Software development of a Fly-By-Wire controller for Fixed-Wing UAVs

Universitat Politècnica de Catalunya

ESEIAAT

Mathematics and Computer Science

Pau Climent Salazar

August 8, 2025

Contents

List of Figures

Introduction

1.1 Objectives

The objective of this work has been to implement a piece of software to run on a small fixed-wing UAV that can perform fly-by wire control of the attitude of the plane with the aim of making the flying experience easier while providing a smoother flight and rejecting any external disturbances to the aircraft such as wind gusts.

This is a very complex project which will benefit from an already existing UAV platform which has been developed for another subject, mainly being centred on the hardware aspect of the drone platform. The code that ends up running on the flight computer, will be in charge of receiving the commands from the transmitter, interpreting them, and combining them with the estimation of the UAV's attitude in order to output a control command that, when executed by the UAV's actuators, will stabilise the airplane while achieving the desired orientation.

The flight computer which has already been developed is based on an Arduino UNO R4 minima development board and uses a peripheral board which contains the necessary sensors, an indicator LED, a module for interfacing with an SD card, and the interfaces with the radio receiver and the plane's actuators. This board is also in charge of distributing the power correctly to the peripheral electronics. The Arduino board will be able to run code using the Arduino libraries, which will make it possible to more easily and robustly interact with the input/output pins, manage the various internal clocks, and effortlessly incorporate other libraries that deal with the communications protocols used by the peripherals.

1.2 Requirements

The first step to take before any major decisions are made it to define the requirements that the final software will have to fulfil. The following list has been arrived at:

- The developed software shall run in real-time on the pre-existing arduino-based flight computer.
- The code must be robust in the face of possible hardware or communications failures, as well as any software bugs which appear during flight.
- The code must be as computationally efficient as possible (while continuing to prioritise robustness), as faster compute frequencies will lead to better attitude estimation and control results.
- The code shall be developed using the C/C++ programming language together with the Arduino software libraries.
- All of the subroutines used for performing algebraic operations, attitude estimation, and attitude control shall be developed from scratch.
- All of the subroutines used for communications with the peripherals will be implemented with the help of external libraries, as long as they are extensively tested, reliable and efficient.
- The flight software will have to stabilise the airplane at all times while achieving the attitude that is set by the user (except in its heading), resembling a Fly-by-wire controller.
- The flight software shall be able to store the relevant flight data in an external memory such that it can be later analysed.

Some additional observations and soft requirements have also been set in order to reduce the space of possibilities in the inital development stages. These include:

- Functional programming will be favoured before Object Oriented Programming, as none of the mathematical *objects* (with the most complex being matrices and quaternions) and their required manipulations in the scope of this project have enough complexity for classes to be required.
- Doubles shall be used for handling decimal values as the accuracy that they provide is much larger when compared with the accuracy of the sensor readings (subject to large amounts of noise).
- Type-agnostic programming will be favoured in order to make the re-usability of the developed code as large as possible for future projects. C++ templates will be favoured before C macros due to their ability to provide type safety, better error checking, and support for complex type manipulations.

1.3 Flight computer hardware overview

As it has been previously stated, the flight computer is based on an Arduino UNO R4 Minima. This board is equipped with a 32-bit Arm Cortex-M4 processor, providing a clock speed of up to 48 MHz. It has a flash memory capacity of 256 KB and 32 KB of SRAM. Additionally, it includes 2 KB of EEPROM for non-volatile data storage (information extracted from Arduino [2024]). This is not a very large amount of memory, hence why memory safety should be a major concern when programming the board. The Arduino supports multiple communication interfaces such as UART, I2C, and SPI (all of which will be used), and offers 14 digital input/output pins, 6 of which can be used as PWM outputs, necessary for interfacing with the plane's actuators.

The flight computer integrates several peripherals, which are core to its functionality. These peripherals include an MPU6050 module, which is a combined 3-axis gyroscope and accelerometer for attitude estimation. An SD card adapter board is used for data logging and storage purposes. Furthermore, the flight computer contains several connectors which are used to interface with the servos (the actuators that move the plane's control surfaces) and the Electronic Speed Controller or ESCs (the motor controller). In order to receive the user's signals, it uses the Fr.Sky NANO V2 remote control receiver, which operates on 2.4Ghz and communicates with the Ardunio using a variant of UART communication called SBUS (in this case, inverted SBUS). Last but not least, the flight computer includes an LED indicator, which can be used to signal whether any error has been detected in the code.

To provide a clearer understanding of the flight computer's layout and connections, a high-level overview diagram is shown in figure 1.3.1, illustrating the interconnections between the Arduino and the various peripherals, as well as the interaction between the said peripherals and the real world.

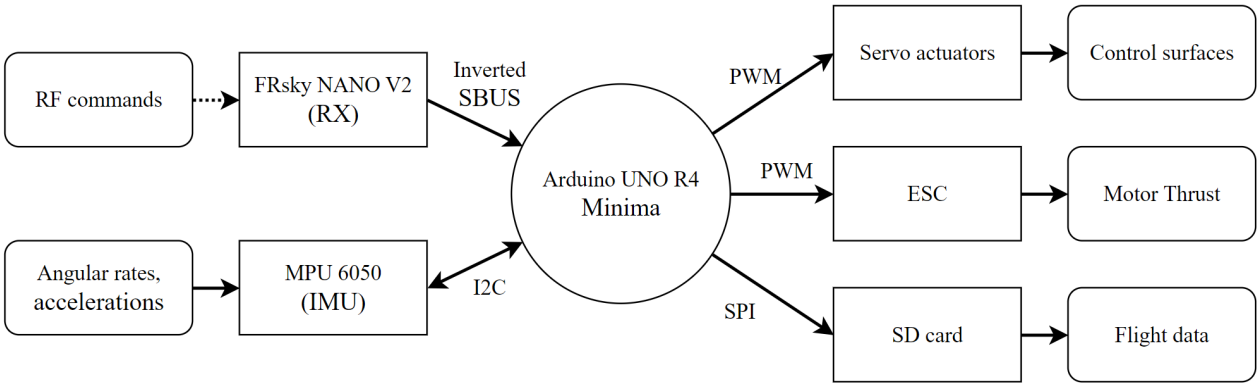

Figure 1.3.1: Top-level diagram of the flight computer, its peripherals, and the communications protocol that each one uses.

An image of the real flight computer can be seen in figure 1.3.2 which includes labels for every major component that it integrates.

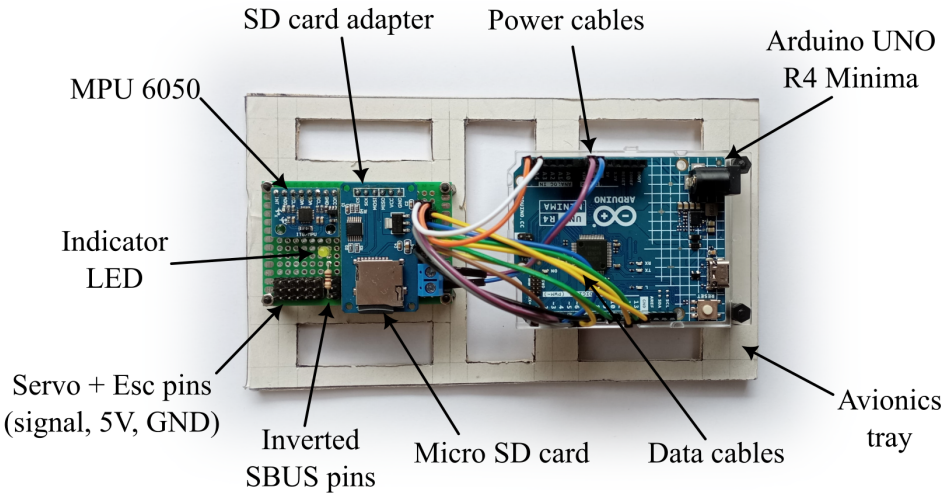

Figure 1.3.2: Image of the flight controller board mounted on the avionics tray, with a provisional wire harness.

1.4 UAV system description

The last thing to take into account before working on the flight control software is how the UAV is going to be controlled. More specifically, the system's kinematics and dynamics must be studied to define the control mechanisms.

The UAV is a fixed-wing platform, which means that it requires forward velocity relative to the air in order to generate a lifting force that offsets its weight. When doing so a drag force will oppose the vehicle's velocity in a manner that is proportional to the square of the airspeed. This drag force is countered by the UAV's powerplant, which is able to generate a thrust force. The amount of thrust that is generated depends on many variables but, most importantly, it depends on the airspeed and the power that is being supplied by the motor to the propeller, which can be controlled with the ESC. In cruise flight conditions, all of the forces cancel out and the aircraft sustains flight in a fixed plane at constant speed.

The problem with achieving cruise flight, especially in small airborne vehicles, is that external disturbances can cause the aircraft to deviate from its nominal cruise flight conditions. For example, a gust of cross-wind can make the aircraft change its attitude in a way that is not desired. The goal of the flight controller will be to maintain the aircraft's attitude to the desired values. This must be accomplished by exerting external torques on the aircraft to stabilise and control the attitude. The most common and effective way of exerting such torques is by moving the control surfaces of the plane, which are small movable parts of the larger aerodynamic surfaces of the plane which modify the magnitude of the forces and torques that they generate.

The UAV that is being dealt with uses a conventional aerodynamic and control surface configuration, which can be seen in figure 1.4.1. These consist of:

- Ailerons: used to generate rolling moments.
- Elevator: used to generate pitching moments.
- Rudder: used to generate yawing moments for coordinating turns.

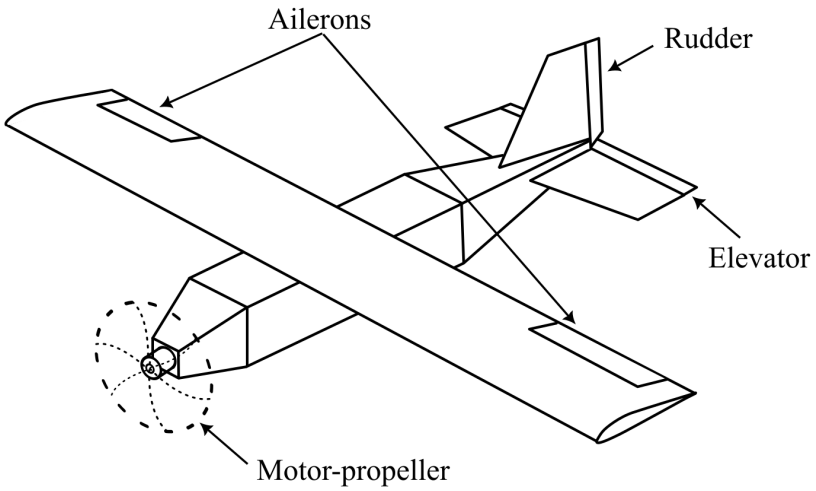

Figure 1.4.1: Illustration of the UAV used in this work, highlighting its control surfaces and power-plant.

A more rigorous control systems design process would involve the modelling of the aircraft's dynamics according to the various actuations that are possible, in order to apply classical control theory for designing a suitable control law. This, however, is not the focus of this work. Instead, a control law that has already been extensively studied in the literature will be selected, and its control values will be tuned heuristically by trial and error. It is still important to describe how the attitude of the aircraft is defined.

In order to do so, a non-rotating or space-fixed reference system must be described, together with a rotating or body-fixed reference system.

The body-fixed refrence system can be defined as having its origin on the centre of mass of the plane, its X-axis pointing towards the nose, its Y-axis pointing towards the right-wing while being perpendicular to the symmetry plane of the UAV, and its Z-axis pointing down and completing a right-handed orthonormal basis.

