Project in Predictive Control and Real Time Systems

Marcus Greiff, Daniel Nilsson

May 5, 2016

Contents

1 Introduction

2 Dynamics

In this project, we consider the non-linear quadcopter equations as derived by Lukkonen et al. [1]. A brief description of the dynamics is given to define terms which will be used in the control scheme. Let

$$\boldsymbol{\xi} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}, \quad \boldsymbol{\eta} = \begin{bmatrix} \phi \\ \theta \\ \psi \end{bmatrix}, \quad \boldsymbol{\omega} = \begin{bmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \\ \omega_4 \end{bmatrix}, \tag{1}$$

where $\boldsymbol{\xi}$ [m] denotes the position of the centre of mass in a global cartesian coordinate system, $\boldsymbol{\eta}$ [rad] is the euler-angles in the body coordinate system and ω_i [rad/s] is the angular speed of the rotor i. For future reference, the basis vectors in the cartesian coordinate system are written $\hat{\cdot}$, and the subindexing $\cdot_B$ refers to the vector of matrix defined in the body coordinate system.

The translation from the global- to the body coordinate system is done by the orthogonal rotation matrix

$$\mathbf{R} = \begin{bmatrix} \cos(\psi)\cos(\phi) & \cos(\psi)\sin(\theta)\sin(\phi) - \sin(\psi)\cos(\phi) & \cos(\psi)\sin(\theta)\cos(\phi) + \sin(\psi)\sin(\phi) \\ \sin(\psi)\cos(\phi) & \sin(\psi)\sin(\theta)\sin(\phi) + \cos(\psi)\cos(\phi) & \sin(\psi)\sin(\theta)\cos(\phi) - \cos(\psi)\sin(\phi) \\ -\sin(\theta) & \cos(\theta)\sin(\phi) & \cos(\theta)\cos(\phi) \end{bmatrix} \tag{2}$$

such that a vector defined in the body system $\mathbf{v}_B$ can be translated to the global coordinate by the mapping

$$\mathbf{v} = \mathbf{R}^{-1}\mathbf{v}_B = \mathbf{R}^T\mathbf{v}_B. \tag{3}$$

The force generated by the rotor i is assumed to be proportional to the rotor speed squared,

$$f_i = c_2\omega_i^2 + c_1\omega_i + c_0 \approx k_i\omega_i^2 \tag{4}$$

in the positive $\hat{\mathbf{z}}_B$ direction with some constant k_i. In previous work, a non-linear regression of measured force as a function of rotor speed yielded $c_1, c_0 < 10^{-4}$. The approximation is deemed good enough, but the coefficients will be identified for our hardware.

Furthermore, we let the torque around each motor axis be written

$$\tau_{M_i} = b\omega_i^2 + I_M\dot{\omega}_i \tag{5}$$

where b is a drag constant and I_M is the rotor inertia. By virtue of symmetry and under the assumption that $k_i \approx k \in \mathbb{R}^+ \ \ \forall i$, the thrust and torque vectors in the body coordinate system can be written

$$\mathbf{T}_B = T\hat{\mathbf{z}}_B = \begin{bmatrix} 0 \\ 0 \\ k\sum_{i=1}^{4}\omega_i^2 \end{bmatrix}, \qquad \boldsymbol{\tau}_B = \begin{bmatrix} \tau_\phi \\ \tau_\theta \\ \tau_\psi \end{bmatrix} = \begin{bmatrix} kl(-\omega_2^2 + \omega_4^2) \\ kl(-\omega_1^2 + \omega_3^2) \\ \sum_{i=1}^{4}\tau_{M_i} \end{bmatrix} \tag{6}$$

To make the model more accurate, we introduce air resistance or drag, which increases with $\dot{\boldsymbol{\xi}}$ similarly to viscous friction. This drag matrix is defined as

$$\mathbf{D} = \begin{bmatrix} D_{11} & 0 & 0 \\ 0 & D_{22} & 0 \\ 0 & 0 & D_{33} \end{bmatrix} \tag{7}$$

where $D_{11} = D_{22} < D_{33}$, and the coefficients remain to be estimated. With the above definitions, the non-linear dynamics of the quadcopter can then be derived from the Newton-Euler equations as

$$\begin{cases} m\ddot{\boldsymbol{\xi}} = m\mathbf{G} + \mathbf{T}_B - \mathbf{D}\dot{\boldsymbol{\xi}} \\ \ddot{\boldsymbol{\eta}} = \mathbf{J}^{-1}(\boldsymbol{\eta})(\boldsymbol{\tau}_B - \mathbf{C}(\boldsymbol{\eta}, \dot{\boldsymbol{\eta}})\dot{\boldsymbol{\eta}}), \end{cases} \tag{8}$$

A brief description of $\mathbf{J}$ and $\mathbf{C}$ matrices can be found in **Section 8**, but the interested reader is referred to [1] for a more thorough derivation of the Newton-Lagrange equations. In the work of Lukkonen, this system was simulated in continuous time, and here we will take an alternate approach in order to implement the dynamics as a discrete time ROS node in Python.

2.1 Non-linear model

By defining the states and control signals as

$$\mathbf{x} = \begin{bmatrix} \boldsymbol{\xi} \\ \dot{\boldsymbol{\xi}} \\ \boldsymbol{\eta} \\ \dot{\boldsymbol{\eta}} \end{bmatrix} \in \mathbb{R}^{12\times 1}, \quad \text{and} \quad \mathbf{u} = \begin{bmatrix} T \\ \tau_\phi \\ \tau_\theta \\ \tau_\psi \end{bmatrix} \in \mathbb{R}^{4\times 1} \tag{9}$$

respectively, the full non-linear system can then be written

$$\begin{aligned} \dot{\mathbf{x}}(t) =& \mathbf{A}_c\mathbf{x}(t) + \mathbf{B}_c\mathbf{u}(t) + \mathbf{G}_c \\ \mathbf{y}(t) =& \mathbf{C}_c\mathbf{x}(t) \end{aligned} \tag{10}$$

with

$$\mathbf{A}_c = \begin{bmatrix} \mathbf{0} & \mathbb{I}_{3\times 3} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & -\frac{1}{m}\mathbf{D} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbb{I}_{3\times 3} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & -\mathbf{J}^{-1}(\boldsymbol{\eta})\mathbf{C}(\boldsymbol{\eta}, \dot{\boldsymbol{\eta}}) \end{bmatrix}, \mathbf{B}_c = \begin{bmatrix} \mathbf{0} & \mathbf{0} \\ \frac{1}{m}\mathbf{R}\hat{\mathbf{z}} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{J}(\boldsymbol{\eta})^{-1} \end{bmatrix}, \mathbf{G}_c \begin{bmatrix} \mathbf{0} \\ \mathbf{G} \\ \mathbf{0} \\ \mathbf{0} \end{bmatrix}, \mathbf{C}_c = \begin{bmatrix} \mathbf{0}_{7\times 2} & \mathbb{I}_{7\times 7} & \mathbf{0}_{7\times 2} \end{bmatrix} \tag{11}$$

The $\mathbf{C}_c$ matrix was chosen to reflect the available sensory information. The height z is measured by a pressure sensor, the angles $\boldsymbol{\eta}$ are retrieved from a gyroscope aboard the quadcopter and the velocities $\dot{\boldsymbol{\xi}}$ are integrated from readings of the combined sensory feedback from the accelerometer and magnetometer. In order to simulate the dynamics, the continuous time system (34) was implemented in Simulink (see `quadcopter_model.m`, **Section 8**), and validated by comparison to the results in [1].

The motors in the physical process are incredibly responsive, going from $||\boldsymbol{\omega}||_\infty \approx 0$ to $||\boldsymbol{\omega}||_\infty \approx 2.5 \cdot 10^4$ in < 180 ms. (see https://www.bitcraze.io/2015/02/measuring-propeller-rpm-part-3/). We have access to RPM measurements and can therefore construct a very fast PID loop for each

motor to keep $\boldsymbol{\omega}^2$ at a desired value. Consequently, it could be an idea to investigate methods of control when using rotor speeds as control signals. If we make the assumption that the PID loops are very fast, and that the copter will be operating close to a hovering position with bounds on the control signals, it might be feasible to assume a 1:1 relashionship between desired rotor speed and actual rotor speed and not include the PID dynamics in the system matrix. The main reason for this is that including the 12 additional states will make the problem too big to solve efficiently with the MPC formulation. This requires a mapping from rotor speeds to thrust and torques, which can be derived from (6),as

$$\begin{bmatrix} T \\ \tau_\phi \\ \tau_\theta \\ \tau_\psi \end{bmatrix} = \mathbf{M}_\omega \boldsymbol{\omega}^2 \quad \text{where} \quad \mathbf{M}_\omega = \begin{bmatrix} k & k & k & k \\ 0 & -kl & 0 & kl \\ -kl & 0 & kl & 0 \\ -b & b & -b & b \end{bmatrix} \quad \text{and} \quad \mathbf{M}_\omega^{-1} = \begin{bmatrix} \frac{1}{4k} & 0 & -\frac{1}{2kl} & \frac{1}{4b} \\ \frac{1}{4k} & -\frac{1}{2kl} & 0 & -\frac{1}{4b} \\ \frac{1}{4k} & 0 & \frac{1}{2kl} & \frac{1}{4b} \\ \frac{1}{4k} & \frac{1}{2kl} & 0 & -\frac{1}{4b} \end{bmatrix} \tag{12}$$

as derived from equation (6). The updated continuous system, with $\mathbf{u}(t) = \boldsymbol{\omega}^2(t)$ is then

$$\begin{aligned} \dot{\mathbf{x}}(t) =& \mathbf{A}^c \mathbf{x}(t) + \mathbf{B}^c \mathbf{M}_\omega \mathbf{u}(t) + \mathbf{G}^c \\ \mathbf{y}(t) =& \mathbf{C}^c \mathbf{x}(t) \end{aligned} \tag{13}$$

No matter which continuous system is used (linear or non-linear), the discrete time system is computed using zero-order hold at a time step h, with the discrete state space representation

$$x(t_k + h) = \mathbf{A}_d x(t_k) + \mathbf{B}_d \mathbf{u}(t_k) + \mathbf{G}_d \tag{14}$$

$$y(t_k) = \mathbf{C}_d x(t_k) \tag{15}$$

$$\tag{16}$$

where

$$\mathbf{A}^d = e^{\mathbf{A}^c h}, \qquad \mathbf{B}^d = \int_0^h e^{\mathbf{A}^c h} ds B^c, \qquad \mathbf{G}^d = h\mathbf{G}^c, \qquad \mathbf{C}^d = \mathbf{C}^c. \tag{17}$$

As the final implementation of the control system was done in ROS based in Python and C++, a script was written do simulate the system using only the scipy and numpy modules (see simulate_system.py, **Section 8**). The result (see Figure **??**).

