Modelling, Simulation, control, and Implementation of a Quadrotor Group 13

Anton Stigemyr Hill David Espedalen Siyu Yi Yinsong Wang

***Abstract*—This report presents the development of a control system for a drone using the Crazyflie 2.0 development platform. The process includes estimating the drone's orientation and correcting biases of the sensor data by implementing filters. The controller is simulated using Simulink and Simscape to obtain relevant parameters, which are then evaluated for optimal control. This is finally implemented in C-code and flashed to the drone.**

***Index Terms*—Crazyflie 2.0, model-based design, quadrotor simulation, LQR control, FreeRTOS**

I. Introduction

In an increasingly automated world, model-based design can be seen in a wide range of systems from heat regulation to control of autonomous drive systems. Even though these systems may seem far from each other at first glance, the control theory behind them works in the same way. This is the reason why control theory is an important issue in modern technology. Model-based design is a tool to speed up the development processes and save time and resources by simulating, analyzing, and optimizing these systems before implementation.

In this Project, the development and implementation of a control system is done for a Quadcopter using the Crazyflie 2.0 development platform from Bitcraze. This is done to get hands-on experience with the theories learned in the MPSYS program at Chalmers. This is an important set of knowledge since to implement control systems in real life some additional steps that lay outside the range of control theory are needed. These include setting up a test environment, converting the control system into code the drone can understand, and uploading the code to the drone. This was done in a Matlab environment in Simulink and Simscape created by the course which was then compiled to C-code and later uploaded to the quadcopter wirelessly via the Crazyradio 2.0 using a VirtualBox. [1] After evaluating the control system, all previously developed parts were implemented directly in C-code to control the drone in real time.

To understand this project, some prior knowledge of basic control systems design is needed however, the concepts will be explained rather thoroughly.

II. Table of variables

What follows below is a table of all variables used in the project in order. If any question arises about a variable, look up its name and read its description.

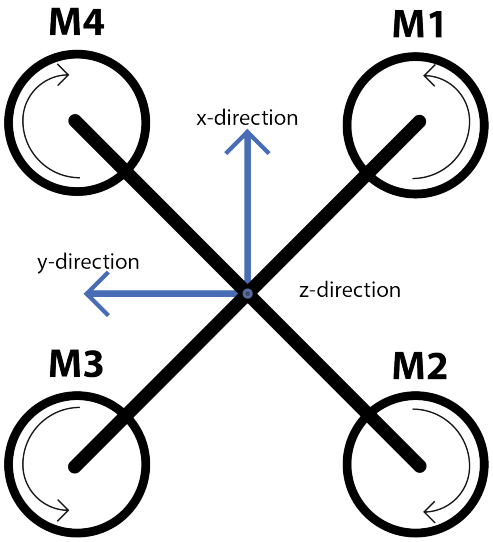

Fig. 1. Drone orientation

TABLE I
Variables

Parameter	Explanation
φ	Roll angle
θ	Pitch angle
ψ	Yaw angle
${}^{B}R_{\omega}$	rotation matrix from body to world
${}^{\omega}R_{B}$	rotation matrix from world to body
h	Sampling time
${}^{\omega}a$	Acceleration with respect to world frame
${}^{B}f$	Force acting on the body
m	Mass of the quadrocopter
γ	Weighting factor
θ_a	Estimation from accelerometer
θ_g	Estimation from gyroscope
y_g	Estimation from the gyroscope (before integration)
T_s	Sampling time
v	Speed
T	Total thrust
F	Air friction coefficient
τ	Torque
L	Lift
D	Drag
d	Distance between motor and origin
K	LQR feedback gain
Q	States weighting matrix
R	Inputs weighting matrix
An	Continuous time A matrix
Bn	Continuous time B matrix
A_d	Discrete time A matrix
B_d	Discrete time B matrix
r	Reference

III. Orientation Estimation

The position of the quadrotor is defined by a state vector with six states, $x, y, z, \phi, \theta, \psi$. Where ϕ, θ, ψ represents the rotation in the three different axes. The rotation is denoted by applying the right-hand rule which easily gives the positive or the negative rotation direction. To get the quadrotor's orientation from each motor, i.e. body, to the world frame a rotation matrix is used. The rotation matrix is given by the product of the three angles rotation matrices. To be more clear, the angles roll (ϕ), pitch (θ), and yaw (ψ). The rotation matrices R_ϕ, R_θ, and R_ψ are multiplied together to get the rotation and transformation from the body to the world frame. $R_{xyz} = R_x(\phi)R_y(\theta)R_z(\psi)$.

