Modelling, Simulation and Control Report

Model Based Control of Cyber-Physical Systems

Carl-Johan Heiker (heikerc) | Sondre Wiersdalen (chanon) | Vide Ramsten (vider) | Lucas Rath (lucasra)

***Abstract*—In this assignment we present mathematical modelling and control of a quadrotor; the Crazyflie 2.0. The modelling and control consists of physical modelling of the quadrotor and estimation of orientation, a complementary filter for the angular measurements, linearization of the physical model and full linear quadratic (LQR) state feedback control with integral action (LQRI). The controller is first implemented using Matlab and Simulink, and later using C.**

We found that selecting a state space subset of controllable states was necessary in order to compute a well working controller in simulation, and the implementation of integral action made reference tracking more precise. On the hardware, the system is working at a satisfactory level when using the C-implemented controller. The Simulink version of the controller was found to have oscillatory behaviour in both stabilization and set point tracking. Simulations of the model and measurements of the implemented system are provided as plots and analyzed. Finally, we analyze our chosen methods and give some ideas for improvement on this project and its procedure.

I. Introduction to control of Quadrocopters

Quadrotors are very versatile and relatively simple aircrafts. As they can be used for many interesting tasks, there is a need to develop controllers that can provide flight stability and set point tracking. A good test bed is the Crazyflie 2.0, which will be studied in this project.

A sketch of the drone is provided in Figure 1.

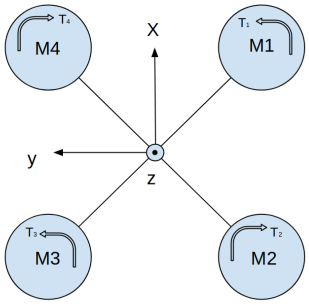

Fig. 1. Crazyflie body frame, motors (M_i) and torque (T_i)

For clarification a complete list of symbols used in the report is provided in Table I.

TABLE I
Symbols (MKS values)

$\boldsymbol{x}$	state vector
s	Laplace variable
p	position
v	velocity
a	acceleration
Θ	Euler angle vector, sequence $XYZ \Rightarrow \varphi, \theta, \psi$
φ	roll angle
θ	pitch angle
ψ	yaw angle
A	system matrix
A_d	discretized system matrix
B	input matrix
B_d	discretized input matrix
C	output matrix
H	selection matrix for controlled outputs
K	state feedback gain
K_r	reference gain
K_I	integral action gain
g	gravitational unit
{B}	body frame
{I}	world/inertial frame
f	force
R	rotation matrix
m	mass
τ	control signal
J	moment of inertial
ω	angular velocity
T	torque
Θ	angle
α	time constant
γ	tuning parameter (complementary filter)
T_s	sampling time
r	reference
z^k	z-transform variable
d	distance
I	variables denoted by I means inertial/world frame
b	variables denoted by b means body frame
$\dagger$	pseudoinverse operand
$atan2$	sign sensitive tangent operator

II. Estimation of orientation

To describe the orientation of the quadrotor, there are two coordinate frames to consider: the *inertial (world) frame* $\{\mathrm{I}\}$ and the *body frame* $\{\mathrm{B}\}$. In this project, the body will be described relative to the fixed inertial frame using Euler angles of sequence X-Y-Z, which corresponds to roll (φ), pitch (θ) and yaw (ψ), respectively. Here, the inertial frame is a standard right hand coordinate system, such that the z-direction points upwards. The rotation matrix that transforms from the body frame to the inertial frame is derived by the sequence of rotations X-Y-Z, according to the Euler angle sequence as

$$R_b = R_x(\varphi)R_y(\theta)R_z(\psi) \tag{1}$$

where

$$\begin{aligned} R_x(\varphi) &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos(\varphi) & -sin(\varphi) \\ 0 & sin(\varphi) & cos(\varphi) \end{bmatrix} \\ R_y(\theta) &= \begin{bmatrix} cos(\theta) & 0 & sin(\theta) \\ 0 & 1 & 0 \\ -sin(\theta) & 0 & cos(\theta) \end{bmatrix} \\ R_z(\psi) &= \begin{bmatrix} cos(\psi) & -sin(\psi) & 0 \\ sin(\psi) & cos(\psi) & 0 \\ 0 & 0 & 1 \end{bmatrix} \end{aligned} \tag{2}$$

are elemental rotations. In three dimensions, the order of multiplication in Equation (1) matters. Using X-Y-Z order, Gimbal lock is avoided in critical positions.

To estimate the Euler angle, an *accelerometer* and a *gyroscope* are used to fuse the data to obtain a better estimate. We start by describing how to derive the Euler angles estimates from the accelerometer and the gyroscope and end this section describing the sensor fusion technique called complementary filtering.

A. Accelerometer observer

The accelerometer measures the reaction forces caused by the accelerations on the sensor in $\{\mathrm{B}\}$ coordinates. These readings are normalized and denoted $\bar{f}^b$, and they also include the reaction force caused by the gravity. This can be translated into

$$\bar{f}^b = \frac{f^b}{mg} = R^b \, \frac{a^I - \vec{g}^I}{g} \tag{3}$$

where a^I and $\vec{g}^I = [0, 0, -g]^T$ are the inertial frame and gravity acceleration vector in $\{\mathrm{I}\}$ as g is the gravitational constant and R^b is the rotation matrix that converts $\{\mathrm{I}\}$ into $\{\mathrm{B}\}$. According to [Sic09], R^b can be calculated from Equation (1) simply as $R^b = R_b^{\mathsf{T}}$.

In the long term, the expected value of the body acceleration a^I is zero, and we can thus estimate the orientation of the body frame based on the accelerometer signal, which will output a vector pointing to $-\vec{g}$.

Replacing $a^I = 0$ and $R^b = R_b^{\mathsf{T}}$ from equation (1) and (2) into equation (3) yields the angular expressions for roll and pitch angles

$$\begin{aligned} \varphi_a &= atan2(\bar{f}_y^b, \bar{f}_z^b) \\ \theta_a &= atan2\Big(- \bar{f}_x^b, \sqrt{(\bar{f}_y^b)^2 + (\bar{f}_z^b)^2}\Big) \end{aligned} \tag{4}$$

Since it is known that the body frame will often accelerate, it is a good practice to normalize the elements of $\bar{f}^b$ before applying Equation (4), such that its norm is expressed unitary as

$$\bar{f}'^b = \frac{\bar{f}^b}{\|\bar{f}^b\|^2} \tag{5}$$

Because the accelerometer measures the acceleration vector in relation to the gravitational vector, it can not be used to detect changes in yaw. Yaw could be measured using a magnetometer, but also by the gyroscope that is fitted on the Crazyflie. Therefore, the Euler angle estimates from the accelerometer can be summarized as

$$\Theta_a = \begin{bmatrix} \varphi_a \\ \theta_a \\ 0 \end{bmatrix} \tag{6}$$

B. Gyroscope observer

The gyroscope measures angular velocity in the body frame w^b, which needs to be manipulated so we can calculate the Euler angles from the sensor readings. The problem is that there is no physical meaning of integrating angular velocity [Sic09, pg 130]. A common approach is to first transform to $\{\mathrm{I}\}$ coordinates with the well-known rotation matrix and then into Euler angle derivatives using a transformation matrix, which depends on the Euler angle sequence used.

$$w = R_b \, w^b \tag{7}$$

$$\dot{\Theta}_g = T_{w_b}^{\dot{\Theta}} \, w \tag{8}$$

The $T_{w_b}^{\dot{\Theta}}$ matrix depends on the Euler sequence used and its derivation can be seen in [Sic09], which results in

$$T_{w_b}^{\dot{\Theta}} = \begin{bmatrix} 1 & 0 & sin(\theta) \\ 0 & cos(\varphi) & -cos(\theta) \cdot sin(\varphi) \\ 0 & sin(\varphi) & cos(\varphi) \cdot cos(\theta) \end{bmatrix} \tag{9}$$

After this, we are able to integrate the Euler angle derivatives to get Euler angles, which are now physically meaningful and write as

$$\Theta_g = \int \dot{\Theta}_g \, dt \tag{10}$$

However, this integration must be performed in discrete time since the sensor outputs are discrete. Performing a forward Euler integration, we can estimate the Euler angles from the gyroscope at each time step

$$\begin{aligned} \Theta_{g,k} &= \Theta_{g,k-1} + T_s \, \dot{\Theta}_{g,k} \\ &= \Theta_{g,k-1} + T_s \, T_{w_b}^{\dot{\Theta}} \, R_b \, w_k^b \end{aligned} \tag{11}$$

where T_s is the sample time.