2.2 Linearised dynamics

Consider the continuous time system with input signal $\mathbf{u} = \boldsymbol{\omega}^2$, it is clear then that the non-linear part of the state-space matrices only concern the $\boldsymbol{\eta}, \dot{\boldsymbol{\eta}}$-states and the input signal. By defining the linearisation point of the euler angles around the stable point $\boldsymbol{\eta}_p = \dot{\boldsymbol{\eta}}_p = 0$, and the rotor velocities around the rotor speed required to hover $\boldsymbol{\omega}_p = \sqrt{\frac{gm}{4k}}[1, 1, 1, 1]^T$, such that the deviation from the linearisation points become

$$\begin{aligned} \Delta\boldsymbol{\eta} =& \boldsymbol{\eta} - \boldsymbol{\eta}_p \\ \Delta\dot{\boldsymbol{\eta}} =& \dot{\boldsymbol{\eta}} - \dot{\boldsymbol{\eta}}_p \\ \Delta\boldsymbol{\omega} =& \boldsymbol{\omega} - \boldsymbol{\omega}_p \end{aligned} \tag{18}$$

Close to this point, the non-linear component of the $\mathbf{A}_c$-matrix ($\mathbf{J}^{-1}\mathbf{C}\dot{\boldsymbol{\eta}}$) can be linearised as

$$\frac{\partial \mathbf{J}^{-1}(\boldsymbol{\eta})\mathbf{C}(\boldsymbol{\eta},\dot{\boldsymbol{\eta}})\dot{\boldsymbol{\eta}}}{\partial \boldsymbol{\eta}}\Big|_{\boldsymbol{\eta}_p,\dot{\boldsymbol{\eta}}_p} = \mathbf{0}_{3\times 3}, \quad \text{and} \quad \frac{\partial \mathbf{J}^{-1}(\boldsymbol{\eta})\mathbf{C}(\boldsymbol{\eta},\dot{\boldsymbol{\eta}})\dot{\boldsymbol{\eta}}}{\partial \dot{\boldsymbol{\eta}}}\Big|_{\boldsymbol{\eta}_p,\dot{\boldsymbol{\eta}}_p} = \mathbf{0}_{3\times 3} \tag{19}$$

implying that the linearized continuous-time system matrix is

$$\tilde{\mathbf{A}}_{\Delta\omega} = \begin{bmatrix} \mathbf{0} & \mathbb{I}_{3\times 3} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & -\frac{1}{m}\mathbf{D} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbb{I}_{3\times 3} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \end{bmatrix}. \tag{20}$$

Using the fact that

$$\mathbf{J}^{-1}(\mathbf{0}) = \begin{bmatrix} \frac{1}{I_{xx}} & 0 & 0 \\ 0 & \frac{1}{I_{yy}} & 0 \\ 0 & 0 & \frac{1}{I_{zz}} \end{bmatrix}, \quad \mathbf{R}(\mathbf{0})\hat{\mathbf{z}}_B = \hat{\mathbf{z}} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \tag{21}$$

we conclude that the linearized continuous-time $\mathbf{B}$-matrix can be written

$$\tilde{\mathbf{B}}_{\Delta\omega} = \frac{\partial \mathbf{B}^c(\boldsymbol{\eta})\mathbf{M}_\omega \boldsymbol{\omega}^2}{\partial \boldsymbol{\omega}}\Big|_{\boldsymbol{\eta}=\mathbf{0},\boldsymbol{\omega}=\boldsymbol{\omega}_p} = \sqrt{\frac{gm}{k}} \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ k & k & k & k \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & \frac{-kl}{I_{xx}} & 0 & \frac{kl}{I_{xx}} \\ \frac{-kl}{I_{yy}} & 0 & \frac{kl}{I_{yy}} & 0 \\ \frac{-b}{I_{zz}} & \frac{b}{I_{zz}} & \frac{-b}{I_{zz}} & \frac{b}{I_{zz}} \end{bmatrix} \tag{22}$$

It is then clear that using this linearised process model

$$\begin{aligned} \dot{\Delta\mathbf{x}} &= \tilde{\mathbf{A}}_{\Delta\omega}\Delta\mathbf{x} + \tilde{\mathbf{B}}_{\Delta\omega}\Delta\boldsymbol{\omega} \\ \Delta\mathbf{y} &= \mathbf{C}_c\Delta\mathbf{x} \end{aligned} \tag{23}$$

the number of observable and controllable states are

$$\begin{cases} \operatorname{rank}\Big(\begin{bmatrix} \tilde{\mathbf{B}}_{\Delta\omega} & \cdots & \tilde{\mathbf{A}}_{\Delta\omega}^{n-1}\tilde{\mathbf{B}}_{\Delta\omega} \end{bmatrix} \Big) = 8 \neq 12 \\ \operatorname{rank}\Big(\begin{bmatrix} \mathbf{C}_c^T & \cdots & (\mathbf{C}_c\tilde{\mathbf{A}}_{\Delta\omega}^{n-1})^T \end{bmatrix}^T \Big) = 8 \neq 12 \end{cases} \tag{24}$$

respectively. Closer investigation shows that we have a pole-zero cancellation of the positional x, y-states and their derivarives, implying that we need to have external motion capture of these states in order to use Kalman filters for state estimation (requiring full observability) or linear quadratic gaussian regulator for reference tracking (requiring full controllability). This could be done by integrating measurements from the accelerometer twice and introducing a complementary filter to decrease positional drift in stationarity, but is best handled by some external of motion capture. Yet another alternative is to augment the model with a stabilising PD-regulator, which will be shown to render the system observable and controllable.

2.3 Rotor-loop dynamics

TODO: Find transfer function from motor current to rotor speed on the form

$$H_{I\to\dot{\omega}}(s) \approx \frac{b_0}{s+a_0} \tag{25}$$

The step response of the system is then

$$\frac{b_0}{s+a_0}\frac{1}{s} = \frac{b_0}{a_0}\Big(\frac{1}{s} - \frac{1}{s+a_0}\Big) \xrightarrow{\mathcal{L}_t} \frac{b_0}{a_0}(e^{-t} - e^{-a_0 t}) \tag{26}$$

where the coefficients can be found with a regression. A simple test would be to measure the response of two unit steps in succession to determine both coefficients but for the purposes of simulation, a rough estimate would be $a_0 = 40$ and $b_0 \approx 40{*}\omega_{max}/I_{max}$, which mathces the specification $||\boldsymbol{\omega}||_\infty \approx 0$ to $||\boldsymbol{\omega}||_\infty \approx 2.5 \cdot 10^4$ in < 180 ms (see https://www.bitcraze.io/2015/02/measuring-propeller-rpm-part-3/). Here, ω_{max} is the maximum rotor speed of the quadcopter and $I_{max} \approx$ 1100 mA is the peak motor current.

2.4 Inner stabilising controller

The system is inherently unstable, as the only truly stable position is when the copter hovers at a certain point in space, i.e. $\dot{\boldsymbol{\xi}} = \boldsymbol{\eta} = \dot{\boldsymbol{\eta}} = 0$. In addition, we know that the x- and y-positions are unobservable and uncontrollable (as shown in the previous section), meaning the we cannot have good positional control using designs based solely on (34). However, if a stabilising inner controller is included, we can not only stabilise the system, but also make the x- and y- states observable in the augmented system. In addition, the stabilising controller may be run on the crazyflie at very high frequencies (currently, the inner stabilising loop is run at 500 Hz) increasing overall performance of the final implementation. In this section, we define three controllers the purpose of creating a stable open loop system in with the control signal

$$\mathbf{u} = \begin{bmatrix} z_{ref} & \phi_{ref} & \theta_{ref} & \psi_{ref} & \dot{z}_{ref} & \dot{\phi}_{ref} & \dot{\theta}_{ref} & \dot{\psi}_{ref} \end{bmatrix}^T \in \mathbb{R}^{8\times 1}, \tag{27}$$

with subindices $(\cdot)_{ref}$ denotes reference values for the respective state. This state space vector is used as the $x, y, \dot{x}, \dot{y}$ states are unobservable in both the linearised and the non-linear dynamics, and cannot be observed in the inner control loop.

A common way of accomplishing the goal is by using a non-model based design, and implementing a PD-controller. Such a controller was derived in e.g. [1] [2], and can be summarised in the scheme

$$\begin{aligned} T &= (g + K_{D,z}(\dot{z}_{ref} - \dot{z})) + K_{P,z}(z_{ref} - z))\frac{m}{\cos(\phi)\cos(\theta)} \\ \tau_\phi &= (K_{D,\phi}(\dot{\phi}_{ref} - \dot{\phi})) + K_{P,\phi}(\phi_{ref} - \phi))I_{xx} \\ \tau_\theta &= (K_{D,\theta}(\dot{\theta}_{ref} - \dot{\theta})) + K_{P,\theta}(\theta_{ref} - \theta))I_{yy} \\ \tau_\psi &= (K_{D,\psi}(\dot{\psi}_{ref} - \dot{\psi})) + K_{P,\psi}(\psi_{ref} - \psi))I_{zz} \end{aligned} \tag{28}$$

In using this controller, setting all references to 0 results in a stable hovering system assuming the state estimation is good. In our implementations, we also include the mapping of thrusts

and torques to references in rotor speeds, given by (12). Note that the controller is non-linear in terms of thrust, and using the PD augmented system for model based outer loop control (such as MPC) does not capture the full PD controller dynamics.