$$R_x(\varphi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos(\varphi) & -sin(\varphi) \\ 0 & sin(\varphi) & cos(\varphi) \end{bmatrix} \tag{1}$$

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

$$R_z(\psi) = \begin{bmatrix} cos(\psi) & -sin(\psi) & 0 \\ sin(\psi) & cos(\psi) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{3}$$

A. Estimate angles from accelerometers

The roll, pitch, and yaw angles are derived using the gyroscope and the accelerometer in the quadrotor. The gyroscope measures the angular velocity, i.e. the rate of change of the quadrotor's orientation. The accelerometer defines the external G-force on the quadrotor sensors and the quadrotor's gravity itself.

$$\begin{bmatrix} {}^Bf_x \\ {}^Bf_y \\ {}^Bf_z \end{bmatrix} = m \cdot \overset{B}{R}_\omega \cdot \left(\begin{bmatrix} {}^\omega a_x \\ {}^\omega a_y \\ {}^\omega a_z \end{bmatrix} - \begin{bmatrix} 0 \\ 0 \\ g \end{bmatrix} \right) \tag{4}$$

When body is at rest, ${}^\omega a_x = 0$, ${}^\omega a_y = 0$ and ${}^\omega a_z = 0$, the accelerometer only measures the acceleration as gravity. The accelerometer was tested which revealed it was calibrated to give -1 when aligned with the earth's gravity field. We then have:

$$\begin{bmatrix} {}^Bf_x \\ {}^Bf_y \\ {}^Bf_z \end{bmatrix} = m \cdot \overset{B}{R}_\omega \cdot \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \tag{5}$$

From this we know that $\overset{B}{R}_\omega = (\overset{\omega}{R}_B)^T$, which gives the following expression for ${}^Bf_x$, ${}^Bf_y$ and ${}^Bf_z$.

$$\begin{aligned} \begin{bmatrix} {}^Bf_x \\ {}^Bf_y \\ {}^Bf_z \end{bmatrix} &= m \cdot R_x^T(\varphi)R_y^T(\theta)R_z^T(\psi) \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \\ &= m \cdot \begin{bmatrix} -sin(\theta) \\ cos(\theta)sin(\varphi) \\ cos(\theta)cos(\varphi) \end{bmatrix} \end{aligned} \tag{6}$$

With this we can solve the angles φ and θ, which gives:

$$\varphi = atan2({}^Bf_y, {}^Bf_z) \tag{7}$$

and

$$\theta = atan2(-{}^Bf_x, \sqrt{({}^Bf_y)^2 + ({}^Bf_z)^2}) \tag{8}$$

B. Estimate angles from gyroscope

The gyroscope gives the angular velocity of the angles, which means that integration, as shown below, on the gyroscope's given values will give the estimated orientation angle.

$$\theta_g = \int_0^t y_g \tag{9}$$

C. Complementary filter

The estimations from the accelerometer are noisy for short time intervals but are accurate for longer periods. The gyroscope is accurate over short time periods but very sensitive to drift over time, therefore a complementary filter is applied to the system where a low-pass filter is used through the accelerometer, to reduce the high frequencies from the accelerometer. A high-pass filter is used through the gyroscope to reduce the low frequencies. When these two filters are applied to the system, the estimation becomes a combined estimation from both the gyroscope and the accelerometer where the bad estimated parts from the gyroscope and the accelerometer are reduced. The complementary filter is applied at discrete time which means that the continuous system needs to be converted to discrete time before using the complementary filter. A chart of the complementary filter is shown in Figure 2.

The two filters, i.e. the low-pass and the high-pass filter defined as in equation 10 and equation 11

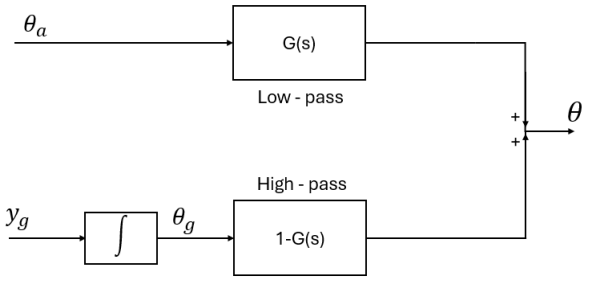

Fig. 2. Complementary filter

$$G(s) = \frac{1}{\alpha s + 1} \tag{10}$$

$$1 - G(s) = 1 - \frac{1}{\alpha s + 1} = \frac{\alpha s}{\alpha s + 1} \tag{11}$$

