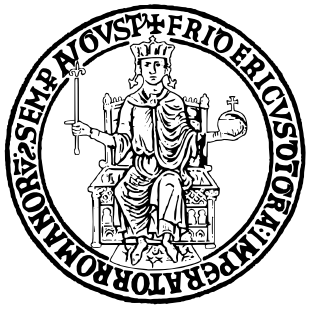
Università degli Studi di Napoli Federico II



Scuola Politecnica e delle Scienze di Base

Dipartimento di Ingegneria Elettrica e Tecnologie dell'Informazione

Corso di Laurea Triennale in Informatica

Experimental Thesis

Simple Kalman Filter for Altitude Estimation on model rockets with embedded system

Supervisor
Prof. Alberto Finzi

Candidate
Andrea Di Donato
N86004934

Academic Year 2024–2025

Abstract

Altitude estimation is a classic problem in the aviation sector, with various solutions in the engineering world. This thesis originated from a real-life problem encountered during the development of the Hermes model rocket for the UniNa Rockets association [20], when our first flight computer was designed. We faced a significant issue with noisy sensor data and a major problem when the rocket's structure failed and the flight computer entered a free-fall motion. After discussing the problem with Professor Alberto Finzi, we arrived at a solution: using a Kalman Filter to enhance sensor data, make a better prediction about the rocket's trajectory, and avoid systematic errors and noisy readings.

Therefore, the main objective was to understand whether the model could work for our problem and if it could run effectively on the embedded system. The thesis explores real-life problems and presents both well-known and custom-made solutions to improve apogee detection. Throughout the thesis, particular attention is given to developing optimized code for the microcontroller, utilizing a safe coding approach with various checks on the running instances and conducting tests at multiple levels.

During the development of this thesis, the code and all created instruments were uploaded to GitHub with an Open Source license. This was done to uphold the ideals of great men who came before me and to allow for future improvements, with the possibility that the code might have a life beyond this thesis project.

"There were open source projects and free software before Linux was there. Linux in many ways is one of the more visible and one of the bigger technical projects in this area, and it changed how people looked at it because Linux took both the practical and ideological approach."

Linus Torvalds

Contents

Chapter 1

Introduction

As technological progress brings us into a world with a growing need for data, the use of sensors has become progressively more relevant, bridging the gap from the physical to the virtual world. The need for a large amount of data for AI training often leads to scraping data from every available device that has a sensor. This approach is compelling due to its adaptability to various problems without requiring a change in the core methodology.

In this thesis, however, we adopt a more "**traditional**" approach that utilises a well-established mathematical tool: the Kalman Filter. This filter has been widely adopted across various technological fields, including everyday devices (cell phones, smartwatches, microwaves, etc.), IoT devices, automation, the automotive industry, and avionics. It is fair to say that the **Kalman Filter** is a mathematical concept that has improved our lives in ways that we might not even realise. The world of avionics, in particular, relies on the Kalman Filter for navigation and control systems. It is essential because it can be integrated and computed on systems that require high speed and high update frequency, while also being mindful of power consumption and limited hardware performance.

The work for this thesis is based on an internal apprenticeship performed with Professor Alberto Finzi. The goal was to study and develop a flight computer capable of filtering data from two sensors, a barometric sensor and an accelerometer, to determine the apogee as accurately as possible, with a functional solution for the STM32 microcontroller. To collect data, the flight computer is equipped with an open-source data logger that writes data to a text file for post-flight analysis.

Despite the Kalman Filter's extensive use in avionics, it is challenging to find general-purpose, low-level solutions that can be directly applied to microcontrollers. Although implementations for Arduino IDE, Python, or Matlab are relatively easy to find, a well-documented and C-based implementation for our specific problem is rare. This scarcity led us to develop a tailor-made solution that can run on low-end devices without the need for a specific Integrated Development Environment.

The data acquired by the flight computer (Figure 1.1) includes the following: pressure, from the LPS22H barometric sensor, which is the primary value used

for altitude calculation, temperature, also from the LPS22H barometric sensor, used in conjunction with pressure to improve altitude calculation,

3-axis acceleration, from the LSM6DSO accelerometer sensor, used to determine acceleration in the direction of gravity.

All of these sensors are mounted on a shield attached to the **Nucleo STM32F401RE** board, with an external data logger used to save the microcontroller's data.

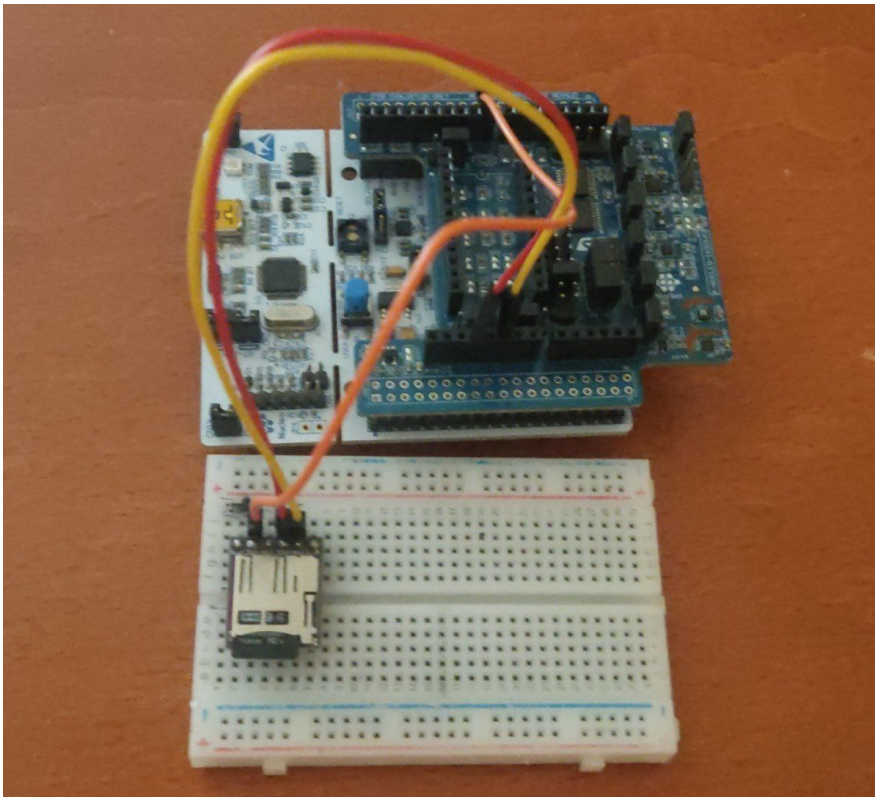


Figure 1.1: Flight Computer early prototype

The primary objective of the apprenticeship was to verify the effectiveness of using the Kalman Filter with real data and to generate optimised, high-performance code for the microcontroller. Special consideration was given to task scheduling and system reliability, as the flight computer is a critical system responsible for recovering the rocket after it reaches its apogee.

The first phase of the apprenticeship involved studying the model provided by the professor using a Python-based simulator. This model was used to analyse the behaviour of the Kalman Filter in various configurations to select the best solution for our specific problem. The approach was iterative, involving changes to the model's configuration and testing with both real and simulated data.

In the second phase, the C code for the Kalman Filter was developed. This included the creation of a matrix library, with careful attention paid to memory safety and optimisation. The code was released to be used by anyone who needs it for their projects.

The final phase of the apprenticeship was the realisation of the flight computer

itself. This involved configuring all sensor peripherals, studying **FreeRTOS** for task management, and integrating the developed libraries. The main challenge was to optimise the code for continuous operation and ensure real-time performance.

The Hermes model rocket (Figure 1.2), the latest model developed by UniNa Rockets, was used as a reference during the internship to provide a starting point and to tune the filters. The model rocket is 1.26 meters long, has a diameter of 9.3 cm, and is powered by a 560-newton thrust engine.

Figure 1.2: Hermes model rocket [19] Source: UniNa Rockets

Chapter 2

Background

The mathematical concepts explained in this chapter are fundamental to this project, which aims to represent the physical world using mathematical equations and physics models. These models are used to simulate rocket motion and to enhance the data.

In the first part of this chapter, we will explore the dynamics of a rocket, considering the nature of the problem and the difficulties encountered during the study process. The second part is dedicated to modelling the system and the trajectory, and to presenting the Kalman Filter and the specific case studied in this thesis.

2.1 Rocket Motion

Understanding rocket motion is the first challenge encountered when working on any aerospace system. It is crucial to take into account all the forces acting on the rocket.

The forces acting on our rocket are illustrated in Figure 2.1.

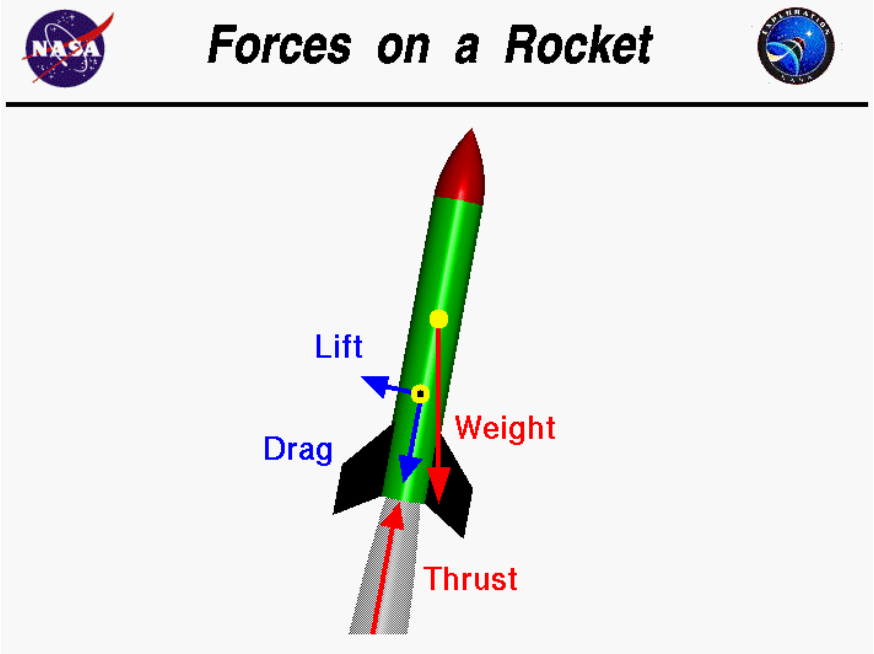


Figure 2.1: Forces acting on a rocket. Source: Glenn Research Department, Beginner Guide to Aeronautics [8].

As shown in this figure, four forces act on the rocket: weight, thrust, drag, and lift. [8]

- **Weight**: depends primarily on the mass of the rocket. Its direction is always perpendicular to the Earth's surface and acts towards the centre of the Earth.

- **Thrust**: depends on the specific engine used, as each engine has a different thrust curve [12]. Thrust is not a linear force, as the force applied to the rocket varies over time. The magnitude and direction of the thrust vector depend on the rocket's attitude, which is one of the main challenges. If the rocket is well-designed and there are no errors, its direction can be considered perpendicular to the Earth's surface and opposite to the weight vector. It is important to remember that thrust is only active for a few seconds, while the ascent phase lasts longer.