As an alternative, a fully linear LQR-controller is proposed for the same purpose, using the linearised model (23) and only including the controllable modes. By the standard approach presented in [3], the cost function is defined as

$$J = \int_0^\infty \mathbf{x}^T(t)\mathbf{Q}\mathbf{x}(t) + \mathbf{u}^T(t)\mathbf{R}\mathbf{u}(t)dt \tag{29}$$

where $\mathbf{Q}, \mathbf{R} > \mathbf{0}$ determines the costs of certain states and control signals respectively. The optimal linear feedback law is then

$$\mathbf{u} = -\underbrace{\mathbf{R}^{-1}\mathbf{B}^T\mathbf{S}}_{\mathbf{K}}\mathbf{x} \tag{30}$$

where the symmetric matrix $\mathbf{S}$ solves the associated Riccati equation,

$$\mathbf{A}^T\mathbf{S} + \mathbf{S}\mathbf{A} - \mathbf{S}\mathbf{B}\mathbf{R}^{-1}\mathbf{B}^T\mathbf{S} + \mathbf{Q} = 0. \tag{31}$$

This method naturally has the drawback of using the linearised model, which only accurately describes the system close to the stable, hovering state. But by bounding the pitch and yaw angles to $\varphi, \theta \in [-0.3, 0.3]$ rad, the controller performs very well. One inherent disadvantage with LQR is that it is a proportional state feedback system, and therefore may give rise to stationary errors. To combat this issue, an LQR scheme with integrator (LQRi) was implemented by introducing four additional states,

$$\mathbf{x}_i = \begin{bmatrix} z_i & \phi_i & \theta_i & \psi_i \end{bmatrix}^T \in \mathbb{R}^{4\times 1} \tag{32}$$

such that

$$\mathbf{x}_i = \int \mathbf{e}(t)dt = \int \mathbf{C}\mathbf{x} - \mathbf{y}dt, \qquad \text{with } \mathbf{C} = \begin{bmatrix} \mathbb{I}_{4\times 4} & \mathbf{0}_{4\times 4} \end{bmatrix} \tag{33}$$

thereby only integrating the positional control errors, $\mathbf{e}(t)$. The reason for not integrating control errors in the velocity is to avoid duplicate states(which in this case renders the system uncontrollable eliminating the possibility of using an LQR scheme).

The extended state space model becomes

$$\begin{aligned} \dot{\mathbf{x}}_e(t) =& \mathbf{A}_e\mathbf{x}(t) + \mathbf{B}_e\mathbf{u}(t) + \mathbf{G}_e \\ \mathbf{y}(t) =& \mathbf{C}_e\mathbf{x}(t) \end{aligned} \tag{34}$$

with the extended state vector $\mathbf{x}_e = \begin{bmatrix} \mathbf{x} & \mathbf{x}_i \end{bmatrix}^T \in \mathbb{R}^{12\times 1}$ and

$$\mathbf{A}_e = \begin{bmatrix} \mathbf{A}_c & \mathbf{0}_{8\times 4} \\ \mathbf{C} & -\mathbb{I}_{4\times 4} \end{bmatrix} \in \mathbb{R}^{12\times 12}, \quad \mathbf{B}_e = \begin{bmatrix} \mathbf{B}_c \\ \mathbf{0}_{4\times 4} \end{bmatrix} \in \mathbb{R}^{12\times 4}, \quad \mathbf{C}_e \in \mathbb{R}^{N\times 12}, \quad \mathbf{G}_e \in \mathbb{R}^{12\times 1} \tag{35}$$

Here, the integer $4 \leq N \leq 8$ denotes the number of observed states (not confined to the four positional used in $\mathbf{C}_e$, but can output velocities as well).

In order to allow large integral gains, the common conditional anti-windup (AW) scheme was implemented so as to set the integral part to zero when having any of the four of the rotor speeds

in saturation. The implemented AW scheme can be summarised in terms of the positional control error,

$$\begin{cases} \mathbf{e}(t) := \mathbf{C}\mathbf{x} - \mathbf{y}, & \text{if} \quad \text{sat}(\boldsymbol{\omega}) - \boldsymbol{\omega} = \mathbf{0} \\ \mathbf{e}(t) := \mathbf{0}, & \text{if} \quad \text{sat}(\boldsymbol{\omega}) - \boldsymbol{\omega} \neq \mathbf{0} \end{cases}, \tag{36}$$

with the saturation function defined as

$$\text{sat}(x) = \begin{cases} x_{max} & x_{max} < x \\ x & x_{min} \leq x \leq x_{max} \\ x_{min} & x < x_{min} \end{cases} \tag{37}$$

Using this effectively disablines integral action when any motor is in a saturated state, and has the added benefit of being very predictable. When setting up the state space model for use in MPC, we would in practice only have to define two different linearised state space models (the LQR when in saturation and the LQRi otherwise). The only downside is the additional four states, which greatly increase the complexity of the optimisation problem in MPC.

The three stabilising controllers were simulated using the parameters described in the work of Lukkonen [1] (for which the above PD regulator was developed and tuned) and at an inner loop rate of 100 Hz (note that the original stabilising controller on the crazyflie runs at 500 Hz). The generated reference trajectory lowpass filtered unit steps the amplitudes $z \in [0, 30]$ [m] and $\phi, \theta, \psi \in [-0.3, 0.3]$ [rad], which is far more aggressive than anything that will be run on the real process (see Figure 1). The LQG regulators were tuned to give reasonable trajectory following in terms of Euler angles, with more pronunciation of yaw and elevation (as the yaw will later be assumed to be 0 in the external loop). As expected the LQG-controller performs well, but also yields the predicted steady state errors. In the depicted simulation, the stationary error were to be of magnitude 7 cm at $t \approx 37$ s, which is unacceptable precise indoor flying. As a contrast, both the PD and LQRi (with AW) yield errors of ≈ 0.1 cm less than 10 seconds after a reference change, with the error still decreasing. We also note that the reference following in elevation is much more precise with the LQRi compared to the PD alternative, and that it is only outperformed by the PD in terms of overshoot when making large reference changes in angular positions. However, an acceptable angular error of < 0.005 [rad] is reached around the same time with both regulators. In conclusion, the PD and LQRi perform almost en par, with a slight edge to the LQRi in terms of altitude changes and in less aggressive trajectories. Both should be implemented and tested on the physical process.

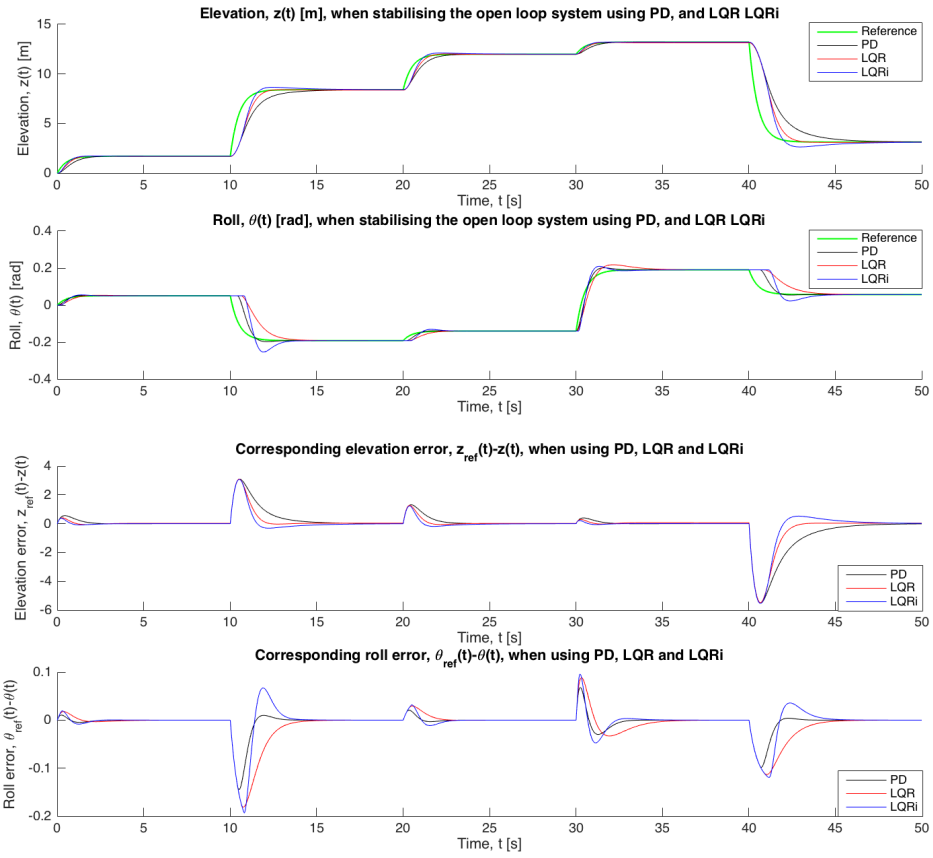

Figure 1: Comparision between the stabilising controllers with simulated elevation and roll (top) and the corresponding control errors (bottom).

3 State estimation

To estimate the states properly, many approaches are considered and compared in simulation before making a real time implementation. As we know from previous sections, the positions and velocities in the x- and y-directions are unobservable in the considered non-linear dynamics. Since these four states are not needed in stabilising the quadcopter, our approach will be to estimate the remaining eight states on the quadcopter and run this in conjunction with a stabilising inner controller (discussed above). This inner loop will be run at ≈ 100 Hz and can be completely independent of external control and estimation. In order to estimate the four unobservable states, a kinect 1 camera will be run on the host computer, yielding positional measurements at ≈ 30 Hz. These measurements will not be good enough to accurately determine the $x-$ and $y-$velocities, for which the acceleration data is required. However, this data will be delayed when sent to the host computer from the crazyflie via bluetooth, and likewise, the estimated positions and speeds will suffer from a time delay when transmitted back to the crazyflie.

The problem in the inner loop is to accurately estimate the eight states

$$\hat{\mathbf{x}}^{(inner)} = \begin{bmatrix} z & \phi & \theta & \psi & \dot{z} & \dot{\phi} & \dot{\theta} & \dot{\psi} \end{bmatrix}^T \tag{38}$$

given the torques and thrusts or the rotor angular speeds, subject to constraints in computational power and the highly non-linear dynamics of the crazyflie. To achieve this, a regular Kalman filter (KF) is tested, using the dynamics linearised around the stable hovering state to compute the filter gain [3]. In addition the extended kalman filter (EKF) is tested. The EKF effectively linearises the system around the previous state estimate on each time step, which is done by computing an expression for the Jacobian offline and then evaluating it on-line. As the dynamics matrix contains inverses, this puts great constraints on how fast the inner loop can be run. Finally, the unscented Kalman filter (UKF) and generic particle filter (GPF) are implemented and tested on the non-linear dynamics.