C. Sensor fusion - complementary filter

The next task is to implement a sensor fusion, so that the best performance of both sensors are combined. One of the possible sensor fusion approaches for this case is the complementary filter.

Since the gyroscope signal is being integrated, it will suffer from numerical drift over the course of simulation time. It

is reasonable to trust these measurements for high frequency changes and filter out low frequency signals. Furthermore, the Euler angles from the accelerometer are calculated assuming that the expected value for the body acceleration is zero. However, this is only true in the long term perspective. Thus, it is reasonable to filter high frequencies of this signal and keep signals at low frequencies. To summarize, the strategy is to high pass the Euler angles from the gyroscope and low pass the angles from the accelerometer with the same cut-off frequency, so that in the end we can add them together and get a more accurate measurement.

The low pass filter is expressed in the Laplace frequency domain as

$$G(s) = \frac{1}{\alpha s + 1} \tag{12}$$

where $1/\alpha$ is the cutoff frequency of the low pass filter. A complementary high pass filter is simply expressed simply as $1 - G(s)$, which has the same cut-off frequency. The sum of these are 1; the filtering is complementary to the components. The expression for the filter in total is

$$\Theta(s) = G(s)\,\Theta_a(s) + (1 - G(s))\,\Theta_g(s) \tag{13}$$

Using Equation (11), (4), (12), (13) and through some algebraic manipulations, the final filter expression can be expressed in discrete time as:

$$\hat{\Theta}_k = (1-\gamma)\,\Theta_{a,k} + \gamma\,\Theta_{g,k} \tag{14}$$

$$= (1-\gamma)\,\Theta_{a,k} + \gamma\,(\Theta_{g,k-1} + T_s\,T^{\dot{\Theta}}_{w_b}\,R_b\,w^b_k) \tag{15}$$

where, again w^b_k is the measurement from the gyro and $\Theta_{a,k}$ is calculated using Equation (4) for the time step k. Also, $\gamma = \frac{\alpha}{T_s + \alpha}$ where $0 < \gamma < 1$.

What remains is to choose a proper value of α for the complementary filter. This value can be tuned empirically. The only restriction for the tuning is that the cut-off frequency should be less than the Nyquist sampling frequency

$$\frac{1}{\alpha} < 0.5\frac{1}{T_s} \quad \Rightarrow \quad \alpha > 0.02 \tag{16}$$

A good value found was $\alpha = 0.05$, which leads to $\gamma = 0.833$.

III. Plant modelling

The plant modelling of the quadrotor using the Simscape toolbox in Simulink is done in an object-oriented, declarative way. This means that we do not need to define equations in a input/output perspective, but just list the equations and the solver will take care of the rest. All the equations to be derived in this sections were inspired by [Cor17].

The first equations to be defined are the linear and angular equations of motions. Both equations need to be defined relative to the inertial frame $\{\mathrm{I}\}$, but they can be written based on any frame coordinates. The linear equation of motion is then defined based on the world frame by Newton's second law, given by

$$m\,a = -\begin{bmatrix} 0 \\ 0 \\ m\,g \end{bmatrix} + R_b \begin{bmatrix} 0 \\ 0 \\ \tau^b \end{bmatrix} \tag{17}$$

$$\dot{v} = a \tag{18}$$

$$\dot{s} = v \tag{19}$$

where τ^b is the total thrust generated by the propellers in $\{\mathrm{B}\}$, R_b is the rotation matrix from body frame $\{\mathrm{B}\}$ to world frame $\{\mathrm{I}\}$, m is the quadrotor mass, g the gravitational constant and a the linear acceleration on $\{\mathrm{I}\}$.

Moreover, we describe the angular equation of motion based on the body frame $\{\mathrm{B}\}$, because the inertia J^b is only constant if expressed in the body frame coordinates.

$$J^b\,\dot{w}^b = -w^b \times (J^b\,w^b) + T^b \tag{20}$$

where w^b is the angular velocity on $\{\mathrm{B}\}$, and T^b the torque generated by the propellers on $\{\mathrm{B}\}$.

The total propeller thrust force is an upward vector defined based on the rotational velocity of the motors $\bar{w}$ and the lift constant k as

$$\tau^b = k\,\bar{w}^2 \tag{21}$$

where $\bar{w}$ is a 4x1 vector corresponding to each propeller and the square is element-wise.

Difference in rotor thrusts generate moments around the quadcopter axis. Roll and pitch rotations are simply generated by difference in motor thrusts around the x- and y-axis, which is written as

$$T^b_x = d\,cos\frac{\pi}{4}\,(\tau^b_3 + \tau^b_4 - \tau^b_1 - \tau^b_2) \tag{22}$$

$$T^b_y = d\,cos\frac{\pi}{4}\,(\tau^b_2 + \tau^b_3 - \tau^b_1 - \tau^b_4) \tag{23}$$

where $d\,cos\frac{\pi}{4}$ is the distance from the propeller centers to the x- or y-axis, which is the same.

Furthermore, the aerodynamic drag generated by the movement of the propellers exerts a reaction torque on the frame about the propeller shaft in the opposite direction of its rotation as

$$T^b_z = b\,(-\bar{w}^2_1 + \bar{w}^2_2 - \bar{w}^2_3 + \bar{w}^2_4) \tag{24}$$

where b is the aerodynamic drag constant. Finally the three torque components can be concatenated, such that

$$T^b = \begin{bmatrix} T^b_x \\ T^b_y \\ T^b_z \end{bmatrix} \tag{25}$$

What remains is to define the equations that couple the angular velocity in $\{\mathrm{I}\}$ coordinates to the Euler angle derivatives. Similar to what have been done in the complementary fusion section, we define this relation as

$$w = R_b\,w^b \tag{26}$$

$$\dot{\Theta} = T^{\dot{\Theta}}_{w_b}\,w \tag{27}$$

$$\Theta = \int \dot{\Theta} dt \tag{28}$$

where, again, Θ is the Euler angle vector and $T_{w_b}^{\dot{\Theta}}$ is defined as in Equation (9).

IV. Linearization of Plant

Since this is a highly non-linear system and we want to implement linear control techniques, we need to linearize the model before be able to construct a good controller for the quadrotor. The states decided to represent the system were

$$\mathbf{x} = \begin{bmatrix} s & v & \Theta & w^b \end{bmatrix}^{\mathsf{T}} \tag{29}$$

where $s, v, \Theta, w^b \in \Re^3$ are the position, velocity, Euler angles, and angular velocity vectors.

The linearization point was decided to be at $x_0 = \vec{0}$, since we assume this is gonna be the point where the quadrotor will operate around most of the time.

In general, the plant model of the system can be described as

$$\dot{\mathbf{x}} = f(\mathbf{x}, u) \tag{30}$$

for which it is known that f is a non-linear function. To linearize this, the equation

$$\dot{\mathbf{x}} \approx \frac{\partial f}{\partial x}\bigg|_{x_0,u_0} \Delta\mathbf{x} + \frac{\partial f}{\partial u}\bigg|_{x_0,u_0} \Delta\mathbf{u} \tag{31}$$

can be used. The new state-space model is now on the form

$$\dot{x} = Ax + Bu \tag{32}$$

Using the linearization method in Equation (31) the state dynamic matrix A and input matrix B were defined as

$$A = \begin{bmatrix} 0 & 0 & 0 & 1.0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1.0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1.0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 9.81 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -9.81 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1.0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1.0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1.0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{33}$$

$$B = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 37.0 & 37.0 & 37.0 & 37.0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ -2333.0 & -2333.0 & 2333.0 & 2333.0 \\ -2277.0 & 2277.0 & 2277.0 & -2277.0 \\ -1277.0 & 1277.0 & -1277.0 & 1277.0 \end{bmatrix} \tag{34}$$

From the linearized model, in Equation (33) it is observed that the input only affects the angular velocity and the acceleration in the z-direction. This is expected, since if the input to the motors change, then the angular velocity should change. This change in angular velocity in turn affects the angles because of the integration in the model. Thrust only affects the z components because the linearization is based around zero angles. Then if the angles are changed this force in z should be affecting the x and y depending on the angle of the quadrotor.