The space-fixed reference system can be defined as also having its origin on the centre of mass of the plane, but its Z-axis points downwards to the centre of the earth (which means that it will only be pseudo-non-rotating), and its X and Y-axis vectors will be normal to each other and to the Z-axis completing a right-handed basis. In order to fully define the space-fixed system, one last degree of freedom must be removed from the given definition. One such option for doing so is making the X-axis point towards the geographic north pole. This option would work well for controlling the heading of the aircraft, but the heading of the aircraft - according to the requirements - should only be stabilised but not controlled. This means that the X-axis of the space-fixed frame can freely rotate such that its unit vector coincides with the projection of the body-fixed X-axis vector onto the space-fixed X-Y plane. In practise, this means that the attitude of the aircraft will only be given in its pitch and roll angles, and its yaw angle will not be accounted for.

The definition of both reference systems can be visualised in the following figure, where b and s denote the body-fixed and space-fixed systems respectively.

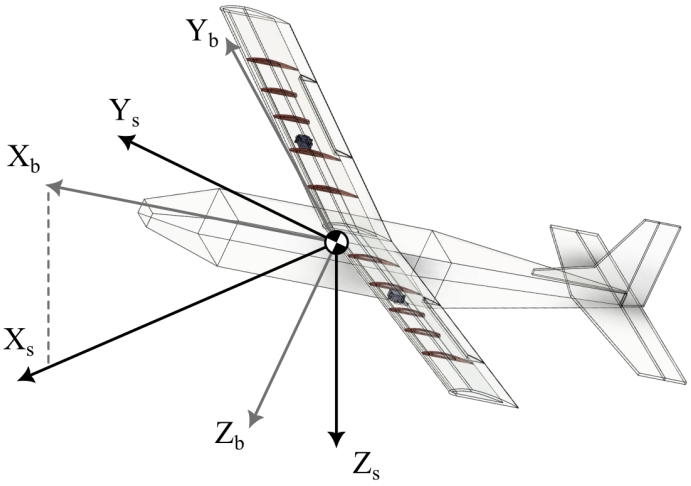

Figure 1.4.2: Illustration of the space-fixed and body-fixed reference systems.

Additionally, while one may have an intuition for what pitch, roll and yaw correspond to, they must be systematically defined. For this work, the Tait-Bryan set of Euler angles are used, in which the attitude transformation is performed by rotating first about the z-axis, then about the y-axis and finally about the x-axis, whose rotation angles correspond to Yaw, Pitch and Roll (ψ, θ and ϕ) respectively.

$$R_{\mathrm{x}}(\phi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos\phi & -sin\phi \\ 0 & sin\phi & cos\phi \end{bmatrix} \qquad R_{\mathrm{y}}(\theta) = \begin{bmatrix} cos\theta & 0 & sin\theta \\ 0 & 1 & 0 \\ -sin\theta & 0 & cos\theta \end{bmatrix} \qquad R_{\mathrm{z}}(\psi) = \begin{bmatrix} cos\psi & -sin\psi & 0 \\ sin\psi & cos\psi & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{1.1}$$

The rotation sequence can be reformulated into a single rotation matrix that encompasses the sequence of the three rotations outlined in equation (1.1). The result can be seen in the following equation which translates points expressed in the body frame to points expressed in the fixed frame.

$$R_{b2s} = R_{\mathrm{z}}(\psi)\, R_{\mathrm{y}}(\theta)\, R_{\mathrm{x}}(\phi) = \begin{bmatrix} cos\theta\, cos\psi & sin\phi\, sin\theta\, cos\psi - cos\phi\, sin\psi & cos\phi\, sin\theta\, cos\psi + sin\phi\, sin\psi \\ cos\theta\, sin\psi & sin\phi\, sin\theta\, sin\psi + cos\phi\, cos\psi & cos\phi\, sin\theta\, sin\psi - sin\phi\, cos\psi \\ -sin\theta & sin\phi\, cos\theta & cos\phi\, cos\theta \end{bmatrix} \tag{1.2}$$

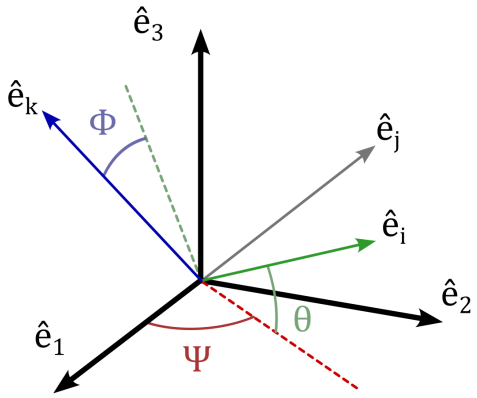

Figure 1.4.3: Illustration of the Tait-Bryan rotation sequence.

Theoretical Bacgkround

2.1 General approach and system architecture

The system that has been outlined in the introduction can sense data from its environment and apply control actions to the aircraft. This presents the right ingredients for closing the feedback loop once the state dynamics (or the physics that make the aircraft react in a particular way) are accounted for, and once a way to estimate the state from the sensor readings is incorporated (in this case a Kalman Filter or KF). If the estimated state is compared with a reference state sent through radio commands to a receiver (denoted as RX), a control law can be applied to correct any deviance in the state. This already serves to categorize the required software architecture as a real-time control system, where an additional data collection subroutine will be added to record the system's data Zalewski [2001] for post-flight analysis. The control-system diagram of the UAV is illustrated in figure 4.0.1, which is analogous in essence to the diagram shown in figure 1.3.1 once the plant dynamics and state estimation are used to close the control loop.

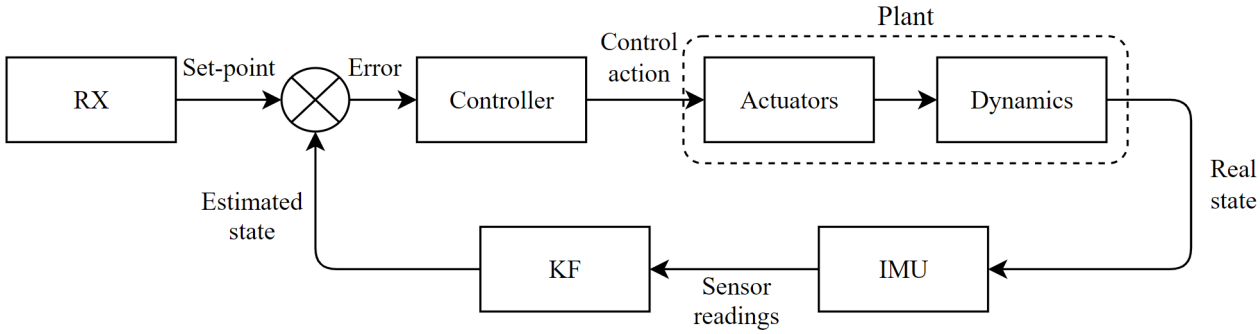

Figure 2.1.1: Illustration of the closed loop attitude controller architecture.

Some in-depth details about the implementation of the entire software should be explained before delving deep into the theory needed and the code itself.

2.1.1 Set-point processing and flight modes

The set-point of the system will be given by the user through a remote controller. The controller that will be used is an FrSky Taranis QX-7, which has two joysticks and additional switches. The control map of this controller can be seen in figure 2.1.2 as given by the manufacturer.

The controller's joysticks will be used for setting the target attitudes or for imposing a direct, manual control of the control surfaces. It is important to be able to switch between the two as the initial tests shall consist of manual flight up to a safe altitude, and then a transition to Fly-By-Wire mode such that the control loop can be tested safely.

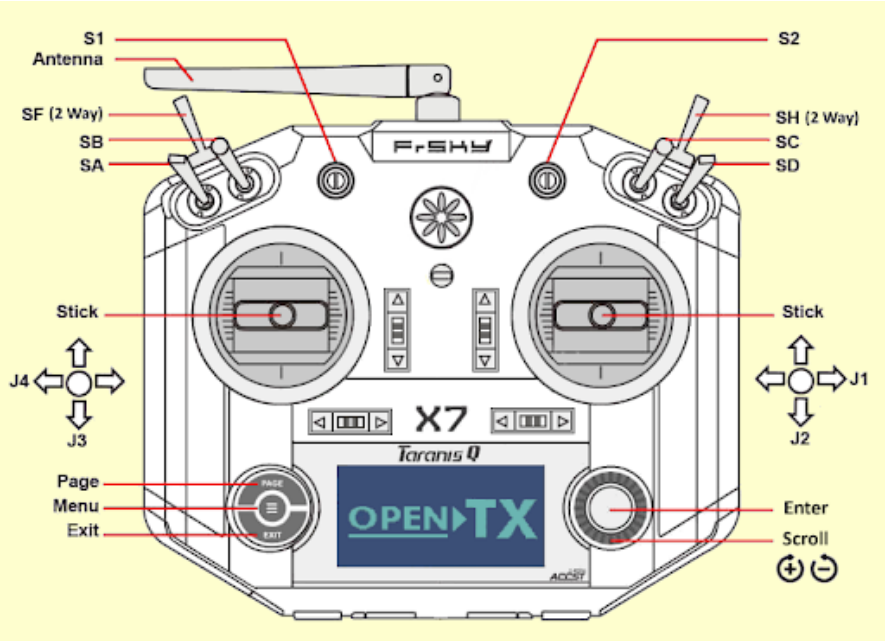

Figure 2.1.2: Control map of the QX-7 remote control.

The set-point, in this case, is equivalent to the desired attitude that the aircraft should have. When the Fly-By-Wire control loop is active, the desired attitude will be processed as follows:

- The general attitude will be given as target roll and pitch angles, which will be changed according to the **J1** and **J2** joystick commands respectively. If the joystick remains at zero, the target angles will remain unchanged, but if the joystick is moved, its position will be mapped to an "angular rate" equivalent, and the target angle will be incremented over time until the joystick is back to zero, where the final target angles will stay constant until further input is given. This can be thought of, in essence as an integral action as defined in equations 2.1 and 2.2, where the subscript t denotes that these are the target angles, J_0 is the middle value of the SBUS command (as it is not centered around 0 by default), and k_J are the scaling constants that will be used to give a faster or slower target angle change rate.

$$\theta_t = \int (J2(t) - J_0) \cdot k_{J2} \; dt \tag{2.1}$$

$$\phi_t = \int (J1(t) - J_0) \cdot k_{J1} \; dt \tag{2.2}$$

- The throttle of the plane will be manually set by the user at all times through **J3**, except in the event of a loss in signal, when the throttle will be automatically set to 0 for safety reasons.
- The yaw rate control will be given by **J4**, which will set a target angular rate, to be achieved by the derivative part of the control law will try to achieve.
- The **SF** switch will be used to change flight modes: it can be set to Fly-By-Wire, or to bypass mode, where the user can take over full manual control of the aircraft.
- The **SH** toggle switch will be used for resetting the target angles back to zero (for cruise flight), as well as resetting the integral term of the controller.

2.1.2 Attitude estimation and control approach

As it has been already stated, the Kalman filter algorithm for state estimation will be used. The state that will be estimated consists of the pitch and roll Euler angles, where the yaw is not required

as the heading is not being controlled (only stabilised through yaw-rate control).

Despite estimating the Euler angles, the control law will use quaternions instead of Euler angles. This begs the question: why not use the same attitude representation scheme for both the estimation and control?