Similarily, the problem in the outer loop is to accurately estimate the states

$$\hat{\mathbf{x}}^{(outer)} = \begin{bmatrix} x & y & z & \dot{x} & \dot{y} & \dot{z} \end{bmatrix}^T \tag{39}$$

given time delayed measurements of quadcopter acceleration, current measurements of position from the kinect and predict the future position and speed of the quadcopter to combat the time delay in transmitting data back to the crazyflie. Fo this purpose a regular KF is tested along with an asynchronous Kalman filter of our own design.

3.1 Discrete Kalman filter (KF)

Initialize: $\mathbf{x}$;
Prediction step;
Correction step;

Algorithm 1: Rough outline of the discrete time Kalman filter update

3.2 Discrete extended kalman filter (EKF)

The EKF is implemented using the nonlinear system representation

$$\begin{aligned} \mathbf{x}_{k+1} &= \mathbf{F}(\mathbf{x}_k, \mathbf{u}_k, \mathbf{v}_{k+1}) \\ \mathbf{y}_k &= \mathbf{H}(\mathbf{x}_k, \mathbf{e}_{k+1}) \end{aligned} \tag{40}$$

denoting the respective state and measurement equation Jacobians as

$$\mathbf{J}_k^F = \frac{\partial \mathbf{F}(\mathbf{x}, \mathbf{u}_k, \mathbf{v})}{\partial \mathbf{x}}\Big|_{\mathbf{x}_k} \qquad \mathbf{J}_k^H = \frac{\partial \mathbf{H}(\mathbf{x}, \mathbf{e})}{\partial \mathbf{x}}\Big|_{\mathbf{x}_k} \tag{41}$$

and again assuming a measurement noise zeros mean gaussian noise, with

$$\mathbf{v}_k \sim \mathcal{N}(0, \mathbf{Q}) \quad \mathbf{e}_k \sim \mathcal{N}(0, \mathbf{R}). \tag{42}$$

Commonly used in GPS systems, the EKF is situational and not an optimal filter as the Jacobians used in the covariance prediction is a first order approximation in a multivariate taylor series expansion. As such, we only get an estimated covariance and not the true covariance, which may cause the state estimation to diverge if the system is non-linear enough and enough time passes. If the system, like the quadcopter process, is badly conditioned and we force it to great extremes (i.e. flying aggressivly) the EKF may fail. For a more thorough theoretical discussion on the EKF, the reader is referred to [4].

In our implementation, the Jacobian $\mathbf{J}_k^F$ is computed offline and stored as a symbolic expression to be evaluated on each iteration (see Algortithm 2). The initial covariance and Expressing the Jacobian to a non-linear system including inverses (recall $\mathbf{J}^{-1}$ in equation (11)) could put constraints on the rate at which the estimations are made, as evaluation of the expression may take considerable time.

Initialize: $\mathbf{x}_0$, $\mathbf{P}_0$, $\mathbf{Q}$, $\mathbf{R}$, $\mathbf{F}(\mathbf{x}, \mathbf{u})$, $\mathbf{C}_d$, $\mathbf{J}^F(\mathbf{x})$;
for *k = 1,...,k_{max}* **do**
  Receive $\mathbf{u}_k, \mathbf{z}_k$;
  Prediction step
  $\hat{\mathbf{x}}_k^- = \mathbf{F}(\hat{\mathbf{x}}_{k-1}, \mathbf{u}_{k-1}, \mathbf{v})$;
  $\mathbf{P}_k^- = \mathbf{J}_{k-1}^F \mathbf{P}_{k-1} \mathbf{J}_{k-1}^F + \mathbf{Q}$;
  Correction step
  $\mathbf{K}_k = \mathbf{P}_k^- \mathbf{C}_d^T (\mathbf{C}_d \mathbf{P}_k^- \mathbf{C}_d^T + \mathbf{R})^{-1}$;
  $\hat{\mathbf{x}}_k = \hat{\mathbf{x}}_k^- + \mathbf{K}_k(\mathbf{y_k} - \mathbf{C}_d \mathbf{x}_k^-)$;
  $\mathbf{P}_k = (\mathbb{I} - \mathbf{K}_k \mathbf{C}_d)\mathbf{P}_k^-$;
end

Algorithm 2: Rough outline of the discrete time extended Kalman filter (EKF)

For the considered dynamics, the measurement function $\mathbf{H}(\mathbf{x}, \mathbf{e})$ is linear and can therefore be replaced by the discrete time measurement matrix in (11), such that $\mathbf{H}(\mathbf{x}, \mathbf{e}) = \mathbf{C}_d \mathbf{x}$, with $\mathbf{J}_k^H = \mathbf{C}_d \forall \mathbf{x}$.

3.3 Unscented kalman filter (UKF)

The unscented Kalman filter is well described in [5], where it is qualitatively compared to the EKF on the chaotic Mackey Glass time series. The interested reader is referred to [5] for more details, but a rough sketch of the algorithm as implemented in Python/ROS/Simulink is given (see Algorithm 3) with a few cliff notes. For consistency, the same assumptions the UKF uses the same system description and assumptions defined in the previous section on the EKF, see equations (40) to (42).

Initialize: $\mathbf{x}_0$, $\mathbf{P}_0$, W^m, W^c, $\mathbf{Q}$, $\mathbf{R}$, $\mathbf{F}(\mathbf{x}, \mathbf{u})$, $\mathbf{H}(\mathbf{x})$;
for *k = 1,...,k_{max}* **do**
  Receive $\mathbf{u}_{k-1}, \mathbf{z}_k$;
  Compute weights and σ-points
  $\sqrt{\mathbf{P}} = \sqrt{L + \lambda} \cdot cholesky(\mathbf{P}_{k-1})^T$;
  $X_\sigma = [\mathbf{x}_{k-1}, \mathbf{x}_{k-1} - \sqrt{\mathbf{P}}, \mathbf{x}_{k-1} + \sqrt{\mathbf{P}}]$;
  Prediction step
  for *i = 1,...,2L+1* **do**
    $X_i^f = \mathbf{F}(X_{\sigma,i}, \mathbf{u}_{k-1}, \mathbf{v}_k)$;
    $Y_i^f = \mathbf{H}(X_{f,i}, \mathbf{e}_k)$;
  end
  $\hat{\mathbf{x}}_k^- = \sum_{i=1}^{2L+1} W_i^m X_i^f$;
  $\hat{\mathbf{y}}_k^- = \sum_{i=1}^{2L+1} W_i^m Y_i^f$;
  $\mathbf{P}_{xx} = \sum_{i=1}^{2L+1} W_i^c (\hat{\mathbf{x}}_k^- - X_i^f)(\hat{\mathbf{x}}_k^- - X_i^f)^T$;
  $\mathbf{P}_{xy} = \sum_{i=1}^{2L+1} W_i^c (\hat{\mathbf{x}}_k^- - X_i^f)(\hat{\mathbf{y}}_k^- - Y_i^f)^T$;
  $\mathbf{P}_{yy} = \sum_{i=1}^{2L+1} W_i^c (\hat{\mathbf{y}}_k^- - Y_i^f)(\hat{\mathbf{y}}_k^- - Y_i^f)^T$;
  Correction step
  $\mathbf{K}_k = \mathbf{P}_{xy}\mathbf{P}_{yy}^{-1}$;
  $\hat{\mathbf{x}}_k = \hat{\mathbf{x}}_k^- + \mathbf{K}_k(\mathbf{y} - \mathbf{C}_d\mathbf{x}_k^-)$;
  $\mathbf{P}_k = \mathbf{P}_{xx} - \mathbf{K}_k\mathbf{P}_{yy}\mathbf{K}_k^T$;
end

Algorithm 3: Rough outline of the implemented discrete time unscented Kalman filter (UKF)

The UKF relies on throwing points (σ-points) in the state space through the nonlinearity and then finding a weighted mean state in which the system is most likely to be. This method has the advantage of not using any approximations (in contrast to the EKF), but performance depends on the system at hand and how the filter is tuned. For instance, if the spread of sigma points around the mean state, α, is set too large the non-linearity may not be portrayed accurately close

to the mean state. In comparison to the EKF, the UKF has the advantage of scaling nicely with the number of states, L, requiring $2L + 2$ σ-points to be run through the non-linearity at each UKF update. This can be compared to the EKF where we would need to evaluate the symbolical expression, or use 2L function evaluations (of $\mathbf{F}$) if computing the Jacobian using a first order central difference scheme, at the cost of greater numerical error in the covariance prediction. The UKF will have to be compared to the alternatives in simulation to make a statement about which to use in the physical process.

3.4 Particle filter (GPF)

Attempted to implement the generic particle filter (see algorithm 3 on page 8 in [6]). To combat degeneracy in the filter the systematic resampling is implemented due to it's simplicity and performance when compared to e.g. multinomial resampling [7] [8].

TODO: Find out why the state amplitude becomes so small after resampling.

3.5 Asynchronous Kalman filter (AKF)

TODO: Write out method ...

3.6 Filter comparison

All considered filters (KF/EKF/UKF/GPF/AKF) were implemented in Simulink as S-functions to allow the use of nestled- and external .m-functions (see `*/kalman_filters/`). As an example, a double integrator with linear dynamics was set up to demonstrate the filters and how they are incorporated into a simulink model (see `*/Examples/double_integrator_filter_test/`). All filters are initialised with an `*_init`-file, in which a KFparam struct variable is created, defining the system and the necessary filter parameters. The code was written with the intent of being reusable in future projects, and is essentially a Simulink state estimation toolbox.

When running the Simulink examples, the state estimation model block can be set with a filter by selecting one of the models in `*/kalman_filters/`. The system is specified in the `init_filters_example.m` file, and the filter parameters are set to yield good estimation of the unknown angular velocities, $\dot{\boldsymbol{\eta}}_k$ (see Figure 2). Simulink is solely used for the purpose of simulating the filters, the more promising filters (KF/UKF/AKF) were then implemented in the Python `crazylib` module which can be used directly in the ROS nodes.