- **Drag**: is dependent on the rocket's velocity, making it another non-linear force. Its direction is parallel to the direction of flight and always opposes the rocket's motion.

- **Lift**: is a force perpendicular to the flight direction. It is used to stabilise the rocket through the use of fins, which are specifically designed for the rocket.

We will not consider all forces that a rocket may encounter (Figure 2.2), such as wind force or the deployment of the parachute, as these are either unpredictable or are accounted for before the launch phase to minimise their influence on rocket motion.

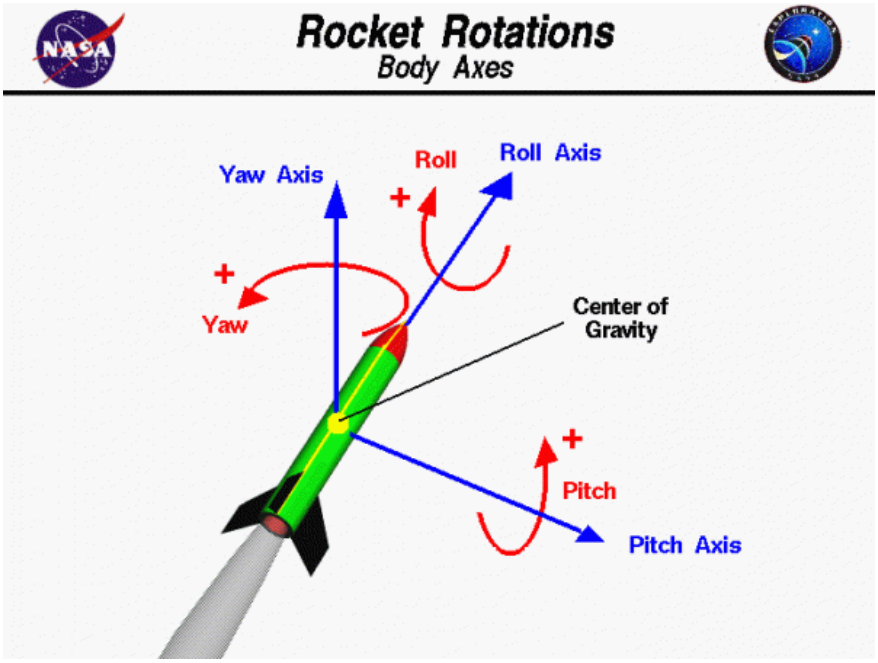


Figure 2.2: Rotations of a rocket. Source: Glenn Research Department, Beginner Guide to Aeronautics [9].

In addition to the forces acting on the rocket, its attitude is an important parameter in rocket flight. In some cases, the rocket needs to make adjustments to its attitude. The two main parameters to consider are roll and pitch. [9]

- **Roll**: is the rotation of the rocket around its longitudinal axis.

- **Yaw**: is the rotation that affects the rocket's vertical alignment, with the nose moving left and right, along its vertical axis.

- **Pitch**: is a rotation that affects the rocket's lateral alignment, with the nose moving up and down along its lateral axis.

These rotations are caused by forces resulting from misaligned engines and fins, fin flexion, or a shifting centre of gravity.

2.2 Apogee Detection

As mentioned previously, apogee detection is a classic problem in rocketry and requires accurate altitude estimation, which presents several challenges.

The primary difficulty lies in the nature of the sensors themselves. Their inherent errors and limitations add complexity to the estimation problem. Let's consider the two main sensors used for altitude estimation: the barometer and the accelerometer.

The barometer is susceptible to noisy and incorrect readings caused by changes in environmental conditions or sensor malfunctions.

Attitude becomes particularly relevant when using a 3-axis accelerometer. Since we only need the acceleration along the z-axis, the rocket's orientation is critical for determining the correct acceleration. Furthermore, the acceleration is zero when the rocket is in free fall, which complicates accurate altitude estimation.

Improving data quality is achieved through the use of filters. As demonstrated in this paper [5], this is neither a simple nor a linear task. In Section 2.4 [5], the authors' testing phase helps explain why conventional filtering approaches are ineffective. These results are particularly interesting because they reveal the real-world problems encountered when working with sensors and why apogee detection is so difficult to calculate accurately. In Section 5.1 [5], the Kalman Filter is proven to be the best option compared to the other proposed methods, showcasing the effectiveness of this mathematical model.

Filtering data is certainly a good method for improving apogee detection, but it is not the only one. In a 2023 paper [15], the authors propose an approach to the problem using unfiltered data, which yields relatively good results for apogee detection. They used a moving average calculation after a simple data preprocessing step. The natural advantages of this approach are its computational speed and low memory footprint, with the obvious drawback of less accurate altitude estimation. Although not filtering data at all is not ideal for our case, using a moving average calculation after the filtering process is a useful technique to achieve even more accurate and memory-efficient apogee detection.

The rocket stages we use as the basis for our design are as follows (Figure 2.3): The use of two parachutes for the Hermes project introduces another challenge:

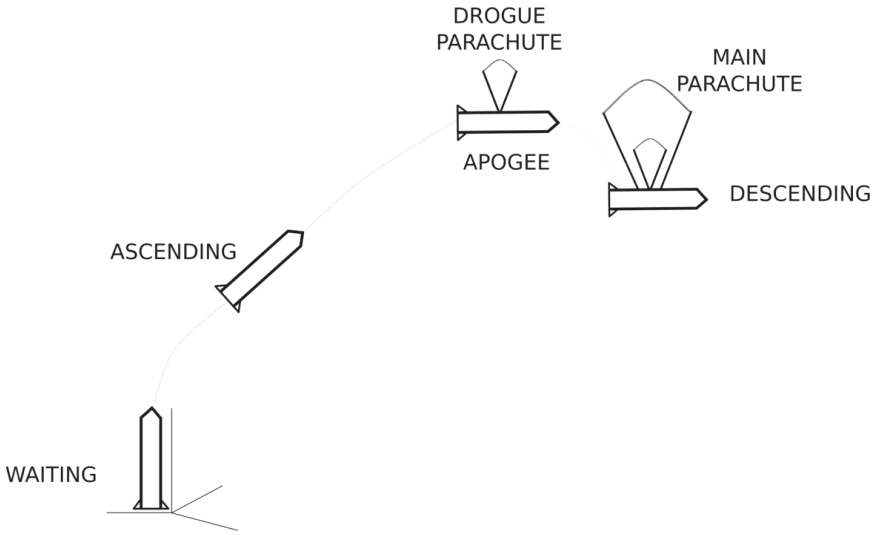


Figure 2.3: Hermes rocket trajectory.

the need for finer altitude precision. The deployment of the main parachute depends on the altitude difference relative to the apogee. Therefore, we not only need to detect the exact moment the rocket reaches apogee but also to correctly detect the second deployment point.

Chapter 3

System Design

System design is one of the most critical steps in any project, as it is the process of identifying and defining the architecture of the components and their relationships. The initial architecture scheme (Figure 3.1) is an abstracted model of the flight computer that highlights how the system works. As we can see, the main functions are data logging and apogee detection, followed by parachute deployment. The first part of this chapter analyzes rocket motion, focusing on

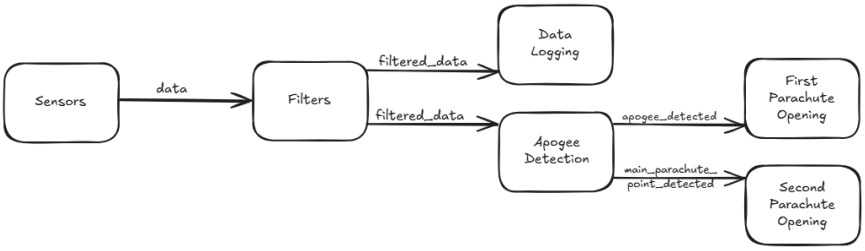


Figure 3.1: High level concept for the flight computer.

the key decisions made to simplify the problem. The second part explores the system architecture, the problem requirements, and the various choices that determine the software and hardware implementation; the decisions made here will influence the subsequent chapters. The third part is dedicated to the Kalman Filter design, which involves creating a model based on the physics of rocket motion and the architectural decisions. This model is also a result of a review of existing literature, consultations with the advisor, and simulations performed using Python code.

3.1 Rocket Domain Specification

Apogee detection is an essential prerequisite for a flight computer in model rocketry. It ensures that the rocket can safely deploy the main parachute at a safe speed and minimises its drift from the launch point. The parachute deployment phases are highly dependent on the rocket's specific metrics and calculations aimed at reducing its speed before impact to avoid damage to the rocket or property [18]. These calculations rely on accurate altitude values, so we need to precisely detect the apogee and avoid both false positives and false negatives.

A typical model rocket uses two parachutes: a drogue parachute to slow the rocket to a safe speed, and a main parachute for the final descent. Apogee detection becomes problematic because sensors are inherently imprecise, and reading errors are common and caused by many factors.

There are three main categories of sensor faults [11]:

- **Inherent Faults**: Occur when measured values differ from a healthy device's measurements. This category is subdivided into:
  - **Constant**: Readings have a consistent value difference.
  - **Short**: A momentary faulty value between two valid ones.
  - **Noise**: Affects subsequent values, increasing the variance of sensor readings.
  - **Drift**: The output of the sensor changes over time, even if the input remains unchanged.
- **Event-Based Faults**: Unexpected events that can affect sensor readings.
- **Isolated Faults**: This category is not relevant to our case, as malicious attacks are not a consideration in model rocketry.

In our case, our sensors (barometer and accelerometer) are particularly affected by noise, which has various causes, including the environment. For example, pressure is not constant over time even when the sensor is stationary, as it is affected by wind and temperature. The accelerometer also has bias issues: even when stationary, the measured value might differ from true gravitational acceleration, and the other axes can show non-zero acceleration.

Event-Based Faults are arguably the most problematic errors that can occur, as they can significantly alter the performance of apogee detection. For example, a large spike in sensor readings could be incorrectly detected as the apogee while the rocket is still in its ascent phase. This could lead to the parachute becoming tangled, damaged, or failing to deploy, depending on the rocket's velocity or dynamics.

This is a real concern based on the analysis of a previous launch by the UniNa Rockets association (Figure 3.2).