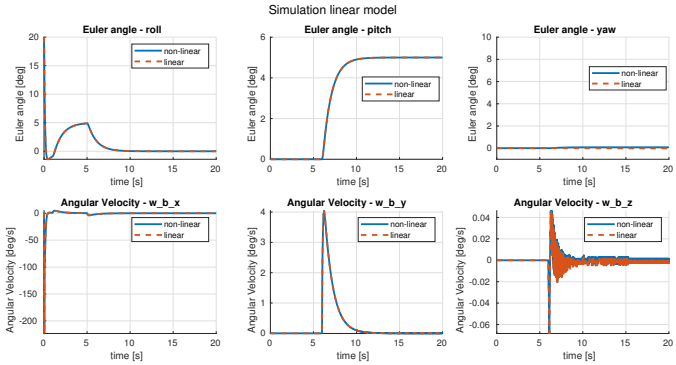

Fig. 2. Simulation of the linear and nonlinear system given a arbitrary reference signal near the linearization point.

We have linearized the system around the point of stability, which is when the angles and angular velocities are zero. Around this point, the linearized model and the non linear model behave in the same way, which can be seen in Figure 2. However, if the model deviates too far from the operating point, the designed control system will be invalid.

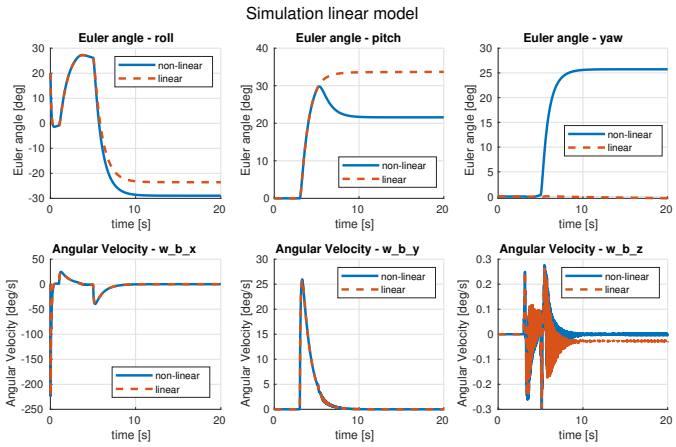

Fig. 3. Comparison between the linear and nonlinear model of the system given a reference signal outside of the valid region of the linearization.

In Figure 3 the system is given a reference angle that is much larger than the linearization can handle. In the graph we can see that the linear model and the non linear model differentiate quite a bit in pitch and yaw angle, also the roll angle is different between the linear and nonlinear.

V. Design of Linear Quadratic Controller

The first step to implementing a *Linear Quadratic Regulator* (LQR) controller is to discretize the linearized model of the plant. The sampling time of the quadrotor is $0.01s$, which is the sample time (T_s) used in the discretization. The method of discretizing used is the *zero-order hold* method, in which the inputs are assumed to be piece-wise constant over the sample time T_s.

The discretization of the model matrices are calculated as

$$\begin{aligned} A_d &= e^{A \cdot T_s} \\ B_d &= \Big(\int_{\tau=0}^{T_s} e^{A\tau} d\tau \Big) B \end{aligned} \tag{35}$$

Because of limitations on the sensors it is not possible to measure *yaw angle* reliably, hence not possible to include it in state feedback. Position and velocity are also not measured, the state vector is therefore reduced by excluding yaw angle ψ from the Euler angle vector $\Theta = [\varphi, \theta, \psi]^T$, position and velocity. This means that the new state vector (in continuous time) is

$$\mathbf{x} = \begin{bmatrix} \varphi \\ \theta \\ w_x^b \\ w_y^b \\ w_z^b \end{bmatrix} \tag{36}$$

Reference tracking is applied on the angular positions and is given to the controller as the vector (SP for set point)

$$r = \begin{bmatrix} \varphi^{SP} \\ \theta^{SP} \\ w_z^{b,SP} \end{bmatrix} \tag{37}$$

This means that the controlled outputs of the plant has to be made to match this reference. The output matrix, C should then be reduced to the matrix

$$H = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{38}$$

Having discretized the plant model, the LQR controller can be calculated by minimizing the infinite time quadratic regulation cost function

$$J = \sum_{k=0}^{\infty} x_k \, Q \, x_k + u_k \, R \, u_k \tag{39}$$

which solution for linear systems is given by the *discrete-time algebraic Riccati equations* (dare)

$$\begin{aligned} K &= (B_d^T P B_d + R)^{-1} (B_d^T P A_d) \\ 0 &= A_d^T P A_d - (A_d^T P B_d)(B_d^T P B_d + R)^{-1} (A_d^T P B_d)^T + Q \\ u^{fb}(k) &= -K \, x(k) \end{aligned} \tag{40}$$

where u_k^{fb} is the feedback control input at time k.

The Q and R matrices in Equation (40) are used as tuning parameters, which can be adjusted so the system meets the desired closed-loop behaviour. In fact, it depends on the relationship between the values in Q and R. As we increase Q, we penalize state deviations from the origin (or set point) more, which will make the system react faster and more aggressively. On the other hand, increasing R will penalize deviations of the control input from the origin and make our system more conservative.

Further, a pre-filter K_r is introduced in order to remove the stationary error between reference and output and can be calculated by solving

$$\lim_{t \to \infty} (y(t) - r(t)) = \lim_{z \to 1} (z-1)(Y(z) - R(z)) = 0 \tag{41}$$

which has a closed-form solution for linear systems

$$K_r = \Big(H \big(I - A_d + B_d K \big)^{-1} B_d \Big)^{\dagger} \tag{42}$$

In addition, it is desired to add a base thrust $u_{basethrust}$ to the motors in the form of a feed-forward controller in order to compensate the gravity effect. As a result, the total control input applied to the quadrotor is given by

$$u(k) = -K \, x(k) + K_r \, r(k) + u_{basethrust}(k) \tag{43}$$

Figure 4 shows then the closed-loop model of the system with LQR feedback controller, feed-forward base thrust compensation and pre-filter.

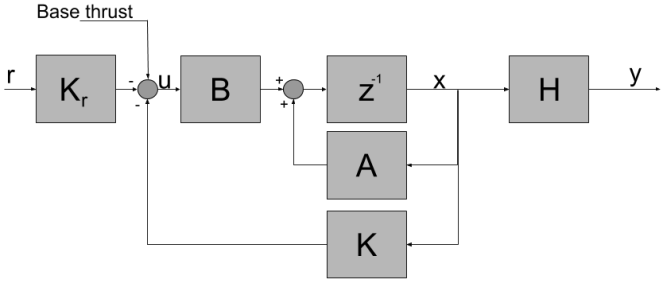

Fig. 4. Block diagram of the state-space model of a system with LQR controller.

This system however can be susceptible to offsets. Since our plant is non-linear and the plant is also susceptible to input disturbance, this could cause the system to not properly regulate to $y = r$. The transformation from thrust to motor input, which is given as an integer, could cause disturbances and/or give an offset to the angles, causing the quadrotor to fail to follow the reference signal correctly.

Introducing integral action in the controller can help removing this offset. This is done by augmenting the original state space by including auxiliary variables (x_l) that integrate the error between the reference and the actual state. The integral action part of a LQR controller is defined as

$$x_l(k+1) = x_l(k) + r(k) - Hx(k) \tag{44}$$

The full discrete state-space now becomes

$$\begin{bmatrix} x(k+1) \\ x_l(k+1) \end{bmatrix} = \begin{bmatrix} A_d & 0 \\ -H & I \end{bmatrix} \begin{bmatrix} x(k) \\ x_l(k) \end{bmatrix} + \begin{bmatrix} B_d \\ 0 \end{bmatrix} u(k) + \begin{bmatrix} 0 \\ I \end{bmatrix} r(k) \tag{45}$$

To get a feedback gain that works with the integral action incorporated, the same approach as in the standard LQR can be used. The equations seen in equation (40) are used, replacing the A and B matrices with the new extended ones seen in equation (45) to get a new $K_{integral}$. This feedback gain is split up in two parts as $K_{integral} = \begin{bmatrix} K & K_I \end{bmatrix}^T$, where K is for the regular state feedback part and K_I is for the integral action feedback. The new control input will then be

$$u(k) = -K \, x(k) + K_I \, x_I(k) + u_{bt}(k) \tag{46}$$

The closed-loop system with LQR feedback controller with integral action and feed-forward base-thrust compensation can be seen in Figure 5.