3.6.1 Inner loop

The filters (KF/EKF/UKF/GPF) were applied and tested on nonlinear quadcopter dynamics, running with an inner stabilising PD loop at 100 Hz and a PID position controller at 50 Hz in the outer loop (see `*/Examples/inner_loop/inner_filter_test/`). To test performance, the measurements of $\boldsymbol{\xi}_k, \dot{\boldsymbol{\xi}}_k, \boldsymbol{\eta}_k$ were corrupted with gaussian zero-mean uncorrelated noise with a variance of ≈ 0.3 [m] on the positional states ($\boldsymbol{\xi}_k$), a variance of 0.1 [m/s] on the velocities ($\dot{\boldsymbol{\xi}}_k$) and a variance of 0.01 on the Euler angles ($\dot{\boldsymbol{\xi}}_k$). The angular velocities were completely omitted and the process noise was assumed to be $\mathbf{0}$ at all times.

As can be seen, the regular KF performs badly when the system is far from the stable state, and the over all error is here visibly the largest. The EKF performs better, but when simulating

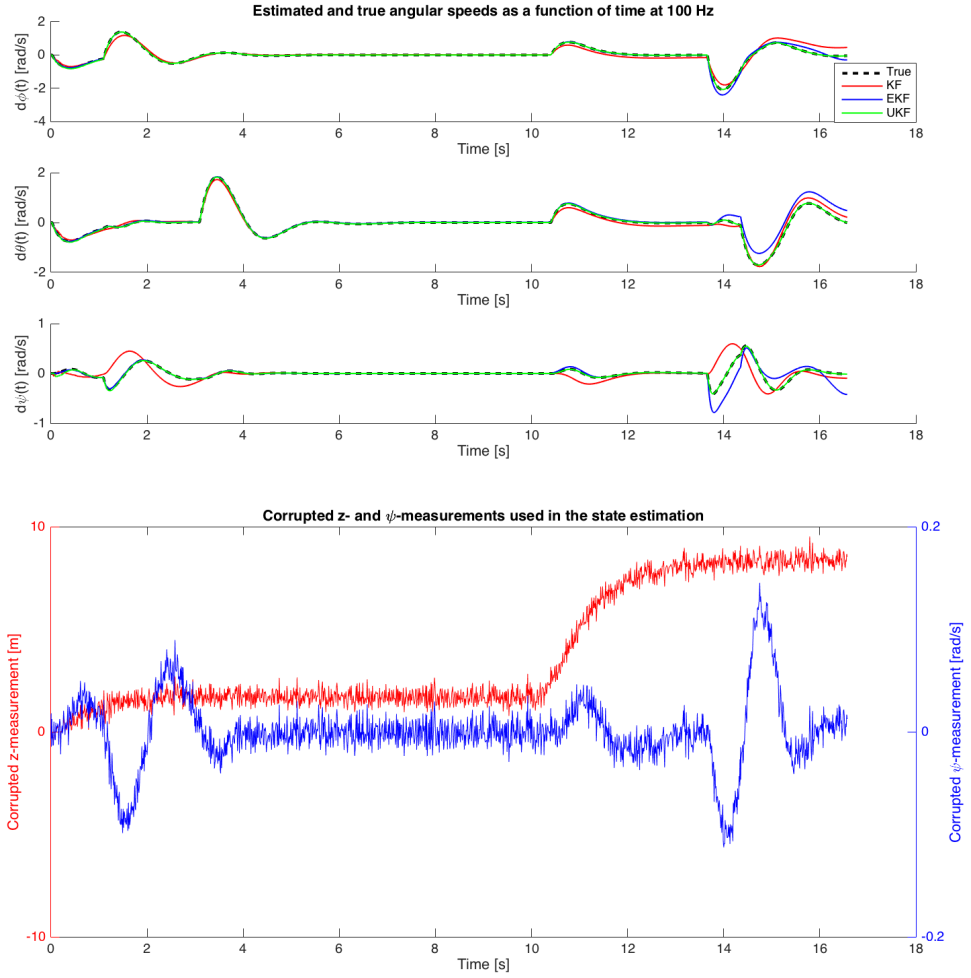

Figure 2: Demonstration of the drawbacks in using a regular Kalman compared to the EKF when estimating an unknown $\dot{\boldsymbol{\eta}}_k$, using (i) the same parameters in both filters (ii) having $\dot{\boldsymbol{\eta}}_k$ unknown and using (iii) corrupted measurements of $\boldsymbol{\xi}, \dot{\boldsymbol{\xi}}_k, \boldsymbol{\eta}_k$.

the system at an inner loop sample rate of 100 Hz, the EKF is demands processing power to a point where it becomes infeasible to implement in the physical process (unless other methods of computing the Jacobian are considered). In addition, the EKF estimation becomes more unstable as the step size increases, and already at a sample period of $T_s = 0.01$ [s], the estimation becomes very poor after ≈ 16 [s] and the covariance matrix requires resetting (at higher sample rates such as 500 Hz, the filter performs well for an entire simulation duration of 50 s). This is due to the approximations used in the covariance updates of the EKF, and it should be noted that the trajectory is very aggressive, and that the EKF may not diverge during the battery lifetime (≈ 5 minutes) if following smother trajectories.

The UKF has to be set with a fairly high $\alpha \approx 0.5$ in order to spread the sigma points enough to properly capture the non-linear dynamics. It is far more economical in terms of computational power than the EKF and follows the true values much more accurately than any of the previously tested filters. This is clear when looking at the the error metric

$$E(\mathbf{x}, \hat{\mathbf{x}}) = \sum_{i}^{3} \int_{0}^{t_s} |\dot{\boldsymbol{\eta}}_i(t) - \dot{\hat{\boldsymbol{\eta}}}_i(t)| dt \tag{43}$$

for the the estimation of the unknown states during a simulation time $t_s = 17$ [s]. Here, $E_{KF} \approx 4.2768$, $E_{EKF} \approx 2.4781$, $E_{UKF} \approx 0.1990$ showing that for this specific problem, the UKF the the better alternative in terms of accuracy and computational power (compared to the EKF). In conclusion, the only viable option is to run the UKF in the inner loop, but the GPF has yet to be tested.

3.6.2 Outer loop

As we are using a linear model, the Kalman Filter will yield optimal estimations assuming the measurement noise is zero mean and gaussian. Consequently, the KF and an asynchronous variation of it was tested to cope with the delays when sending data to and from the crazyflie. To achieve this, two basic methods are considered. The first is to ignore the time delayed accelerometer data, and simply consider a 3-dimensional discrete time double integrator

$$\mathbf{x}_{k+1} = \begin{bmatrix} \mathbb{I} & h \cdot \mathbb{I} \\ \mathbf{0} & \mathbb{I} \end{bmatrix} \mathbf{x}_k, \qquad \mathbf{y}_k = \begin{bmatrix} \mathbb{I} & \mathbf{0} \end{bmatrix} \mathbf{x}_k, \qquad \mathbf{0}, \mathbb{I} \in \mathbb{R}^{3\times 3} \tag{44}$$

to filter the measured positions from the Kinect and estimate the velocities, with

$$\mathbf{x}_k = \begin{bmatrix} x & y & z & \dot{x} & \dot{y} & \dot{z} \end{bmatrix}^T. \tag{45}$$

The second approach is to use discrete time triple-integrator model,

$$\mathbf{x}_{k+1} = \begin{bmatrix} \mathbb{I} & h \cdot \mathbb{I} & \frac{h^2}{2} \cdot \mathbb{I} \\ \mathbf{0} & \mathbb{I} & h \cdot \mathbb{I} \\ \mathbf{0} & \mathbf{0} & \mathbb{I} \end{bmatrix} \mathbf{x}_k, \qquad \mathbf{y}_k = \begin{bmatrix} \mathbb{I} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbb{I} \end{bmatrix} \mathbf{x}_k, \qquad \mathbf{0}, \mathbb{I} \in \mathbb{R}^{3\times 3} \tag{46}$$

where

$$\mathbf{x}_k = \begin{bmatrix} x & y & z & \dot{x} & \dot{y} & \dot{z} & \ddot{x} & \ddot{y} & \ddot{z} \end{bmatrix}^T. \tag{47}$$

The KF and AKF are then applied to the measurements as seen on the host computer, i.e. the noisy positions (no delay) and, if applicable, vey noisy acceleration measurements delayed by

some time $t_1 \sim \mathcal{N}(0.03, 0.002)$ [s], where the variance captures non-deterministic behaviour of the ROS publishers and subscribers. The estimated outputs of the filters are then delayed by t_1 again (to simulate being sent back to the crazyflie), and are then compared to the true values. As a simple test, a reasonably aggressive one dimensional acceleration and the corresponding position were used as ground truth values, and the filter was run at 30 Hz corresponds roughly to how fast data is generated by the kinect 1 camera. The estimated states were then moving average filtered (unweighted MA of order 10) which has a negligible phase lag at the frequencies of 0.4 Hz (see Figure 3). Note that neither method is dependent of complete system dynamics, and won't require too much bandwidth in the communication between the host and the crazyflie.

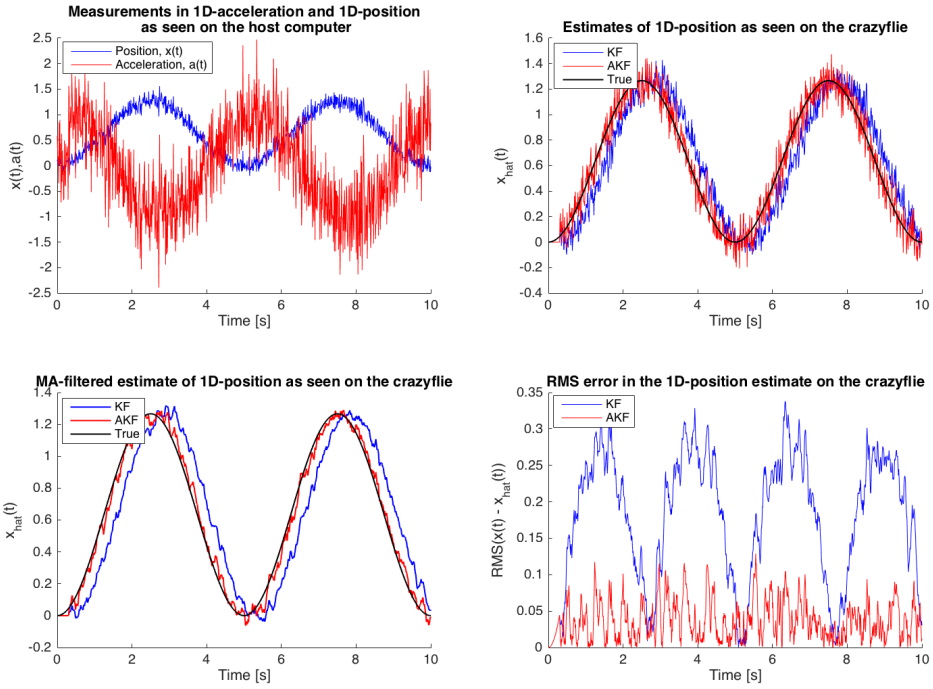

Figure 3: Comparison between KF and AKF using a 1D discrete triple integrator. *Top left*: The measured acceleration and position as seen on the host computer, the true values have been heavily corrupted by noise and the acceleration is delayed by t_1. *Top right*: The estimated positions using the KF and AKF as seen on the crazyflie compared to the true position. Here the positional estimates from the kalman filters have been delayed another t_1 before reaching the quadcopter. *Bottom left*: The MA-filtered estimates (unweighted, order 10) as seen on the crazyflie compared to the true position (black). *Bottom right*: The RMS error between the estimated position and true position when the data reaches the crazyflie with KF (blue) and AKF (red).