It is very clear that these spikes are misleading and create problems in accurately determining the apogee and the main parachute deployment point. Even

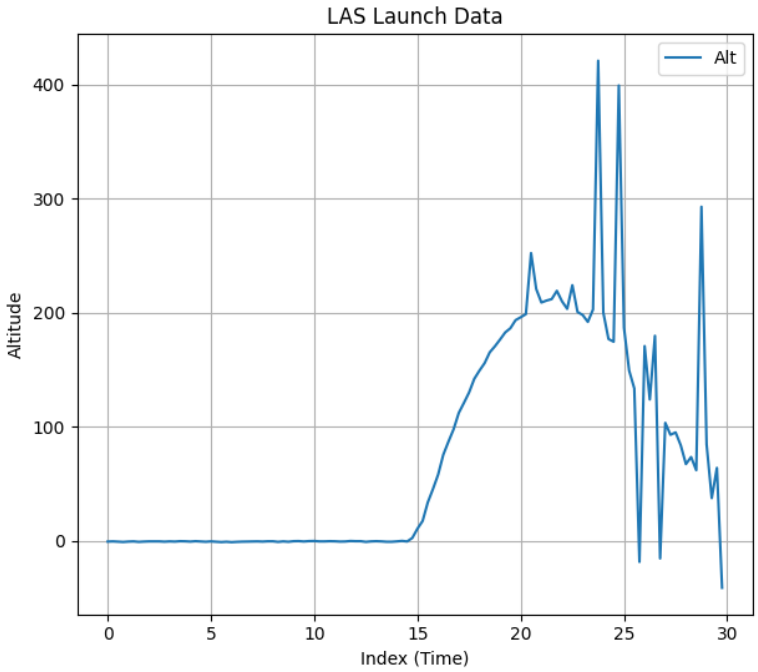


Figure 3.2: Altitude data from LAS-01 first launch.

if the drogue parachute was unaffected by the first spike, the main one could deploy at the wrong altitude or not at all.

Using an accelerometer helps us understand the direction of the rocket and whether it is rising or falling, which can help remove spikes. However, the accelerometer introduces other problems. Due to the nature of the sensor, its orientation affects its reading, and since we only need to operate on the z-axis for apogee detection, the rocket's attitude becomes a relevant factor. The only useful acceleration is the one parallel to the gravitational acceleration. The accelerations on the other axes, while not used for apogee detection, still need to be considered because the rocket's attitude changes during flight.

In scientific literature, this issue is often solved using a gyroscope, which reduces errors and helps to determine the absolute acceleration vector rather than just the relative one read by the sensor. [10]

3.2 System Architecture

After considering all the possible problems related to the system domain of apogee detection, it is time to evaluate the requirements needed to build a well-suited flight computer for our task.

3.2.1 Requirements Engineering

For a better understanding, it is essential to list all the requirements, both functional and non-functional, of our system. This facilitates the design phase and helps to avoid time-consuming iterations resulting from naive design errors and lack of foresight. In every project, the requirements phase is necessary to list all the functionalities and constraints that a system must satisfy. The flight computer (the main system considered in this thesis) has a mix of software and hardware requirements that need to be considered together due to the system's integrated nature.

Functional requirements are the services that a system should provide:

- **Data retrieval**: The flight computer (FC) must retrieve data from sensors.
- **Apogee Detection**: Based on sensor data, the FC must detect the apogee of the rocket's trajectory.
- **Parachute Opening**: The FC must be able to deploy the parachute.
- **Data logging**: Sensor data must be stored for consultation after launch.

Non-functional requirements are the constraints on the services:

- **High-speed**: Calculations and operations must be completed in a short time. High latency can cause significant problems for real-time operations and system actuators.
- **FPU**: A floating-point unit is required for all calculations that require higher precision.
- **Real-time**: The FC must meet real-time constraints on operations due to the critical nature of the system.
- **Low power consumption**: The FC must draw a low current to avoid exhausting the battery before the flight ends.
- **Low memory**: The software needs to run using as little memory as possible.
- **Communication protocols**: The FC must support the main communication protocols (USART and I2C).

- **DMA**: Direct Memory Access (DMA) should be used for communication operations to avoid blocking modes.
- **Data filtering**: Sensor data cannot be assumed to be accurate a priori. Filters are essential to improve reliability and avoid detection errors.

3.2.2 Hardware Design

With the requirements defined, it is now possible to specify the FC's hardware. This phase influences the software design because the program will be specifically tailored for the chosen device (with appropriate abstraction). In this type of application, the software is designed to be adapted to specific hardware with features necessary for the correct functioning on a model rocket or, more generally, on a vehicle.

After the requirement specification, the hardware definition becomes easier. The choice of the main component, the computing unit, is straightforward: a microcontroller is the best option in this circumstance. Personal experience with microcontrollers for a long time helped to make this choice, but they are generally the best option due to their ease of use and lower resource requirements compared to more sophisticated hardware running an operating system.

The chosen microcontroller is the STM32F401RE [17], which is well-documented and widely used in student education due to its versatility and processing power. To meet the architectural requirements, we can check if all the constraints are fulfilled:

- **High-speed**: The Arm 32-bit Cortex-M4 is capable of frequencies up to 84 MHz, with an adaptive real-time accelerator (reducing latency on instruction fetches for the Cortex-M4).
- **FPU**: The Cortex-M4 has a built-in floating-point unit, which supports single-precision data processing.
- **Low power consumption**: In running mode, without considering peripheral consumption, the STM32F401RE uses 146 µA/MHz.
- **Communication protocols**: The microcontroller supports I2C, USART, SPI, SDIO, and advanced connectivity (USB 2.0).
- **DMA**: It has 16-stream DMA controllers with FIFOs and burst support.

The choice of sensors was also guided by experience, as the old flight computer used by the UniNa Rockets association relied solely on the quality of the raw data. At that time, we had not yet figured out how to implement a high-level filter. Therefore, these sensors are embedded in a module called IKS01A3 [16], which contains various sensors. The two that will be used are:

- **LSM6DSO**: A 3-axis accelerometer (±2/±4/±8/±16 g) and 3D gyroscope sensor, supporting I2C and SPI, with an acceleration sensing frequency up to 6.66 kHz and a maximum current draw of 4 mA.
- **LPS22HH**: A barometric sensor (260-1260 hPa), supporting I2C and SPI, with a sensing frequency up to 75 Hz and a typical current draw of 0.012 mA.

The chosen data logging device is the Naze32 F3 Blackbox. As the name suggests, this is a black-box device that logs data from the UART, avoiding the need to configure a file system on the microcontroller. This is a common option used in the avionics field, generally for quadcopters.

The parachute deployment is the final hardware functionality to be implemented. A MOSFET is the chosen solution, which, in combination with a gunpowder charge, creates high pressure within the rocket body to release the parachute.

Finally, it was possible to create a high-level scheme of the flight computer, based on our design (Figure 3.3).

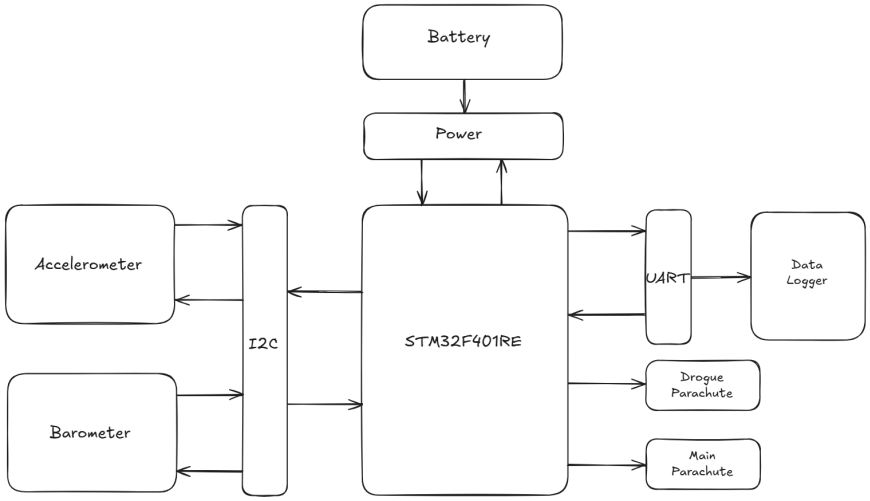


Figure 3.3: High-level flight computer scheme.

The components communicate on shared buses, exchanging information with the microcontroller, which acts as the central computational unit. This unit processes the data and computes the rocket's state.

Rocket state recognition is the main function of the flight computer, along with the deployment of the parachute, to ensure a successful landing of the model rocket.

3.3 Software Design

Due to the critical nature of the environment in which the rocket operates, the software design must adapt to ensure flawless operation under normal conditions and to minimise errors.

Good software design practices were a priority to reduce errors and improve the system's ability to recover from them. A well-written code requires proper planning, and many decisions have been made to simplify implementation and create a modular, scalable, and highly maintainable solution. The solutions adopted are as follows:

- **Kalman Filter library**: Creating a Kalman Filter library specifically for our problem (3.4.1), while ensuring it is modular and adaptable for future changes or improvements.
- **Matrix library**: Not using the standard math library to reduce system memory utilisation, safely manage matrices, and optimise computations.
- **Filters**: Using not only the Kalman Filter but also other filters to enhance the data and reduce faults.
- **Store last altitude data**: Storing multiple recent altitude values to reduce false positives in apogee detection.
- **Real-Time OS**: The use of a real-time OS grants better reliability, system performance, and more readable code. It also provides the opportunity to define critical sections for resource access and to manage interrupts and DMAs properly. These features are crucial for safety-critical operations like a model rocket launch.
- **DMA**: Using DMA to increase CPU idle time, avoiding collisions and missed deadlines when retrieving sensor data.
- **Raw Data Log**: Saving raw data for later analysis after launches. This is particularly useful for comparing the accuracy of the on-board filters against filtering performed on a computer's CPU, and for investigating any issues if the launch fails.

The code structure was represented as a sequence diagram (Figure 3.4) to visualise how functions interact and to serve as an initial software design model.

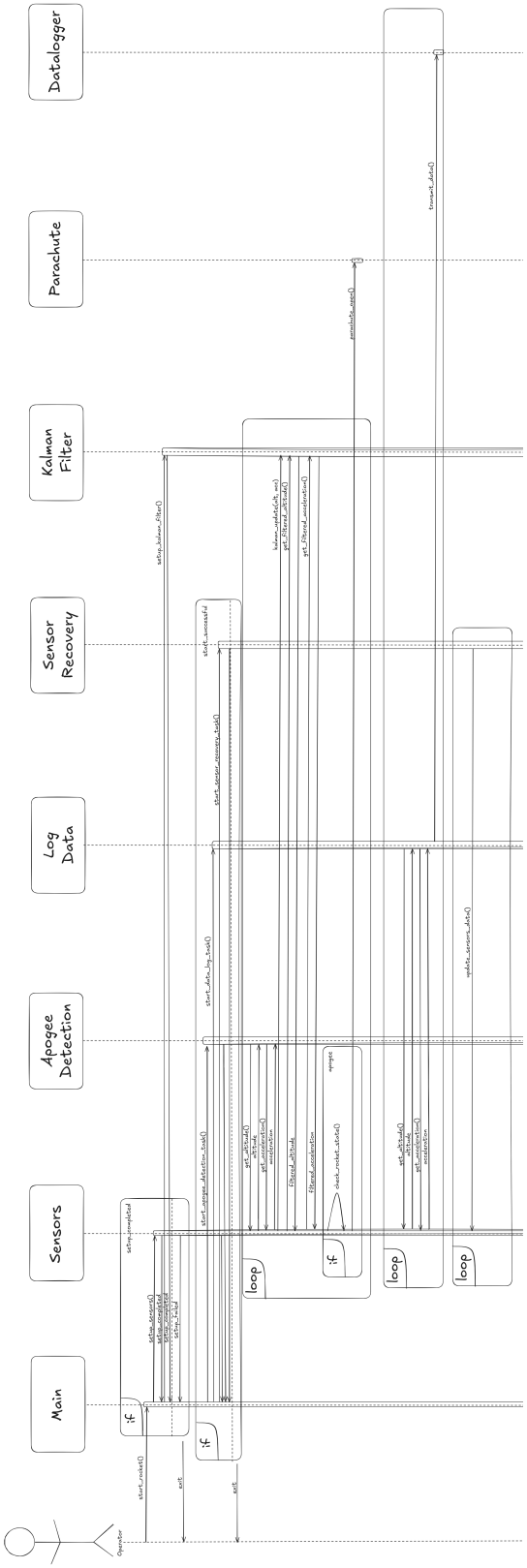


Figure 3.4: Flight Computer code sequence diagram.