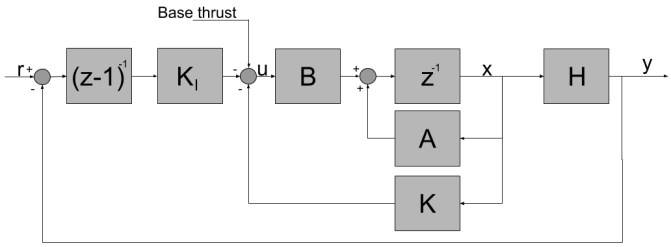

Fig. 5. Block diagram of the state-space of a system with LQR controller with integral action.

VI. Evaluation of the control design against the simulated model and physical system

A. LQR Controller with Integral Action: Stabilization from Offset

The simulations in this section focus on stabilization by driving initial angles to zero. The gravitational acceleration is then set to zero, eliminating the possibility to free-fall. After tuning the controller to values that fit both set point tracking and stabilization, plots were generated to evaluate the performance of the controller in simulation and on the hardware.

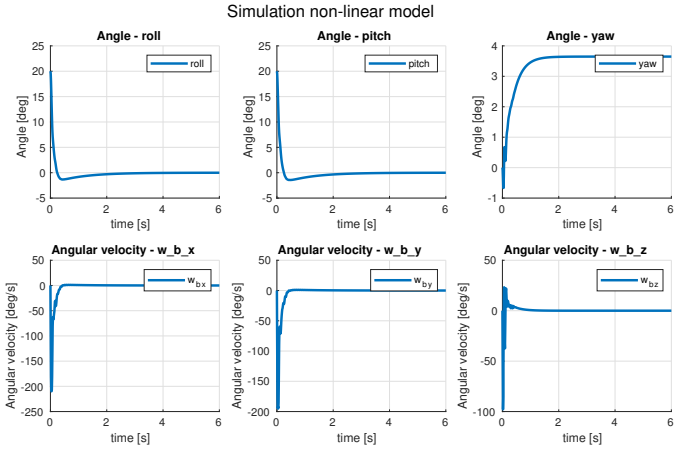

Fig. 6. Simulation of non linear model with LQRI controller. Showing Euler angles and angular velocity in body frame. Reference signal is $[\varphi, \theta, w_z^b]^\mathsf{T} = [0°, 0°, 0°]^\mathsf{T}$.

In Figure 6, it can be seen how the controller stabilizes the system very fast in terms of angles and angular velocity. In these figures, the position states are not included, since they are all zero and not controlled. It should be noted how the yaw angle stabilizes at a different value, since it is not given a setpoint, but the yaw *rate* still stabilizes to zero.

In the actual system, which can be seen in Figure 7, the angles oscillated quite a bit. This could be caused by the filter not being fast enough to properly stabilize to a complete standstill. Also to note in the figure (at the red arrow marks) the system is agitated by an external force. From this we can see that the system somewhat stabilizes to zero, except for the oscillations in the angles.

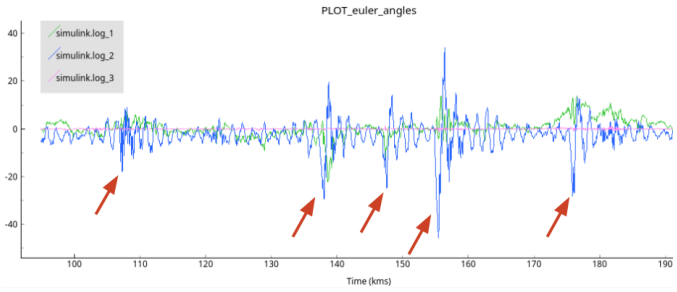

Fig. 7. Measurements from physical system with LQRI controller, the red arrows point out disturbances externally applied to the system.

B. LQR Controller with Integral Action: Setpoint Tracking

The given setpoint is 30 degrees in roll and pitch, with zero velocity in yaw rate.

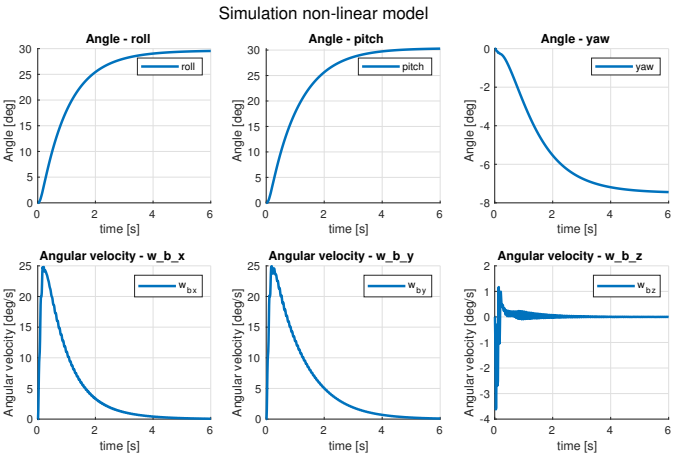

Fig. 8. Non linear model simulation of tracking a given setpoint with LQRI. Showing Euler angles and angular velocity in body frame. Reference signal is $[\varphi, \theta, w_z^b]^\mathsf{T} = [30°, 30°, 0°]^\mathsf{T}$.

In Figure 8, one can note that the simulated system follows the given set point and stabilizes around it after around six seconds. The angular velocities also stabilize around zero while the yaw angle stabilizes around some arbitrary value.

Note that this exact behaviour cannot be tested on the quadrotor because it is suspended in the y axis in these tests.

Fig. 9. Measurements of the angles from physical system when tracking a given setpoint using the LQRI controller.

In Figure 9, it can be seen how the same oscillations occur at the setpoint, which is 30 degrees. The crazyflie is able to keep the setpoint, but the oscillations are very high.

VII. C-IMPLEMENTATION

A C-implementation of an LQR controller was done in the FreeRTOS framework. When tuning the parameters, it was concluded that the closed-loop behaviour was more stable and presented better performance when using a pre-filter instead of an integral action.

In this project three specific tasks was developed to run in parallel on the crazyflie:

- C-implementation of the complementary filter
- C-implementation of the LQR controller
- C-implementation of a set-point generator

The tasks are listed in a decreasing order of priority. Since stability is a number one priority the filter comes before the controller because no matter how good a controller is, it will still be unreliable if the measurement is unreliable.

The processor has the clock frequency of $168MHz$, but the real time operating system has a predefined frequency in which it checks if a task is ready to execute, called the interrupt frequency. This is defined by describing the time interval between each interruption, called "ticks", of a predefined time. In this case, one tick is equal to $1ms$, and the operating system thus reads the sensors at a frequency of $1kHz$.

Since the complementary filter has the highest priority, this means that the reading of the sensors will always hijack the processor. Because of this, the complementary filter thread has to be put to sleep between updates so that the other threads have a chance of updating. In addition, because the complementary filter task and the controller task share access to the state space, the reading and writing to the state space must be regulated by semaphores.

A. Complementary filter

Since the complementary filter was already implemented in `Matlab`, the decision to mostly use hard coded matrices makes the implementation much easier, and faster.

Before the tasks of acquiring sensor data and updating the output angles the sensor have to be calibrated. A function `sensorAreCalibrated()` already exists, so before the tasks read the sensor this function has to output a `true` value. The sensors only need to be calibrated once, so this part can be outside of the task loop.

The filter function is then responsible for reading the sensors, filtering and writing new values in the global state variable. The pseudo-code for the filter implementation in `C` is shown in code 1.

It is important to notice here that the complementary filter will, at every time-step, estimate the orientation and always save these values to local variables. This is necessary because it will need the previous orientation in order to estimate the new one. On the other hand, it will only write those values to global variables if no other thread has taken the semaphore, i.e. if other thread is reading the global state space variable at that moment. If this happens, the filter will simply skip writing to global variables for that period. Indeed, we assume this strategy is better than changing the global variables while some other thread is in the processes of reading, which may cause inconsistency in the algorithm.