4 Motion planning

In this part of the project, we consider the polynomial generation method advanced by Roy et.al. cite XX. The general method is presented to make the Matlab code coherent, using the same nomenclature as in the code. The idea is to set up a constrained QP problem and solve it using Matlab's `quadprog` to find a minimum snap trajectory. The code will later be rewritten in Python/ROS with CVXGEN, and additional constraints will be enforced to ensure that the trajectory always stay in safe convex regions of space. These regions can conceivably be computed using IRIS, but will be assumed to be known initially.

Let the trajectory be composed of n polynomials $P_1(t), ..., P_n(t)$, where

$$P_k(t) = \sum_{i=0}^{N} p_i t^i, \qquad t \in [0, T_k] \tag{48}$$

with a maximum degree of $\deg(P_k) = N$, and a corresponding coefficient vector $\mathbf{p}_{(k)} = [p_{k,0}, ..., p_{k,N}]$. The problem is then to minimise a cost function for every polynomial spline

$$J(T_k) = \int_0^{T_k} c_0 P_k(t)^2 + c_1 P_k'(t)^2 + ... + c_N P_k^{(N)}(t)^2 dt = \mathbf{p}_{(k)}^T \mathbf{Q}_{(k)} \mathbf{p}_k \tag{49}$$

such that continuity is preserved and boundary conditions are enforced. In cite XX, the hessian matrix corresponding to a polynomial was derived by differentiating the term in the cost function containing the r^{th} derivative of $P_{(k)}(t)$ with regards to the polynomial coefficients p_i, i.e. finding

$$\mathbf{Q}_{r,(k)} = \frac{\partial^2}{\partial p_i \partial p_j} \int_0^{T_k} P_k^{(r)}(t)^2 dt \tag{50}$$

and constructing the matrix

$$\mathbf{Q}_{(k)} = \mathbf{Q}_{(k)} = \sum_{r=0}^{N} c_r \mathbf{Q}_{r,(k)} \in \mathbb{R}^{(N+1)\times(N+1)} \tag{51}$$

The complete constrained QP-formulation, including all n polynomials is then

$$\text{Minimize}\Big(\sum_{k=1}^{n} J(T_k) \Big) \qquad \text{subject to} \qquad \mathbf{A}\mathbf{p} - \mathbf{b} = 0 \tag{52}$$

where

$$\tag{53}$$

$$\sum_{k=1}^{n} J(T_k) = \begin{bmatrix} \mathbf{p}_{(1)} & \cdots & \mathbf{p}_{(n)} \end{bmatrix} \begin{bmatrix} \mathbf{Q}_{(1)} & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & \mathbf{Q}_{(n)} \end{bmatrix} \begin{bmatrix} \mathbf{p}_{(1)} \\ \vdots \\ \mathbf{p}_{(n)} \end{bmatrix} = \mathbf{p}^T \mathbf{Q} \mathbf{p}. \tag{54}$$

For the k^{th} polynomial, the r^{th} derivative can be written

$$P_k^{(r)}(t) = \sum_{n=r}^{N} \Big(\prod_{m=0}^{r-1} (n-m) \Big) p_{k,n} t^{n-r}. \tag{55}$$

Using this formula, boundary conditions can be enforced for each spline by finding a matrix

$$\mathbf{A}_{(k)}\mathbf{p}_{(k)} = \mathbf{b}_{(k)} \tag{56}$$

for every know derivative at the time $t = 0$ (collected in $\mathbf{A}_0$) and time $t = T_k$ (collected in $\mathbf{A}_T$). With n_c boundary conditions for the k^{th} spline, then

$$\mathbf{A}_{(k)} = \begin{bmatrix} \mathbf{A}_{0,k} \\ \mathbf{A}_{T,k} \end{bmatrix} \in \mathbb{R}^{n_c \times N+1} \quad \text{and} \quad \mathbf{b}_{(k)} = \begin{bmatrix} \mathbf{b}_{0,k} \\ \mathbf{b}_{T,k} \end{bmatrix} \in \mathbb{R}^{n_c \times 1} \tag{57}$$

For the remaining, free boundary endpoints where no fixed derivative is specified, the splines on each side of a boundary point are set equal by enforcing

$$\mathbf{A}_{T,k}\mathbf{p}_k - \mathbf{A}_{0,k+1}\mathbf{p}_{k+1} = 0. \tag{58}$$

A general method of generating the **Q**- and **A**-matrices was implemented in matlab (see `get_Q`, `get_A`), called by the `compute_splines` method which used `quadprog`'s interior point method to find a solution given a set of points, times and a cost vector. This, and the demo script `splines_2_1D_example.m` is located in the /crazy_trajectory directory and is almost in a good state. The problem can be solved using various polynomial degrees and finds connecting splines, but the continuity conditions in free points is still not working properly (see Figure 4). Clearly, the jerk squared is minimised and the the polynomial splines are continuos, but the derivatives are not. We have yet to figure out why.

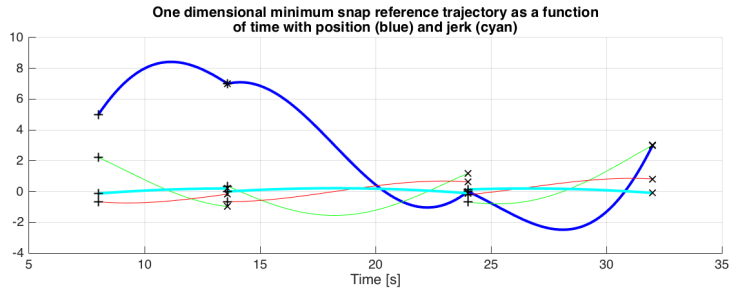

Figure 4: A reference trajectory composed of three splines with a maximum order of $N = 5$ and cost vector $c = [0, 0, 0, 1, 0, 0]$ (minimum snap), enforcing positional endpoint conditions and at unevenly spaced times, with $t = [8, 13.6, 24, 32]$.

5 Outer control

TODO: write about outer loop

5.1 MPC

For the MPC control, the system dynamics is augmented with stabilising PD control in **Section 2.4**, and linearised around a stable point. The system was implemented in Matlab without

state estimation using the MPC-tools 1.0 developed at LTH cite XX as a proof of concept (see `/Examples/quadcopter_mpc_position_test.slx`). The result was equivalent to that of the PID-PD controller when using a fixed reference point for all states on the prediction horizon, and can be expected to improved if using time-varying reference generated in the motion planning. As the MPC-tools solution is (i) incredibly slow, (ii) unable to handle varying references on the prediction horizon and (iii) not apt for a Python/C-realtime implementation in ROS, alternatives were investigated. The two candidates are CVXgen and QPgen, which both generate fast QP solvers in C, and can be set up to suffice (i).

The MPC-problem formulation, with a time varying reference, $\mathbf{r}_k$, and a control signal, $\mathbf{u}_k$, kept close to the angular velocity required to hover, $\mathbf{u}^s$, can be written

$$\begin{aligned} \min \Big(\sum_{k=1}^{m} (\mathbf{x}_k - \mathbf{r}_k)^T \mathbf{Q} (\mathbf{x}_k - \mathbf{r}_k) + \sum_{k=1}^{n} (\mathbf{u}_k - \mathbf{u}^s)^T \mathbf{R} (\mathbf{u}_k - \mathbf{u}^s) \Big) \\ \mathbf{x}_{k+1} = \mathbf{A}^d \mathbf{x}_k + \mathbf{B}^d \mathbf{u}_k, \qquad k = 0, ..., m \\ |\mathbf{x}_{7,k+1}| < x_{max}, \qquad k = 1, ..., m \cdot \\ |\mathbf{x}_{9,k+1}| < x_{max}, \qquad k = 1, ..., m \\ |\mathbf{u}_k - \mathbf{u}^s| < u_{max}, \qquad k = 1, ..., n \\ ||\mathbf{u}_k - \mathbf{u}_{k-1}||_\infty < S_{max}, \qquad k = 1, ..., n \end{aligned} \tag{59}$$

Where $\mathbf{Q} \in \mathbb{R}_+^{10\times10}, \mathbf{R} \in \mathbb{R}_+^{4\times4}$ are diagonal cost matrices, $\mathbf{A}^d \in \mathbb{R}^{10\times10}, \mathbf{B}^d \in \mathbb{R}^{10\times4}$ are the ZOH-sampled, discrete time system matrices of the augmented state space model. The remaining constraints bound the pitch and yaw, bound the deviation in control signal from the linearisation point $\mathbf{u}^s$, and the final puts a hard constraint on how fast we allow the control signal to vary. This final constraint can be set very high or removed completely, as the motors are incredibly responsive.