This diagram shows how functions work and communicate with each other using shared memory and communication protocols.

3.4 Filters Design

The analysis of the rocket domain and the various hardware components has improved our understanding of the problem. In this section, the focus is on filters. The implemented filters are specifically designed to enhance sensor readings, allowing for faster and more precise apogee detection.

The choice of these filters is the result of experimental tests based on both real data and simulated rocket trajectories. Extensive research was also conducted in the scientific literature to find a suitable solution for this type of application. The results of the filtering process are compared and discussed, and the filters are explained in detail to provide an understanding of their mathematical implications.

3.4.1 Kalman Filter

The Kalman Filter is an algorithm used to estimate the state of the system, based on measurements and statistical noise: the values are enhanced due to the awareness of the filter about the errors and the dynamic motion of the system. In the avionics field, it is well known to be used to estimate the position and attitude [13] of the system. For these reasons, Kalman Filter was not only the obvious solution, but also the easiest and most reliable in a flight computer.

The starting issue is that the barometer returns pressure and temperature, but the altitude is the needed parameter. The formula for converting measured pressure to altitude is given by NASA [14]:

$$h = \frac{T_0}{L_0}\left[\left(\frac{P_0}{P}\right)^{\frac{gM}{RL_0}} - 1\right] \tag{3.1}$$

But, for simplicity, the formula used in the software is as follows:

$$h = \left(\left(\frac{P_0}{P}\right)^{\frac{1}{5.257}} - 1\right)\frac{T + 273.15}{0.0065} \tag{3.2}$$

constants have been replaced, and the only parameters that are left unresolved.

- **h**: altitude relative to the starting point.
- $\boldsymbol{P_0}$: pressure measured at the starting point.
- **P**: raw pressure measured by a barometer.
- **T**: raw temperature measured by a barometer.

The rocket model for this thesis is simpler than the one shown in Chapter 2: the FC just needs to acknowledge his altitude, and the benefit coming from the attitude knowledge is not worth the extra complexity. In a preliminary stage of data analysis, it became clear that using just the cinematic laws was more than enough, and it was not necessary to go on with dynamic models.

So, the rocket motion becomes a single-dimensional model, with the z-axis the only axis considered. Below, there are the cinematic law for the rocket motion:

$$h_k = h_0 + v_0 \cdot t_k + \frac{1}{2} \cdot a_K \cdot t^2 \tag{3.3}$$

$$v_k = v_0 + a_k \cdot t \tag{3.4}$$

$$a_k = a - g \tag{3.5}$$

$$\tag{3.6}$$

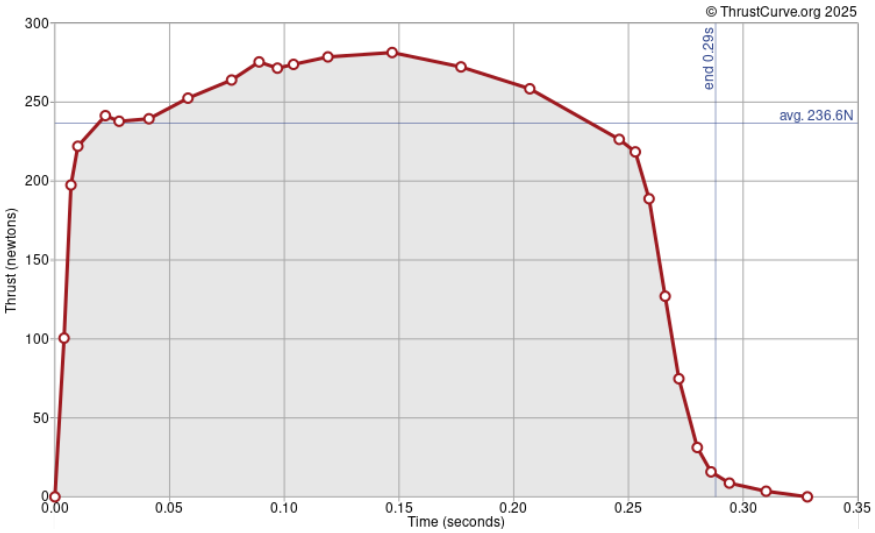


Figure 3.5: Cesaroni 68F240-15A engine thrust curve. [1]

Despite the laws just stated, this is not a uniformly accelerated motion since the engine has a thrust curve that is not linear(Figure 3.5).

This implies that approximations are needed, but we are going to talk about it in the next chapters.

The Kalman Filter equations are listed below:

$$x_k = A \cdot x_{k-1} \qquad \text{State Vector Update} \qquad (3.7)$$

$$P_k = A \cdot P_{k-1} \cdot A^T + Q \qquad \text{Uncertainty Matrix Update} \qquad (3.8)$$

$$y = z_k - H \cdot x_k \qquad \text{Measurement Residual} \qquad (3.9)$$

$$K = P_k H^T (H P_k H^T + R)^{-1} \qquad \text{Kalman Gain} \qquad (3.10)$$

$$x_k = x_k + K \cdot y \qquad \text{State Estimation Update} \qquad (3.11)$$

$$P_k = (I - KH) P_k \qquad \text{Covariance Matrix Correction} \qquad (3.12)$$

The Kalman Filter is made by 6 equations divided in two steps: Prediction and Correction.

- **Prediction**: in this step, Kalman predicts the next altitude value starting from the previous one and updates the uncertainty based on the previous uncertainty and process noise (3.7)(3.8).
- **Correction**: in this step, the prediction is corrected based on a statistical approach, considering sensor noise and updated uncertainty (3.9)(3.10)(3.11)(3.12).

Now, take an in-depth look at what the meanings of these equations are, also explicitly stating the dimensions and their values:

The state vector represents the state of a system at a certain time. It contains parameters tracked by the Kalman Filter: altitude, velocity and acceleration.

$$\mathbf{x} = \begin{pmatrix} h \\ v \\ a \end{pmatrix}$$

The state transition is the model used to track the position of the rocket over-time, and is related to the motion equation (3.3)(3.4)(3.5): important to notice that the state transition matrix differs a bit from the equation, referred to this equation (3.5), because the acceleration considered as Kalman input is the total acceleration.

$$\mathbf{A} = \begin{pmatrix} 1 & \Delta t & \frac{\Delta t^2}{2} \\ 0 & 1 & \Delta t \\ 0 & 0 & 1 \end{pmatrix}$$

The Uncertainty matrix is the estimated covariance of the rocket state representing, so, the uncertainty about the position, velocity and acceleration that are stored in the state vector.

$$\mathbf{P} = \begin{pmatrix} \sigma_h^2 & 0 & 0 \\ 0 & \sigma_v^2 & 0 \\ 0 & 0 & \sigma_a^2 \end{pmatrix}$$

The Process Noise matrix is the uncertainty of a mathematical model, if the model is more or less accurate in describing the rocket motion.

$$\mathbf{Q} = \begin{pmatrix} Q_h & 0 & 0 \\ 0 & Q_v & 0 \\ 0 & 0 & Q_a \end{pmatrix}$$

The Observation matrix tracks the data measured by sensors, used to recover altitude and acceleration, to confront with the measured data.

$$\mathbf{H} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

The Measure Noise matrix expresses the noise in the sensor's readings.

$$\mathbf{R} = \begin{pmatrix} R_h & 0 \\ 0 & R_a \end{pmatrix}$$

Measured Vector contains the data taken by sensors.

$$\mathbf{z} = \begin{pmatrix} z_h \\ z_a \end{pmatrix}$$

3.4.2 Other Filters

Further analysis led me to implement other filters to avoid problems and prevent the Kalman Filter from diverging. Simplifying the rocket's motion also requires some data manipulation before insertion into the filter:

- **Altitude Converter**: The conversion of pressure to altitude was discussed in the Kalman Filter section (3.2).

```
float get_altitude()
{
float pressure = get_pressure();
float temperature = get_temperature();
return ( ( powf( ( 1013.25f / pressure ), 1/5.257f ) - 1.0f
    ) ) * ( temperature + 273.15f ) / 0.0065f;
}
```

- **Low Pass Filter**: This is a standard approach in sensor filtering and is considered to have low computational and time complexity, yielding very good results for cutting out spikes. It's used to filter outlier values and is tuned based on the rocket's expected trajectory.

```
if( altitude - last_altitude_vector[index_altitude_vector] >
     ALTITUDE_ERROR_THRESHOLD )
kf_update( last_altitude_vector[index_altitude_vector],
    acceleration );
else
kf_update( altitude, acceleration );
```

- **Z-Axis Filter**: As the mathematical background chapter showed, the rocket does not remain perfectly parallel to the Z-axis. Thus, simplifying the rocket's motion implies obtaining vertical acceleration without knowing the attitude of the system. A very easy workaround is to simply use the magnitude of the 3-dimensional acceleration vector, since the acceleration on the other axes is typically very low and can be ignored in normal cases.

```
acceleration get_acceleration_taxis()
{
acceleration acc;
acc.ax = (((float)((int16_t)(((uint16_t)buffer_lsm6dso[1] <<
     8) + buffer_lsm6dso[0])) * XL_SENS_FS_8G) / 1000.0f) *
    GRAVITY;
acc.ay = (((float)((int16_t)(((uint16_t)buffer_lsm6dso[3] <<
     8) + buffer_lsm6dso[2])) * XL_SENS_FS_8G) / 1000.0f) *
    GRAVITY;
acc.az = (((float)((int16_t)(((uint16_t)buffer_lsm6dso[5] <<
     8) + buffer_lsm6dso[4])) * XL_SENS_FS_8G) / 1000.0f) *
    GRAVITY;

return acc;
}
```

- **Rocket Acceleration Filter**: This solution comes from a particular feature of accelerometric sensors: their output is zero in free fall. This behavior could cause the Kalman Filter to miscompute the result, trusting the accelerometer too much. To avoid this problem, this non-standard solution was specifically designed for this system domain.

```
1 float get_acceleration_expected_value( float acceleration )
2 {
3 if( acceleration >= 11 )
4 return acceleration;
5 return -9.81;
6 }
```

To summarise how all the filters interact, a summary diagram was created (Figure 3.6): every piece of acquired data passes through a multitude of filters before entering the Kalman Filter. This process helps to prevent the filter from diverging due to faulty values.

