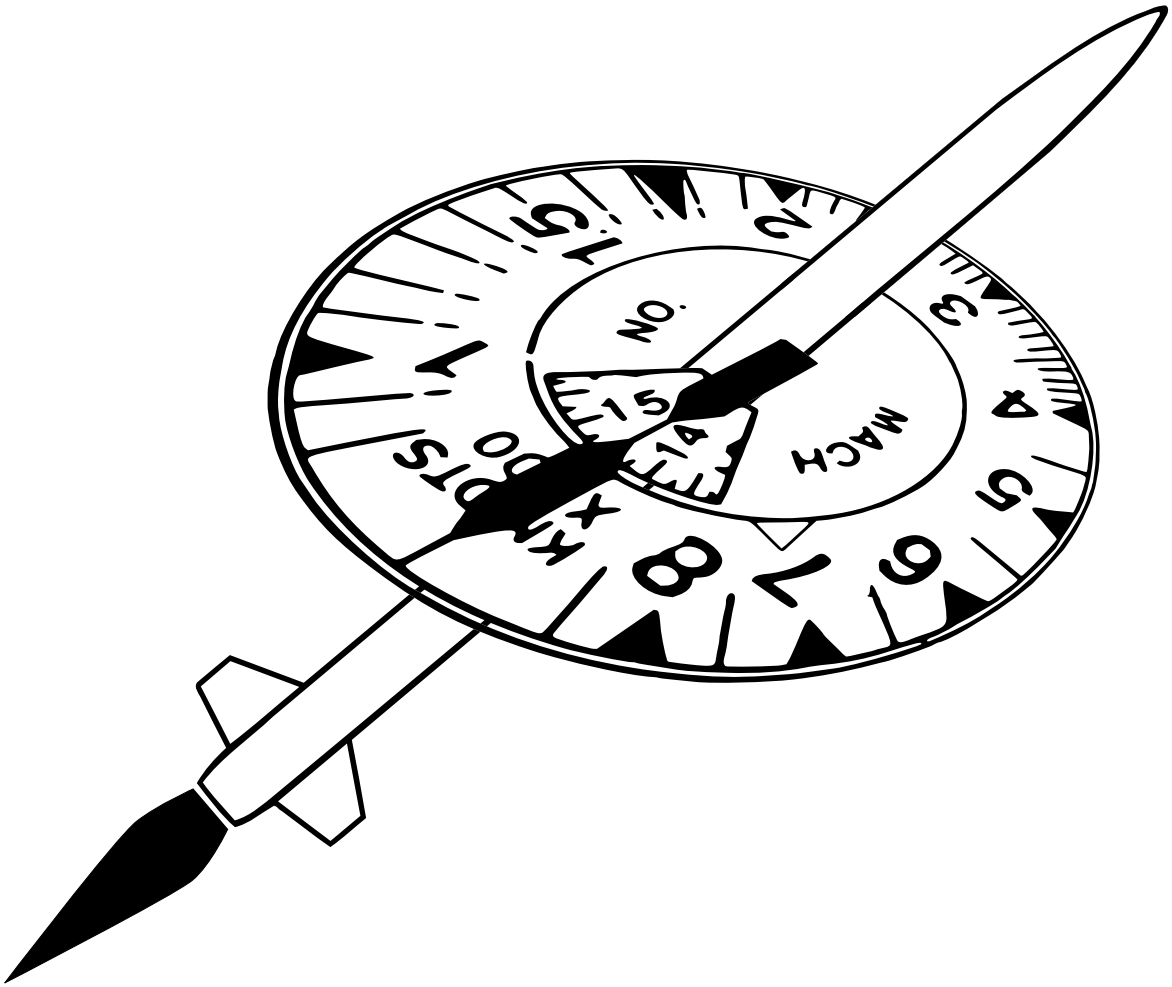
Lucerne University of Applied Sciences & Arts

ARIS: Data Fusion for Sounding Rocket

Semester Thesis



Author:	Aron Schmied
Department:	Electrical Engineering
Supervisor:	Prof. Marcel Joss
Expert:	Prof. Thomas Hunziker
Industrial Partner:	Internal
Submission Date:	December 21, 2018
Classification:	Access

Declaration

Hereby, I declare that I have composed the presented paper independently on my own and without any other resources than the ones indicated. All thoughts taken directly or indirectly from sources are properly denoted as such. This paper has neither been previously submitted to another authority nor has it been published yet.

Horw, December 21, 2018

Abstract

ARIS, a collaboration of students, is participating in the Spaceport America Cup in summer 2019 where a sounding rocket has to be developed and built which reaches an apogee of exactly 10'000 feet. Information about its altitude and velocity in vertical direction needs to be accessible at any time.