One of the main goals, personally, when setting out to complete this project was to try to adapt quaternion control laws that had been previously used for rotary-wing UAVs to a fixed-wing UAV, so the control logic should use quaternions as a base. However, performing attitude estimation with quaternions is a much more complex task than estimating it with Euler angles: one of the best quaternion estimation algorithms thus far is the Optimal-REQUEST algorithm, which applies discrete Kalman Filtering techniques (which will also be done in the final approach chosen) to Davenport's q-method for solving Wahba's problem (the problem of finding the attitude quaternion that matches at least two vector readings such as acceleration and magnetic field), as described in Choukroun et al. [2004]. It is the solution to Wahba's problem that adds such a major complexity to the quaternion estimation problem, as the solution to an eigenvalue problem that requires an iterative method must be included. This results in more accurate attitude estimation at the cost of much larger complexity and computational time (Navarro Trastoy [2019]). For this reason, applying the Kalman Filter to an estimation of only the pitch and roll Euler angles becomes a much lighter task, computationally, which still provides good results.

Once the Euler angles have been estimated, these can be efficiently converted to their respective quaternion, such that the quaternion control law can be applied. While not a particularly elegant solution, this has been found to work better than initially expected.

The next sections focus on explaining the theory behind both the Kalman Filter and its implementation for this specific application, and quaternions when used for attitude representation and control.

2.2 Attitude estimation with Kalman Filter

The Kalman Filter algorithm has been selected for the attitude estimation process as it is able to combine data from multiple sensors, by having a prediction model and correcting said prediction first by comparing it to an observed state. To do so, this algorithm models the noise in the sensor measurements and in the propagation model as consisting of random numbers sampled from a Gaussian distribution. If the sensors' noise is properly characterised, this filter can come up with an optimal weight by which to trust each sensor more or less, selectively choosing the best data at each moment. This approach has been guaranteed to obtain estimation results that converge towards the real state of the system, if the stated hypotheses are valid.

The model whose state is to be estimated can be written in discrete time according to the following equations, where x_k corresponds to the system's state at time instant t_k, and y_k corresponds to the measured state.

$$\begin{cases} x_k = Ax_{k-1} + Bu_{k-1} + w_{k-1} \\ y_k = Hx_k + v_k \end{cases} \tag{2.3}$$

Regarding the noise terms in equation 2.3, w_k and v_k correspond to the white noise of zero mean value with given variances Q and R respectively (which are assumed to be constant).

The solution to the state estimation of such a discrete system which minimises the estimated state error over time yields the two-step solution, consisting of a prediction step followed by a correction step. In the prediction step, both the system's state and the covariance matrix are propagated forwards in time according to the knowledge of the plant. This takes place in equations 2.4 and 2.5

respectively.

$$\hat{x}_k^- = A\hat{x}_{k-1} + Bu_{k-1} \tag{2.4}$$
$$P_k^- = AP_{k-1}A^T + Q \tag{2.5}$$

The predicted covariance and state variables correspond to P_k^- and $\hat{x}_k^-$ respectively. The predictions must then be corrected given the knowledge of the sensors' behaviour. To do so, the Kalman gain must first be computed.

$$K_k = P_k^- H^T \left(HP_k^- H^T + R\right) \tag{2.6}$$

The Kalman gain can be applied to the residue of the output equation in 2.3. Adding this term to the predicted state yields the corrected state $\hat{x}_k$. The error covariance matrix can also be corrected given the newly found Kalman gain, yielding P_k.

$$\hat{x}_k = \hat{x}_k^- + K_k \left(z_k - H\hat{x}_k^-\right)^{-1} \tag{2.7}$$
$$P_k = \left(I - K_k H\right) H_k \tag{2.8}$$

Complete algorithm

The resulting algorithm is a recursive one, which means that it only needs the immediately previous values to perform the state estimation, avoiding the need for large memory buffers. The steps and structure of the Kalman Filter algorithm can be summarized in this diagram.

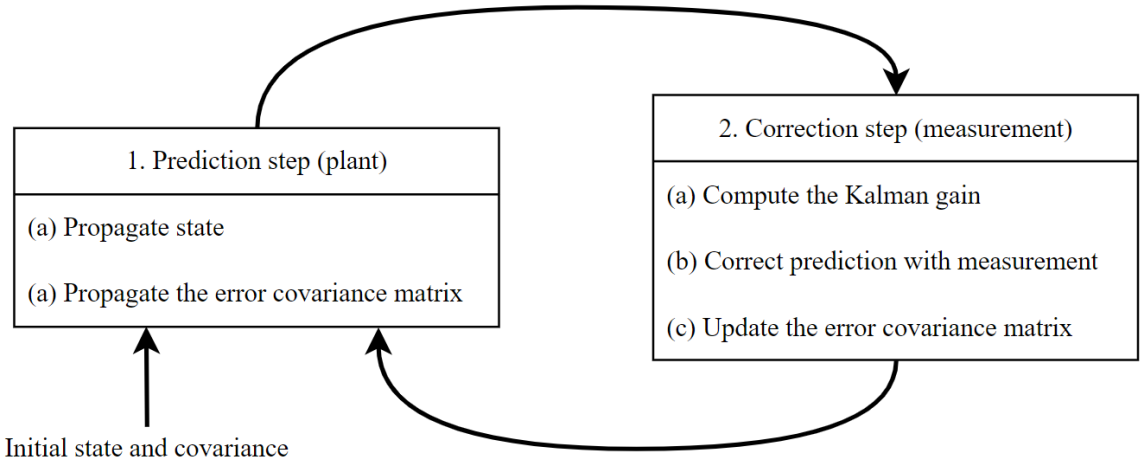

Figure 2.2.1: Diagram of the Kalman Filter algorithm.

2.2.1 Sensor description

The sensor suite that is incorporated in the Flight Computer consists of the MPU 6050 IMU. As it has already been stated, this package contains rate gyroscopes and linear accelerometers, one for each axis (as well as a temperature sensor which will be unused).

There are certain problems associated with each of the sensor's working principles, which must be taken into account when performing state estimation. These are mainly sensor bias, wrong or variable scaling factors, non-orthogonality errors or sensor misalignment, and random measurement noise. These possible errors are dealt with by embedding the knowledge of their existence in the models used for the Kalman Filter design.

Rate gyroscopes

Rate gyroscopes are sensors that measure angular rates about one axis. Each of the three gyroscopes included in the MPU605 will be aligned with each of the body axes. The angular body rates can be used for estimating the Euler angles' time derivatives, a process that can be simplified through linearisation (as detailed in section 2.2.2). Then, the Euler angles can be propagated by the gyroscope through numerical time integration of the angular rates.

The process of numerical integration, however, means that all of the noise and biases are accumulated in the integration process, generating an error that grows larger over time. This phenomenon is known as *gyro drift*. This issue is accentuated by the fact that the MPU6050 can be considered a low quality MEMS (Micro Electro-Mechanical Sensor), as it is based on the coriolis effect (where much more accurate gyroscopes based on other physical phenomena are offered as alternatives).

The equation for integrating the angular rate in discrete time can be applied to each Euler angle, which is going to be generalized in equation 2.9 as angle α, and where the gyroscope's bias $\dot{\alpha}_{bk}$ is included as part of the integration.

$$\alpha_k = \alpha_{k-1} + T_s \ (\dot{\alpha}_k + \dot{\alpha}_{bk}) \tag{2.9}$$

Regarding the rest of the terms, T_s is the sampling time, and $\dot{\alpha}_k$ is the measured angular rate at the present time k.

Linear accelerometers

Linear accelerometers measure the linear accelerations to which a compliant spring-mass system is subjected, again based on MEMS technology. Once again, the linear acceleration is measured in each of the body axes. When the sensor unit is static, the acceleration vector can be assumed to point in the same direction as the gravity vector. The biggest issue with this assumption happens when any external accelerations, including vibrations, are present, as they will vastly affect the acceleration vector's direction, introducing major disturbances in the readings. The main advantage of accelerometers for attitude estimation, however, is that they provide an absolute measurement, immune to drift.

The measurements of the accelerometers can be converted to the roll and pitch euler angles or ϕ and θ respectively with equations 2.11 and 2.10, where the acceleration vector is $\mathbf{A_b} = \{A_{bx}, A_{by}, A_{bz}\}$.

$$\phi = arctan\left(\frac{A_{by}}{A_{bz}}\right) \tag{2.10}$$

$$\theta = arctan\left(\frac{-A_{bx}}{\sqrt{A_{by}^2 + A_{bz}^2}}\right) \tag{2.11}$$

2.2.2 Observer model

The matrix that relates the body angular rates (angular rate of the B-frame with respect to the E-frame, expressed in the B-frame), denoted as $\omega = [p, q, r]^T$ can also be related with the derivatives of the Euler angles denoted as $\eta = [\dot{\phi}, \dot{\theta}, \dot{\psi}]^T$ by the equation 2.12.

$$\omega = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos\,\phi & sin\,\phi\ cos\,\theta \\ 0 & -sin\,\phi & cos\,\phi\ cos\,\theta \end{bmatrix} \dot{\eta} \tag{2.12}$$

Since the UAV will be operating around the cruise flight set-point which consists of level flight, that is with low values of ϕ and θ, the above expression can be approximated with a Taylor's series, and

the matrix relating the two terms becomes the identity matrix, yielding the relationship shown in equation 2.13.

$$\omega \approx \dot{\eta} \tag{2.13}$$

The observer model can then be decoupled from the kinematics, and it will transform directly into the gyroscope sensor model. It should be added that the small angles approximation can be escaped by using an Extended Kalman Filter, but the estimation results encountered have been more than sufficiently accurate even with very large angles.

A state space model is devised with the goal of estimating the integral of the angular rate in the following time step, α_{k+1}, as well as the bias angular rate $\dot{\alpha}_{b\,k+1}$. The generic observer model for the gyroscope with the measured state is the following. This is what was meant by encoding information about the sensor error in the model, as the Kalman filter will estimate not only the body's attitude, but also the bias in the sensors.

$$\begin{cases} \begin{bmatrix} \alpha_{k+1} \\ \dot{\alpha}_{b\,k+1} \end{bmatrix} = \begin{bmatrix} 1 & -T_s \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \alpha_k \\ \dot{\alpha}_{b\,k} \end{bmatrix} + \begin{bmatrix} T_s \\ 0 \end{bmatrix} \dot{\alpha}_k \\ y_k = \alpha_{\text{measured }k} = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} \alpha_k \\ \dot{\alpha}_{b\,k} \end{bmatrix} \end{cases} \tag{2.14}$$

By examining the equations in system of equations 2.14, the necessary A, B and C system matrices can be obtained for the Kalman Filter equations. The model that has been obtained can be used along equations from 2.4 to 2.8 for performing the complete attitude estimation, when applied to both Euler angles.

$$A = \begin{bmatrix} 1 & -T_s \\ 0 & 1 \end{bmatrix} \tag{2.15}$$

$$B = \begin{bmatrix} T_s \\ 0 \end{bmatrix} \tag{2.16}$$

$$H = \begin{bmatrix} 1 & 0 \end{bmatrix} \tag{2.17}$$

2.3 Use of quaternions for attitude control

Any transformation that comes from any rotation sequence has one major problem: Gimbal Lock, which consists of mathematical singularities rendering one rotation axis ineffective. When dealing with the Tait-Bryan zyx rotation sequence, this problem occurs when the pitch angle is rotated by $\pm$ 90º, as the axis of rotation in the last rotation becomes aligned with that of the first one losing one degree of freedom, as is illustrated in figure 2.3.1.

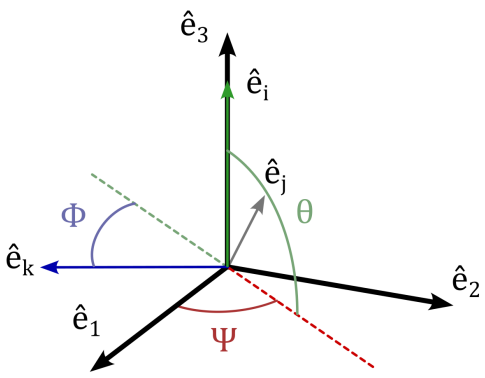

Figure 2.3.1: Illustration of Gimbal Lock.