Now let θ_a be the estimation from the accelerometer and θ_g be the estimation from the gyroscope after one integration, $\theta_g = \int_0^t y_g dt$. From Figure 2 the estimated θ after the complementary filter is calculated as:

$$\theta(S) = G(S) \cdot \theta_a(S) + (1 - G(S)) \cdot \theta_g(S) \tag{12}$$

The estimated θ is then discretized using the Euler-backward method. We get the following expression, where h is the sampling time.

$$\theta_k = (1 - \gamma) \cdot \theta_{a,k} + \gamma \cdot (\theta_{k-1} + h \cdot Y_{g,k}) \tag{13}$$

D. Validation of complementary filter

The complementary filter's response to the sensor data can be seen in Figure 3 and 4. There the drift component is taken care of which is especially noticed in the pitch case.

From the plots, it can be seen that the pitch angle from the integrated gyroscope value is drifting over time. By applying the complementary filter this can be eliminated by taking the gyroscope values through the high-pass filter. The roll angle seems to not be affected by the gyroscope but tends to drift over time. Overall we can conclude that a complementary filter will result in a better estimation rather than using the gyroscope as the orientation estimation.

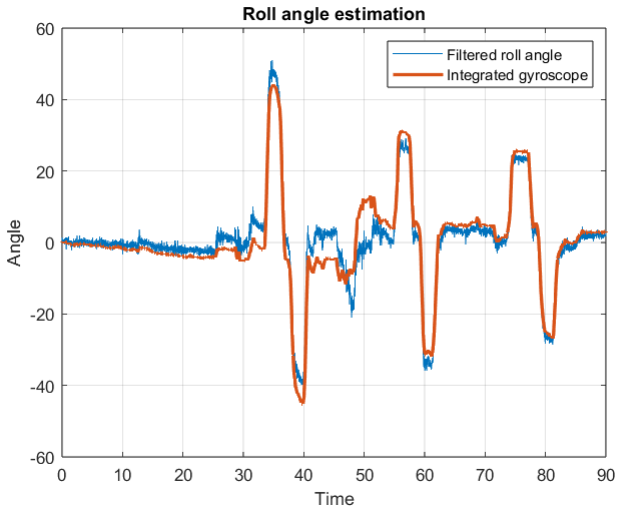

Fig. 3. Plot over the estimated filtered roll angle and the integrated gyroscope roll angle

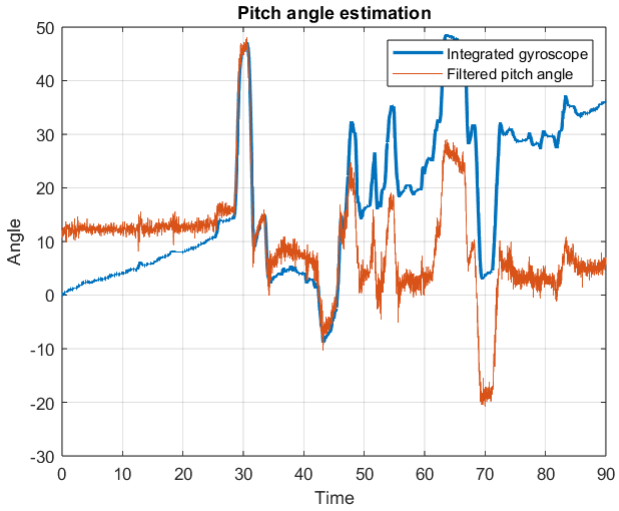

Fig. 4. Plot over the estimated filtered pitch angle and the integrated gyroscope pitch angle

IV. Plant Modeling

When modeling the plant, Matlab's Simscape was used. This is a tool to model systems using only equations. The difference between Simulink and Simscape is that in Simscape, block diagrams are not used. Instead, the relevant relations between variables are stated and then all possible equations are extracted automatically.