Listing 1. Pseudo-code for complementary filter thread

```
wait until sensors are calibrated
while true
    Wait until sensor readings are ready
    read sensor buffer and copy to local variables
    estimate orientation (complementary filter)
    save orientation to local_state_space
    if state_space_semaphore is available
        take state_space_semaphore
        update the global_state_space variable
        give state_space_semaphore back
    sleep until next time-step
```

B. LQR controller

The state space is updated by the complementary filter, and the reference signal $r(k)$ is defined using the set point generator. Using these values, the control signal will be computed according to the LQR control law with reference, which reads as

$$\mathbf{u}(k) = K_r\mathbf{r}(k) - K\mathbf{x}(k) \tag{47}$$

As in the complementary filter, a problem is that `C` does not have a straight forward implementation of vector multiplication. Because the *structure* of the control law will not change, the matrix vector multiplications will always be the same structurally but with different state space values and references. By predefining the result of the control law, no subroutines will need to be implemented. In Matlab, it is simple to symbolically compute the control law, convert it into C-code and then writing the control law directly into the control task.

Before the computed control law is sent to the motors, a safety function was implemented to stop the output if the angular readings are outside the reasonable scope of the linearized controller.

Schematically, the `updateControlInput()` executes the actions shown in pseudo-code 2.

Listing 2. Pseudo-code for control update thread

```
wait until state_space_semaphore can be taken
take state_space_semaphore
copy global_state_space to local variables
give state_space_semaphore back

wait until reference_semaphore can be taken
take reference_semaphore
copy global_reference to local variables
give reference_semaphore back

Compute control input
if not safe (global_state_space)
    shut down
else
    write control input to motor values
sleep until next time-step
```

Because the threads might operate at different frequencies, the purpose of the semaphores used in this thread is to make sure that the global state space and reference variables will not

be updated while the control thread is reading these values, which might cause inconsistency in the values being read.

C. Set point generator

The set point generator function is the one with the lowest priority. This is because updating the set point values is not so crucial for stability than updating the state space variables or updating the control input. The set point function has a quite simple structure as the function `commanderGetPoint()` is already implemented.

This function gets the set point from the smart phone application for the crazyflie, but it can also be used to send values from the computer to the quadrotor. This property can thus also be used in tuning the controller live. When the task runs, it reads the set point input and then sets the global variable `setpoint_state` to that value. This global variable can then be read by the motor input function in order to track the set point. The pseudo-code for this threads is shown in Pseudo-code 3.

Listing 3. Pseudo-code for set point update thread

```
1  if set_point_semaphore is available
2      take set_point_semaphore
3      update the global_reference variable
4      give set_point_semaphore back
5  sleep until next time-step
```

In the same as the complementary filter thread, the update set point thread only writes the reference to global variables if no other thread is in the process of reading them. If that happens, it simply skip writing for that time-step.

D. Semaphore distribution

The motor input function and the complementary filter function both use the same global variable, which means that they could try to access the variable at the same time. To make sure that the functions do not interrupt each other at the crucial point of reading and writing, a semaphore system is set up to ensure that a complete reading or writing to the variable is done before the other function tries to access the variable.

The same problem arises when the set point function is updating the reference and the motor input function tries to read the reference. Therefore, a second semaphore is introduced to guard the shared variable.

E. Evaluation of C-implementation

Using the C-code implemented controller to stabilize the quadrotor gave satisfying results, and plots from two flight tests were generated.

Figure 10 describes the pitch and roll, along with the motor signals normalized to be shown as percentage of maximum thrust. In this test, the quadrotor is sent a constant thrust which is calculated to keep it hovering for as long as possible, but eventually land.

As can be seen, the controller is very well tuned and keeps the quadrotor stable from takeoff until around 50 seconds, where it is close to landing. Here, the ground effect makes the air ”slippery” and this is seen in figure 10 in that the angle deviations get larger. When the crazyflie lands, the

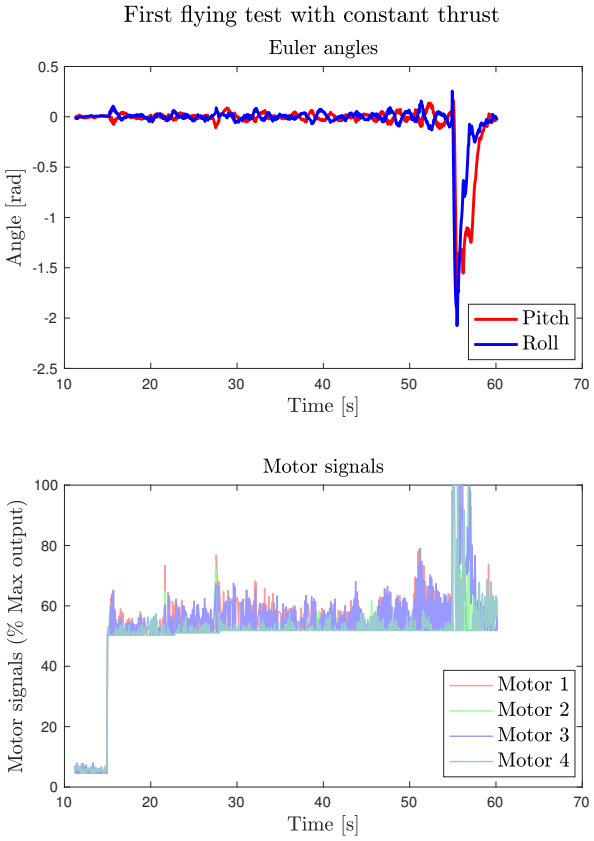

Fig. 10. Angle measurements and motor signals from the quadrotor using the C-implemented controller, first test.

unmodeled forces from the collision with the ground affect the accelerometer gyroscope readings. These are interpreted as large angular deviations in the figure which also leads to larger control signals. As the thrust stays constant, it can also be seen how the quadrotor bounces back up and stabilizes to zero again after hitting the ground.

To compare, another test was made and the results are plotted in figure 11.

In this test, the quadrotor took off from the hand, in order to see if it would achieve a stable lift off. There are some initial deviations, and by implication higher initial motor signals. The crazyflie achieves stability, but later lands unevenly and reads different values in roll and pitch. This leads to higher control action for a longer time period when it bounces back up in the air due to the constant thrust signal.

VIII. Conclusions

From the described differences between the simulation and the measurements from the physical system, we are reminded of the fact that the simulation never represents the physical system perfectly. The controller based on the Simulink model with generated C-code seems to perform a bit differently from the from the controller with our own C-implemented version.

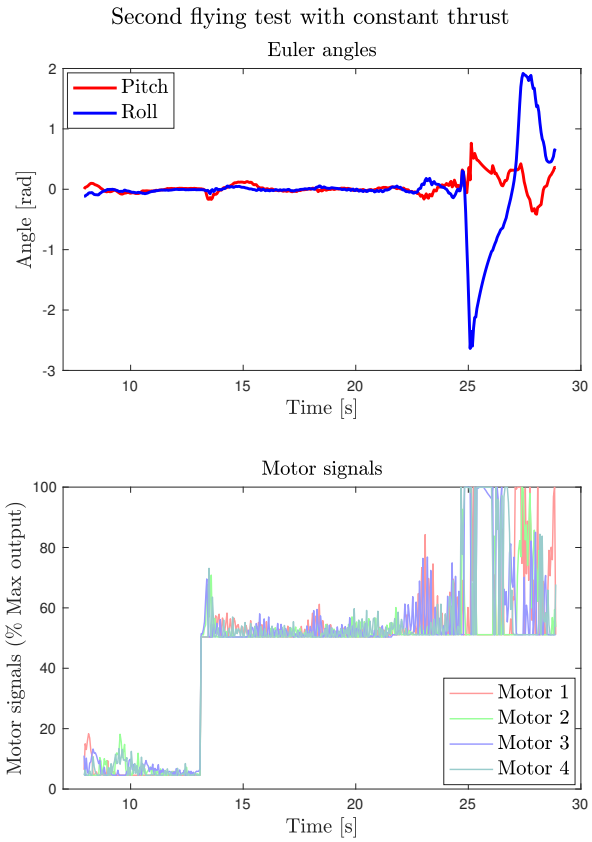

Fig. 11. Angle measurements and motor signals from the quadrotor using the C-implemented controller, second test.

A positive outcome regarding the Simulink version is that the integral action did work well when simulating set point tracking.

In comparison, the C-implemented controller is very stable in flight and also accepts set point tracking in roll, pitch, yaw and thrust via the smartphone app hand controller. Thus, the more successful implementation is the C-controller without integral action, but this version could not have been achieved without first modelling, simulating and computing crucial parameters such as LQR gain and transformation matrices in Matlab and Simulink. These tools are suited for developing the control system, but the sharp version should be implemented efficiently in C.

Afterwards, there are some areas in the development of the system in which further work can be made to achieve an even better controller than the one implemented in this project.

