet* file also contains videos of test flights.

A.2 Github

The Visual Odometry was developed while using *Git* to do version control. *Github* was used to synchronize files between the embedded Linux computer and a workstation. The repository is at: `https://github.com/skrogh/oDroidDrone`

The repository might be updated in the future to also include the flight controller software and future developments on the code.

Bibliography

Harris, C. and Stephens, M.: 1988, A combined corner and edge detector, *In Proc. of Fourth Alvey Vision Conference*, pp. 147–151.

Li, M., Kim, B. and Mourikis, A. I.: 2013, Real-time motion estimation on a cellphone using inertial sensing and a rolling-shutter camera, *Proceedings of the IEEE International Conference on Robotics and Automation*, Karlsruhe, Germany, pp. 4697–4704.

Li, M. and Mourikis, A. I.: 2012, Improving the accuracy of ekf-based visual-inertial odometry, *Proceedings of the IEEE International Conference on Robotics and Automation*, Minneapolis, MN, pp. 828–835.

Lucas, B. D. and Kanade, T.: 1981, Iterative image registration technique with an application to stereo vision., *Proceedings of the 7th International Joint Conference on Artificial Intelligence* **2**, 674–679.

Madgwick, S. O. H., Harrison, A. J. L. and Vaidyanathan, R.: 2011, Estimation of imu and marg orientation using a gradient descent algorithm, *2011 IEEE INTERNATIONAL CONFERENCE ON REHABILITATION ROBOTICS (ICORR)* pp. –.

Madgwick, S., Vaidyanathan, R. and Harrison, A.: 2010, An efficient orientation filter for inertial measurement units (imus) and magnetic angular rate and gravity (marg) sensor arrays, *Technical report*, Department of Mechanical Engineering.
URL: *http://www.x-io.co.uk/res/doc/madgwick_internal_report.pdf*

Mourikis, A. I. and Roumeliotis, S. I.: 2006, A multi-state constraint kalman filter for vision-aided inertial navigation, *Technical report*, Dept. of Computer Science and Engineering, University of Minnesota. www.cs.umn.edu/∼mourikis/tech_reports/TR_MSCKF.pdf.

Shelley, M.: 2014, *Monocular visual inertial odometry on a mobile device*, Master's thesis, Technical University Munich, Germany.

Shi, J. and Tomasi, C.: 1994, Good features to track, *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition* pp. 593–600.

Thrun, S., Burgard, W. and Fox, D.: 2005, *Probabilistic Robotics*, MIT Press.

Wang, Jian and Holzapfel, Florian and Xargay, Enric and Hovakimyan Naira: 2013, Non-cascaded dynamic inversion design for quadrotor position control with L1 augmentation.

Wikipedia, t. f. e.: 2015, Singular value decomposition.
URL: *https://en.wikipedia.org/wiki/Singular_value_decomposition*