A Bachelor thesis from 2018 proposed a sensor fusion algorithm, implemented as a Kalman filter, which estimates the current state of the rocket. The aim of this study is to improve the proposed algorithm in terms of performance, computing time and memory usage and to port the improved algorithm from Matlab onto an STM32 ARM Cortex M4 microcontroller. Furthermore, the implementation needs to be verified and a concept on determing the algorithm parameters must be established.

A different system model than initially proposed was derived to increase the feasibility of porting the algorithm and to reduce the consumption of resources. Furthermore, sequential Kalman filtering was applied to further decrease the number of additions and multiplications and hence decrase computing time. In order to verify the algorithm and determine the parameters, a Matlab script was implemented which can be used to execute these tasks.

Tests were conducted in Matlab and on the embedded system to verify the system model and the firmware implementation as well as measurements to determine the resource consumption on the embedded system. The results have shown that the presented system model can be applied, however, the parameters need to be set according to the sensors that are ultimately used. Also, a reduction in computing time and stack usage by 94% and 83%, respectively, was achieved compared to the proposed algorithm.

Contents

List of Figures

List of Tables

Abbreviation

AGL	altitude above ground level
AR Model	autoregressive model
ARIS	Akademische Raumfahrtinitiative Schweiz
BCS	best case scenario
ENU	east, north and up coordinate frame
EPFL	École Polytechnique Fédérale de Lausanne
ETH Zurich	Swiss Federal Institute of Technology in Zurich
FPU	floating point unit
GPS	Global Positioning System
HMI	human-machine interface
HSLU	Lucerne University of Applied Sciences & Arts
IDE	integrated development environment
IMU	inertial measurement unit
MCU	microcontroller unit
NED	north, east and down coordinate frame
OS	operating system
RAM	random-access memory
RTOS	real time operating system
STM	STMicroelectronics
WCS	worst case scenario

List of Symbols

Measurement and State Variables

δh_{GPS}, δh_{p_1}, δh_{p_2}, δa	disturbance on measurements
ω_x, ω_y, ω_z	angular rate components
$\sigma^2_{a_z}$, $\sigma^2_{a_{off}}$, σ^2_p, $\sigma^2_{hp_1}$, $\sigma^2_{hp_2}$	variances of measurement and system noise
φ	pitch angle
a, a_x, a_y, a_z	acceleration
a_{off}	acceleration measurement offset
h	altitude
h_{GPS}	measured GPS altitude
KPv	pressure gradient
p, p_1, p_2	barometric pressure
p^n, x_E, x_N, x_U	position in the local Cartesian navigation frame
p_x, p_y, p_z	GPS position
v, v_z	velocity
w_{a_z}, $w_{a_{off}}$, w_p	system noise components

Mathematical Variables

$\hat{P}_k$	error covariance matrix of prediction
$\hat{x}_k$	predicted system state
$\tilde{P}_k$	error covariance matrix of estimation
$\tilde{x}_k$	estimated system state
A, B, G, H	continuous time system matrices
A_d, B_d, G_d	discrete time system matrices
K, K_k	continuous and discrete Kalman gain
P, P_k	continuous and discrete error covariance matrix
Q, Q_k	continuous and discrete system noise covariance matrix
Q_B	observability matrix
R, R_k	continuous and discrete measurement noise covariance matrix
r_k	innovation

T_s	algorithm iteration rate
u, u_k	continuous and discrete time system input vector
v, v_k	continuous and discrete time measurement noise vector
w, w_k	continuous and discrete time system noise vector
x, x_k	continuous and discrete time state vector
y, y_k	continuous and discrete time measurement vector

Atmospheric Pressure Model

g	gravitational acceleration
M	molar mass of earth's air
p_0	pressure at ground level
R	universal gas constant
T_0	temperature at ground level
T_{grad}	temperature gradient for actual weather conditions

1 Introduction

In summer 2019, the ARIS (Akademische Raumfahrtinitiative Schweiz), a collaboration of students from the ETH in Zurich and Lucerne University of Applied Sciences & Arts (HSLU), is participating for the second time in the Spaceport America Cup. This competition is executed in different categories, but the main goal consists of developing, building, and launching a sounding rocket.

With this year's project called "HEIDI", ARIS is competing in the 10'000 ft. commercial-off-the-shelf solid rocket engine category. This means, the developed rocket has to reach a target apogee of exactly 10'000 ft. above ground. Furthermore, the rocket must have a payload capacity of 8.8 lb. and provide real time telemetry.

In terms of the competition, there are points awarded for the entry form, the progress updates, the technical report, design, implementation, and the flight performance. With 500 points out of 1'000 in total, the flight performance is weighted of highest importance. These 500 possible points are further divided into 150 points for the successful recovery and 350 points for the actual apogee reached by the rocket with one point loss every 2.5 meters off target [1].

As seen, reaching the target apogee of 10'000 ft. is vital for the success of this project; hence a sophisticated solution must be implemented.