For the plant, all equations should be stated in the world frame and therefore we will have to use the rotation matrices developed in the previous chapter to switch between frames. First of we begin with the law of motion. To do this in the world frame the following equation is used.

$$ma = \overset{B}{R}_{\omega} \begin{bmatrix} 0 \\ 0 \\ T \end{bmatrix} - \begin{bmatrix} 0 \\ 0 \\ mg \end{bmatrix} - Fv \tag{14}$$

Next, we need to define the angular equations of motion. Once again we begin in the body frame and later move it to the world. The torque generated in the body frame can be expressed in x, y, and z directions. The torque in the x-direction (Roll-torque) is generated by a positive input from motors 3 and 4 while motors 1 and 2 contribute to a negative torque. This thrust is calculated by lift times motor speed squared, $T_i = L\omega_i^2$. The thrust is then multiplied by $cos(\frac{\pi}{4})$ to capture the moment applied on the x-axis. The same method is used for the y-axis case. In this case, however, the motors contribute in different directions, changing the signs. The rotation around the z-axis is only dependent on the conservation of angular momentum. This can be calculated by the drag times motor speed squared and as $\tau_i = D\omega_i^2 = \frac{D}{L}T_i$ giving the full expression:

$$\tau_{tot} = \begin{bmatrix} \tau_x \\ \tau_y \\ \tau_z \end{bmatrix} = \begin{bmatrix} d \cdot cos(\frac{\pi}{4}) \cdot (T_3 + T_4 - T_1 - T_2) \\ d \cdot cos(\frac{\pi}{4}) \cdot (T_2 + T_3 - T_1 - T_4) \\ \frac{D}{L} \cdot (T_2 + T_4 - T_1 - T_3) \end{bmatrix} \tag{15}$$

Now Newton's second law of rotation in vector space can be used

$$J \cdot \dot{\omega} = -\omega \times (J\omega) + \tau_{tot} \tag{16}$$

The last step is then to multiply this with the Euler angle transform matrix to get it to the world frame.

$$\omega_b = \begin{bmatrix} 1 & 0 & sin(\theta) \\ 0 & cos(\varphi) & -sin(\varphi)cos(\theta) \\ 0 & sin(\varphi) & cos(\varphi)cos(\theta) \end{bmatrix} \omega \tag{17}$$

To check whether the Simscape model worked as intended, two different test cases were used in an open-loop system. For the first one motor 1 and motor 2 were set to 10 at time 4s which results in the drone rotating with a negative roll. This can be seen in the plots for case 1 in Figure 5 and 6. For the other case, motor 1 was set to 10000 at time 4s, and motor 3 was set to 2000 at time 6s. Now all angles are affected at 4s since the drone starts spinning in all directions and the accelerations jump from -1 to 1 because of that as well which can be seen in Figure 7 and 8.