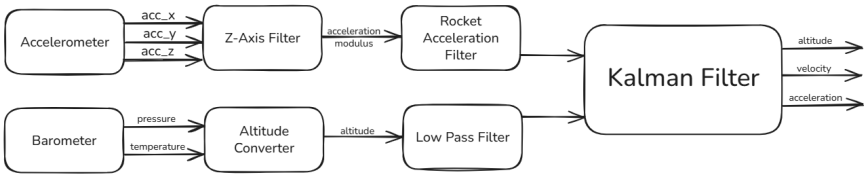


Figure 3.6: Filters scheme used on the flight computer.

3.5 Apogee Detection

As explained in Section 2.2, calculating the moving average is a good method to detect the apogee, even without using filters. However, filters unlock the full potential of this approach; refined data improves accuracy, making this calculation more reliable. There are four states:

- **WAITING**: This is when the rocket is on the launch pad. To avoid premature parachute deployment, the flight computer will not activate the parachute unless the acceleration exceeds a certain threshold.
- **ASCENDING**: The flight computer is actively trying to detect the apogee to deploy the first parachute.
- **APOGEE**: After the first parachute is deployed, the detection point for the second parachute is shifted by a certain amount of meters from the first.
- **DESCENDING**: This is an idle phase where no action is required.

```
1  void check_rocket_state()
2  {
3    float last_altitudes_sum = 0, apogee = 0;
4    for( unsigned short int i = 0; i < SIZE_ALTITUDE_VECTOR; ++i )
5    {
6      last_altitudes_sum += last_altitude_vector[i];
7      if( last_altitude_vector[i] > apogee )
8                  apogee = last_altitude_vector[i];
9    }
10   float avg_altitude = last_altitudes_sum / SIZE_ALTITUDE_VECTOR;
11   switch (rocket_state)
12   {
13     case WAITING:
14       if( rocket_state == WAITING && get_acceleration() >
     ACCELERATION_WAKEUP_THRESHOLD )
15         rocket_state = ASCENDING; break;
16     case ASCENDING:
17       if( avg_altitude + APOGEE_THRESHOLD > last_altitude_vector[
     index_altitude_vector]  )
18       {
19         open_first_parachute();
20         apogee = max(last_altitude_vector, SIZE_ALTITUDE_VECTOR);
21         rocket_state = APOGEE;
22       }; break;
23     case APOGEE:
24                 if( apogee - last_altitude_vector[
     index_altitude_vector] > SECOND_PARACHUTE_ALTITUDE )
25                 {
26                     open_second_parachute();
27                     rocket_state = DESCENDING;
28                 }; break;
29     case DESCENDING: break;
```

```
30                default: break;
31    }
32 }
```

Chapter 4

Embedded System Integration

After designing the flight computer, the embedded system integration step involves integrating the hardware and software systems into a real-world application. While the previous chapters defined the overall system, this chapter will focus on the transmutation of the software design into code, highlighting key decisions made to increase code understandability and maintainability.

A good software implementation is crucial for flawless code execution, especially in cases where fast and optimised code is necessary for low execution time and to reduce task deadline collision.

Memory management is another important part to consider when working on a microcontroller that runs on low memory. For this reason, using proprietary libraries helps minimise memory usage and provides the opportunity to set case-specific boundaries.

4.1 Software Implementation

The purpose of my internship was to prove the effectiveness of Kalman filtering on a microcontroller and the feasibility of porting a Kalman Filter to an embedded system. This involved managing the system with acquired values, filtering, and task scheduling.

Good practices were applied to the software design and turning them into code was a meticulous process. Particular attention was paid to reducing memory consumption and improving security. Faster computation also means lower power consumption, so optimising code execution is not only necessary for task scheduling but also a good way to idle the CPU, lowering the consumption of peripherals and the microcontroller.

4.1.1 Matrix Library

The most straightforward solution to work with matrices was to use the standard GNU GSL library, a very widespread and widely used library. However, a general-purpose solution did not fit our specific problem well, which is why creating this custom library was the best choice, given our focus on optimisation. This basic matrix library is open-source and available on GitHub [2].

The reasons behind the development of this library are:

- **Type**: One of the main issues with the standard C matrix library is that the working type for matrices and operations is double. As shown in the hardware design (3.2.2), the microcontroller is not capable of hardware-accelerated double precision floating-point computation. Since the Floating-Point Unit (FPU) only works on single precision, double precision operations must be emulated via software. Software emulation requires more resources and time, in addition to being less precise.

- **Dimensions**: The Kalman Filter works with many matrices and matrix operations, which require checks on matrix dimensions to avoid memory leaks or stack overflows. Moreover, tracking all dimensions throughout the process is cumbersome with a highly-coupled solution, such as computing Kalman equations directly with loop cycles instead of functions.

```
typedef struct MatrixFloatStruct{
unsigned short int rows;
unsigned short int cols;
float** p_matrix;
}matrix_float;
```

- **Memory**: Another core factor in this decision was memory optimisation. The code went through various iterations to achieve the best possible optimisation. This solution was developed to standardise the double loop (and other operations), as shown in this example:

```
void basic_double_loop_matrices( matrix_float * result,
    const matrix_float * matrix1, const matrix_float *
    matrix2, MatricesBasicOperation operation )
{

for( unsigned short int i = 0; i < result->rows; i++ )
{

  for( unsigned short int j = 0; j < result->cols; j++ )
  {

     operation( &result->p_matrix[i][j], ( const
    matrix_float * ) matrix1, ( const matrix_float * )
    matrix2, i, j);

  }
}

}
```

- **Security**: All allocations and operations are safe and prevent incorrect actions, such as on null pointers or between two invalid matrices. For example, in the matrix product function, the following checks are made:

```
matrix_float* product_matrices( matrix_float * matrix1,
    matrix_float * matrix2 )
{
if( matrix1 == NULL || matrix2 == NULL )
return NULL;

if( !( matrix1->cols == matrix2->rows ) )
return NULL;

matrix_float* matrix = create_empty_float_matrix( matrix1->
    rows, matrix2->cols );

if( matrix == NULL )
return NULL;
```

- **Time**: Time is another important metric for safety-critical systems. For this reason, the direct method for a 2x2 matrix was used for the calculation of the inverse matrix, because its time complexity is O(1), compared to the Gauss-Jordan method, which requires O(n^3).

```
if (matrix->rows == 2 && matrix->cols == 2) {
float a = matrix->p_matrix[0][0];
float b = matrix->p_matrix[0][1];
float c = matrix->p_matrix[1][0];
float d = matrix->p_matrix[1][1];

float det = a * d - b * c;

if (_fab(det) < EPSILON) {
return NULL;
}

matrix_float* inverse = create_empty_float_matrix(2, 2);
if (inverse == NULL) {
return NULL;
}

inverse->p_matrix[0][0] = d / det;
inverse->p_matrix[0][1] = -b / det;
inverse->p_matrix[1][0] = -c / det;
inverse->p_matrix[1][1] = a / det;

return inverse;
}
```

4.1.2 Kalman Filter Library

As was mentioned for the matrix library, there were online solutions similar to the Kalman Filter described in the previous chapter (3.4.1). However, these were either not optimal or diverged from our specified model. Therefore, the only viable solution was to implement the Kalman Filter using the newly created library. This implementation was straightforward, as the library made the code drafting very clear.

Writing both the matrix library and the Kalman Filter library ensured that the solution was perfectly tailored to our system.

The Kalman Filter had three main functions:

- **Initial Setup**: This function initialises the Kalman Filter with starting values for its matrices.

```
void kf_setup_initial_value()
{
//State Vector Initialization
x_pre = create_empty_float_matrix( 3, 1 );

//Transition Matrix Initialization
A = transition_matrix_init();

//Covariance Matrix Initialization
P_pre = covariance_matrix_init();
P_k_n = create_empty_float_matrix( 3, 3 );

//Process Noise Initialization
Q_k = process_noise_matrix_init();

//Observation Matrix Initialization
H_k = observation_matrix_init();

//Measure Noise Matrix Initialization
R_noise = measure_noise_matrix_init();

//Kalman Gain Initialization
K_k = create_empty_float_matrix( 3, 2 );

//Measured and Expected Observation Matrix Initialization
z_k = create_empty_float_matrix( 2, 1 );
y_k = create_empty_float_matrix( 2, 1 );

//Identity Matrix Initialization
I = create_identity_matrix(3);

}
```

- **Kalman Update**: updating matrices with computation of the next predicted state and changing covariance.

```
void kf_update( const float altitude, const float
    acceleration )
```

```
{
   //Temporary matrices to update in safe mode
   matrix_float *x_k_n_tmp = NULL;
   matrix_float *P_k_n_tmp = NULL;
   matrix_float *y_k_tmp = NULL;
   matrix_float *K_k_tmp = NULL;
   matrix_float *x_new = NULL;
   matrix_float *P_new = NULL;


   //Updating Measured Vector
   z_k->p_matrix[0][0] = altitude;
   z_k->p_matrix[1][0] = acceleration;


   //State Vector Update
   x_k_n_tmp = product_matrices(A, x_pre);
   if (x_k_n_tmp == NULL) goto cleanup;

   //Uncertainty Matrix Update
   matrix_float *A_T = transpose_matrix(A);
   matrix_float *AP = product_matrices(A, P_pre);
   matrix_float *APA_T = product_matrices(AP, A_T);
   P_k_n_tmp = sum_matrices(APA_T, Q_k);

   free_matrix_float(A_T);
   free_matrix_float(AP);
   free_matrix_float(APA_T);

   if (P_k_n_tmp == NULL) goto cleanup;

   //Measurement Residual
   matrix_float *Hx = product_matrices(H_k, x_k_n_tmp);
   y_k_tmp = difference_matrices(z_k, Hx);
   free_matrix_float(Hx);

   if (y_k_tmp == NULL) goto cleanup;

   //Kalman Gain
   matrix_float *H_T = transpose_matrix(H_k);
   matrix_float *PH_T = product_matrices(P_k_n_tmp, H_T);
   matrix_float *HPH_T = product_matrices(H_k, PH_T);
   matrix_float *S = sum_matrices(HPH_T, R_noise);

   //Secure Inversion 2x2
   matrix_float *S_inv = inverse_matrix_2x2(S);

   free_matrix_float(HPH_T);
   free_matrix_float(S);

   if (S_inv == NULL)
   {
      // Correction ignored to avoid fault cause by singular
    matrix
```

```
55          x_new = x_k_n_tmp;
56          P_new = P_k_n_tmp;
57          x_k_n_tmp = NULL;
58          P_k_n_tmp = NULL;
59
60          free_matrix_float(H_T);
61          free_matrix_float(PH_T);
62          free_matrix_float(y_k_tmp);
63
64          y_k_tmp = NULL;
65          goto update_state;
66      }
67
68      K_k_tmp = product_matrices(PH_T, S_inv);
69
70      free_matrix_float(H_T);
71      free_matrix_float(PH_T);
72      free_matrix_float(S_inv);
73
74      if (K_k_tmp == NULL) goto cleanup;
75
76      //State Estimation Update
77      matrix_float *Ky = product_matrices(K_k_tmp, y_k_tmp);
78      x_new = sum_matrices(x_k_n_tmp, Ky);
79      free_matrix_float(Ky);
80
81      if (x_new == NULL) goto cleanup;
82
83      // Covariance Matrix Correction (Joseph Form stable)
84      // P_new = (I - K*H) * P_k_n * (I - K*H)' + K*R*K'
85      matrix_float *KH = product_matrices(K_k_tmp, H_k);
86      matrix_float *IKH = difference_matrices(I, KH);
87      matrix_float *IKH_T = transpose_matrix(IKH);
88      matrix_float *term1 = product_matrices(product_matrices(
        IKH, P_k_n_tmp), IKH_T);
89
90      matrix_float *K_T = transpose_matrix(K_k_tmp);
91      matrix_float *KR = product_matrices(K_k_tmp, R_noise);
92      matrix_float *term2 = product_matrices(KR, K_T);
93
94      P_new = sum_matrices(term1, term2);
95
96      free_matrix_float(KH);
97      free_matrix_float(IKH);
98      free_matrix_float(IKH_T);
99      free_matrix_float(term1);
100     free_matrix_float(K_T);
101     free_matrix_float(KR);
102     free_matrix_float(term2);
103
104     if (P_new == NULL)
105     {
106         free_matrix_float(x_new);
107         x_new = NULL;
```

```
108         goto cleanup;
109     }
110 
111 update_state:
112     free_matrix_float(x_pre);
113     free_matrix_float(P_pre);
114     x_pre = x_new;
115     P_pre = P_new;
116 
117     x_new = NULL;
118     P_new = NULL;
119 
120 cleanup:
121     free_matrix_float(x_k_n_tmp);
122     free_matrix_float(P_k_n_tmp);
123     free_matrix_float(y_k_tmp);
124     free_matrix_float(K_k_tmp);
125     free_matrix_float(x_new);
126     free_matrix_float(P_new);
127 }
```