The current approach is to aim at a slightly higher altitude than 10'000 ft. by burning the engine and then using air brakes on the rocket to control the trajectory to the target apogee. To reliably control the trajectory of the rocket, the control algorithm needs accurate information about the current altitude and velocity. In the last project, this was done by integrating the measured acceleration once for the speed, respectively twice for the current altitude. This approach, however, is inaccurate due to the fact that the high vibration during the engine burn phase disturbs the sensor and the measurement error accumulates over time because of the integration. The solution to this is to use multiple sensors which measure different physical quantities such as absolute pressure, position with GPS and acceleration. An algorithm known by the name of "state estimator" can then be used to fuse these sensor readings together to produce more reliable and more precise information than each sensor can produce on its own.

For this reason, Michael Kurmann described and compared different versions of state estimators in his Bachelor thesis at Lucerne University of Applied Sciences & Arts in 2018 [2]. By implementing and testing those algorithms in Matlab, he concluded that a simple, discrete Kalman filter would fulfil the requirements given by ARIS. However, he also addressed unsolved questions concerning the accuracy and neglect of certain factors which are different in a real environment from how they are described in his model.

The aim of this industrial project is to improve the algorithm, which was derived by Kurmann, and to port it to an embedded system, so it can be used in next year's competition.

1.1 Task Description

The general task of improving and porting an already existing sensor fusion algorithm from Matlab can be divided into four sub-tasks which were approached in the methodology section.

Firstly, knowledge had to be gathered to solve the stated problem. A summary of the applied concepts and terms is provided in chapter 2.

Secondly, chapter 3 focuses on the improvement of the proposed algorithm in Matlab. The proposed algorithm is described and known issues are discussed in section 3.1. In section 3.2, an improved algorithm is presented which resolves the stated issues.

Thirdly, the process of preparing the algorithm in Matlab for converting it into C is described. For this conversion, a tool from Matlab is used. The code was optimized first and then formatted the right way, so it could be converted. Furthermore, the algorithm needed to be embedded into the firmware architecture used on the microcontroller. The used OS (operating system), the interfaces to other running tasks on that OS, the allocated memory and the realization of the timing issue were all points which needed to be considered when executing this step. Additionally, a concept on how to deal with sensor failure as well as an implementation of the detection of the current flight phase were part of this step. This is described in chapter 4.

In a final step, a concept had to be derived on how to choose the parameters of the sensor fusion algorithm. Furthermore, the correctness of the algorithm had to be verified. This included not only the algorithm in Matlab, but also the implementation on the embedded system. Since a real device test could not be conducted, an acceptable simulated test environment had to be developed where simulated and real cases could be tested. These steps are covered in chapter 5.

1.2 Requirements

Since the concept of project HEIDI did not fully exist during the course of this work, assumptions concerning the requirements had to be made. These assumptions are based on the previous project. Furthermore, requirements could be deduced from Kurmann's Bachelor thesis, respectively the requirements given by the competition.

System Load

The system load describes how many cycles per second the algorithm can use to calculate the state of the rocket from the given sensor readings. This limitation exists due to a finite frequency of the processor which can perform only a limited amount of operations per second. Additionally, the core has to be shared with the OS, the sensor sampling, the data logging and the telemetry.

The processor in use is a 32-bit ARM Cortex M4 with floating-point unit running at 168 MHz. Assuming the algorithm can use one third of the computational power, 56 million clock cycles can be used per second. However, reaching this value should be avoided.

Precision

Since every 2.5 meters off target result in a point deduction, the goal is to have a precision which is smaller than that. Since the final precision depends on many factors, such as the sensors used and the accuracy of the control algorithm, it is hard to define a number which describes the precision of the algorithm. Furthermore, it makes little sense to give a conclusion about the accuracy of the sensor fusion algorithm by using simulated sensor value. Section 5.1 addresses this difficult topic on how to verify the derived solution without knowing what the optimal estimation is.

However, what can be defined is the average difference between the results from the implementation on the embedded system and the results from the Matlab algorithm. This difference should not exceed 1 cm in average.

Modularity

The implemented solution needs to be expandable and replaceable. This is important because the improvement of the algorithm, either in this project or in a future genaration of ARIS, has to be granted. Furthermore, since the concept of project HEIDI was not existing during the process of this work, changes might need to be implemented quickly.

Summary

These requirements are summarized in table 1.

Table 1: Summary of requirements

Requirement	**Description**	**Aim**
Computational time	Clock cycles needed per second	<56 M/s
Stack usage	Stack used by the algorithm	<10 Kbyte
Precision	Difference: Matlab - C Implementation	<1 cm

1.3 Target System

As discussed previously, the concept of HEIDI was not known at the beginning of this project. Therefore, an assumption based on last year's project, called TELL, as well as Kurmann's thesis had to be made.