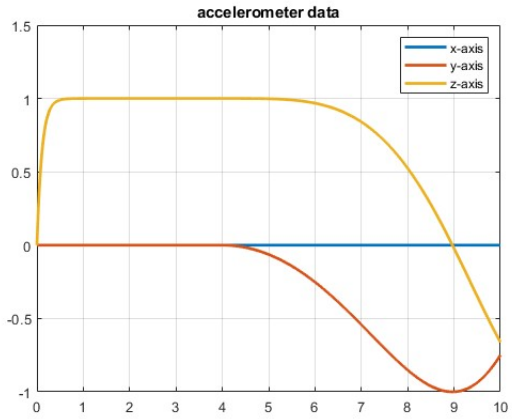

Fig. 5. Accelerometer data for case 1

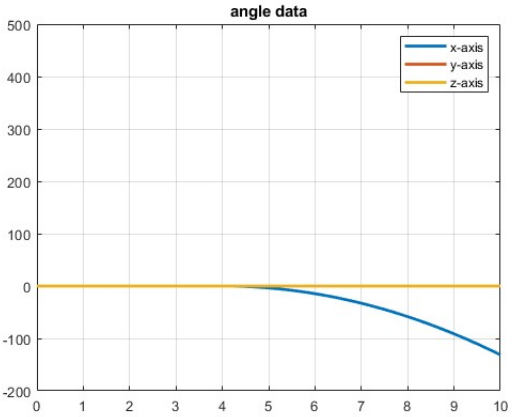

Fig. 6. Angle data for case 1

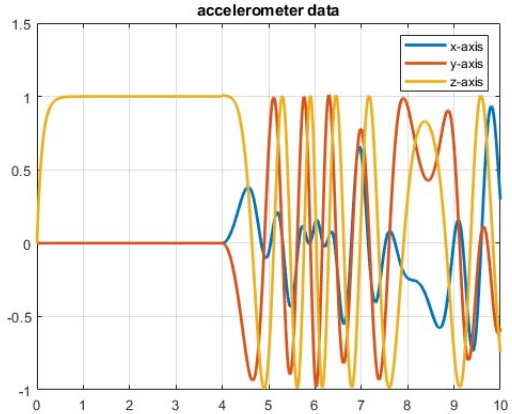

Fig. 7. Accelerometer data for case 2

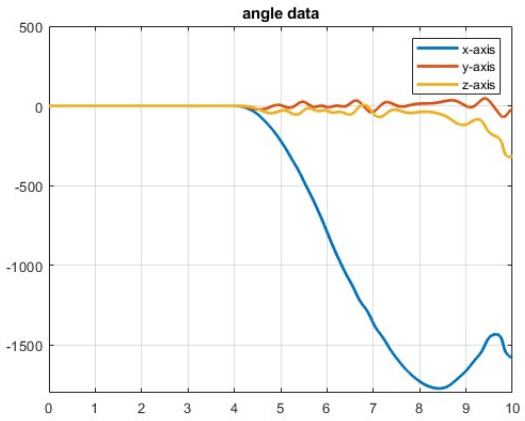

Fig. 8. Angle data for case 2

V. Plant Linearization

According to the quadrotor model, the system is nonlinear, which means direct control application is not possible. To enable the use of an LQ controller, the system must first be linearized.

We have chosen the state representation for this system as

$$x = \begin{bmatrix} \varphi \\ \theta \\ \dot{\varphi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} \tag{18}$$

where the model of the plant is described by the equation $\dot{x} = f(x, u)$. The dynamics of the system are captured by the function

$$\dot{x} = f(x, u) = \begin{bmatrix} \dot{\varphi} \\ \dot{\theta} \\ \ddot{\varphi} \\ \ddot{\theta} \\ \ddot{\psi} \end{bmatrix} \tag{19}$$

The input to the system consists of the thrust generated by four motors. To linearize the model, we assume that the operating point is where the thrust from the four motors exactly counterbalances the force of gravity, allowing the system to hover in mid-air. Therefore, it is set as

$$u_{\text{lin}} = \begin{bmatrix} mg/4 \\ mg/4 \\ mg/4 \\ mg/4 \end{bmatrix} = \begin{bmatrix} 0.0662 \\ 0.0662 \\ 0.0662 \\ 0.0662 \end{bmatrix} \tag{20}$$

and the initial state is

$$x_0 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{21}$$

Subsequently, we can derive the state-space model equations as follows:

$$\dot{x} = \underbrace{\left.\frac{\partial f}{\partial x}\right|_{x_o, u_{\text{lin}}}}_{A} \Delta x + \underbrace{\left.\frac{\partial f}{\partial u}\right|_{x_o, u_{\text{lin}}}}_{B} \Delta u \tag{22}$$

Using the Jacobian method, we calculated the matrices A_n, B_n as follows:

$$A_n = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$B_n = 10^3 \times \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ -2.8376 & -2.8376 & 2.8376 & 2.8376 \\ -1.9141 & 1.9141 & 1.9141 & -1.9141 \\ -0.4282 & 0.4282 & -0.4282 & 0.4282 \end{bmatrix}$$

Now, using the linearized matrices A_n and B_n, we can calculate the discretized model. Given a sampling time of $T_s = 0.01$ seconds, the discrete-time system matrices, A_d and B_d, can be computed as follows: [2]

$$A_d = e^{A_n T_s} \tag{23}$$

$$B_d = \left(\int_{\tau=0}^{T_s} e^{A_n \tau} d\tau \right) B_n \tag{24}$$

the matrices A_d, B_d as follows:

$$A_d = \begin{bmatrix} 1 & 0 & 0.01 & 0 & 0 \\ 0 & 1 & 0 & 0.01 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$B_d = \begin{bmatrix} -0.1419 & -0.1419 & 0.1419 & 0.1419 \\ -0.0957 & 0.0957 & 0.0957 & -0.0957 \\ -28.3756 & -28.3756 & 28.3756 & 28.3756 \\ -19.1414 & 19.1414 & 19.1414 & -19.1414 \\ -4.2821 & 4.2821 & -4.2821 & 4.2821 \end{bmatrix}$$

VI. Design of the Linear Quadratic Regulator (LQR) Controller

The Linear Quadratic Controller(LQR) could be developed following the block diagram in Figure 9.