This issue has propelled engineers to search for new ways of representing orientations in space, from which quaternions have resulted in one of the most used options. Consequently, several quaternion-based control laws have emerged over the past decades, especially having been used for the control of satellites' attitude, and more recently, applied to quadrotors. Euler angle based control laws could also be used, quaternions will be used for the control portion of the software with the aim to test the control laws typically used for quadrotor UAVs when applied to fixed-wing ones.

2.3.1 Quaternion algebra

Quaternions are, in essence, an extension of complex number into a total of four dimensions, which can be thought of a real one and three different imaginary ones. The Hamiltonian convention of quaternions is used in this work. This convention was first adopted at the time of its invention and places the real component as the first value of the quaternion. As a concequence, the identity $ij = k$ (in reference to equation (2.19)) is obtained.

A quaternion $\mathbf{q}$ which follows the Hamiltonian convention has the following definition, where $a, b, c, d \in \mathbb{R}$, and $i, j, k \in \mathbb{I}$.

$$\mathbf{q} = a + bi + cj + dk \tag{2.18}$$

$$i^2 = j^2 = k^2 = ijk = -1 \tag{2.19}$$

Quaternions tend to be expressed as vectors of four components. It is also useful to divide quaternion vectors into their scalar q_w and imaginary or vectorial $\mathbf{q_v}$ subparts.

$$\mathbf{q} := \begin{bmatrix} q_w \\ \mathbf{q}_v \end{bmatrix} := \begin{bmatrix} q_w \\ q_x \\ q_y \\ q_z \end{bmatrix} \tag{2.20}$$

A single quaternion can encode a transformation between coordinate frames. The definition of a passive rotation is used for obtaining the attitude quaternions (as opposed to active rotations). This definition follows equation:

$$\mathbf{q}_b = \mathbf{q}_{attitude} \otimes \mathbf{q}_a \otimes \mathbf{q}_{attitude}^{-1} \tag{2.21}$$

Expression of attitude with quaternions

When expressing coordinates which are initially given in the body system of reference, with respect to the space-fixed system of reference, the following definition is applied.

$$\mathbf{q}_s = \mathbf{q}_{b2s} \otimes \mathbf{q}_b \otimes \mathbf{q}_{b2s}^{-1} \tag{2.22}$$

In this expression, the attitude quaternion corresponds to $\mathbf{q}_{b2s}$. The quaternion identities necessary for performing this operation are discussed ahead.

Quaternion product

The expression for quaternion products can be derived from the very definition set in equation (2.19). It will be denoted by the $\otimes$ symbol which, when developed, takes the following form.

$$\mathbf{q^a} \otimes \mathbf{q^b} = \begin{bmatrix} q_w^a q_w^b - \mathbf{q_v^a}^T \mathbf{q_v^b} \\ q_w^a \mathbf{q_v^b} + q_w^b \mathbf{q_v^a} + \mathbf{q_v^a} \times \mathbf{q_v^b} \end{bmatrix} = \begin{bmatrix} q_w^a q_w^b - q_x^a q_x^b - q_y^a q_y^b - q_z^a q_z^b \\ q_w^a q_x^b + q_x^a q_w^b + q_y^a q_z^b - q_z^a q_y^b \\ q_w^a q_y^b - q_x^a q_z^b + q_y^a q_w^b + q_z^a q_x^b \\ q_w^a q_z^b + q_x^a q_y^b - q_y^a q_x^b + q_z^a q_w^b \end{bmatrix} \tag{2.23}$$

The quaternion product is non-commutative and associative.

Quaternion norm

The norm of a quaternion has the following definition.

$$||\mathbf{q}|| = \sqrt{q_w^2 + q_x^2 + q_y^2 + q_z^2} \tag{2.24}$$

By definition, all attitude quaternions must be unitary, so attitude quaternions shall be normalised to ensure a correct functioning of the control laws.

Quaternion conjugate

The conjugate of a given quaternion can be found by negating its imaginary components. When it comes to rotation quaternions, obtaining its conjugate is equivalent to obtaining the quaternion that expresses the opposite rotation. As an example, this would mean that $\mathbf{q}_{b2f}^* = \mathbf{q}_{f2b}$, which allows for easier inversion of the rotation when compared to computing the inverse of a rotation matrix.

$$\mathbf{q}^* = \begin{bmatrix} q_w \\ -\mathbf{q}_v \end{bmatrix} \tag{2.25}$$

Quaternion inverse

The inverse of a quaternion can be found by obtaining its conjugate and dividing it by the square of its norm. This operation is necessary for performing attitude transformations.

$$\mathbf{q^{-1}} = \frac{\mathbf{q}^*}{||\mathbf{q}||^2} \tag{2.26}$$

In the case of dealing with unit quaternions, as are rotation quaternions, computing their inverse is equal to computing their conjugate.

Quaternion rotation composition

When dealing with multiple reference frames, quaternions expressing different concatenated rotations can be joined into a single quaternion by performing quaternion products. As an example, the quaternion relating the space-fixed frame with the orbital frame, $\mathbf{q_{s2o}}$, and the quaternion relating the orbital frame with the body frame, $\mathbf{q_{o2b}}$, can be grouped into a single quaternion relating the space-fixed with the body frame, $\mathbf{q_{s2b}}$.

$$\mathbf{q_{f2b}} = \mathbf{q_{s2o}} \otimes \mathbf{q_{o2b}} \tag{2.27}$$

This concept is essential for computing the error quaternion that will be used later in attitude control algorithms.

Transformation of Euler angles to quaternion

In order to convert Euler angles into a rotation quaternion, the same rotations can be applied in the same sequence using the equivalent rotation quaterions. Doing so yields the following expressions.

$$q_w = \cos\left(\frac{\phi}{2}\right)\cos\left(\frac{\theta}{2}\right)\cos\left(\frac{\psi}{2}\right) + \sin\left(\frac{\phi}{2}\right)\sin\left(\frac{\theta}{2}\right)\sin\left(\frac{\psi}{2}\right) \tag{2.28}$$

$$q_x = \sin\left(\frac{\phi}{2}\right)\cos\left(\frac{\theta}{2}\right)\cos\left(\frac{\psi}{2}\right) - \cos\left(\frac{\phi}{2}\right)\sin\left(\frac{\theta}{2}\right)\sin\left(\frac{\psi}{2}\right) \tag{2.29}$$

$$q_y = \cos\left(\frac{\phi}{2}\right)\sin\left(\frac{\theta}{2}\right)\cos\left(\frac{\psi}{2}\right) + \sin\left(\frac{\phi}{2}\right)\cos\left(\frac{\theta}{2}\right)\sin\left(\frac{\psi}{2}\right) \tag{2.30}$$

$$q_z = \cos\left(\frac{\phi}{2}\right)\cos\left(\frac{\theta}{2}\right)\sin\left(\frac{\psi}{2}\right) - \sin\left(\frac{\phi}{2}\right)\sin\left(\frac{\theta}{2}\right)\cos\left(\frac{\psi}{2}\right) \tag{2.31}$$

2.4 Quaternion control law

In order to generate a suitable control output, it is necessary to first compute an error which contains information on how close or how far appart the current attitude is from its setpoint. In linear control schemes, this can be found by a simple subtraction, as would be the case if using Euler angles. When dealing with quaternions, however, the error can be computed using quaternion algebra together with the property of rotation compositions. More specifically, the rotation sequence that relates the space-fixed and the target reference systems can be linked with the current attitude.

$$\mathbf{q}_{s2t} = \mathbf{q}_{s2b} \otimes \mathbf{q}_{b2t} \tag{2.32}$$

The error quaternnion, denoted by $\delta\mathbf{q}$ can be obtained as the quaternion that relates the target attitude with the body attitude.

$$\delta\mathbf{q} = \mathbf{q}_{b2t} = \mathbf{q}_{s2b}^{-1} \otimes \mathbf{q}_{s2t} \tag{2.33}$$

When the current attitude is exactly the desired one, the error quaternion will have a scalar δq_w part of 1 and a vectorial $\delta\mathbf{q}_v$ part consisting of zeros. Instead, when there is a deviation between the current and desired attitudes, the scalar part will decrease its magnitude, and the vectorial part will contain information about the angle by which the two reference systems are deviated (by a factor of $\sin\varphi/2$, with φ being the total error angle), as well as the rotation axis by which the body must rotate to become perfectly aligned (expressed in body coordinates). In fact, if an angular acceleration were applied exclusively to this axis, the desired attitude would be reached. This contains invaluable information about how to command various torques to the actuators.

An additional aspect that must be taken into account by the control algorithm to correctly converge to the desired attitude is the angular rate error $\delta\omega$ of the UAV, as it is necessary for the its angular rate ω to match the desired ω_{ref}, which will be zero in all axes except for the Z axis, that corresponds with the yawing motion (when dealing with small angles).

$$\delta\omega(t) = \omega_{\text{ref}}^{b}(t) - \omega(t) \tag{2.34}$$

The final control law will make use of both the vector part of the quaternion error as well as the angular rate error, conforming what is commonly referred to in the literature as a nonlinear quaternion PD controller. The outputs of this equation will be directly mapped with the torque that should be provided.

By integrating both the vectorial part of the quaternion error, and the angular rate error, a non-linear PD controller as originally proposed by Fresk and Nikolakopoulos [2013] is obtained, where $\Gamma_{PD}(t)$ is the resulting torque, $\mathbf{K}_p$ and $\mathbf{K}_d$ are the control gain matrices, which will be positive diagonal matrices.

$$\boldsymbol{\Gamma} = -\mathbf{K}_p\, \delta\mathbf{q}_v(t) - \mathbf{K}_d\, \delta\omega(t) \tag{2.35}$$

This law, however, cannot be directly applied to the UAV treated in this work, as this law was thought for use in a quadrotor, where the control inputs can be easily mapped to the resulting torque, making it a robust control law that can be applied with little knowledge about the plant. Modelling a fixed-wing UAV is a much more complex task, as it requires large amounts of aerodynamic analysis and experimental data to be gathered, and the resuting model is a very complex one which depends on more external information such as airspeed, air density, etc. In order to cope with the model uncertainty, two major modifications are going to be made:

- The output of the controller is not going to be directly mapped to the desired torque, but instead to the control surface deflection angle ξ, which will correspond to the rudder for ξ, the elevator for ξ, and the aileron for ξ. This eliminates the need for a complex torque model achieving similar results.

- An integral term will be added to the control equation, which will assess the integral of $\delta \mathbf{q}_v$ over time. This will make sure that the steady state error is eliminated despite the lack of an aerodynamic model of the airplane.

The specific ξ deflection angles can be visualized for the tail section in figure 2.4.1.

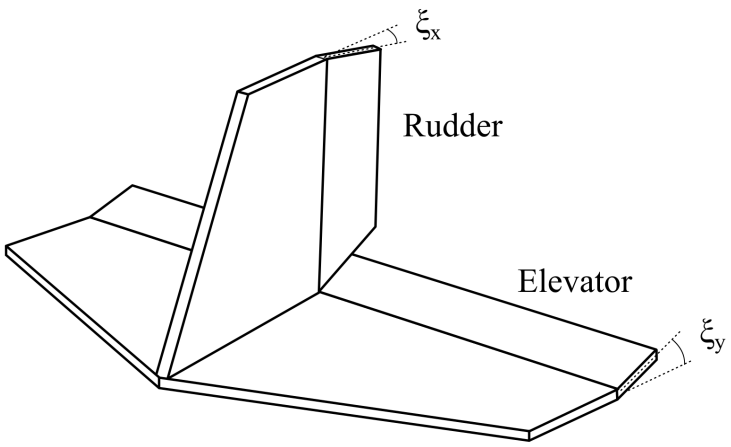

Figure 2.4.1: Illustration of the rudder and elevator control surfaces deflection.