1.3.1 Sensors and Microcontrollers

In terms of hardware, it was assumed that based on the last project, the same sensors and microcontroller unit (MCU) are used. For the firmware target device, this would be an STM32F407 ARM Cortex M4 MCU from ST Microelectronics. This MCU is equipped with 512 Kbyte to 1 Mbyte flash, 192 Kbyte RAM, a floating-point unit (FPU) as well as a maximum clock frequency of 168 MHz.

For the sensors, it was expected that similar components are used as in the last competition. These sensors, their output as well as the expected sampling frequency are introduced in chapter 3.1.1, where also the thesis of Kurmann is presented.

1.3.2 Firmware Architecture

Since the system load of the control algorithm is assumed to become rather heavy, ARIS decided to use two STM32F407 MCUs for project HEIDI to share this load.

It is planned that the first MCU will contain the sensor fusion algorithm, the sensor sampling task, the telemetry and sensor logging component, the human machine interface (HMI) control as well as the so-called data aggregation task, which handles all the sub components and the communication to the other microcontroller.

The control algorithm, which is bound to calculate the airbrake position, as well as the servo control implementation which moves the brakes, are located on the second MCU. This structure can be seen in figure 1.

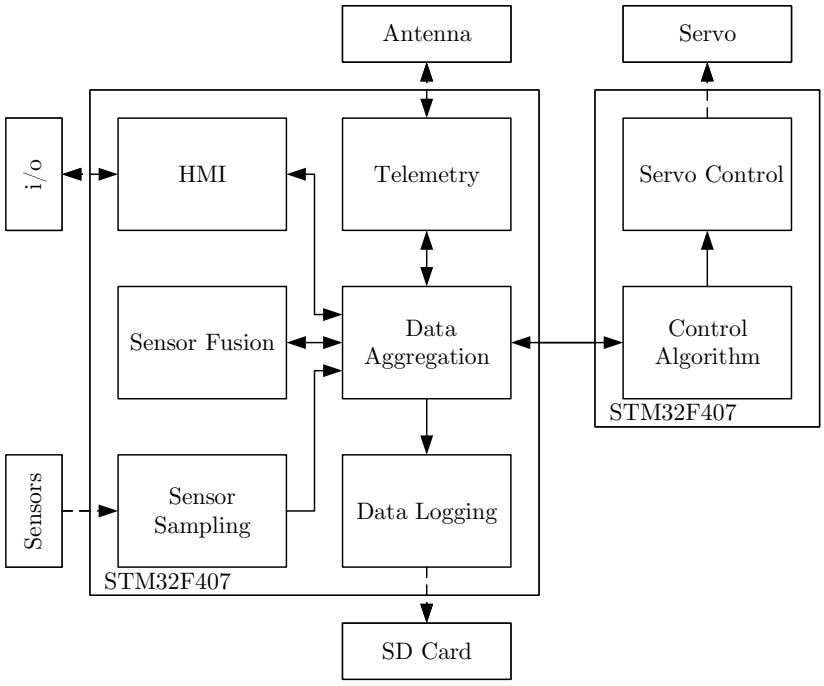


Figure 1: Concept of firmware structure by ARIS for project HEIDI by using two MCUs

2 Theoretical Background

In the course of this report, specific terms and concepts are used. In this chapter, a theoretical background on these terms and concepts is given. This includes the basics of state estimation by using Kalman filtering as well as the problem of inertial navigation.

2.1 Introduction to State Estimation

As explained earlier, the aim of the algorithm, which must be implemented, is to fuse sensor data together and to estimate a state of a physical system. In literature, such an algorithm can also be called a state estimator [3, p. 26]. There are different variations of state estimators, but the most important among these is the Kalman filter, developed and published in the 1960s by Rudolf E. Kalman [4, p. 101]. At that time, the Kalman filter was used in the Apollo program to estimate the position and speed of the Apollo lunar module [5]. This shows the importance of this algorithm and its initial practical use, which is not far away from the application of this project.

The Kalman filter is an optimal linear quadratic estimator which observes measurements over time and combines them with a prediction. The prediction is calculated by using the last estimation and a model of the physical system. The combination is then made based on the accuracy of the measurement and the previously established prediction. The resulting estimation will then be more accurate than the prediction and the measurement [5, p. 17]. This process can also be seen in figure 2 with $\tilde{x}_k$ being the estimation of the new state, based on the propagation of the last state $\tilde{x}_{k-1}$ through the system model to the prediction $\hat{x}_k$ and the measurement y_k with the respective uncertainties represented by the radius of the cirles and the width of the bell curves, which describe the probability density function of the gaussian distribution.