- **Getting values**: computed values from the filter are returned in safe mode, without using pointers and avoiding race condition problems.

```
1  float get_filtered_altitude()
2  {
3      return x_pre->p_matrix[0][0];
4  }
5  
6  float get_filtered_velocity()
7  {
8      return x_pre->p_matrix[1][0];
9  }
10 
11 float get_filtered_acceleration()
12 {
13     return x_pre->p_matrix[2][0];
```

4.1.3 Embedded System

The last part of the software implementation is the integration of the code on board. This step is not just about unifying the code but also about microcontroller management.

The main choice for the microcontroller's software was to adopt the STM32Cube Hardware Abstraction Layer (HAL). This foundational software provides an abstraction layer for the microcontroller, offering APIs for its basic functions, and reducing the need for low-level configurations. This approach provided both an easy-to-use interface for code implementation, a safe and tested environment that reduces human errors in configurations.

FreeRTOS[6] was the second layer used and was selected for a specific reason: to guarantee reliability in a safety-critical system like this. FreeRTOS is an open-source, real-time operating system (RTOS) used on both microcontrollers and microprocessors. This was a core decision to enable multithreading in our system and to avoid memory race conditions that an interrupt-based approach could have caused.

The chosen task scheduling approach was preemptive because it ensures the execution of higher-priority tasks without risking high-priority task starvation. High-priority tasks never enter a Blocked or Suspended state. Preemption becomes crucial because FreeRTOS doesn't have a built-in Rate Monotonic Scheduling. While it would theoretically be possible to use a fixed-priority approach and implement it manually, it was not necessary for our case.

For managing race conditions, mutexes were preferred, even though they are slightly heavier on the CPU, because they are easier to use and debug.

To retrieve data from sensors and prevent the CPU from remaining in a busy-wait state, DMA (Direct Memory Access) was the obvious choice. This further reduces the possibility of task collisions.

The tasks implemented in the software were divided based on their logical function to avoid collisions between functionalities and were assigned priorities based on the code they contained:

- **apogeeDetection**: This is a high-priority task (real-time) because it is responsible for the parachute's deployment. In this task, altitude and acceleration from the sensors are filtered (3.4.2), and the Kalman Filter update function is called. The rocket's state is also checked within this task, and the altitude value is stored in a circular queue and used with an algorithm to detect the apogee.

```
1 void StartApogeeDetection(void *argument)
2 {
3   kf_setup_initial_value();
4   osDelay(1500);
5   for(;;)
6   {
7   const float altitude = get_altitude(),
8       acceleration = get_acceleration_expected_value(
    get_acceleration() );
```

```
9
10    if( altitude - last_altitude_vector[index_altitude_vector]
        > ALTITUDE_ERROR_THRESHOLD )
11      kf_update( last_altitude_vector[index_altitude_vector],
       acceleration );
12    else
13      kf_update( altitude, acceleration );
14
15    filtered_altitude = get_filtered_altitude();
16    filtered_acceleration = get_filtered_acceleration();
17
18    index_altitude_vector = ( index_altitude_vector + 1 ) %
       SIZE_ALTITUDE_VECTOR;
19    last_altitude_vector[index_altitude_vector] =
       filtered_altitude;
20
21    check_rocket_state();
22
23     osDelay(250);
24    }
25  }
26
27
```

- **sensorRecovery**: This is the second highest priority task and is used to acquire data from the sensors. A mutex is indispensable here as it manages the communication flow: the barometer and accelerometer share the same I2C bus. For this reason, when the DMA begins communication with one sensor, the CPU is set to idle mode, waiting for the mutex to be released once the data recovery is complete.

```
1  void StartSensorRecovery(void *argument)
2  {
3    for(;;)
4    {
5      //Read pressure and temperature with DMA
6      HAL_I2C_Mem_Read_DMA( &hi2c1, LPS22HH_ADDRESS,
       LPS22HH_PRESS_XL, 1, buffer_lps22hh, 5 );
7
8      osSemaphoreAcquire( I2cDmaRxSemaphoreHandle,
       osWaitForever );
9
10     //Read acceleration with DMA
11     HAL_I2C_Mem_Read_DMA( &hi2c1, LSM6DSO_ADDRESS,
       LSM6DSO_OUTX_L_A, 1, buffer_lsm6dso, 6 );
12
13     osSemaphoreAcquire( I2cDmaRxSemaphoreHandle,
       osWaitForever );
14
15      osDelay( 20 );
16    }
17 }
18
19
```

- **taskLogging**: The lowest priority is assigned to this task, which performs no critical operations. Its sole purpose is to send data via UART to the datalogger.

```
void StartTaskLogging(void *argument)
{
  float time = 0;
  char buffer_out[250];

  for(;;)
  {
    const float pressure = get_pressure();
    const float temperature = get_temperature();

    sprintf( buffer_out, "t:%.2f,p:%f,T:%f,h:%f,a:%f,fh:%f,
   fa:%f\r\n",
        time, pressure, temperature, get_altitude(),
        get_acceleration(), filtered_altitude,
   filtered_acceleration );

    osSemaphoreAcquire( UartDmaTxSemaphoreHandle,
   osWaitForever );
    HAL_StatusTypeDef status =
        HAL_UART_Transmit_DMA( &huart1, (uint8_t*)buffer_out
   , strlen( buffer_out ) );

    if( status != HAL_OK )
    {
      osSemaphoreRelease( UartDmaTxSemaphoreHandle );
    }

    time += 0.25f;

    osDelay( 250 );
  }
}

```

4.2 Implementation Issues

Two main problems were encountered during code implementation:

- **Singular Matrix**: All matrix operations are safe, consisting of basic mathematical operations like sums, products, or transposing elements, with the exception of matrix inversion. There are some cases where matrix inversion fails due to a phenomenon called a singular matrix, which occurs when the determinant of the matrix is zero. In our code, this returns a null pointer, causing a "NULL POINTER" exception when the task attempts to read the filtered value. To resolve this problem, a safe update for the Kalman Filter was implemented, as shown in the Kalman Filter section (4.1.2, in the update phase).

- **Deadlock**: In the sensorRecovery task, an extra mutex lock was incorrectly added. Since the mutex was configured to wait indefinitely, the task was executed only once and never repeated. This caused the data to not refresh, and the flight computer would be locked on the same data for its entire runtime. Removing the extra mutex lock was sufficient to solve the problem.

Chapter 5

Testing

Testing was the most time-consuming task during my internship, involving analysing data, comparing results, and modifying the code. These iterations were used to fix code errors and obtain better filtered data by adjusting the initial values for the Kalman Filter.

In addition to the Kalman Filter tests, code testing was also performed to verify the functionality of the libraries, which helped in code modification. This process was carried out with every change to ensure that everything worked correctly.

5.1 Code Testing

When it comes to code testing, there are various solutions for different programming languages. C has several major frameworks, such as CUnit. However, after encountering numerous issues with these frameworks, switching to a proprietary solution became the only option, which is why a non-conventional testing suite was developed.

There are various types of tests that can be performed on programming languages, but the two main ones are the following:

- **Black-Box**: This approach involves treating the code as a black box, ignoring its internal structure, and proceeding as an end-user would. This testing is based on considering the possible values a function can take and checking if they work properly.
- **White-Box**: This consists of a teardown of the code, focusing on its structure and creating a representation of the code flow. This can be referred to as statement coverage, branch coverage, or path coverage.

While the White-Box approach can be useful for better understanding a function's flow, the Black-Box approach was preferred for this project because it was easier to implement, more reliable, and simpler to understand.

The Black-Box approach has different types of classification for testing. Equivalence Class is a fundamental part that consists of defining the range of values to be analysed and identifying which values are relevant to the library functions. The methodology used was N-Wect, which involves testing the limit cases to check if the result is the expected one.

An example of these tests can be seen below: **Dimensions**: The Kalman Filter works with many matrices and matrix operations, which requires checking their dimensions to avoid memory leaks or stack overflows. Furthermore, tracking all dimensions throughout the process is cumbersome with a highly-coupled solution, such as computing Kalman equations directly within loop cycles instead of using dedicated functions.

```
1  void invertible_matrix()
2  {
3    matrix_float* matrix_invertible = create_empty_float_matrix( 2,
       2 );
4    matrix_invertible->p_matrix[0][0] = 2;
5    matrix_invertible->p_matrix[0][1] = 7;
6    matrix_invertible->p_matrix[1][0] = 2;
7    matrix_invertible->p_matrix[1][1] = 8;
8
9    matrix_float* matrix_invertible_inverse = inverse_matrix(
      matrix_invertible );
10   if( matrix_invertible_inverse == NULL )
11   {
12     fail += 1;
13     return;
14   }
```

```
15
16    success += 1;
17    return;
18  }
19
20  void not_invertible_matrix()
21  {
22    matrix_float* matrix_invertible = create_empty_float_matrix( 2,
        2 );
23    matrix_invertible->p_matrix[0][0] = 2;
24    matrix_invertible->p_matrix[0][1] = 2;
25    matrix_invertible->p_matrix[1][0] = 1;
26    matrix_invertible->p_matrix[1][1] = 1;
27
28    matrix_float* matrix_invertible_inverse = inverse_matrix(
       matrix_invertible );
29    if( matrix_invertible_inverse == NULL )
30    {
31      success += 1;
32      return;
33    }
34
35    fail += 1;
36    return;
37  }
38
39  void null_invertible_matrix()
40  {
41    matrix_float* matrix_invertible = NULL;
42
43    matrix_float* matrix_invertible_inverse = inverse_matrix(
       matrix_invertible );
44    if( matrix_invertible_inverse == NULL )
45    {
46      success += 1;
47      return;
48    }
49
50    fail += 1;
51    return;
52  }
53
```

5.2 Kalman Filter Testing

Testing the Kalman Filter was a very long and tedious process because it required data to simulate real-life conditions. For this reason, data generated by the Flight Dynamics department of UniNa Rockets, based on simulations of the last launched rocket, was used.