When both modifications are added, the final quaternion-based non-linear PID control law is obtained, where the $\mathbf{K}_i$ diagonal matrix is also added.

$$\xi = -\mathbf{K}_p \, \delta \mathbf{q}_v(t) - \mathbf{K}_i \int \delta \mathbf{q}_v(t) dt - \mathbf{K}_d \, \delta \omega(t) \tag{2.36}$$

Software Development

3.1 Final code structure

The final code can run in real-time, estimate the attitude from the sensors' data and compute an adequate control action that is then sent to the actuators to execute. Several external aspects are taken into consideration when going through all of these steps. One such example is managing the different flight modes, of which three are defined:

- Failsafe mode: this occurs whenever the signal to the remote controller has been lost. This is given by the receiver (communicated through sbus) and it will make all of the servos go to their neutral position, and the throttle to be null, with the hopes that the natural stability of the plane will make it glide back to the ground in such an emergency scenario.
- Manual bypass mode: in this mode, there is connection with the remote controller, and the received inputs are directly mapped to the actuator commands. This is how regular RC aircraft are flown.
- Fly-by-wire mode: in this mode, the flight control loop is on, and the pilot's commands are mapped to the desired attitude. This is the main flight mode.

3.2 Main code and config files

The main code file, named `main.ino`, follows the typical arduino code structure, consisting of an initial section where the external libraries and additional source files are included, together with the declaration of the global variables. Then, the `void setup()` function can be found, which the arduino compiler ensures that it is the first code fragment to be executed, only once. Lastly, the `void loop()` function is declared where the main bulk of the code occurs, which mostly resembles the structure laid out in 4.0.1.

3.2.1 Configuration file

In order to make the main code specifically cleaner and better structured, the declaration of configuration constants has been moved to a separate `config.h` file, which can be found in section 5.2. Special care has been put into selecting the most suitable data types for each variable, declaring most integers with specific C-native types of a fixed size (such as const uint8_t for the pin numbers and register adresses, as these are very low.

One additional function of complex nature included in this configuration file is the `MPU_config()` function, which performs the following tasks:

- It initializes communications with the MPU 6050 over I2C using the wire library and sets an initial reset to low, in order to make the device start to communicate its data.
- It then sets the built-in gyroscope low-pass filter to a bandwidth of 96 Hz.
- Afterwards, it sets the range of the gyroscope and accelerometer data to $\pm$ 250 ^{o}/s and $\pm$ 4g respectively.

In order to execute all of these commands, the `MPU6050`'s register map (InvenSense Inc. [2013]) has been looked at in close detail.

3.2.2 Main code utils

Some of the functions that are of a more utilitary and general nature have been stored in the `utils.h` and `utils.cpp` files, which can be found in sections 5.3 and 5.4 in the annex. These files include the functions for flashing the indicator LEDs, logging the flight data to the SD card, as well as a saturation function (for limiting any number to a minumum and maximum values) as well as a generic map function (which essentially performs linear interpolation). The generic map function was the first place where the benefits of using templates for function declarations was found to be useful, as the stock map function that comes with the arduino library could not handle inputs of different types which were advantageous to use in this case.

3.2.3 External libraries

As it has been stated, external libraries have been used to make the final implementation process easier. The list of the libraries and what they are used for includes:

- `PWMServo.h`: used for sending a target angle to the control surfaces and a target throttle value to the motor.
- `Wire.h`: used for communications over I2C with the MPU 6050.
- `sbus.h`: used for asynchronous SBUS communications with the remote control receiver.
- `math.h`: used for basic mathematical functions.
- `SPI.h`: used for communications over Serial Peripheral Interface (SPI) with the SD card.
- `SD.h`: used for handling the write process and the various files to the SD card.
- `stdint.h`: used for the C-specific types that have been declared.

3.3 Algebraic functions

A small source file with the function of performing basic algebra functions on arrays of numbers has been created. The functions are as general as possible, so the size of the input matrices has to be specified, as C/C++ arrays decay to the pointers of the first element in the array, and no information about its size can be passed other than explicitly. This file offers the functionalities of:

- `sign()`: returns the sign of a number.
- `dot_product()`: returns the result of the dot product between two numbers.
- `vector_norm()`: returns the euclidean norm of an input array.
- `matrix_sum()`: returns the element-wise sum of two arrays of the same size.
- `matrix_multiply()`: performs matrix products.
- `matrix_transpose()`: transposes an input matrix.
- `normalize_vector()`: uses the previous `vector_norm()` function to transform the input vector to its unit vector equivalent.

All of these functions are extensively used in other more complex source files with good results. The functions themselves are not complex at all and all use standard algebraic methods to perform these functions. The algebra functions files can be found in sections 5.5 and 5.6.

3.4 Quaternion functions

To perform the necessary quaternion computations, two files have been created, similar to the algebraic functions source files. The algebraic concepts that they use are more complex, but all of it is explained in detail in section 2.3.1. The specific functions that this file contains are:

- `quaternion_product()`: performs the product between two quaternions.
- `normalize_quaternion()`: normalizes a quaternion, both in the sense that it makes its norm equal to one, and in that it makes changes the sign of the scalar part to be positive.
- `quaternion_error()`: computes the error quaternion between two different input quaternions.

The files where this code is housed can be found in sections 5.7 and 5.8.

3.5 Estimation functions

The functions that are responsible for performing the attitude estimation computations have also been defined in a separate file from the main, as these comprise a complex process. The first functions that are declared for the state estimation are the functions responsible for computing the attitude angles with the accelerometer sensors, which correspond to *observing the measured state* in terms of the Kalman Filter. These functions are:

- `pitch_Accelerometer()`: computes the pitch angle using the accelerometer data.
- `roll_Accelerometer()`: computes the roll angle using the accelerometer data.

The equations and meaning behind these functions are explored in detail in section 2.2. The next major function that is included in the estimation functions files is the main function which implements the Kalman Filter equations, which is named as `kalman_filter`. This function incorporates the observer model, described in equations from 2.14 to 2.17, and then applies the Kalman Filter two-step prediction and correction process for estimating the Euler angles, the bias in the gyroscope readings, and the error covariance matrix P, as illustrated in figure 2.2.1. All of this has also been explained in further detail in section 2.2.

Lastly, the function `euler2quat()` is implemented, which transforms the input euler angles into their corresponding quaternion using equations from 2.28 to 2.31.

The files which fouse these functions can be found in sections 5.9 and 5.10.

3.6 Control functions

The last of the separate files for external functionalities houses the control lawm which is relatively simple in nature and is therefore comprised in a single (regular) function, `quaternionPID()`, which when given the vectorial components of the quaternion error, its integral, the un-biased angular rates, as well as the target states and the control gains, computes the unlimited control surface deflections to be commanded.

This function is housed in the files found in sections 5.11 and 5.12.

3.7 Additional observations

Some of the additional challenges that have been encountered in the programming process will be commented here. The first one, is reading the correct data from the sensors. To do so, the wire library can request packets of data from the sensor via I2C when running the command

`Wire.requestFrom(MPU_6050, 6, true);`. This requests information from the six following memory registers within the MPU6050, and stores them in an internal memory buffer.

When looking at the datasheet, it can be seen that these six registers contain the full raw data for each axis, as two eight-bit registers per one axis. Therefore, the individual axis data must be obtained in binary by fetching the next register data in such a way that the pairs of registers conform a single binary number of 16-bits of length. This is done by first reading the first register, shifting it to the left by one byte, and concatenating it with the next register. This operation is done with the following code (in the case of the accelerometer:

```
int16_t ACCEL_OUT_X = Wire.read() << 8 | Wire.read();
int16_t ACCEL_OUT_Y = Wire.read() << 8 | Wire.read();
int16_t ACCEL_OUT_Z = Wire.read() << 8 | Wire.read();
```

Another important aspect when programming the software was time management. One alternative was to use software interrupts to execute the code at a fixed frequency, but the drawback of this was the need to implement possible timeouts in case that an error occurs in one computation step which causes delays in the control loop. While this may seem more robust, the possible timeouts will cause the sampling time to vary, and if this is not measured, it can make the estimation values inaccurate in the case of timeouts. It therefore was opted to use to run at the maximum frequency possible, and to count the time taken for each iteration. The calculation of the previous time and the reset of the time counter are done next to one another, and right after extracting the gyroscope's data, since the angular rate is the most time-critical measurement done. The Arduino library facilitates the time management by offering the following functions:

```
delta_time = (double)(micros() - gyro_time) / 1000000.0;
gyro_time = micros();
```

In order to save the flight data at a more moderate rate, however, a separate timer is declared as `sd_timer`, which when it reaches the sd sample time (in this case set to 0.15 seconds), it logs the sd data, resets to zero and starts counting again.

Results and conclusions

The final code has been extensively tested on the ground, achieving successful responses of the control surfaces to the attitude errors that have been imposed. This can be seen in the validation video that has been handed in together with this report.

Figure 4.0.1: Screenshot of the validation video, the airplane responds to the right roll and pitch down with an up elevator and an aileron response set to correct the wrong attitude (wrong as in non-leveled flight). Watch video for full context.

The transitions between the different flight modes are also tested in the video, proving a successful state and variables management.

Additionally, the Euler angle estimation has also been empirically validated by setting the flight computer at known angles and reading the estimated angles at the arduino serial terminal.

One of the biggest problems that is yet to be solved, is the data recording, as it has not been possible to successfully write the data itself onto a data file on the sd card, but only the header. Additional resources will go into finding out why before doing a full test-flight, but for now, it has been commented in the main code to not cause any interferences.

In conclusion, this project has successfully achieved most of its objectives of developing and implementing robust fly-by-wire control software for a small fixed-wing UAV. Leveraging an existing UAV platform focused on hardware, the project centered on the development of real-time, efficient software running on an Arduino-based flight computer. The effectiveness of the estimation and control scheme will have to be tested on the UAV platform once it is fully finished and the data logging problems have been sorted.

Lastly, programming this codebase has provided immense insight into real-time software development in embedded system with time-critical applications. Special care has been placed on the data types chosen and their manipulation, as well as to the eventual adaptability of the codebase.

Bibliography

Arduino. *Arduino® UNO R4 Minima Product Reference Manual*. Arduino, June 2024. URL `https://docs.arduino.cc/resources/datasheets/ABX00080-datasheet.pdf`. Datasheet.

Daniel Choukroun, I. Bar-Itzhack, and Yaakov Oshman. Optimal-request algorithm for attitude determination. *Journal of Guidance Control and Dynamics - J GUID CONTROL DYNAM*, 27: 418–425, 05 2004. doi: 10.2514/1.10337.

Emil Fresk and George Nikolakopoulos. Full quaternion based attitude control for a quadrotor. In *2013 European Control Conference (ECC)*, pages 3864–3869, 2013. doi: 10.23919/ECC.2013.6669617.

InvenSense Inc. *MPU-6000 and MPU-6050 Register Map and Descriptions*. InvenSense Inc., 1197 Borregas Ave, Sunnyvale, CA 94089, U.S.A., 08 2013. URL `http://www.invensense.com`. Document Number: RM-MPU-6000A-00, Revision: 4.2, Release Date: 08/19/2013.

A. Navarro Trastoy. Attitude determination using quaternion-based algorithms for 3cat-4. Master's thesis, Universitat Politècnica de Catalunya, Barcelona, 2019.