With this formulation with $m, n = 10$, the QP solver generated by CVXgen has 5364 non-zero KKT entries, which is slightly above the recommended 4000. To evaluate the feasibility of the CVXgen solver with the above formulation, a Simulink implementation was made using the MEX-solver, replacing the MPC-tools solver with a rewritten S-function (see `MPC_CVX.m`). The solver seems to work perfectly with the simulink environment, and is, as suspected, incredibly much faster than the already implemented `qp_is.m` solver of the MPC-tools. The implementation is still not complete, and can presently only be run as an open loop system, where $\mathbf{u}_0$ is set to

In order to use The MPC solver efficiently, the reference trajectory generated by the optimisation in **Section 4** has to be evaluated on the prediction horizon, i.e, at each cycle of the MPC-loop, we need to find a matrix $\mathbf{R}_{ref} = \begin{bmatrix} \mathbf{r}_1 & \cdots & \mathbf{r}_m \end{bmatrix}$, which evaluates the splines. The splines are in an array of polynomial coefficients, $\mathbf{P}$, defined on $t \in [t_0, t_f]$. Numerical evaluation needs to be done m times at a period time of T_s, starting at the current time, t_c. Special caution needs to be taken if a sample time $t_k \neq [t_0, t_f]$, in which case the quadcopter is set to hover in the appropriate end of the trajectory.

For these purposes, the scripts `reftraj_compute_example.m` and `reftraj_eval_example.m` were written. The former computes a trajectory in space with fixed positions at certain points in time, the derivatives are left free and currently, there is an issue with continuity, as described in **Section 4**. The latter forms the $\mathbf{R}_{ref}$-matrix, and the resulting reference samples on the prediction horizon is plotted (see Figure 5).

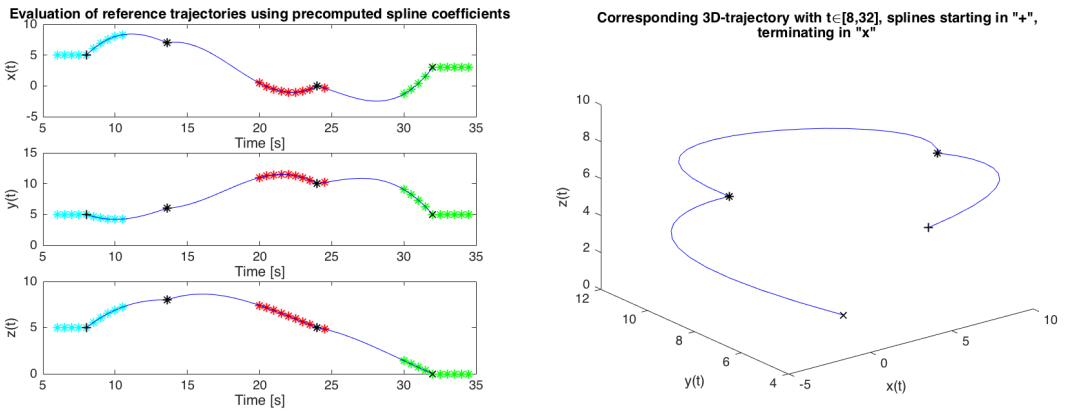

Figure 5: A reference trajectory composed of three 3D-splines with a maximum order of $N = 5$ and cost vector $c = [0, 0, 0, 1, 0, 0]$ (minimum snap), enforcing positional endpoint conditions and at unevenly spaced times, with $t_0 = 8$ and $t_f = 32$. The splines were evaluated with $T_s = 0.5$, $m = 10$, $t_c = 6 \Rightarrow$ some $t_k < t_0$ (cyan), $t_c = 20 \Rightarrow$ all $t_k \in [t_0, t_f]$ (red) and $t_c = 30 \Rightarrow$ some $t_k > t_f$ (green). As can be seen, the algorithm properly evaluates the splines at evenly spaced times and makes the reference hover at a trajectory endpoint if $t_k \neq [t_0, t_f]$. Here, only the positional elements of $\mathbf{R}_{ref}$ are plotted, note the reference velocities and anles are also evaluated.

TODO

1. ~~Create simplified linearised system model for use in MPC~~ (see eg. [9]).
2. ~~Validate by comparison to the results in~~ [9].
3. ~~Set up MPC controller with Simulink MPC-library~~ (see eg. [9]).
4. ~~Validate by comparison to the results in~~ [9].
5. Set up MPC controller with CVXgen and S-fucntions (see eg. [9] [10]).
6. Validate by comparison to the results in Simulink.
7. System identification.
8. Simulate system with proper parameters.
9. Compare the four different implementations based on speed and stability.

5.2 $\mathcal{L}_1$-control

Here we consider control of the linearized system The $\mathbf{\Gamma}$-projection operator for two vectors $\theta, y \in \mathbb{R}^k$ is defined as

$$\text{Proj}_{\mathbf{\Gamma}}(\theta, y, f) = \begin{cases} \mathbf{\Gamma} y - \mathbf{\Gamma} \dfrac{\nabla f(\theta)(\nabla f(\theta))^T}{||\nabla f(\theta)||_2} \mathbf{\Gamma} y f(\theta) & \text{if } \ f(\theta) > 0 \ \text{ and } \ y^T \nabla f(\theta) > 0 \\ \mathbf{\Gamma} y & \text{otherwise.} \end{cases} \tag{60}$$

where $\mathbf{\Gamma} = \mathbb{I}_{k \times k} \Gamma$ for some scalar $\Gamma > 0$ (typically $\Gamma \approx 10^5$) and $f(\theta)$ is a convex function [11]. By solving the Lyapunov equation

$$\mathbf{A}_m \mathbf{X} + \mathbf{X} \mathbf{A}_m^T + \mathbf{Q} = 0, \tag{61}$$

for $\mathbf{P} = \mathbf{P}^T$, with some arbitrary $\mathbf{Q} > 0$, the feedback controller

$$\begin{cases} u(t) = \hat{\theta}^T x(t) + k_g r(t) \\ \dot{\hat{\theta}}(t) = \text{Proj}_{\mathbf{\Gamma}}(\hat{\theta}^T(t), x(t) \tilde{x}^T(t) \mathbf{X} b) \end{cases} \tag{62}$$

can be constructed, where $\tilde{x} = \hat{x} - x$ is the state estimation error, k_g is a gain and $r(t)$ is the reference signal. By designing the companion system

$$\begin{cases} \dot{\hat{x}}(t) = \mathbf{A}_m \hat{x}(t) + b(u(t) - \hat{\theta}^T(t) x(t)) \\ y(t) = c^T \hat{x}(t) \end{cases} \tag{63}$$

it can be shown (by Theorem 2 [12]) that the state estimation error,

$$\lim_{t \to \infty} \tilde{x} = 0. \tag{64}$$

By a corollary of the theorem, choosing

$$k_g = -\frac{1}{c^T \mathbf{A}_m^{-1} b} \Rightarrow \lim_{t \to \infty} y(t) = r \tag{65}$$

if $r \equiv$ constant.

TODO

1. ~~Define general control structure.~~
2. Create Simulink projection operator (see eg. [13])
3. Validate projection operator against benchmark Simulink models (eg. [12]).
4. Define robustness metrics (see eg. [13] [14])
5. Create script for computing the $\mathcal{L}_1$-gain (see eg. [13]).
6. Validate script against benchmark Simulink models (eg. [12]).
7. Simulate control.

6 Control system summary

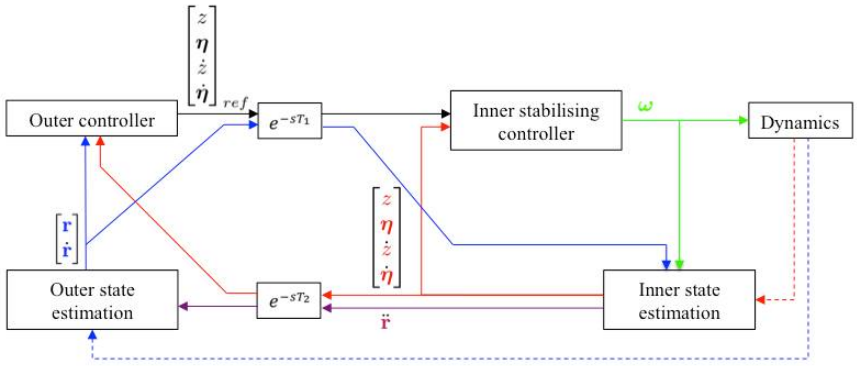

Figure 6: Overview of the entire system with system measurements of positions **r** (blue dashed) from the kinect and measurements from the accelerometer, magnetometer, gyroscope and pressure sensor (red dashed).

7 ROS implemetation

Wolfgang's crazyros driver [15]

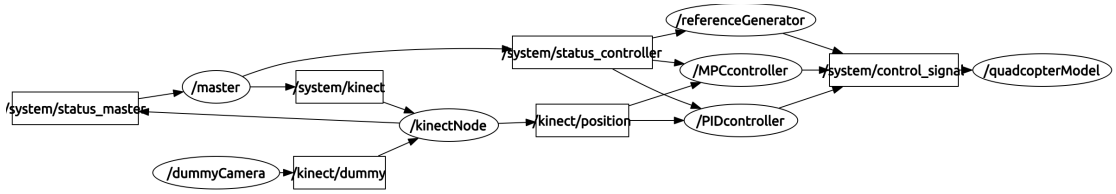

Figure 7: The current ROS structure as generated in `rqt_plot` with nodes (ellipses), topics (boxes) and the connecting arrows intdicating the flow of information.

7.1 Kinect node

The kinect node subscribes to the /camera/depth/image_rect-topic, to which the "openni"-driver publishes information in the data type `Image`, which is part of the standard ROS messages library `sensor_msgs`. This node handles the raw camera data and publishes the quadcopter position to the topic /kinect/position in the standard data type `Point` of the `geom_msgs` library. The reason for this choice of data type is to enable realt time plotting of the position with rviz and rqt_plot. In this node the outer state estimation is done in order to minimise the non-deterministic effects

of inter node communication via publishing (done in UDP/TCP). Consequently, the purpose of the kinect node is to (i) calibrate the camera, (ii) handle and filter image data from the "openni"-driver and (iii) facilitate communication between the kinect and the master.

7.1.1 Calibration

The camera is calibrated by taking first taking 30 consecutive samples of the background noise, from which a mean depth is computed. When running, the measured depth matrix is subtracted with the background noise so that only a handful of pixels are zero separate (or rather above some threshold $\epsilon \approx 2$ depth units). This set of pixels, S, is then used to compute the depth of the copter as the mean of the depths of all pixels in S. Similarity, the quadcopter position in the image is computed as the mean of all x and y pixel indices of the points in S.