The filter validation process was performed with two programs developed during the internship: one in which the Kalman Filter was executed in Python and another where the filter was executed on the Kalman Filter C library (4.1.2).

5.2.1 Python Testing

The first step was to validate the Kalman Filter model (3.4.1) using a high-level Python program. The following Python libraries were used:

- **tkinter**: This is a library used to create a simple GUI for visualising data and Kalman Filter results.
- **matplotlib**: This is one of the most well-known and widely used libraries in data science, providing a straightforward solution for plotting data.
- **numpy**: This library was responsible for performing the matrix operations and served as a comparison point for the Kalman Filter C library.

The program was divided into three modules:

- **SimpleKFBarometerAccelerometer.py**: This is the main module that runs the primary loop. This part is responsible for displaying the GUI, retrieving data from files, updating the Kalman Filter class with simulated values, and plotting them on the screen. Furthermore, it includes a button to take a screenshot, which was useful for this thesis.
- **KalmanFilter.py**: This is the class that tracks matrices, performs operations between them, and returns filtered values.
- **KalmanFilterConstants.py**: The initial setup values for the Kalman Filter are stored in this file. This was an important part of my testing process because the tuning of the filter involved changing values in this file. Now, let's have a look at the values.

  Starting with **Initial Uncertainty** to setup the **Covariance Matrix**

```
1 #Initial Uncertainty
2 DELTA_H_0_INIT = 1;
3 DELTA_V_0_INIT = 10;
4 DELTA_A_0_INIT = 100;
5
```

```
#Three sigma rule
RHO_SQUARED_H_INIT = pow( ( DELTA_H_0_INIT / 3 ), 2 );
RHO_SQUARED_V_INIT = pow( ( DELTA_V_0_INIT / 3 ), 2 );
RHO_SQUARED_A_INIT = pow( ( DELTA_A_0_INIT / 3 ), 2 );
```

Important to note that the three-sigma rule [7] was used. In statistics, this rule is related to the Normal distribution (or Gaussian distribution).

The values chosen for the **Process Noise Matrix** were derived from considerations about the trust in our model. In theory, many factors are not considered in our simplified model, so it could have been good practice to increase the values, indicating low confidence in the model. However, through testing, it became clear that trusting the model was a better decision.

```
H_NOISE = pow( 0.01,  2 ); #Expected 0.01 m^2 of noise for
    altitude
V_NOISE = pow( 0.02,  2 ); #Expected 0.04 m^2/s^2 of noise
    for velocity
A_NOISE = pow( 0.001, 2 )
```

Last is the **Sensor Noise Matrix**, used to consider the noise added by sensor to measured values:

```
#RMS Noise of sensor
RMS_Altitude = 0.06; #Meters
RMS_Acceleration = 0.003; #Meters per squared second

#Covariance matrix of measure noise
R_NOISE_Altitude = pow( RMS_Altitude, 2 );
R_NOISE_Acceleration= pow( RMS_Acceleration, 2 );
```

These values where taken directly to producers datasheet[16].

5.2.2 C Testing

Similar to the Python-based Kalman Filter testing (5.2.1), a Python GUI was used as the user interface for the C testing. This program was created with the intention of testing the functionality of the Kalman Filter library with the C Matrix library. This part was particularly useful because it allowed me to find errors in the code execution and function calls.

This program differs from the previous one (5.2.1) in that the Kalman Filter runs directly in C, and Python is used only for plotting the data. The workflow of this program is as follows:

- **Compiling C source code**: After starting the Python script, clicking on the "Run C and Plot" button begins the process of compiling all the necessary C files.
- **Executing Kalman Filter**: The code automatically runs after compilation and filters the values, returning the filtered data as a CSV file.
- **Plotting data**: The data are then plotted, making it possible to view the filtered data compared to the raw data.

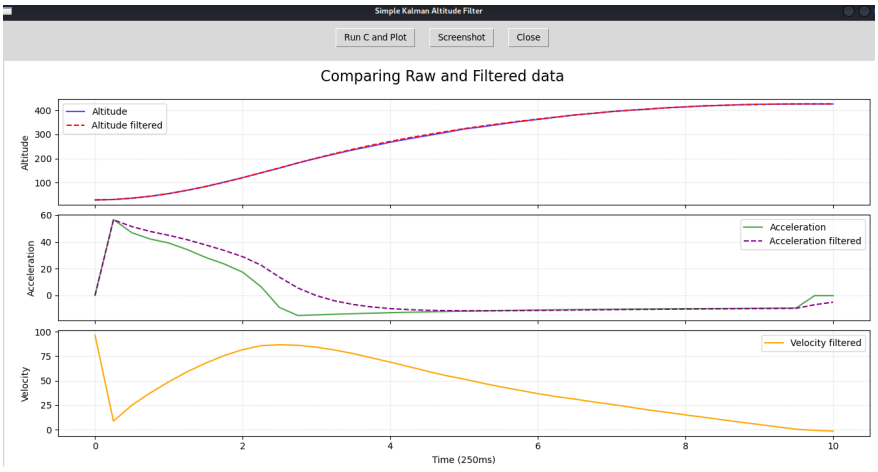


Figure 5.1: Simple Altitude Kalman Filter main window.[3]

5.2.3 Test Results

The test results were also used for the software design because the approach utilised was Test-Driven Development [4]. This involved starting with a failed test and then implementing the code to pass the test. This methodology was particularly useful in the filter design and led me to make crucial decisions based on the results of altered data. For example, the low-pass filter or the rocket acceleration filter (3.4.2). To demonstrate the effectiveness of this filter, simulations were conducted, and the results are shown below.

In normal conditions, with a sampling time of 0.25 seconds, the values read by the sensors are close to the true value, and the resulting Kalman-filtered values show high convergence with the measured values (Figure 5.2).

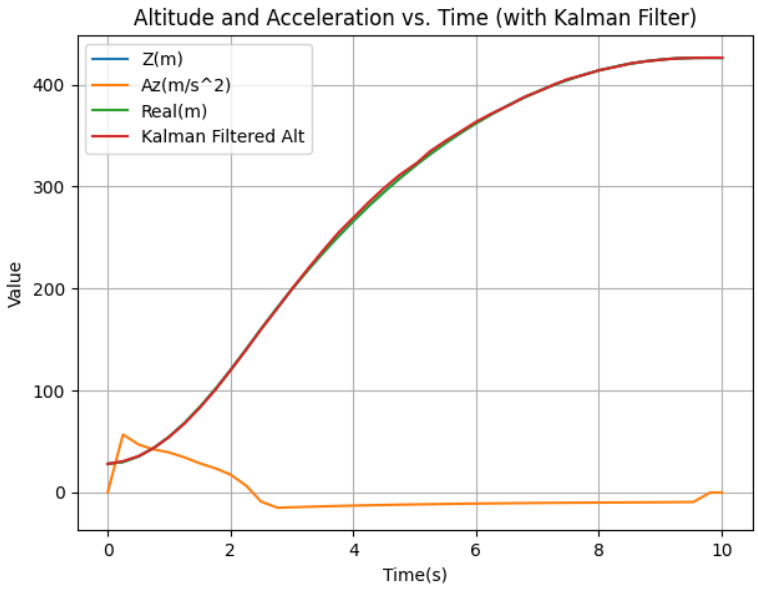


Figure 5.2: Optimal case with measurements close to real values.

However, this scenario is not representative of a real-world situation because many sources of error are ignored, such as sensor noise or pressure changes due to parachute deployment. For this reason, adding random spikes to the input data helped to better test the Kalman Filter. As can be seen (Figure 5.3), the Kalman Filter now diverges from the rocket's trajectory, making the filtering completely unreliable.

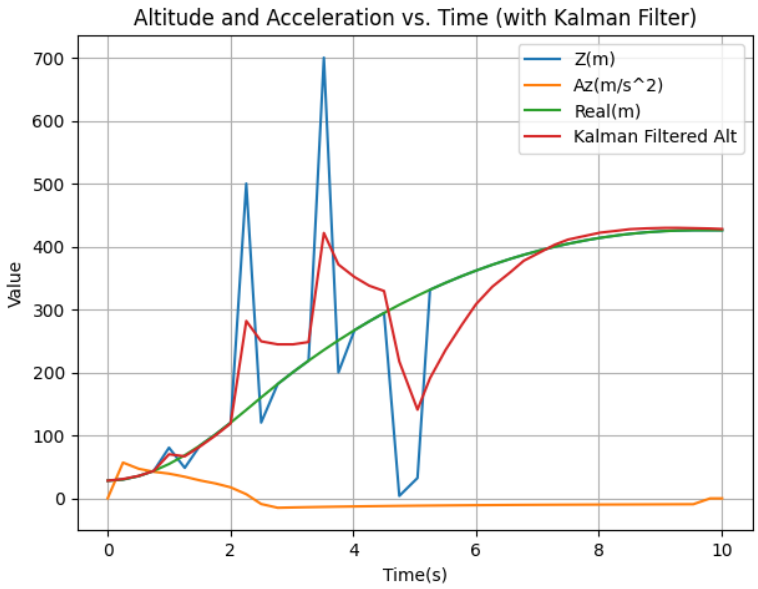


Figure 5.3: Random spikes added to the optimal case.

Solving this problem with a low-pass filter was the simplest and fastest solution, making the filtering process more stable and reliable. As you can see (Figure 5.4),

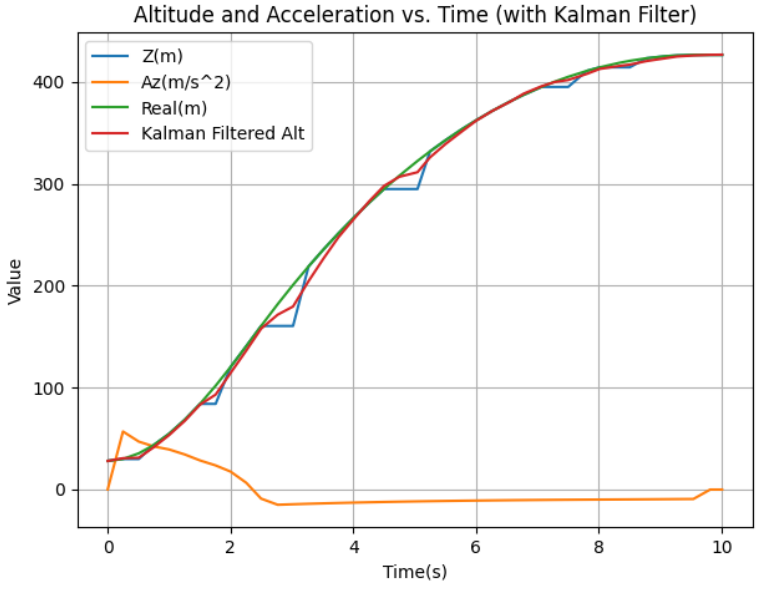


Figure 5.4: Low-pass filter implemented to reduce spikes.

the filter now works much better, with both noise reduction and spike reduction.