Janusz Zalewski. Real-time software architectures and design patterns: Fundamental concepts and their consequences. *Annual Reviews in Control*, 25:133–146, 12 2001. doi: 10.1016/S1367-5788(01)80001-8.

Appendix

5.1 main.ino

```
// Include libraries to be used
#include <PWMServo.h>
#include <Wire.h>
#include "sbus.h"
#include <math.h>
#include <SPI.h>
#include <SD.h>
#include <stdint.h>

// Include custom header files
#include "config.h"
#include "utils.h"
#include "algebra.h"
#include "quaternions.h"
#include "estimation.h"
#include "control.h"

// Initialize error flag of MPU 6050
int error = 1;

// Initialize SBUS objects for reading and writing on RX0
bfs::SbusRx sbus_rx(&Serial1);
bfs::SbusTx sbus_tx(&Serial1);
bfs::SbusData data;

// Declare control action
double control_action[3] = {0.0, 0.0, 0.0};

// Initialize servo objects
PWMServo elevator, throttle, aileron, rudder;

// Initialize Time management variables
double delta_time = 0.0;
unsigned long gyro_time = 0;

// Initialize data file object
File dataFile;
double sd_timer = 0.0;

// Declare global state variables
bool failsafe_mode = false;
bool manual_bypass = true;

// Declare the RX commands to be read from the receiver
unsigned int RX_cmd[8] = {0};
/*
0 -> Throttle
1 -> Ailerons
2 -> Elevator
3 -> Rudder
4 -> Large Left Toggle switch
5 -> Large Right Toggle switch
*/
```

```
54
55 // Initialize the accelerometer and gyroscope readings
56 double ACC_values[3];
57 double GYRO_values[3];
58
59 // Initialize the control surface angle values
60 double elevator_angle = 0.0;
61 double throttle_angle = 0.0;
62 double aileron_angle = 0.0;
63 double rudder_angle = 0.0;
64
65 // Initialize variables for Kalman Filter
66 double X_roll[2] = {0.0, 0.0};
67 double X_pitch[2] = {0.0, 0.0};
68
69 // Declare attitude, target and error quaternion, as well as angular body
     rates
70 double quaternion_f2b[4] = {1.0, 0.0, 0.0, 0.0};
71 double quaternion_f2t[4] = {1.0, 0.0, 0.0, 0.0};
72 double quaternion_t2b[4] = {1.0, 0.0, 0.0, 0.0};
73 double quaternion_t2b_int[4] = {0.0, 0.0, 0.0, 0.0};
74 double angular_rate[3] = {0.0, 0.0, 0.0};
75
76 // Declare target pitch and roll Euler angles, as well as target angular
     rate
77 double pitch_tg = 0.0;
78 double roll_tg = 0.0;
79 double yaw_rate_tg = 0.0;
80
81 // Begin setup
82 void setup() {
83   // Initialize USB serial communications
84   Serial.begin(9600);
85
86   // Perform IMU setup
87   Wire.begin();
88   error = 1;
89   while (error != 0){
90     Serial.println("MPU error");
91     error = MPU_config();
92   }
93
94   // Attach PWM control pins
95   elevator.attach(elevator_pin, min_servo, max_servo);
96   throttle.attach(throttle_pin, min_esc, max_esc);
97   aileron.attach(aileron_pin, min_servo, max_servo);
98   rudder.attach(rudder_pin, min_servo, max_servo);
99
100  // Begin SBUS communications
101  sbus_rx.Begin();
102  sbus_tx.Begin();
103  delay(100);
104
105  // Begin SD card communications
106  /*
107  // see if the card is present and can be initialized:
108  if (!SD.begin(chipSelect)) {
109    Serial.println("Card failed, or not present");
```

```
110   }
111   Serial.println("card initialized.");
112
113   // Write header to data file
114   dataFile = SD.open(fileName, FILE_WRITE);
115   if (dataFile) {
116
117     // Log header to CSV
118     //log_data(delta_time, failsafe_mode, manual_bypass, X_pitch[0], X_roll
          [0], angular_rate, quaternion_f2b, quaternion_f2t, quaternion_t2b,
          throttle_angle / 0.18, elevator_angle, rudder_angle, aileron_angle,
          false, dataFile);
119     //delay(50);
120     dataFile.println("Timestamp, SensorValue1, SensorValue2");
121     // Close the file after writing the data
122     dataFile.close();
123
124   } else {
125     Serial.print("Error opening SD card \n");
126   }
127   */
128
129   // Flash LED to indicate MPU connection steps performed
130   flash_led(statusLED, 5, 50);
131   flash_led(LED_BUILTIN, 5, 50);
132
133 }
134
135 void loop () {
136   // Begin transmission with the IMU over the Accelerometer register
137   Wire.beginTransmission(MPU_6050);
138   Wire.write(ACCEL_XOUT_H);
139   error = Wire.endTransmission(false);
140   if (error != 0) {
141     Serial.println("MPU error");
142     MPU_config();
143     flash_led(statusLED, 2, 10);
144   }
145
146   // Extract accelerometer data
147   Wire.requestFrom(MPU_6050, 6, true); // Extract the 6 following registers
        where all the acc data is stored
148   int16_t ACCEL_OUT_X = Wire.read() << 8 | Wire.read();
149   int16_t ACCEL_OUT_Y = Wire.read() << 8 | Wire.read();
150   int16_t ACCEL_OUT_Z = Wire.read() << 8 | Wire.read();
151
152   // Begin transmission with the IMU over the Accelerometer register
153   Wire.beginTransmission(MPU_6050);
154   Wire.write(GYRO_XOUT_H);
155   Wire.endTransmission(true);
156
157   // Store time in microseconds right before requesting gyroscope data
158   delta_time = (double)(micros() - gyro_time) / 1000000.0;
159   gyro_time = micros();
160   sd_timer += delta_time;
161
162   // Extrac gyroscope data
163   Wire.requestFrom(MPU_6050, 6, true); // Extract the 6 following registers
```

```
        where all the acc data is stored
164   int16_t GYRO_OUT_X = Wire.read() << 8 | Wire.read();
165   int16_t GYRO_OUT_Y = Wire.read() << 8 | Wire.read();
166   int16_t GYRO_OUT_Z = Wire.read() << 8 | Wire.read();
167
168   // Scale accelerometer data
169   ACC_values[0] = - double(ACCEL_OUT_Y) / acc_scaling;
170   ACC_values[1] = - double(ACCEL_OUT_X) / acc_scaling;
171   ACC_values[2] = + double(ACCEL_OUT_Z) / acc_scaling;
172
173   // Scale gyroscope data
174   GYRO_values[0] = + double(GYRO_OUT_Y) / gyro_scaling;
175   GYRO_values[1] = + double(GYRO_OUT_X) / gyro_scaling;
176   GYRO_values[2] = - double(GYRO_OUT_Z) / gyro_scaling;
177
178   // Perform Kalman Filtering
179   double Y_pitch = pitch_Accelerometer(ACC_values[0], ACC_values[1],
          ACC_values[2]);
180   double Y_roll = roll_Accelerometer(ACC_values[1], ACC_values[2]);
181   kalman_filter(delta_time, GYRO_values[0], Q_roll, R_roll, Y_roll, X_roll,
          P_roll);
182   kalman_filter(delta_time, GYRO_values[1], Q_pitch, R_pitch, Y_pitch,
          X_pitch, P_pitch);
183
184   Serial.print(X_pitch[0] * 180.0 / M_PI);
185   Serial.print(", ");
186   Serial.println(X_roll[0] * 180.0 / M_PI);
187
188   // Remove bias from angular rate estimation
189   angular_rate[0] = GYRO_values[0] - X_roll[1];
190   angular_rate[1] = GYRO_values[1] - X_pitch[1];
191   angular_rate[2] = GYRO_values[2];
192
193   // If receiver is connected
194   if (sbus_rx.Read()) {
195     // Grab the received data
196     data = sbus_rx.data();
197
198     // If RX detects failsafe, set mode to failsafe
199     if(data.failsafe == 1){
200       failsafe_mode = true;
201
202       // Set servo commands to neutral and throttle to zero
203       rudder_angle = map(mid_sbus, min_sbus, max_sbus, -90.0, 90.0);
204       elevator_angle = map(mid_sbus, min_sbus, max_sbus, -90.0, 90.0);
205       aileron_angle = map(mid_sbus, min_sbus, max_sbus, -90.0, 90.0);
206       throttle_angle = 0.0;
207     }
208
209     else{
210
211       // Read all channel data
212       for(int ch_counter = 0; ch_counter < 8; ch_counter ++){
213
214         // Read the data and set to middle values for deadband
215         RX_cmd[ch_counter] = constrain(data.ch[ch_counter], min_sbus,
              max_sbus);
216         if(data.ch[ch_counter] < mid_sbus + deadband_sbus && data.ch[
```

```
            ch_counter] > mid_sbus - deadband_sbus)
217         RX_cmd[ch_counter] = mid_sbus;
218     }
219
220     // Check manual bypass switch
221     manual_bypass = (RX_cmd[4] > mid_sbus) ? false : true;
222
223     // Check attitude reset switch
224     if (RX_cmd[5] > mid_sbus){
225       // Set target attitude to zero
226       pitch_tg = 0.0;
227       roll_tg = 0.0;
228
229       // Set quaternion error integral to zero
230       for(int i = 0; i < 4; i++){
231         quaternion_t2b_int[i] = 0.0;
232       }
233
234     }
235
236     // Set commands to the servos
237     if(manual_bypass == true || error != 0){
238       // Set quaternion error integral to zero
239       for(int i = 0; i < 4; i++){
240         quaternion_t2b_int[i] = 0.0;
241       }
242
243       // Set servo commands to direct manual values
244       rudder_angle = map(RX_cmd[3], min_sbus, max_sbus, min_rudder,
              max_rudder);
245       elevator_angle = map(RX_cmd[2], min_sbus, max_sbus, min_elevator,
              max_elevator);
246       aileron_angle = map(RX_cmd[1], min_sbus, max_sbus, min_aileron,
              max_aileron);
247     }
248     else{
249
250       // Update the target pitch and roll
251       pitch_tg -= map_Generic((double)RX_cmd[2], min_sbus, max_sbus, -
              max_pitch_rate, max_pitch_rate) * delta_time;
252       roll_tg += map_Generic((double)RX_cmd[1], min_sbus, max_sbus, -
              max_roll_rate, max_roll_rate) * delta_time;
253       yaw_rate_tg = map_Generic((double)RX_cmd[3], min_sbus, max_sbus, -
              max_yaw_rate, max_yaw_rate);
254
255       Serial.print(pitch_tg);
256       Serial.print(" ");
257       Serial.print(roll_tg);
258       Serial.print(" ");
259       Serial.println(yaw_rate_tg);
260       //Serial.println((double) RX_cmd[2]);
261
262       // Update the current and target quaternion
263       euler2quat(0.0, X_pitch[0], X_roll[0], quaternion_f2b);
264       euler2quat(0.0, pitch_tg, roll_tg, quaternion_f2t);
265
266       // Compute the quaternion error
267       quaternion_error(quaternion_f2t, quaternion_f2b, quaternion_t2b);
```

```
268
269         // Update quaternion error integral
270         for(int i = 0; i < 4; i++){
271           quaternion_t2b_int[i] += quaternion_t2b[i] * delta_time;
272         }
273
274         // Apply quaternion PID
275         quaternionPID(quaternion_t2b, quaternion_t2b_int, angular_rate,
                yaw_rate_tg, Kp, Ki, Kd, control_action);
276
277         // Set servo commands to stabilized values with control loop
278         elevator_angle = - saturate(control_action[1], (double)
                min_elevator, (double) max_elevator);
279         aileron_angle = saturate(control_action[0], (double) min_aileron, (
                double) max_aileron);
280         rudder_angle = saturate(control_action[2], (double) min_rudder, (
                double) max_rudder);
281       }
282
283       // Set command to the throttle
284       throttle_angle = map(RX_cmd[0], min_sbus, max_sbus, 0.0, 180.0);
285
286       // Saturate command values
287       throttle_angle = saturate(throttle_angle, (double) min_throttle, (
              double) max_throttle);
288
289     }
290
291   }
292
293   // Send servo commands through PWM
294   elevator.write(- elevator_angle + 90.0);
295   throttle.write(throttle_angle);
296   aileron.write(aileron_angle + 90.0);
297   rudder.write(- rudder_angle + 90.0);
298
299   // Write data to SD card
300   /*
301   if (sd_timer > sd_sampling){
302
303     // Reset SD timer
304     sd_timer = 0.0;
305
306     // Write to data file
307     //dataFile = SD.open(fileName, FILE_WRITE);
308     dataFile = SD.open("data.txt", FILE_WRITE);
309     if (dataFile) {
310
311       // Log header to CSV
312       //log_data(delta_time, failsafe_mode, manual_bypass, X_pitch[0],
              X_roll[0], angular_rate, quaternion_f2b, quaternion_f2t,
              quaternion_t2b, throttle_angle / 0.18, elevator_angle,
              rudder_angle, aileron_angle, false, dataFile);
313       delay(50);
314       dataFile.println("Timestamp, SensorValue1, SensorValue2");
315
316       // Close the file after writing the data
317       dataFile.close();
```

```
318
319       } else {
320         Serial.print("Error opening SD card \n");
321       }
322     }
323     */
324 }
```