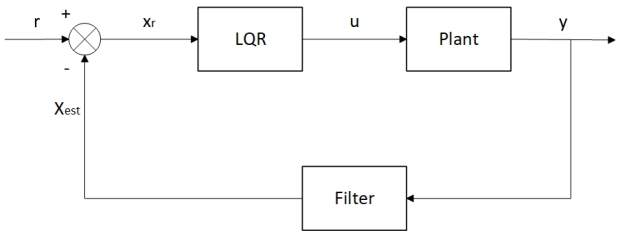

Fig. 9. LQR block diagram

Reference tracking is applied on the angular pitch and roll as well as yaw rate as angular velocity.

$$r = \begin{bmatrix} \varphi \\ \theta \\ \dot{\psi} \end{bmatrix} \tag{25}$$

Using the discrete-time state-space model, we design our LQR controller by focusing on minimizing the infinite time quadratic regulation cost function. This function is critical for evaluating the performance of the controller across an unlimited time horizon. The cost function J is defined as:

$$J = \min \sum_{k=0}^{\infty} x_k^\top Q x_k + u_k^\top R u_k \tag{26}$$

where x_k represents the state vector at step k, and u_k denotes the control input at step k. The matrices Q and R are weighting factors that balance the state error and control effort, respectively.

Next, we define the feedback gain, which is crucial for determining the effectiveness of the control response:

$$u(k) = -K\hat{x}(k) \tag{27}$$

Here, $\hat{x}(k)$ represents the estimate of the state at time k, and K is the feedback gain matrix calculated using the discrete-time Riccati equations:

$$\begin{aligned} K &= (B_d^\top S B_d + Q)^{-1} B_d^\top S A_d \\ S &= A_d^\top S A_d + Q - A_d^\top S B_d (B_d^\top S B_d + R)^{-1} B_d^\top S A_d \end{aligned} \tag{28}$$

These equations ensure that the feedback gain K optimizes the control actions based on the dynamics of the system encapsulated by matrices A and B.

The choice of Q and R matrices profoundly influences the controller's behavior. By adjusting Q, we prioritize the reduction of state deviations, leading to a more aggressive control action. Conversely, increasing R makes the system more conservative, reducing the magnitude of the control input to minimize effort and potential overshoot. These adjustments allow for fine-tuning the controller to meet specific performance criteria and system demands.

Then we can implement our theory into Simulink to establish our own block which we can see in Figure 10

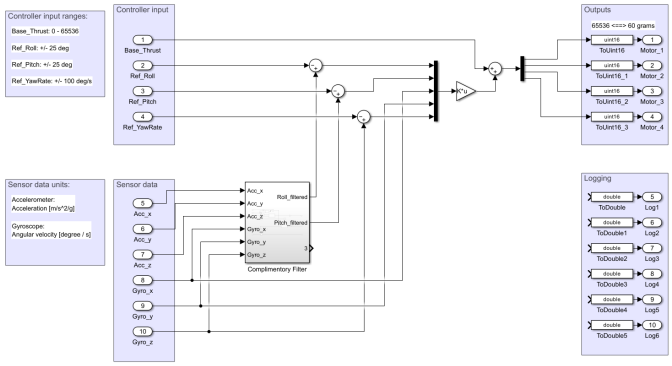

Fig. 10. Simulink block diagram

VII. Implement C-code Based on FreeRTOS

We introduce the FreeRTOS framework when we do the C-code implementation of an LQR controller for Crazyflie.

In this project, there are three tasks that need to be done which are given priority from high to low according to the following sequence:

- The complementary filter
- The LQR controller
- The reference task creator

Given that stability is the top priority, the filter precedes the controller because, regardless of the controller's quality, it will be unreliable if the measurement is flawed. The frequency is determined by the interval between each interruption, referred to as "ticks," with each tick in this case being 1 ms. Consequently, the operating system reads the sensors at a frequency of 1kHz.

Since the complementary filter has the highest priority, sensor readings will always take precedence over the processor. Therefore, the complementary filter thread must be put to sleep between updates to allow other threads a chance to update. Additionally, because both the complementary filter task and the controller task access the state space, reading and writing to the state space must be regulated by semaphores.

A. Complementary filter

The complementary filter is designed by reading values from the sensor calibration function, applying a filter, and writing the updated values to the global variable state. The filtering process has been implemented in the following way:

The process begins by waiting for the sensors to be calibrated. Once calibrated, it enters an infinite loop where it waits for new sensor readings to be ready. Upon receiving the readings, it copies the data to local variables and uses a complementary filter to estimate the orientation. This estimated orientation is saved to a local state space. The system then checks if the state space semaphore is available, and if so, it updates the global state space variable with the new data and releases the semaphore. Finally, the process sleeps until the next time step before repeating the loop.

Additionally, the function $vTaskDelayUntil$ is used to ensure the filter operates at 1kHz. The estimated state can be written to the global variables states when no other tasks are accessing the variable.