A. Keeping the Crazyflie Static

When testing the Simlulink LQRI implementation, the Crazyflie was tied up along the y-axis to keep it fixed in the air in order to test the ability of the LQRI to stabilize the system around the setpoint or point of linearization. However, this system is not ideal.

First of all, this system limits the movement so that the control of the roll and yaw angle cannot be tested. In the reduced state space that the LQR is based upon, we do not have control over yaw, but we do control yaw rate. Since we would like to show that we can stabilize or have set point tracking in pitch, roll and yaw rate, one suggestion is to fix the origin point of the underside of the crazyflie on a small ball joint on a podium. It would then be free to move in roll, pitch and yaw while still being kept in the air.

Secondly, the string that we have tied to the quadrotor is also elastic. This means that when the quadrotor wobbles, this can create standing waves in which resonance may occur and throw the system into unexpected and potentially uncontrollable states. The fact that the strings are hand tied to the crazyflie, asymmetrically. This creates a torsion along the line which also create disturbances which are not modelled.

B. Actuator Limitations

Another problem that seems to emerge when running the controller on the hardware is that the motors take some time to act according to the controller signals. We think that the reason for this is that the actuators are not modelled with enough precision. The assumption that we make is that the applied voltage versus speed relationship is linear, which is not true in all regions of the DC-motor. A desired motor thrust will not be achieved according to the same dynamics in a non-linear model compared to a linear motor model, which translates to different amounts of time depending on if you give the directive in a linear or non linear region of the motor dynamics. An example of this is that if a base thrust is applied, the controller is more responsive since the motors are operating in a more linear region compared to operating from standstill.

A solution to this problem would be to incorporate a classic DC-motor model into the dynamics of the crazyflie.

C. Aerodynamics

Another parameter that could be of some importance is the fact that we have excluded air resistance from the model. In some sense, given that the air resistance acts as a damping factor, we are modelling a system that could oscillate or accelerate forever. Even if the air resistance might be negligible in most cases, this may affect dynamics in simulation, not to mention the extreme case of physical free fall if one was to drop the Crazyflie from a height.

D. Transformation of Euler Angles for Derivatives and Integrals of Angles

In many cases throughout this project, problems have arisen as a result of not considering that it makes no physical sense to try to directly differentiate or integrate angles expressed in Euler angles [Sic09]. As a result, a transformation matrix was used for example in computing the angular velocity from the sensor values in the state space, but also to integrate them in the integral action section.

E. C-implemented controller

Regarding the C-implemented controller, we are able to control the crazyflie using a smartphone. The crazyflie is able to lift and fly in a stable fashion, which means that the setpoint tracking and control is working. Roll, pitch and yaw setpoint tracking also works with precision. However, in some tests, the Crazyflie accelerated directionally in the xy-plane; this required the testing to be done in large open spaces to confirm that the controller was working correctly.

One thing that can cause this directional movement in the quadrotor is the battery placement. As battery weight is a substantial part of the crazyflie total weight, the placement of this affects the center of gravity considerably. However, this placement is neither fixed nor modeled, and the possible displacements can thus cause an unmodelled weight distribution offset which introduces an angle bias. This will ultimately cause the quadrotor to drift in some direction.

Another issue is the drift that the gyroscope produces. In the complementary filter, the same cutoff frequency is used for the low-pass filter on the accelerometer and the high-pass filter for the gyroscope. A compromise has to be done, where the noise and drift will be minimized as much as possible. However, there will never be a point where both of these issues are completely fixed. The current solution is to introduce a *forgetting factor* on the gyroscope readings, that makes the system disregard old values which are otherwise ”stored” because of the integration.

Another solution to minimize both errors would be to use a more complex Kalman filter on the measurements instead which would improve the performance of the angle estimation, but time constraints did not allow us to try this solution. This solution would also require an error analysis to determine which type of Kalman filter to use (linear or non-linear).

REFERENCES

[Cor17] Peter Corke. *Robotics, vision and control*. Springer Berlin Heidelberg, New York, NY, 2017.

[Sic09] Bruno Siciliano, editor. *Robotics: modelling, planning and control*. Advanced textbooks in control and signal processing. Springer, London, 2009. OCLC: ocn144222188.

APPENDIX

Please see the pdf attached to the end of this document.