The angle of the camera is calibrated by taking a set of N measurement points, $\mathbf{p}_i = \begin{bmatrix} x_i, y_i, z_i \end{bmatrix}$, in the background depth matrix. From these points, the center of mass is computed as

$$\bar{\mathbf{p}} = \begin{bmatrix} \bar{x}, \bar{y}, \bar{z} \end{bmatrix} = \frac{1}{N} \sum_{i=1}^{N} \mathbf{p}_i, \tag{66}$$

and a matrix for each point's deviation from the center of mass is set up

$$\mathbf{P} = \begin{bmatrix} \mathbf{p}_1 \\ \vdots \\ \mathbf{p}_N \end{bmatrix} - \begin{bmatrix} \bar{\mathbf{p}} \\ \vdots \\ \bar{\mathbf{p}} \end{bmatrix} \in \mathbb{R}^{N \times 3}, \tag{67}$$

The matrix $\mathbf{P}$ is then factorised using the singlar value decomposition (SVD), from which, the left-singular vector corresponding to the smallest of the three singular values is the normal of the best fitting plane in a least-squares sense, $\bar{\mathbf{n}}_{xyz}$. As we are only interested in the angle α in the xz-plane, the normal is projected onto this plane, resulting in the normal $\bar{\mathbf{n}}_{xz}$ (see Figure 8). From this, the angle is simply computed using the cosine dot product definition

$$\alpha = \arccos\left(\frac{\bar{\mathbf{n}}_{xz} \cdot \hat{\mathbf{x}}_c}{||\bar{\mathbf{n}}_{xz}||_2} \right). \tag{68}$$

Once the angle has been computed, we are ready to detect moving objects. By subtracting the background data from each measured image and computing the center of mass of all the points in S, we get an estimate of the quadcopter's center of mass in terms of depth and pixel indices. This point is then mapped into the camera coordinate system [m], where the depth unit is mapped to a distance from the camera to the quadcopter [m] is used to translate the position of the copter to a point in space in the camera coordinate system, $\mathbf{p}_c$, with origin at the camera. This point is then translated into the global coordinate system, $\mathbf{p}$, (in which the controller operates) by means of the transformation

$$\mathbf{p} = \begin{bmatrix} \cos(\alpha) & 0 & -\sin(\alpha) \\ 0 & 1 & 0 \\ \sin(\alpha) & 0 & \cos(\alpha) \end{bmatrix} \mathbf{p}_c. \tag{69}$$

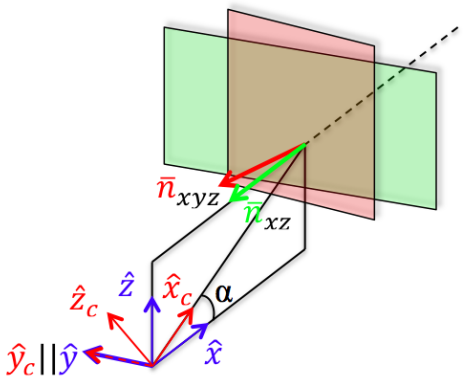

Figure 8: .

7.1.2 Filtering

Once a positional measurement has been established in the global coordinate system, a Kalman filter is applied to smooth out the measurement and estimate the velocities of the quadcopter. The kalman filters were implemented in the `crazylib` module in the /crazy_ros/modules/ directory, from which the discrete kalman update is called to update the estimation and covariance matrix for every new measurement. A big problem in the estimation is the cases when the copter is temporarily lost in the frame (which occurs frequently when flying 2 [m] from the camera). This problem was amended in practice by putting a small paper box on the bottom of the quadcopter, but also in the code by only using the complete kalman update when a good measurement is registered.

This can be accomplished by (1) setting the measurement covariance matrix very high for bad measurements (rendering the kalman gain very small) or (2) by simply terminating the kalman update after prediction when a bad measurement is registered. Here, the second alternative is implemented, and the criteria for a bad measurement is defined by comparing the difference of the previous estimation and the current measurement (in the two-norm) and comparing this to the variance of the positions. By setting the initial covariance as a diagonal matrix (or toeplitz) with a main diagonal ≥ 1, formulating the condition this way will allow higher two-norm deviations when beginning the filtering (which is desired, as the initial condition will never be perfect). The condition for a bad measurement on the k^{th} update can then be written

$$|| \left[\hat{x}^{k-1}, \hat{y}^{k-1}, \hat{z}^{k-1}\right]^T - \left[x^k, y^k, z^k\right]^T ||_2 < \alpha || \left[\mathbf{P}_{11}^{k-1}, \mathbf{P}_{22}^{k-1}, \mathbf{P}_{33}^{k-1}\right] ||_2 \tag{70}$$

As a proof of concept, and to illustrate how the filter handles some of the problems, a regular Kalman filter was used on the 3D-double integrator model (not using quadcopter accelerometer data) when simply swinging the quadcopter in a thin line. In the experiment, $\alpha = 5$, $\mathbf{Q} = 0.1 \cdot \mathbb{I} \in \mathbb{R}^{6\times6}$, $\mathbf{R} = 5 \cdot \mathbb{I} \in \mathbb{R}^{3\times3}$, and the quadcopter is swung in the xz-plane during $\approx 30s$. The data was saved from the topics (as published in ROS) and then analysed and plotted offline in Matlab(see Figure 9). Here we note that the filter handles outlying measurements very well (see the spikes

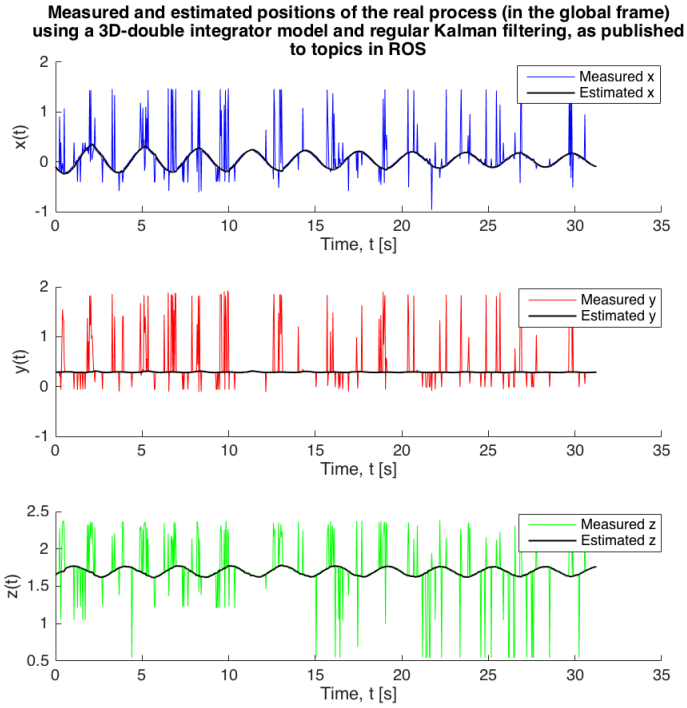
Measured and estimated positions of the real process (in the global frame) using a 3D-double integrator model and regular Kalman filtering, as published to topics in ROS
Measured x
Estimated x
x(t)
Time, t [s]
Measured y
Estimated y
y(t)
Time, t [s]
Measured z
Estimated z
z(t)
Time, t [s]

Figure 9: .

in the measurement signals) and that the estimated positions are smooth and on a decaying sinusoidal form, as expected when the entire system then resembles a common pendulum.

7.1.3 Communication

References

[1] T. Luukkonen, "Modelling and control of quadcopter," *Independent research project in applied mathematics, Espoo*, 2011.

[2] İ. Dikmen, A. Arīsoy, and H. Temeltaş, "Attitude control of a quadrotor," in *Recent Advances in Space Technologies, 2009. RAST'09. 4th International Conference on.* IEEE, 2009, pp. 722–727.

[3] T. Glad and L. Ljung, *Control theory.* CRC press, 2000.

[4] G. A. Terejanu, "Extended kalman filter tutorial," *Online]. Disponible: http://users. ices. utexas. edu/˜ terejanu/files/tutorialEKF. pdf*, 2008.

[5] E. A. Wan and R. Van Der Merwe, "The unscented kalman filter for nonlinear estimation," in *Adaptive Systems for Signal Processing, Communications, and Control Symposium 2000. AS-SPCC. The IEEE 2000.* Ieee, 2000, pp. 153–158.

[6] M. S. Arulampalam, S. Maskell, N. Gordon, and T. Clapp, "A tutorial on particle filters for online nonlinear/non-gaussian bayesian tracking," *Signal Processing, IEEE Transactions on*, vol. 50, no. 2, pp. 174–188, 2002.

[7] R. Douc and O. Cappé, "Comparison of resampling schemes for particle filtering," in *Image and Signal Processing and Analysis, 2005. ISPA 2005. Proceedings of the 4th International Symposium on.* IEEE, 2005, pp. 64–69.

[8] J. D. Hol, T. B. Schon, and F. Gustafsson, "On resampling algorithms for particle filters," in *Nonlinear Statistical Signal Processing Workshop, 2006 IEEE.* IEEE, 2006, pp. 79–82.

[9] P. Bouffard, "On-board model predictive control of a quadrotor helicopter: Design, implementation, and experiments," Master's thesis, EECS Department, University of California, Berkeley, Dec 2012. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-241.html

[10] J. Mattingley and S. Boyd, "Cvxgen: A code generator for embedded convex optimization," *Optimization and Engineering*, vol. 13, no. 1, pp. 1–27, 2012.

[11] E. Lavretsky, T. E. Gibson, and A. M. Annaswamy, "Projection operator in adaptive systems," 2011.

[12] C. Cao and N. Hovakimyan, "Design and analysis of a novel l1 adaptive controller, part i: Control signal and asymptotic stability," in *American Control Conference, 2006.* IEEE, 2006, pp. 3397–3402.

[13] N. Hovakimyan, "L1 tutorial." [Online]. Available: http://naira-hovakimyan.mechse.illinois.edu/l1-adaptive-control-tutorials/

[14] M. Q. Huynh, W. Zhao, and L. Xie, "L 1 adaptive control for quadcopter: Design and implementation," in *Control Automation Robotics & Vision (ICARCV), 2014 13th International Conference on.* IEEE, 2014, pp. 1496–1501.

[15] W. Hoenig, C. Milanes, L. Scaria, T. Phan, M. Bolas, and N. Ayanian, "Mixed reality for robotics," in *IEEE/RSJ Intl Conf. Intelligent Robots and Systems*, Hamburg, Germany, Sept 2015, pp. 5382 – 5387.

8 Appendix