B. LQR controller

The controller function reads the sensor data, the estimated state from the states, and the reference from the states then implements LQR and sends out the motor signal. Note that the roll and pitch yaw rates come directly from the sensor and need to be converted from angle values to radians before being added to the state. The LQR gain values for the LQR controller were introduced then. Since C lacks a straightforward implementation of matrix multiplication, the multiplication was performed manually when porting the algorithm to C. The function $vTaskDelayUntil$ ensures that the controller operates at 100Hz, matching the system's frequency.

Due to the possibility of threads operating at different frequencies, semaphores in this thread are used to ensure that the global state space and reference variables are not updated while the control thread is reading them, preventing inconsistencies in the values being read.

C. Reference task creator

The reference generator simply assigns a value to the global variable $setpoint$, which serves as the reference for the controller to track.

D. FreeRTOS setting for global variables

The different tasks share the same global variables in the program, meaning they could attempt to access these variables simultaneously. To prevent task interference and ensure data integrity, semaphores are used to protect the global variable data transfers. Semaphores for each global variable are created using '**xSemaphoreCreateBinary**' in the main function. Within the tasks, '**xSemaphoreTake**' is executed before accessing the global variables to ensure no other tasks are operating on them. After the transmission, '**xSemaphoreGive**' is executed to release the semaphore for the corresponding variable. This semaphore system ensures that a complete read or write operation is finished before another task can access the variable.

VIII. Evaluation of Control Design

A. Criteria we use to evaluate the controller

Assuming correct implementation, both approaches in our case should produce identical results. We evaluate our controller in two different ways:

- The simulation result
- The real drone work performance

B. Generate code from Simulink

The controller was subsequently tested using the model. Figure 11 illustrates the system's ability to resist interference. A disturbance was introduced, and it was evident that the system promptly adjusted to mitigate its effects.

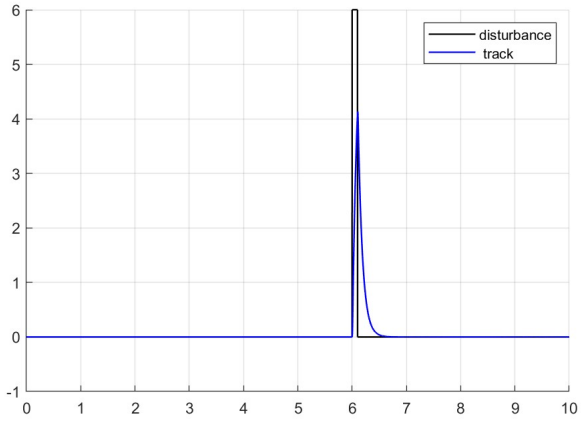

Fig. 11. Disturbance test of the controller

In the final part of the experiment, we fine-tuned the controller by changing values in the Q and R matrices. Specifically, we focused on adjusting the roll and pitch parts in the Q matrix. When the R matrix is small, the system responds slowly. Conversely, if the Q matrix is too big, the system overshoots. To get the best performance, we found results in Figure 12, showing the most favorable tuning.

Utilizing the implemented controller from Simulink-generated code to stabilize the quadrotor yielded satisfactory outcomes.

C. Write C-code in FreeRTOS frame

Following almost the same process in Simulink, we found that using the matrix described in Equation 29 is the way that

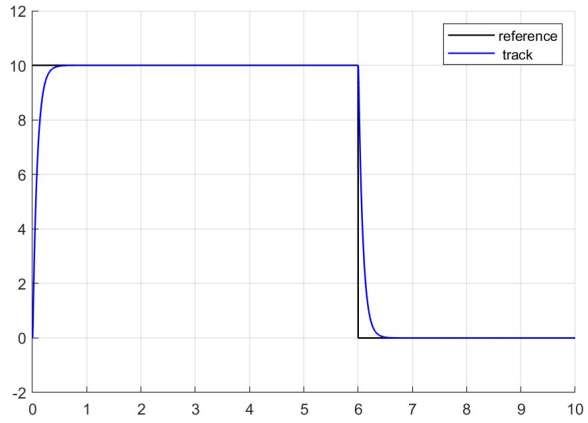

Fig. 12. Track following test of the controller

leads to the best performance of the simulation results shown in Figure 13.

$$K = \begin{bmatrix} -358.26 & -520.97 & -69.62 & -100.91 & -0.40 \\ -358.26 & 520.97 & -69.62 & 100.91 & 0.40 \\ 358.26 & 520.97 & 69.62 & 100.91 & -0.40 \\ 358.26 & -520.97 & 69.62 & -100.91 & 0.40 \end{bmatrix} \tag{29}$$