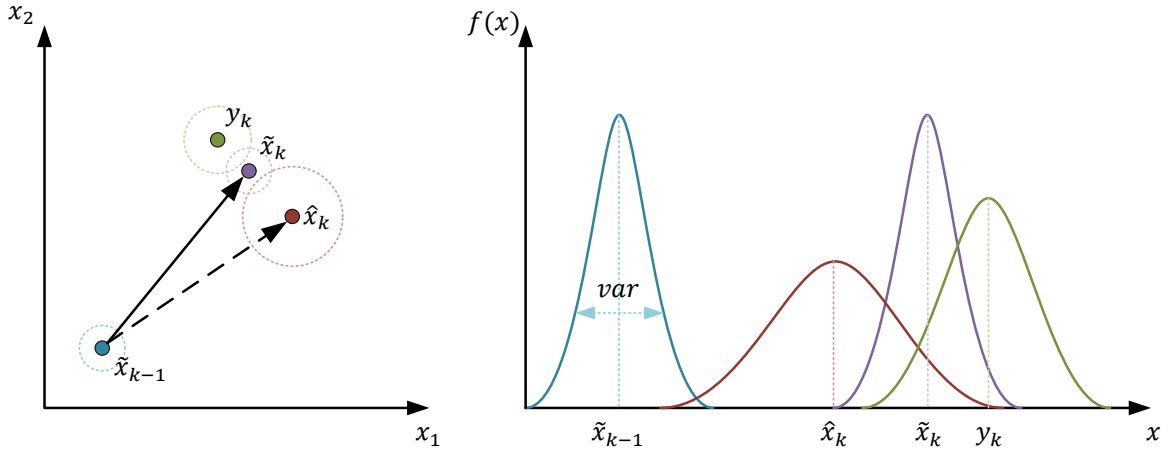


Figure 2: Visual propagation of the Kalman filter algorithm

2.1.1 Continuous State Space Model

At first, a mathematical model of the dynamic system, which needs to be estimated, must be derived, since the Kalman filter is based on that model. Such model can be represented by a state space in which multiple first order differential equations are combined. If furthermore the equations are linear and time invariant, they can be written in matrix form.

Formula 2.1 shows the continuous time equations used for the Kalman filter algorithm. The vector x represents the state, u the system input vector and y the measurement vector.

Matrix A equals the state transition matrix, B the system input model, G the system noise matrix and H the measurement model. Furthermore, the vector w describes the noise on the system and v the measurement noise. The noise processes w and v are white with zero mean and the variance and covariance described by Q, respectively R [6, p. 124].

$$\begin{aligned} \dot{x} &= Ax + Bu + Gw \\ y &= Hx + v \end{aligned} \tag{2.1}$$

$$\begin{aligned} w &\sim \mathcal{N}(0,\ Q) \\ v &\sim \mathcal{N}(0,\ R) \end{aligned} \tag{2.2}$$

2.1.2 Discrete State Space Model

Since the algorithm is running on an embedded system with a finite iteration period T_s, the continuous time state space model needs to be discretised. This can be done by solving the homogeneous differential equation given in the state space representation for time 0 to T_s [5, p. 9]. From that, equation 2.3 results. The time discrete matrices are marked with a lower case d. Furthermore, the lowercase letter indices at the vectors show that the state at the discrete time k is derived from the previous state at $k-1$ multiplied with the state transition matrix A_d and the system input with the input matrix B_d.

$$\begin{aligned} A_d &= e^{AT_s} \\ B_d &= A_d B \end{aligned} \tag{2.3}$$

$$G_d = A_d G \tag{2.4}$$

$$G_d = \int_0^{T_s} e^{Av} G dv \tag{2.5}$$

$$\begin{aligned} x_k &= A_d x_{k-1} + B_d u_k + G_d w_k \\ y_k &= H x_k + v_k \end{aligned} \tag{2.6}$$

$$\begin{aligned} w_k &\sim \mathcal{N}(0,\ Q_k) \\ v_k &\sim \mathcal{N}(0,\ R_k) \end{aligned} \tag{2.7}$$

Note that the measurement model equation to calculate y does not need to be discretised since the measurement vector is a scalar of the time discrete state vector.

Marchthaler and Dingler [5] mention multiple methods for the discretization of the matrix G. Equation 2.4 assumes that the noise is sampled with a sequence of perfect Dirac Delta functions. Therefore, the assumption that $w(t) \approx w_d(t)$ is made. In equation 2.5, it is assumed that the noise on w is partially constant which means that it does not change between the sampling interval. In the scope of this report, latter is assumed.

2.1.3 Observability

Haugen [4] mentions that the system model must be observable, otherwise, the Kalman filter will not work. Therefore, the determinant of the observability matrix Q_B must be determined and checked if it is unequal zero. If the observability matrix is not symmetric, its rank can be determined and compared to the order of the system model. If the rank equals the number of states in the state vector, the model is observable.

$$\begin{aligned} \det\left(Q_B\right) &= \det\left(\begin{bmatrix} H \\ \vdots \\ HA_{v\times i}^{v-1} \end{bmatrix}\right) \neq 0 \\ rank(Q_B) &= rank\left(\begin{bmatrix} H \\ \vdots \\ HA_{v\times i}^{v-1} \end{bmatrix}\right) = size(x) \end{aligned} \tag{2.8}$$