5.3 Real-world Validation

The final step of the internship was to test the flight computer in a real-world scenario, specifically to test the effectiveness of the software in a controlled environment. Initially, the intention was to test it with a model rocket launch, but this became a ballistic launch. Consequently, with no real flight scenario available, the controlled scenario became the only viable option, in the hope that a full flight test will one day be possible.

It is important to note that even if these tests do not represent a real flight scenario, the software behaviour is accurately tested. This is especially true given that as the rocket gets close to apogee, its velocity is very low, and this is the moment when filtering is crucial to get values as close as possible to the rocket's true altitude.

5.3.1 Bucket Testing

As a first experiment, the flight computer was tested using a bucket attached to a cord. To simulate the rocket's ascent and descent, the bucket was moved up and down from a balcony, simulating the rocket's trajectory. Although this might raise doubts about the quality of the experiment, it is important to note that when a rocket approaches the apogee, its velocity is very low. The nonlinear changes caused by the rocket's structure (e.g., parachute deployment) can be compared to the bucket's "bumps" caused by sudden changes in hand movement. As we can see (Figure 5.5), after an initial divergence caused by the uncertainty

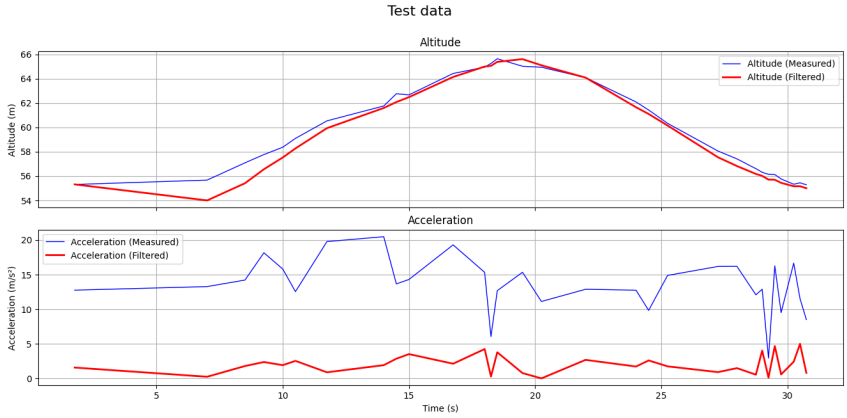


Figure 5.5: Data from the bucket test.

of the first values read, the filter converged to the measured values, returning a better trajectory than the raw data.

It is important to state that the experiment was conducted without tuning the Kalman Filter for the bucket's specific trajectory, so the initial divergence is to be expected. Furthermore, the sampling rate was too fast for the linear motion of the bucket and needed to be lowered to improve the filtering results. However, the test was overall successful.

Chapter 6

Conclusions

The apogee detection for a model rocket is a problem with no definitive solution that is optimal in all situations. In this thesis, we explored many different solutions for our problem, always taking into account the specific context.

In fact, the foundational knowledge, supported by references in scientific literature, helped us frame the behaviour of the rocket model, laying the foundation for the design. Furthermore, a key part of the analysis was understanding that the rocket's attitude was not only relevant but also crucial.

The design phase is one of the most delicate tasks, especially for systems that require extensive testing to anticipate all possible issues. This is why Test-Driven Development proved to be an excellent option for designing and modelling the software based on all possible test outcomes, iterating until the results were satisfying. In this way, problem domain-specific filters (3.4.2) were conceived based on the paths discovered by testing. The software and hardware were strictly related because the rocket's physical resources were limited in terms of space and weight, and the hardware's limitations required a fine-tuned software design made specifically for that hardware.

The code implementation of the libraries was one of the easiest phases, requiring very few code fixes or debugging sessions. Even though the libraries themselves ran fairly smoothly, testing the Kalman Filter library on the microcontroller required a significant amount of time due to an issue with the inverse matrix function, which returned a null pointer when passing a singular matrix. This problem was caused by misleading testing that did not account for the case of a nearly motionless rocket (the WAITING state) and also by the FPU approximation in floating-point calculations. Implementing a safe update for the matrices was a successful workaround and demonstrated that this is a solid approach for real-life scenarios.

There are also several limitations to this thesis that could be improved in the future. Starting with the mathematical model itself, which presents a significant opportunity to substantially improve the Kalman Filter's precision. Moreover, a real-world flight scenario would have provided invaluable feedback, but this was not possible due to technical difficulties.

Indeed, real-life validation showed that the thesis's core premise was proven:

a Kalman Filter for altitude estimation is feasible on an embedded system, improving apogee detection without requiring more complex systems.

6.1 Open Issues

While the project exceeded expectations and met all initial requirements, it is not free from problems. These are a collection of known open issues.

- **Acceleration approximation**: The acceleration filter used to account for the rocket's attitude significantly reduces the Kalman Filter's accuracy when the rocket is near apogee.
- **Preemptive scheduling**: This is not the optimal choice for this type of real-time system. A fixed-priority approach with Rate Monotonic Scheduling would reduce power consumption and is clearer during the development phase, as it ensures the execution of all tasks, not just the high-priority ones.
- **Limited failure recovery**: There are very few error recovery procedures in place to prevent a complete system failure mid-flight. This is a significant problem for applications that require high levels of reliability.
- **No real model rocket test**: This was caused by a complete structural failure of the Hermes launch, which led to the disappearance of the flight computer.
- **Null pointer exception**: For a more robust Kalman Filter, some null pointer checks are missing and need to be implemented for a fully safe update. Matrix initialisation also needs a fail-safe update mode to prevent a failure in the initialisation sequence.
- **Dynamic Allocation**: In an RTOS, using dynamic allocation is not considered good practice because it fragments memory and cannot be checked before runtime.

6.2 Future Development

These are possible ideas for future additions to this project:

- **Less powerful microcontroller**: The STM32F401RE handles the problem without any time or memory issues, so it would be interesting to transition to a less powerful microcontroller to see if the software runs as smoothly. This could lead to lower power consumption and a reduction in production costs.
- **EKF**: An Extended Kalman Filter (EKF) could be used not only to track the rocket's altitude but also its attitude, improving the model's performance and reducing process noise.
- **GNSS/GPS**: It could be interesting to analyse the differences between pressure-calculated altitude and GNSS/GPS altitude and implement it in a Simple Kalman Filter without changing the model.
- **More peripherals**: Adding more peripherals could create scheduling problems or require more race-condition handling, increasing the overall system complexity.
- **Different sensors**: Using different sensors would require different values for the Measure Noise Matrix and could lead to a different Kalman Filter tuning.
- **ML**: Apogee detection could also be solved with a machine learning model trained for binary classification. This could provide an interesting comparison between the moving average model discussed in the apogee detection (3.5) and a machine learning model.

Chapter 7

Bibliography

[1] Cesaroni Technology. *Cesaroni 68F240-15A*. URL: https://www.thrustcurve.org/motors/Cesaroni/68F240-15A/ (visited on 08/16/2025).

[2] Di Donato Andrea. *C Matrix Float*. URL: https://github.com/D1D02/c-matrix-float (visited on 08/19/2025).

[3] Di Donato Andrea. *Simple Altitude Kalman Filter*. URL: https://github.com/D1D02/SimpleAltitudeKalmanFilter (visited on 08/21/2025).

[4] Dua Agha, Rashida Sohail. *Test Driven Development*. URL: Test%20Driven%20Development%20and%20Its%20Impact%20on%20Program%20Design%20and%20Software%20Quality:%20A%20Systematic%20Literature%20Review (visited on 08/21/2025).

[5] Evan Dougal, Julia Kwok, Edward Luckett. *Digital Detection of Rocket Apogee*. URL: https://repository.rice.edu/server/api/core/bitstreams/0fbefe49-d831-4891-b83f-d8c2f05ce2f9/content (visited on 08/28/2025).

[6] freertos.org. *FreeRTOS website*. URL: https://www.freertos.org/ (visited on 08/21/2025).

[7] Friedrich PUKELSHEIM. *The Three Sigma Rule*. URL: https://d-nb.info/1186632208/34 (visited on 08/21/2025).

[8] Glenn Research Center. *Forces on a Rocket*. URL: https://www1.grc.nasa.gov/beginners-guide-to-aeronautics/four-rocket-forces/ (visited on 08/10/2025).

[9] Glenn Research Center. *Rotations on a Rocket*. URL: https://www1.grc.nasa.gov/beginners-guide-to-aeronautics/rocket-rotations/ (visited on 08/10/2025).

[10] Håvard Fjær Grip, Thor I. Fossen, Tor A. Johansen, Ali Saberi. *Attitude Estimation Using Biased Gyro and Vector Measurements With Time-Varying Reference Vector*. URL: https://www-robotics.jpl.nasa.gov/media/documents/Grip_Attitude2012.pdf (visited on 08/13/2025).

[11] Kamali Gupta,Deepali Gupta, Shalli Rani. *An Insight Survey on Sensor Errors and Fault Detection Techniques in Smart Spaces*. URL: https://www.researchgate.net/publication/377052102_An_Insight_Survey_on_Sensor_Errors_and_Fault_Detection_Techniques_in_Smart_Spaces (visited on 08/12/2025).

[12] Penn Kim and Slaton William. *Measuring Model Rocket Engine Thrust Curves*. 2010.

[13] Md Samiul Haque Motayed. *Sensor Fusion for Drone Position and Attitude Estimation using Extended Kalman Filter*. URL: https://www.researchgate.net/publication/393590343_Sensor_Fusion_for_Drone_Position_and_Attitude_Estimation_using_Extended_Kalman_Filter (visited on 08/16/2025).

[14] NASA. *U.S. Standard Atmosphere, 1976*. URL: https://www.ngdc.noaa.gov/stp/space-weather/online-publications/miscellaneous/us-standard-atmosphere-1976/us-standard-atmosphere_st76-1562_noaa.pdf (visited on 08/16/2025).

[15] Paul W. Munyao, Patrick I. Muiruri, Shohei Aoki. *Utilizing Moving Average for Apogee Detection inUnfiltered Data: An Analysis*. URL: https://sri.jkuat.ac.ke/jkuatsri/index.php/sri/article/view/692/478 (visited on 08/28/2025).

[16] STM32 microelectronics. *IKSO1A3 module specifications*. URL: https://www.st.com/en/ecosystems/x-nucleo-iks01a3.html (visited on 08/16/2025).

[17] STM32 microelectronics. *STM32F401RE datasheet*. URL: https://www.st.com/resource/en/datasheet/stm32f401re.pdf (visited on 08/15/2025).

[18] Tim Van Milligan. *Increasing the Descent Time of Rocket Parachutes*. URL: https://www.apogeerockets.com/downloads/Technical_Publications/Tech_Pub_03.pdf (visited on 08/10/2025).

[19] UniNa Rockets. *Hermes Rocket Model*. URL: https://uninarockets.it/projects.html (visited on 08/28/2025).

[20] UniNa Rockets. *UniNa Rockets Website*. URL: http://www.uninarockets.it/ (visited on 08/21/2025).