Utilizing the implemented controller in C-code to stabilize

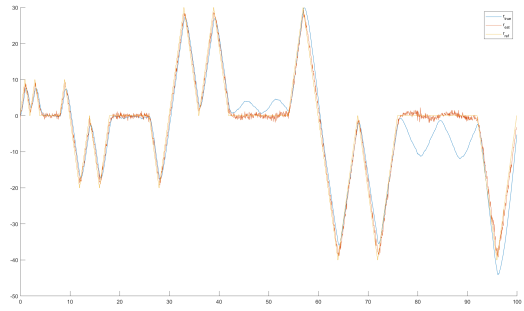

Fig. 13. Simulation test of C-code

the quadrotor yielded satisfactory outcomes.

D. Compare real implementation between them

The differences highlighted between the simulation and measurements from the physical system serve as a reminder that simulations never perfectly represent real-world systems. The controller based on the Simulink model, with generated C-code, appears to behave somewhat differently from our C-implemented version of the controller.

One possible aspect of the Simulink version is that the LQR gain doesn't work effectively to facilitate set point tracking during simulation.

On the other hand, the C-implemented controller demonstrates remarkable stability during flight and efficiently accepts set point tracking in roll, pitch, yaw rate, and thrust. Consequently, the more successful implementation appears to be the C-controller with better LQR gain. However, it's important to acknowledge that this version couldn't have been achieved without initially modeling, simulating, and computing crucial parameters such as the LQR gain matrix in Matlab and Simulink. While these tools are well-suited for developing the control system, the final implementation in C must be executed efficiently.

IX. Conclusions

In this project, we have successfully developed and implemented a sophisticated control system for the Quadcopter Crayfile. Utilizing the Crazyfile 2.0 platform, our team integrated advanced LQR control strategies that enhanced stability and performance, demonstrating the practical application of control theory in real-world scenarios.

A. Challenges in this project

Through our modeling, simulation, and control implementation, several challenges arose, particularly with the alignment between theoretical models and real-world behavior.

1) The role of Transform matrix: Throughout this project, it became evident that directly differentiating or integrating angles represented in Euler angles does not hold physical significance. Therefore, a transform matrix was utilized to compute angular velocity from sensor data in the state space, as well as to handle the integration of these values in the integral action section.

2) Practical Challenges and Simulation Limitations in Quadcopter Testing: Other challenges included the uneven weight distribution of the Quadcopter Crazyflie and the way the force from the fishing string was applied. Additionally, disturbances such as the drone hitting the table during takeoff in physical flight tests added further complexity to our control challenges.

We also encountered numerous issues during the control simulation phase. When attempting to use data from actual flights, we found that the control components seemed ineffective. We traced this ineffectiveness to the version of our modeling tool, MATLAB Simulink. This experience underscores the importance of maintaining a unified environment for consistent and reliable modeling.

3) Integrating and Tuning LQR Controllers: Another challenge we encountered was in designing the complementary filter and LQR controller using C code within the FreeRTOS framework. By testing several parameters, including the gamma value and the K matrix, leveraging parameters generated from the Simulink model, as well as repeatedly converting between the time domain and frequency domain, we ultimately achieved favorable results. This process not only enhanced the system's responsiveness and stability but also highlighted the value of extensive experimentation and simulation.

However, by meticulously modeling and simulating each component of the Quadcopter Crazyflie, we gained a deep

understanding of its operation, which makes it easier to modify or enhance the system. This detailed approach has proven immensely beneficial for our ongoing development efforts.

B. Further improvements

Enhancing the LQR controller with a Kalman filter results in the creation of a Linear Quadratic Gaussian (LQG) controller. The addition of a Kalman filter could improve the system by providing robust state estimation capabilities. This is crucial, as the Kalman filter excels in filtering out noise from the system measurements, thereby ensuring more accurate and reliable state estimates. As a result, the LQG controller can adapt more effectively to real-world conditions by continually updating its state estimates based on incoming data.

Therefore, with such improvements, we expect to achieve better results.

REFERENCES

[1] SSY191 Course Team, "SSY191 Model-based development of cyber-physical systems," From the SSY191_Gettingstarted, Chalmers University of Technology, 2023.

[2] T. Glad and L. Ljung, *Control Theory - Multivariable and Nonlinear Methods*, Taylor & Francis, ISBN: 978-0-748-40878-8.