5.2 config.h

```
#ifndef CONFIG_H
#define CONFIG_H

// Include libraries
#include <stdint.h>

// Specify control gains
const double Kp[3] = {150.0, 150.0, 0.0};
const double Ki[3] = {225.0, 225.0, 0.0};
const double Kd[3] = {5.0, 5.0, 15.0};

// Specify name of the data file
const char *fileName = "data.txt";
const double sd_sampling = 0.25;

// Define output pins
const uint8_t elevator_pin = 3;
const uint8_t throttle_pin = 5;
const uint8_t aileron_pin  = 6;
const uint8_t rudder_pin   = 9;
const uint8_t chipSelect   = 10;
const uint8_t statusLED    = 7;

// Declare MPU 6050 addresses
const uint8_t MPU_6050     = 0x68;
const uint8_t PWR_MGMT_1   = 0x6B;
const uint8_t CONFIG       = 0x1A;
const uint8_t GYRO_CONFIG  = 0x1B;
const uint8_t ACC_CONFIG   = 0x1C;
const uint8_t ACCEL_XOUT_H = 0x3B;
const uint8_t GYRO_XOUT_H  = 0x43;

// Define scaling values of the IMU sensors
const double acc_scaling  = 8192.f;    // 2^16 / 8 (16 bits max value / +-4
    g)
const double gyro_scaling = 7509.8724; // 2^16 / 500 * 180 / pi (16 bits
    max value / +-250º/s in rad/s)

// Define uncertainty matrices of the MPU6050 sensors for the roll axis
double Q_roll[2][2] = {{0.001, 0.0}, {0.0, 0.003}};
double P_roll[2][2] = {{1.f, 0.0}, {0.0, 1.f}};
double R_roll = 30.f;

// Define uncertainty matrices of the MPU6050 sensors for the pitch axis
double Q_pitch[2][2] = {{0.001, 0.0}, {0.0, 0.003}};
double P_pitch[2][2] = {{1.f, 0.0}, {0.0, 1.f}};
double R_pitch = 30.f;

// Define minimum and maximum SBUS values
const uint16_t min_sbus = 180;
const uint16_t max_sbus = 1800;
const uint16_t mid_sbus = 990;
const uint16_t deadband_sbus = 10;

// Define maximum absolute rotation rates for processing the RC inputs
const double max_pitch_rate = 300 * M_PI / 180.0;
```

```
55 const double max_roll_rate = 150 * M_PI / 180.0;
56 const double max_yaw_rate = 10 * M_PI / 180.0;
57
58 // Declare min and max pulse length for servo PWM
59 const uint16_t min_servo = 600;
60 const uint16_t max_servo = 2300;
61 const uint16_t min_esc = 1000;
62 const uint16_t max_esc = 2000;
63
64 // Declare min and max angle values for servos and throttle (180deg max
     range)
65 const int16_t min_elevator = -30;
66 const int16_t max_elevator = 30;
67 const int16_t min_throttle = 0;
68 const int16_t max_throttle = 180;
69 const int16_t min_aileron = -25;
70 const int16_t max_aileron = 25;
71 const int16_t min_rudder = -20;
72 const int16_t max_rudder = 20;
73
74 // Define MPU configuration function
75 int MPU_config(){
76   // Initialize temporary error and error flag
77   int error_temp = 0;
78   int error_flag = 0;
79
80   // Write reset to low
81   Wire.beginTransmission(MPU_6050);
82   Wire.write(PWR_MGMT_1); // Write to the power management 1 adress
83   Wire.write(0x00); // Set DEVICE_RESET (bit 7) to 0 - don't reset MPU 6050
       ??
84   error_temp = Wire.endTransmission(true);
85   error_flag = (error_temp != 0) ? error_temp : error_flag;
86
87   // Set filter bandwidth values to ~ 96Hz
88   Wire.beginTransmission(MPU_6050);
89   Wire.write(CONFIG); // Write to the gyro config 1 adress
90   Wire.write(0x02); // Set DLPF_CFG to 02 (acc - 94Hz @ 3ms, gyro - 98Hz @
       2.8ms)
91   error_temp = Wire.endTransmission(true);
92   error_flag = (error_temp != 0) ? error_temp : error_flag;
93
94   // Set gyroscope configuration to +- 250 deg/s
95   Wire.beginTransmission(MPU_6050);
96   Wire.write(GYRO_CONFIG); // Write to the gyro config adress
97   Wire.write(0x00); // Set FS_SEL to 00 (+-250deg/s)
98   error_temp = Wire.endTransmission(true);
99   error_flag = (error_temp != 0) ? error_temp : error_flag;
100
101  // Set accelerometer configuration to +- 4g
102  Wire.beginTransmission(MPU_6050);
103  Wire.write(ACC_CONFIG); // Write to the acc config adress
104  Wire.write(0x08); // Set AFS_SEL to 01 (+-4g)
105  error_temp = Wire.endTransmission(true);
106  error_flag = (error_temp != 0) ? error_temp : error_flag;
107
108  return error_flag;
109 }
```

```
110
111 #endif // CONFIG_H
```

5.3 utils.h

```
#ifndef UTILS_H
#define UTILS_H

// Include libraries

#include "Arduino.h"
#include <SD.h>
#include <SPI.h>

// Define regular functions

void log_data(double dt, bool failsafe, bool bypass, double pitch, double roll, double angular_rate[3], double current_quaternion[4], double target_quaternion[4], double error_quaternion[4], double throttle_percent, double elevator_ang, double rudder_ang, double aileron_ang, bool header, File& dataFile);
void flash_led(int pin, int repetitions, int period);

// Define template functions

template <typename Tval, typename Tlim>
Tval saturate(Tval value, Tlim min, Tlim max){
  // Function to saturate a given input
  value = (value > max) ? max : value;
  value = (value < min) ? min : value;
  return value;
}

template <typename Xres, typename Xin, typename Xout>
Xres map_Generic(Xres x, Xin in_min, Xin in_max, Xout out_min, Xout out_max) {

    return (x - in_min) * (out_max - out_min) / (in_max - in_min) + out_min;
}

#endif // UTILS_H
```

5.4 utils.cpp

```
#include "utils.h"

void log_data(double dt, bool failsafe, bool bypass, double pitch, double
    roll, double angular_rate[3], double current_quaternion[4], double
    target_quaternion[4], double error_quaternion[4], double
    throttle_percent, double elevator_ang, double rudder_ang, double
    aileron_ang, bool header, File& dataFile) {

  if (dataFile) {

    if (header) {
      // Create header information
      String header_str = "dt, flight_mode, pitch, roll, ang_rate_x,
          ang_rate_y, ang_rate_z, ";
      header_str += "quat_s2b_w, quat_s2b_x, quat_s2b_y, quat_s2b_z, ";
      header_str += "quat_s2t_w, quat_s2t_x, quat_s2t_y, quat_s2t_z, ";
      header_str += "quat_b2t_w, quat_b2t_x, quat_b2t_y, quat_b2t_z, ";
      header_str += "throttle_percent, elevator_ang, rudder_ang,
          aileron_ang";
      dataFile.println(header_str);
    }
    else {
      // Create data information
      int mode = (failsafe) ? 0 : ((bypass) ? 1 : 2);

      String data_str = "";
      data_str += String(dt) + ", ";
      data_str += String(mode) + ", ";
      data_str += String(pitch * 180 / PI) + ", ";
      data_str += String(roll * 180 / PI) + ", ";
      data_str += String(angular_rate[0] * 180 / PI) + ", ";
      data_str += String(angular_rate[1] * 180 / PI) + ", ";
      data_str += String(angular_rate[2] * 180 / PI) + ", ";
      data_str += String(current_quaternion[0]) + ", ";
      data_str += String(current_quaternion[1]) + ", ";
      data_str += String(current_quaternion[2]) + ", ";
      data_str += String(current_quaternion[3]) + ", ";
      data_str += String(target_quaternion[0]) + ", ";
      data_str += String(target_quaternion[1]) + ", ";
      data_str += String(target_quaternion[2]) + ", ";
      data_str += String(target_quaternion[3]) + ", ";
      data_str += String(error_quaternion[0]) + ", ";
      data_str += String(error_quaternion[1]) + ", ";
      data_str += String(error_quaternion[2]) + ", ";
      data_str += String(error_quaternion[3]) + ", ";
      data_str += String(throttle_percent) + ", ";
      data_str += String(elevator_ang) + ", ";
      data_str += String(rudder_ang) + ", ";
      data_str += String(aileron_ang);

      dataFile.println(data_str);
    }

  }
  else{
    Serial.println("SD error");
```

```
51   }
52 }
53
54 void flash_led(int pin, int repetitions, int period){
55   // initialize digital pin LED_BUILTIN as an output.
56   pinMode(pin, OUTPUT);
57   for(int i = 0; i < repetitions; i++){
58     digitalWrite(pin, HIGH);   // turn the LED on (HIGH is the voltage
         level)
59     delay(period);                          // wait for 50 ms
60     digitalWrite(pin, LOW);    // turn the LED off by making the voltage
         LOW
61     delay(period);                          // wait for 50 ms
62   }
63 }
```

5.5 algebra.h

```
#ifndef ALGEBRA_H
#define ALGEBRA_H

#include <math.h>

#include "algebra.tpp"

template <typename T>
T sign(T number);
template <typename T>
T dot_product(T *vec1, T *vec2, int size);

template <typename T>
T vector_norm(T *vector, int size);

template <typename T>
void matrix_sum(T* mat1, T* mat2, int rows, int cols, T* result);

template <typename T>
void matrix_multiply(T *mat1, T *mat2, int rows1, int cols1, int cols2, T *
    result);

template <typename T>
void matrix_transpose(T *mat, int rows, int cols, T *result);

template <typename T>
void normalize_vector(T *vector, int size);

#endif // ALGEBRA_H
```