2.1.4 Prediction Step

Now, the Kalman filter equations can be formulated. First, with the estimation $\widetilde{x}_{k-1}$ from the previous iteration and the system input u_k, a prediction is calculated based on the system model. Furthermore, the system noise covariance matrix Q, which consists of the system noise variances, is added to the transformed error covariance matrix $\widetilde{P}_{k-1}$ from the last estimation. The concluding estimation error covariance matrix $\hat{P}_k$ describes the uncertainty of the prediction $\hat{x}_k$. $\hat{P}_k$ and $\hat{x}_k$ are called the "a priori" state estimation and the "a priori" error covariance matrix.

$$\begin{aligned} \hat{x}_k &= A_d\widetilde{x}_{k-1} + Bu_k \\ \hat{P}_k &= A_d\widetilde{P}_{k-1}A_d^T + G_dQ \end{aligned} \tag{2.9}$$

2.1.5 Update Step

Following the prediction step comes the update step. First, the Kalman gain K_k is calculated with equation 2.11. Assumed R and P are diagonal matrices and the measurement model H is an identity matrix, the Kalman gain matrix can consist of any values between 0 and 1. By looking at extreme cases for P and R and letting them be 1x1 matrices, the following observation can be made:

- For $R_k \gg \hat{P}_k \Rightarrow K_k \to 0$: The prediction is much more reliable than the measurement.
- For $R_k \ll \hat{P}_k \Rightarrow K_k \to 1$: The measurement is much more reliable than the prediction.

If one now considers r_k to be the innovation in each iteration, a scalar of this expression is added to the predicted state in equation 2.12 which equals the "a posteriori" state $\widetilde{x}_k$. Furthermore, the "a posteriori" error covariance matrix $\widetilde{P}_k$ can be calculated from the Kalman gain with equation 2.13.

$$r_k = (y_k - H\hat{x}_k) \tag{2.10}$$

Combining this with the observation made about the Kalman gain, the conclusion that the prediction tends to stay the same if it is more reliable than the measurement and that the prediction is becoming the measurement if the measurement is more reliable, can be made.

$$K_k = \hat{P}_k H^T \left(H \hat{P}_k H^T + R_k \right)^{-1} \tag{2.11}$$

$$\widetilde{x}_k = \hat{x}_k + K_k \left(y_k - H \hat{x}_k \right) = \hat{x}_k + K_k r_k \tag{2.12}$$

$$\widetilde{P}_k = \left(I - K_k H \right) \hat{P}_k \tag{2.13}$$

For the next iteration, the "a posteriori" estimation can now be set as the previous estimation.

$$\begin{aligned} \widetilde{P}_{k-1} &= \widetilde{P}_k \\ \widetilde{x}_{k-1} &= \widetilde{x}_k \end{aligned} \tag{2.14}$$

2.1.6 Multiple Sampling Rates

When using Kalman filtering, measurements can arrive at different rates. To perform the update step, multiple measurement models are needed. If there are for example r sensors, for each sensor, a measurement model can be derived.

$$H_1, H_2, \ldots, H_r$$

Each of them with a respective measurement covariance matrix.

$$R_1, R_2, \ldots, R_r$$

However, for each of those models, observability must hold.

$$rank\left(Q_{Bn}\right) = \begin{bmatrix} H_n \\ \vdots \\ H_n A^{v-1} \end{bmatrix} \neq 0, \; n \, \epsilon \, \{1, \; \ldots, \; r\} \tag{2.15}$$

2.1.7 Notation in Literature

It must be mentioned that there are many different notations concerning state space representation and Kalman filtering in literature. Table 2 gives an overview over these notations.

Table 2: List of different notations in literature

Term	This Report	Other Notations
Measurement vector	y	z [7]
State transition matrix	A	F [8, 9, 3, 10, 6]
Observation model	H	C [4, 2, 5]
State error covariance matrix	P	Σ [9]
Kalman gain	K	L [9]

2.2 Inertial Navigation System

A brief introduction is given on inertial navigation system to introduce the terms of the different navigation frames and strapdown navigation as well as the concepts behind these topics.

2.2.1 Coordinate Frames

First, the relevant reference frames must be defined. Since an altitude of 10'000 ft. above ground level (AGL) needs to be reached [1, p. 22], the so called local Cartesian navigation frame is used to represent the current state of the rocket.

Local Cartesian Navigation Frame

In the local Cartesian navigation frame, the origin can be set at arbitrary coordinates. In this project, that will be the projected point on the ground where the rocket is launched.