: Appendix A

```
#include "your_code.h"

#define ALPHA 0.01f
#define g 9.81f
#define DT 0.01f
// #define GAMMA (ALPHA / (ALPHA + DT))
float GAMMA = (ALPHA / (ALPHA + DT));

#define DEBUG_MODULE "STAB"
#define DEG2RAD (3.1415f/180.f)
#define MIN(a,b) (((a)<(b))?(a):(b))
#define MAX(a,b) (((a)>(b))?(a):(b))
#define THRUST_CONV (4.0f*65536.0f/(0.06f*9.81f))
#define ACC_RANGE_PERC (0.25f)
#define SAFETY_MAX_ANG (60.f*DEG2RAD)
#define BASE_THRUST (0.f)

sensorData_t sensorData;
float state[5];
setpoint_t setpoint;
state_t setpoint_state;

// onlinetuning parameters
float factor_ref  = 30.f;
float fac_fb_rp   = 10.f;
float fac_fb_v    = 1.2f;
float fac_fb_yr   = 1.f;
float GYR_FORGET_FAC = 0.99f;
float ADD_THRUST = 0;

// array for storing angle output from filter
float angle_filter[3] = {0,0,0};
float angle_acc[3]    = {0,0,0};
float angle_gyro[3]   = {0,0,0};
float sensor_gyro[3]  = {0,0,0};
float sensor_acc[3]   = {0,0,0};
float motor_values[4] = {0,0,0,0};

SemaphoreHandle_t stateSemaphore, refSemaphore, calSemaphore;


// Thread to read from setpoint
void updateReference(void *pvParameters){
        TickType_t lastWakeTime = xTaskGetTickCount ();
        while(1){
                if( xSemaphoreTake( refSemaphore, (TickType_t) 1 ) == pdTRUE ){
                        commanderGetSetpoint(&setpoint, &setpoint_state);
                        xSemaphoreGive( refSemaphore );
                }
                vTaskDelayUntil(&lastWakeTime, F2T(RATE_100_HZ));
        }
}

// Thread to calculate control input and set those values to the motors
void updateControlInput(void *pvParameters){

        TickType_t lastWakeTime = xTaskGetTickCount ();
        DEBUG_PRINT("STAB - CONTROLLER STARTED AND WAITING FOR CALIBRATION.\n");
        xSemaphoreTake(calSemaphore, portMAX_DELAY);
        DEBUG_PRINT("STAB - CONTROLLER CALIBRATED.\n");

        float u_fb[4], u_ref[4], ref[3], l_state[5];
        float safety_bool = 1.f;
        while(1){

                // read shared resource
```

```
                xSemaphoreTake( stateSemaphore, portMAX_DELAY );
                l_state[0] = state[0];
                l_state[1] = state[1];
                l_state[2] = state[2];
                l_state[3] = state[3];
                l_state[4] = state[4];
                xSemaphoreGive(stateSemaphore);

                // u_fb = -K * x
                u_fb[0] = + 0.003384f*l_state[0]*fac_fb_rp + 0.003483f*l_state[1]*fac_fb_rp +
                 ↪  0.01073f*l_state[2]*fac_fb_v + 0.01105f*l_state[3]*fac_fb_v + 0.01737f*l_state[4]*fac_fb_yr;
                   u_fb[1] = + 0.003384f*l_state[0]*fac_fb_rp - 0.003483f*l_state[1]*fac_fb_rp +
                    ↪  0.01073f*l_state[2]*fac_fb_v - 0.01105f*l_state[3]*fac_fb_v - 0.01737f*l_state[4]*fac_fb_yr;
                   u_fb[2] = - 0.003384f*l_state[0]*fac_fb_rp - 0.003483f*l_state[1]*fac_fb_rp -
                    ↪  0.01073f*l_state[2]*fac_fb_v - 0.01105f*l_state[3]*fac_fb_v + 0.01737f*l_state[4]*fac_fb_yr;
                   u_fb[3] = - 0.003384f*l_state[0]*fac_fb_rp + 0.003483f*l_state[1]*fac_fb_rp -
                    ↪  0.01073f*l_state[2]*fac_fb_v + 0.01105f*l_state[3]*fac_fb_v - 0.01737f*l_state[4]*fac_fb_yr;

                // u_ref = Kr * ref
                xSemaphoreTake( refSemaphore, portMAX_DELAY        );
                ref[0] =  setpoint.attitude.roll    * DEG2RAD;
                ref[1] = -setpoint.attitude.pitch   * DEG2RAD;
                ref[2] =  setpoint.attitudeRate.yaw * DEG2RAD;
                xSemaphoreGive( refSemaphore );
                u_ref[0] = -0.0034f*ref[0] -0.0035f*ref[1] -0.0174f*ref[2];
                   u_ref[1] = -0.0034f*ref[0] +0.0035f*ref[1] +0.0174f*ref[2];
            u_ref[2] =  0.0034f*ref[0] +0.0035f*ref[1] -0.0174f*ref[2];
            u_ref[3] =  0.0034f*ref[0] -0.0035f*ref[1] +0.0174f*ref[2];

                // Control input = u_fb + u_ref + ref_thrust
                motor_values[0] = ( u_fb[0] + u_ref[0]*factor_ref ) * THRUST_CONV;
                motor_values[1] = ( u_fb[1] + u_ref[1]*factor_ref ) * THRUST_CONV;
                motor_values[2] = ( u_fb[2] + u_ref[2]*factor_ref ) * THRUST_CONV;
                motor_values[3] = ( u_fb[3] + u_ref[3]*factor_ref ) * THRUST_CONV;

                motor_values[0] = MIN(65500.f, MAX(motor_values[0], 0.f) + BASE_THRUST + setpoint.thrust + ADD_THRUST
                 ↪  );
                motor_values[1] = MIN(65500.f, MAX(motor_values[1], 0.f) + BASE_THRUST + setpoint.thrust + ADD_THRUST
                 ↪  );
                motor_values[2] = MIN(65500.f, MAX(motor_values[2], 0.f) + BASE_THRUST + setpoint.thrust + ADD_THRUST
                 ↪  );
                motor_values[3] = MIN(65500.f, MAX(motor_values[3], 0.f) + BASE_THRUST + setpoint.thrust + ADD_THRUST
                 ↪  );

                // Safety function: turns off all motors if roll or pitch is outside limit
                if (l_state[0] > SAFETY_MAX_ANG || l_state[0] < -SAFETY_MAX_ANG ||
                    l_state[1] > SAFETY_MAX_ANG || l_state[0] < -SAFETY_MAX_ANG){
                        safety_bool = 0.f;
                }

                // Add Base thrust and saturate signals before converting to uint16_t
                motorsSetRatio(MOTOR_M1, (uint16_t) motor_values[0]*safety_bool );
                motorsSetRatio(MOTOR_M2, (uint16_t) motor_values[1]*safety_bool );
                motorsSetRatio(MOTOR_M3, (uint16_t) motor_values[2]*safety_bool );
                motorsSetRatio(MOTOR_M4, (uint16_t) motor_values[3]*safety_bool );

                // sleep until next time step
                vTaskDelayUntil(&lastWakeTime, F2T(RATE_100_HZ));
        }
}

void compFilter(void *pvParameters){

        TickType_t lastWakeTime;
        float norm;

        // Wait for calibration
        DEBUG_PRINT("STAB - WAITING FOR CALIBRATION.\n");
```

```
        lastWakeTime = xTaskGetTickCount ();
        while(!sensorsAreCalibrated()) {
                vTaskDelayUntil(&lastWakeTime, F2T(RATE_100_HZ));
        }
        xSemaphoreGive( calSemaphore );
        DEBUG_PRINT("STAB - CALIBRATED.\n");
        DEBUG_PRINT("STAB - FILTERING.\n");
        lastWakeTime = xTaskGetTickCount ();
        while(1){

                // read from sensors
                sensorsWaitDataReady();
                sensorsAcquire(&sensorData, 1);

                // Acceleration observer
                norm = sqrt(pow(sensorData.acc.x, 2) + pow(sensorData.acc.y, 2) + pow(sensorData.acc.z, 2));
                // normalize acceleration measurements
                if (norm != 0){
                        sensor_acc[0] = sensorData.acc.x / norm;
                        sensor_acc[1] = sensorData.acc.y / norm;
                        sensor_acc[2] = sensorData.acc.z / norm;
                }
                // calculate rotation angles based on acceleration sensor
                angle_acc[0] = atan2(sensor_acc[1], sensor_acc[2]);
                angle_acc[1] = atan2(-sensor_acc[0], sqrt(pow(sensor_acc[1],2) + pow(sensor_acc[2],2)) );
                angle_acc[2] = 0;

                // Gyro observer
                sensor_gyro[0] = DEG2RAD*sensorData.gyro.x;
                sensor_gyro[1] = DEG2RAD*sensorData.gyro.y;
                sensor_gyro[2] = DEG2RAD*sensorData.gyro.z;

                // equations imported from Matlab
                // and translated into C
                angle_gyro[0] = angle_gyro[0] - sensor_gyro[2] * (-DT * sin(angle_filter[1]) + (DT *
                 ↪  pow(cos(angle_filter[0]), 2.0) * cos(angle_filter[1]) * sin(angle_filter[1])) /
                 ↪  (pow(cos(angle_filter[0]), 2.0) * cos(angle_filter[1]) + cos(angle_filter[1]) *
                 ↪  pow(sin(angle_filter[0]), 2.0)) + (DT * cos(angle_filter[1]) * pow(sin(angle_filter[0]), 2.0) *
                 ↪  sin(angle_filter[1])) / (pow(cos(angle_filter[0]), 2.0) * cos(angle_filter[1]) +
                 ↪  cos(angle_filter[1]) * pow(sin(angle_filter[0]), 2.0))) + sensor_gyro[0] * (DT *
                 ↪  cos(angle_filter[2]) * cos(angle_filter[1]) - (DT * cos(angle_filter[0]) * sin(angle_filter[1])