5.6 algebra.tpp

```
#ifndef ALGEBRA_TPP
#define ALGEBRA_TPP

#include "algebra.h"

template <typename T>
T sign(T number) {
    return (number > 0.0) - (number < 0.0);
}

template <typename T>
T dot_product(T *vec1, T *vec2, int size){
  // Perform dot product

  T result = 0.0;
  for(int i = 0; i < size; i++)
    result += vec1[i] * vec2[i];

  return result;
}

template <typename T>
T vector_norm(T *vector, int size){
    // Function to find the norm of a given vector
    // Compute the array's norm
    T norm = 0.0;
    T norm_squared = 0.0;
    for(int i = 0; i < size; i++)
        norm_squared += pow(vector[i], 2);
    norm = sqrt(norm_squared);

    return norm;
}

template <typename T>
void matrix_sum(T* mat1, T* mat2, int rows, int cols, T* result) {
  // Perform matrix sum
  for (int i = 0; i < rows; ++i) {
    for (int j = 0; j < cols; ++j) {
      int idx = i * cols + j;
      result[idx] = mat1[idx] + mat2[idx];
    }
  }
}

template <typename T>
void matrix_multiply(T *mat1, T *mat2, int rows1, int cols1, int cols2, T *
    result) {
    // Perform matrix multiplication
    for (int i = 0; i < rows1; ++i) {
        for (int j = 0; j < cols2; ++j) {
            T sum = 0.0;
            for (int k = 0; k < cols1; ++k) {
                sum += mat1[i * cols1 + k] * mat2[k * cols2 + j];
            }
            result[i * cols2 + j] = sum;
```

```
56          }
57      }
58  }
59
60  template <typename T>
61  void matrix_transpose(T *mat, int rows, int cols, T *result) {
62      // Perform matrix transpose
63
64      for (int i = 0; i < rows; ++i) {
65          for (int j = 0; j < cols; ++j) {
66              result[j * rows + i] = mat[i * cols + j];
67          }
68      }
69  }
70
71  template <typename T>
72  void normalize_vector(T *vector, int size){
73      // Function to normalize a given vector
74      // The input-output array should be a 1 dimensional float array
75
76      // Compute the array's norm
77      T norm = vector_norm(vector, size);
78
79      // Divide individual elements by the array's norm
80      for(int i = 0; i < size; i++)
81          vector[i] /= norm;
82  }
83
84  #endif // ALGEBRA_TPP
```

5.7 quaternions.h

```
// Declare the header guards

#ifndef QUATERNIONS_H
#define QUATERNIONS_H

// Include libraries

#include <math.h>

// Include other source files

#include "algebra.h"
#include "quaternions.tpp"

// Declare and define functions

template <typename T>
void quaternion_product(T input_p[4], T input_q[4], T* output);

template <typename T>
void normalize_quaternion(T* output);

template <typename T>
void quaternion_error(T q_desired[4], T q_actual[4], T* output);

#endif // QUATERNION_FUNCTIONS_H
```

5.8 quaternions.tpp

```cpp
#ifndef QUATERNIONS_TPP
#define QUATERNIONS_TPP

#include "quaternions.h"

template <typename T>
void quaternion_product(T input_p[4], T input_q[4], T* output){
    // Function for computing the product between two quaternions
    // Input arrays must have a size of 4

    // Extract the real parts of the quaternions
    T pw = input_p[0];
    T qw = input_q[0];

    // Extract the imaginary parts of the quaternions
    T pv[3] = {input_p[1], input_p[2], input_p[3]};
    T qv[3] = {input_q[1], input_q[2], input_q[3]};

    // Compute the dot product of the imaginary parts of the input
       quaternions
    T dot_product_pqv = 0;
    for(int i = 0; i < 3; i++)
        dot_product_pqv = dot_product_pqv + pv[i] * qv[i];

    // Compute elements of the quaternion
    output[0] = pw * qw - dot_product_pqv;
    output[1] = pw * qv[0] + qw * pv[0] + pv[1] * qv[2] - pv[2] * qv[1];
    output[2] = pw * qv[1] + qw * pv[1] + pv[2] * qv[0] - pv[0] * qv[2];
    output[3] = pw * qv[2] + qw * pv[2] + pv[0] * qv[1] - pv[1] * qv[0];
}

template <typename T>
void normalize_quaternion(T* output){
    // Function for normalizing the attitude quaternion after numerical
       integration
    // The output must enter as a 1D array of length 4

    // If scalar part is negative, multiply by -1
    T sign = 1;
    if(output[0] < 0)
        sign = - sign;

    for(int i = 0; i < 4; i++)
        output[i] *= sign;

    // Normalize the array (of size 4 <--> quaternion)
    normalize_vector(output, 4);

}

template <typename T>
void quaternion_error(T q_desired[4], T q_actual[4], T* output){
    // Function for computing the error between two quaternions
    // Input arrays must have a size of 4

    // Normalize both quaternions
```

```
55      normalize_quaternion(q_desired);
56      normalize_quaternion(q_actual);
57
58      // Extract the real parts of the quaternions
59      T q_desired_inverse[4];
60      q_desired_inverse[0] = + q_desired[0];
61      q_desired_inverse[1] = - q_desired[1];
62      q_desired_inverse[2] = - q_desired[2];
63      q_desired_inverse[3] = - q_desired[3];
64
65      // Apply the error formula
66      quaternion_product(q_desired_inverse, q_actual, output);
67
68      normalize_quaternion(output);
69  }
70
71  #endif // QUATERNIONS_TPP
```

5.9 estimation.h

```
#ifndef ESTIMATION_H
#define ESTIMATION_H

// Include libraries
#include <math.h>

// Include other source files
#include "algebra.h"
#include "estimation.tpp"

template <typename T>
T pitch_Accelerometer(T AccX, T AccY, T AccZ);

template <typename T>
T roll_Accelerometer(T AccY, T AccZ);

template <typename T>
void kalman_filter(T delta_time, T W_gyro, T Q[2][2], T R, T Y, T
    X_estimated[2], T P_estimated[2][2]);

template <typename T>
void euler2quat(T yaw, T pitch, T roll, T* output);

#endif // ESTIMATION_H
```

5.10 estimation.tpp

```
#ifndef ESTIMATION_TPP
#define ESTIMATION_TPP

#include "estimation.h"

template <typename T>
T pitch_Accelerometer(T AccX, T AccY, T AccZ) {
    return std::atan2(-AccX, std::sqrt(AccY * AccY + AccZ * AccZ));
}

template <typename T>
T roll_Accelerometer(T AccY, T AccZ) {
    return std::atan2(AccY, AccZ);
}

template <typename T>
void kalman_filter(T delta_time, T W_gyro, T Q[2][2], T R, T Y, T
    X_estimated[2], T P_estimated[2][2]){

  // Store temporary X previous
  T X_previous[2];
  for(int i = 0; i < 2; i++){
    X_previous[i] = X_estimated[i];
  }

  // Store temporary P previous
  T P_previous[2][2];
  for(int i = 0; i < 2; i++){
    for(int j = 0; j < 2; j++){
      P_previous[i][j] = P_estimated[i][j];
    }
  }

  // Construct Kalman filter model matrices
  T A[2][2] = {{1.0, -delta_time}, {0.0, 1.0}};
  T B[2] = {delta_time, 0.0};
  T C[2] = {1.0, 0.0};

  // Perform prediction - X_predicted = A * X_previous + B * W_angle;
  T AX_prev[2];
  matrix_multiply((T*) A, (T*) X_previous, 2, 2, 1, (T*) AX_prev);
  T BW_ang[2] = {B[0] * W_gyro, B[1] * W_gyro};
  T X_predicted[2] = {AX_prev[0] + BW_ang[0], AX_prev[1] + BW_ang[1]};

  // Propagate the error covariance matrix - P_predicted = A * P_previous *
      A' + Q;
  T A_transposed[2][2];
  matrix_transpose((T*) A, 2, 2, (T*) A_transposed);
  T AP[2][2];
  matrix_multiply((T*) A, (T*) P_previous, 2, 2, 2, (T*) AP);
  T APAt[2][2];
  matrix_multiply((T*) AP, (T*) A_transposed, 2, 2, 2, (T*) APAt);
  T P_predicted[2][2];
  matrix_sum((T*) APAt, (T*) Q, 2, 2, (T*) P_predicted);

  // Compute the Kalman Gain - /K_gain = P_predicted * C' / (C *
```

```
      P_predicted * C' + R);
  T PCt[2];
  matrix_multiply((T*) P_predicted, (T*) C, 2, 2, 1, (T*) PCt);
  T CPCtR = dot_product((T*) C, (T*) PCt, 2) + R;
  T K_gain[2] = {PCt[0] / CPCtR, PCt[1] / CPCtR};

  // Update estimate with measurement - X_estimated = X_predicted + K_gain
     * (Y - C * X_predicted);
  T YmCX = Y - dot_product((T*) C, (T*) X_predicted, 2);
  T K_prod[2] = {K_gain[0] * YmCX, K_gain[1] * YmCX};
  matrix_sum((T*) X_predicted, (T*) K_prod, 2, 1, (T*) X_estimated);

  // Update the error covariance - P_estimated = (eye(size(P_previous)) -
     K_gain * C) * P_predicted;
  T Identity_2x2[2][2] = {{1.f, 0.f}, {0.f, 1.f}};
  T mKC[2][2];
  for(int i = 0; i < 2; i++){
    for(int j = 0; j < 2; j++){
      mKC[i][j] = - K_gain[i] * C[j];
    }
  }
  T ImKC[2][2];
  matrix_sum((T*) Identity_2x2, (T*) mKC, 2, 2, (T*) ImKC);
  matrix_multiply((T*) ImKC, (T*) P_predicted, 2, 2, 2, (T*) P_estimated);

}

template <typename T>
void euler2quat(T yaw, T pitch, T roll, T* output) {

  T cr = std::cos(roll * 0.5);
  T sr = std::sin(roll * 0.5);
  T cp = std::cos(pitch * 0.5);
  T sp = std::sin(pitch * 0.5);
  T cy = std::cos(yaw * 0.5);
  T sy = std::sin(yaw * 0.5);

  output[0] = cr * cp * cy + sr * sp * sy;
  output[1] = sr * cp * cy - cr * sp * sy;
  output[2] = cr * sp * cy + sr * cp * sy;
  output[3] = cr * cp * sy - sr * sp * cy;

}

#endif // ESTIMATION_TPP
```

5.11 control.h

```
#ifndef CONTROL_H
#define CONTROL_H

// Include libraries
#include <math.h>
#include "algebra.h"

// Define the various functions to be used
void quaternionPID(double err_quaternion[4], double int_quaternion[4],
    double ang_rate[3], double ref_rate_yaw, const double Kp[3], const
    double Ki[3], const double Kd[3], double* output);

#endif // CONTROL_H
```

5.12 control.cpp

```cpp
#include "control.h"

void quaternionPID(double err_quaternion[4], double int_quaternion[4],
    double ang_rate[3], double ref_rate_yaw, const double Kp[3], const
    double Ki[3], const double Kd[3], double* output){
  // Non-linear quaternion PID function

  // Declare signs of quaternions (account for unwinding)
  int att_sign, int_sign;

  // Declare P, I and D terms
  double P_term, I_term, D_term;

  // Apply PID equation to each axis
  for(int i = 0; i < 3; i++){
    att_sign = sign(err_quaternion[0]);
    int_sign = sign(int_quaternion[0]);
    P_term = att_sign * Kp[i] * err_quaternion[i + 1];
    I_term = int_sign * Ki[i] * int_quaternion[i + 1];
    D_term = - Kd[i] * (ref_rate_yaw - ang_rate[i]);
    output[i] = - (P_term + I_term + D_term);
  }
}
```