Every physical entity in this navigation frame is relative to this reference point. There are two conventions to describe that frame. The North East Down (NED) as well as the East North Up (ENU) reference frame. In the scope of this project, ENU coordinates are used. Furthermore, just the vertical component is needed for the control algorithm which means the others are ignored. In this report, physical quantities which are in this frame are either indicated by an uppercase n, if the represented frame needs to be highlighted, or not indicated at all. The position p of an object with its components x in the local Cartesian navigation frame can thus be described as in equation 2.16.

$$p^n = [x_E,\ x_N,\ x_U]^T \tag{2.16}$$

Body Fixed Frame

The body fixed frame is in relation to the orientation of the vehicle. In terms of this project, it is defined that its z-axis points in direction of the rocket's cone. In the scope of this report, all the physical quantities which are represented in the body fixed frame and which need to be highlighted so are marked with an uppercase b.

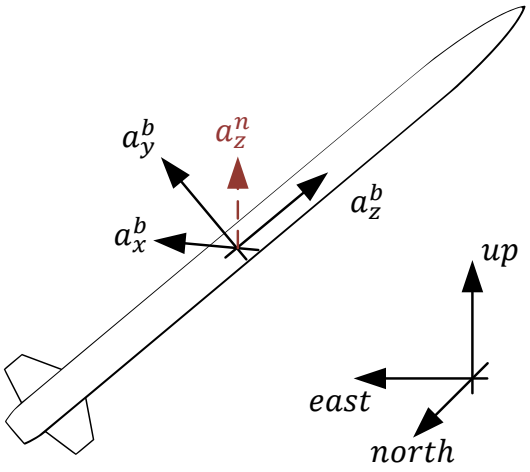


Figure 3: Comparison of the body fixed frame, with the respective acceleration components, and the local Cartesian navigation frame.

2.2.2 Strapdown Navigation

Another term which needs to be introduced in the scope of this report is strapdown navigation. Unlike gimballed gyrostabilized platforms, where an accelerometer is mounted on a gyroscope and the measured accelerations are directly in the navigation frame representation, in strapdown navigation, the inertial measurement unit (IMU), consisting of an accelerometer and gyro sensor, is strapped on the body of the vehicle. This implies that all measurements from the IMU are in respect to the body fixed frame [11, p. 27].

By using the measured angular rates from the gyro sensor, integrating them and perform further calculations, the so-called Euler angles can be derived. A transformation matrix can be calculated to transform the acceleration measurements from the body fixed frame to the navigation frame [9, p. 56].

3 Navigation Algorithm

The core of this project is the implementation of a state estimation algorithm for a sounding rocket. As stated in section 1.3, multiple sensors are used in the rocket to accomplish this goal. In his Bachelor thesis, Kurmann [2] compared different approaches to fuse these sensor readings together. However, the proposed "best performing system" did not consider certain factors. The proposed model and its problems are summarized in section 3.1. In section 3.2, a better model is introduced which solves the issues stated.

3.1 Background

The described "best performing system", derived by Kurmann, can be summarized by looking at the used sensors as well as the system model derived from them. It needs to be mentioned that in the competition of summer 2018, not the same sensors were used as in project HEIDI. The implemented algorithm was laid out to support these previously used sensors because the decision of which components will be used was not final until the end of this project.

3.1.1 Previously Used Sensors

One GPS module, two digital barometric pressure sensors, a gyroscope and an accelerometer served as a basis for Kurmann's thesis. A list of these sensors, the respective type, the provided sensor output and the used sampling frequencies can be found in table 3.

Regarding the GPS module, the output p_x and p_y describe the position of the rocket in the horizontal plane, whereas p_z holds information about the current altitude. In this report, p_z is therefore called h_{GPS}. In terms of the accelerometer, the subscripts refer to the orientation of the sensor. Since the sensor was mounted such that its z-axis points to the cone of the rocket, the measurement a_z refers to the acceleration in the direction of motion. One can recall the term strapdown navigation from section 2.2.2 for this reason. In the same category, the angular rate ω can be placed, produced by the gyroscope. Both barometers produce the barometric pressure p in hPa.

Table 3: Relevant sensors for the state estimation algorithm

Description	**Type**	**Output**	**Sampling Frequency [2, p. 7]**
GPS Module	NEO-M8T	p_x, p_y, p_z	1 Hz
Barometer 1	2SMPB-02E	p	100 Hz
Barometer 2	LPS22HBTR	p	100 Hz
Accelerometer	ADXL357	a_x, a_y, a_z	1 kHz
Gyroscope	ITG-7301	ω_x, ω_y, ω_z	1 kHz

As described in section 2.2.2, the accelerometer, as well as the gyroscope measurements are in the body fixed coordinate frame and therefore need to be transformed to the local Cartesian coordinate frame before using them in the state representation of the rocket.