                 ↪  * (sin(angle_filter[0]) * sin(angle_filter[2]) - cos(angle_filter[0]) * cos(angle_filter[2]) *
                 ↪  sin(angle_filter[1]))) / (pow(cos(angle_filter[0]), 2.0) * cos(angle_filter[1]) +
                 ↪  cos(angle_filter[1]) * pow(sin(angle_filter[0]), 2.0)) + (DT * sin(angle_filter[0]) *
                 ↪  sin(angle_filter[1]) * (cos(angle_filter[0]) * sin(angle_filter[2]) + cos(angle_filter[2]) *
                 ↪  sin(angle_filter[0]) * sin(angle_filter[1]))) / (pow(cos(angle_filter[0]), 2.0) *
                 ↪  cos(angle_filter[1]) + cos(angle_filter[1]) * pow(sin(angle_filter[0]), 2.0))) - sensor_gyro[1]
                 ↪  * (DT * cos(angle_filter[1]) * sin(angle_filter[2]) + (DT * cos(angle_filter[0]) *
                 ↪  sin(angle_filter[1]) * (cos(angle_filter[2]) * sin(angle_filter[0]) + cos(angle_filter[0]) *
                 ↪  sin(angle_filter[2]) * sin(angle_filter[1]))) / (pow(cos(angle_filter[0]), 2.0) *
                 ↪  cos(angle_filter[1]) + cos(angle_filter[1]) * pow(sin(angle_filter[0]), 2.0)) - (DT *
                 ↪  sin(angle_filter[0]) * sin(angle_filter[1]) * (cos(angle_filter[0]) * cos(angle_filter[2]) -
                 ↪  sin(angle_filter[0]) * sin(angle_filter[2]) * sin(angle_filter[1]))) /
                 ↪  (pow(cos(angle_filter[0]), 2.0) * cos(angle_filter[1]) + cos(angle_filter[1]) *
                 ↪  pow(sin(angle_filter[0]), 2.0)));
                angle_gyro[1] = angle_gyro[1] + sensor_gyro[0] * ((DT * cos(angle_filter[0]) * (cos(angle_filter[0])
                 ↪  * sin(angle_filter[2]) + cos(angle_filter[2]) * sin(angle_filter[0]) * sin(angle_filter[1]))) /
                 ↪  (pow(cos(angle_filter[0]), 2.0) + pow(sin(angle_filter[0]), 2.0)) + (DT * sin(angle_filter[0]) *
                 ↪  (sin(angle_filter[0]) * sin(angle_filter[2]) - cos(angle_filter[0]) * cos(angle_filter[2]) *
                 ↪  sin(angle_filter[1]))) / (pow(cos(angle_filter[0]), 2.0) + pow(sin(angle_filter[0]), 2.0))) +
                 ↪  sensor_gyro[1] * ((DT * cos(angle_filter[0]) * (cos(angle_filter[0]) * cos(angle_filter[2]) -
                 ↪  sin(angle_filter[0]) * sin(angle_filter[2]) * sin(angle_filter[1]))) /
                 ↪  (pow(cos(angle_filter[0]), 2.0) + pow(sin(angle_filter[0]), 2.0)) + (DT * sin(angle_filter[0]) *
                 ↪  (cos(angle_filter[2]) * sin(angle_filter[0]) + cos(angle_filter[0]) * sin(angle_filter[2]) *
                 ↪  sin(angle_filter[1]))) / (pow(cos(angle_filter[0]), 2.0) + pow(sin(angle_filter[0]), 2.0)));
```

```
                angle_gyro[2] = angle_gyro[2] + sensor_gyro[2] * ((DT * pow(cos(angle_filter[0]), 2.0) *
                ↪  cos(angle_filter[1])) / (pow(cos(angle_filter[0]), 2.0) * cos(angle_filter[1]) +
                ↪  cos(angle_filter[1]) * pow(sin(angle_filter[0]), 2.0)) + (DT * cos(angle_filter[1]) *
                ↪  pow(sin(angle_filter[0]), 2.0)) / (pow(cos(angle_filter[0]), 2.0) * cos(angle_filter[1]) +
                ↪  cos(angle_filter[1]) * pow(sin(angle_filter[0]), 2.0))) + sensor_gyro[0] * ((DT *
                ↪  cos(angle_filter[0]) * (sin(angle_filter[0]) * sin(angle_filter[2]) - cos(angle_filter[0]) *
                ↪  cos(angle_filter[2]) * sin(angle_filter[1]))) / (pow(cos(angle_filter[0]), 2.0) *
                ↪  cos(angle_filter[1]) + cos(angle_filter[1]) * pow(sin(angle_filter[0]), 2.0)) - (DT *
                ↪  sin(angle_filter[0]) * (cos(angle_filter[0]) * sin(angle_filter[2]) + cos(angle_filter[2]) *
                ↪  sin(angle_filter[0]) * sin(angle_filter[1]))) / (pow(cos(angle_filter[0]), 2.0) *
                ↪  cos(angle_filter[1]) + cos(angle_filter[1]) * pow(sin(angle_filter[0]), 2.0))) + sensor_gyro[1]
                ↪  * ((DT * cos(angle_filter[0]) * (cos(angle_filter[2]) * sin(angle_filter[0]) +
                ↪  cos(angle_filter[0]) * sin(angle_filter[2]) * sin(angle_filter[1]))) /
                ↪  (pow(cos(angle_filter[0]), 2.0) * cos(angle_filter[1]) + cos(angle_filter[1]) *
                ↪  pow(sin(angle_filter[0]), 2.0)) - (DT * sin(angle_filter[0]) * (cos(angle_filter[0]) *
                ↪  cos(angle_filter[2]) - sin(angle_filter[0]) * sin(angle_filter[2]) * sin(angle_filter[1]))) /
                ↪  (pow(cos(angle_filter[0]), 2.0) * cos(angle_filter[1]) + cos(angle_filter[1]) *
                ↪  pow(sin(angle_filter[0]), 2.0)));

                angle_gyro[0] = angle_gyro[0] * GYR_FORGET_FAC;
                angle_gyro[1] = angle_gyro[1] * GYR_FORGET_FAC;
                angle_gyro[2] = angle_gyro[2] * GYR_FORGET_FAC;

                // Complementary filter
                angle_filter[0] = (1 - GAMMA) * angle_acc[0] + GAMMA * angle_gyro[0];
                angle_filter[1] = (1 - GAMMA) * angle_acc[1] + GAMMA * angle_gyro[1];
                angle_filter[2] = (1 - GAMMA) * angle_acc[2] + GAMMA * angle_gyro[2];

                // Save shared resource
                if( xSemaphoreTake( stateSemaphore, (TickType_t) 1 ) == pdTRUE ){
                        state[0] = angle_filter[0];
                        state[1] = angle_filter[1];
                        state[2] = sensor_gyro[0];
                        state[3] = sensor_gyro[1];
                        state[4] = sensor_gyro[2];
                        xSemaphoreGive(stateSemaphore);
                }

                // Give the semaphore to the control thread

                // sleep until next time step
                vTaskDelayUntil(&lastWakeTime, F2T(RATE_100_HZ));
        }
}

/*         Main function that is called after initing the sensor and motors
         Executes three threads which will:
                estimate states based on measurements
                calculate control input and inputs to the motors
                read from reference (cellphone)
*/
void yourCodeInit(void){
        stateSemaphore = xSemaphoreCreateBinary();
        xSemaphoreGive( stateSemaphore );
        refSemaphore = xSemaphoreCreateBinary();
        xSemaphoreGive( refSemaphore );
        calSemaphore = xSemaphoreCreateBinary();
        xSemaphoreGive( calSemaphore );
        xSemaphoreTake( calSemaphore, (TickType_t) 1 );

        xTaskCreate(compFilter, "Complementary Filter", 1024, NULL, 4, NULL);
        xTaskCreate(updateControlInput, "Controller", 1024, NULL, 3, NULL);
        xTaskCreate(updateReference, "Reference", 1024, NULL, 2, NULL);
}


// Add ONLINE tuning parameters to the Graphic User Interface
PARAM_GROUP_START(AA_STAB)
```

```
PARAM_ADD(PARAM_FLOAT, factor_ref, &factor_ref)
PARAM_ADD(PARAM_FLOAT, fac_fb_rp, &fac_fb_rp)
PARAM_ADD(PARAM_FLOAT, fac_fb_v, &fac_fb_v)
PARAM_ADD(PARAM_FLOAT, fac_fb_yr, &fac_fb_yr)
PARAM_ADD(PARAM_FLOAT, GAMMA, &GAMMA)
PARAM_ADD(PARAM_FLOAT, GYR_FORGET_FAC, &GYR_FORGET_FAC)
PARAM_ADD(PARAM_FLOAT, ADD_THRUST, &ADD_THRUST)
PARAM_GROUP_STOP(AA_STAB)


// Add signals to be ploted in the GUI
LOG_GROUP_START(AA_thrust)
LOG_ADD(LOG_FLOAT, tx, &setpoint.thrust)
LOG_ADD(LOG_FLOAT, ty, &setpoint.thrust)
LOG_ADD(LOG_FLOAT, tz, &setpoint.thrust)
LOG_GROUP_STOP(AA_thrust)

LOG_GROUP_START(AA_gyro)
LOG_ADD(LOG_FLOAT, x, &sensorData.gyro.x)
LOG_ADD(LOG_FLOAT, y, &sensorData.gyro.y)
LOG_ADD(LOG_FLOAT, z, &sensorData.gyro.z)
LOG_GROUP_STOP(AA_gyro)

LOG_GROUP_START(AA_angle_filter)
LOG_ADD(LOG_FLOAT, roll,  &angle_filter[0])
LOG_ADD(LOG_FLOAT, pitch, &angle_filter[1])
LOG_ADD(LOG_FLOAT, yaw,   &angle_filter[2])
LOG_GROUP_STOP(AA_angle_filter)

LOG_GROUP_START(AA_angle_acc)
LOG_ADD(LOG_FLOAT, roll,  &angle_acc[0])
LOG_ADD(LOG_FLOAT, pitch, &angle_acc[1])
LOG_ADD(LOG_FLOAT, yaw,   &angle_acc[2])
LOG_GROUP_STOP(AA_angle_acc)

LOG_GROUP_START(AA_angle_gyro)
LOG_ADD(LOG_FLOAT, roll,  &angle_gyro[0])
LOG_ADD(LOG_FLOAT, pitch, &angle_gyro[1])
LOG_ADD(LOG_FLOAT, yaw,   &angle_gyro[2])
LOG_GROUP_STOP(AA_angle_gyro)

LOG_GROUP_START(AA_motor)
LOG_ADD(LOG_FLOAT, motor_1, &motor_values[0])
LOG_ADD(LOG_FLOAT, motor_2, &motor_values[1])
LOG_ADD(LOG_FLOAT, motor_3, &motor_values[2])
LOG_ADD(LOG_FLOAT, motor_4, &motor_values[3])
LOG_GROUP_STOP(AA_motor)
```