Kurmann used the pitch angle φ, which describes the tilt of the rocket relative to the vertical axis and which can be determined over the angular rates, to perform this transformation (compare equation 3.1). However, the process of deriving the pitch angle from the angular rates is not described in Kurmann's thesis.

$$a_z^n = a_z^b cos(\varphi) \tag{3.1}$$

Furthermore, to calculate the altitude from the barometric pressure, the earth atmospheric pressure model for altitudes below 11'000 meters, which is the troposphere, was used [12].

$$p = p_0 * \left(1 - \frac{T_{grad} * h}{T_0}\right)^{\frac{M*g}{R*T_{grad}}} \Longrightarrow h = \frac{T_0}{T_{grad}} \left(1 - \left(\frac{p}{p_0}\right)^{\frac{R*T_{grad}}{M*g}}\right) \tag{3.2}$$

where:

p_0 = Pressure at ground level
T_0 = Temperature at ground level
T_{grad} = Temperature gradient for actual weather conditions
M = Molar mass or Earth's air: $0.0289644 \frac{kg}{mol}$
g = Gravitational acceleration: $9.81 \frac{m}{s^2}$
R = Universal gas constant: $8.31 \frac{J}{mol*K}$

Kurmann [2, p. 49] stated concerns regarding the Temperature gradient T_{grad} which is weather dependent. In this report, no further investigation was conducted to allay to these concerns.

3.1.2 Previous System Model

The recommended "best system" is now presented. It was shown by Kurmann that this system, after being fed with test data, showed an average error compared to the ground truth which was around 1 meter better than the other constellations of the state and measurement vectors [2, p. 42].

In the state vector (3.3), the current altitude of the rocket h, its vertical speed v_z and acceleration a_z, as well as the acceleration offset a_{off} and the current barometric pressure p at that altitude can be found.

$$x = \begin{bmatrix} h \\ v_z \\ a_z \\ a_{off} \\ p \end{bmatrix} \tag{3.3}$$

Literature shows that it is common practice to model the offset, also called bias, of the accelerometer in the state vector [9, p. 23][5, p. 139][3, p. 48]. Reason behind this is that in general, the noise on the state variables should have zero mean (compare with section 2.1.1). Accelerometers, however, show a time variant bias which drifts depending on changes in temperature [5, p. 139]. This bias error is then further amplified since the acceleration is integrated twice to find the current position. By considering the bias in the state vector, this effect can be reduced.

The consideration of the ambient barometric pressure though does not have the same backing in literature. The intention by Kurmann behind this was to eliminate the fact that the altitude has a nonlinear relation with the atmospheric pressure. A nonlinear relation between states and measurements would not have been compatible with the standard Kalman filter 2.2.1. Haugen [4, p. 116] mentions that "environmental forces" can be represented in the

state vector, however, with the intention of using this estimation as a "feedforward control signal" for the controller. This, after all, is not the case here.

The measurement vector y (3.4) includes the measured altitude from the GPS module h_{GPS}, the measured and converted acceleration in the navigation frame a (3.1), the two measured barometric pressures p_1 and p_2 as well as the altitude, calculated from the estimated pressure in the state vector x (3.3) by usinng equation 3.2.

$$y = \begin{bmatrix} h_{GPS} \\ a \\ p_1 \\ p_2 \\ h(p) \end{bmatrix} \tag{3.4}$$

$$u = [0] \tag{3.5}$$

State augmentation was used to model the rocket dynamics. Therefore, the system input vector u (3.5) was set to zero. When using state augmentation, it is assumed that a state has an almost constant rate (in this case the velocity v_z) and an additional state is introduced (in this case the acceleration a_z). Changes in the acceleration are now just modelled with the system noise w which has a variance Q (3.6) [4, p. 116] since the derivative of the acceleration is set zero.

$$Q = \begin{bmatrix} \sigma_{a_z}^2 & 0 & 0 \\ 0 & \sigma_{a_{off}}^2 & 0 \\ 0 & 0 & \sigma_p^2 \end{bmatrix} \tag{3.6}$$

where:

$\sigma_{a_z}^2, \sigma_{a_{off}}^2, \sigma_p^2$ = variances of each individual system noise

This can also be seen in the state space representation of the differential equation 3.7. Note that KPv equals the rate dp/dh.

$$\begin{bmatrix} \dot{h} \\ \dot{v}_z \\ \dot{a}_z \\ \dot{a}_{off} \\ \dot{p} \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & KPv & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} h \\ v_z \\ a_z \\ a_{off} \\ p \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} w_{a_z} \\ w_{a_{off_z}} \\ w_p \end{bmatrix} \tag{3.7}$$

where:

$w_{a_z},\ w_{a_{off_z}},\ w_p$ = noise on the system input
$\dot{h}, \dot{v}_z, \dot{a}_z, \dot{a}_{off}, \dot{p}$ = rate of each state

The change in the acceleration can now be written with $\dot{a}_z$ being the derivative of the ground truth acceleration in vertical direction.

$$\dot{a}_z = w_{a_z} \tag{3.8}$$

With equations 3.9 and 3.10, Kurmann derived a formulation for the variance of the acceleration (equ. 3.11).

$$w_{a_z} \sim \mathcal{N}\left(0, \sigma_{a_z}^2\right) \tag{3.9}$$

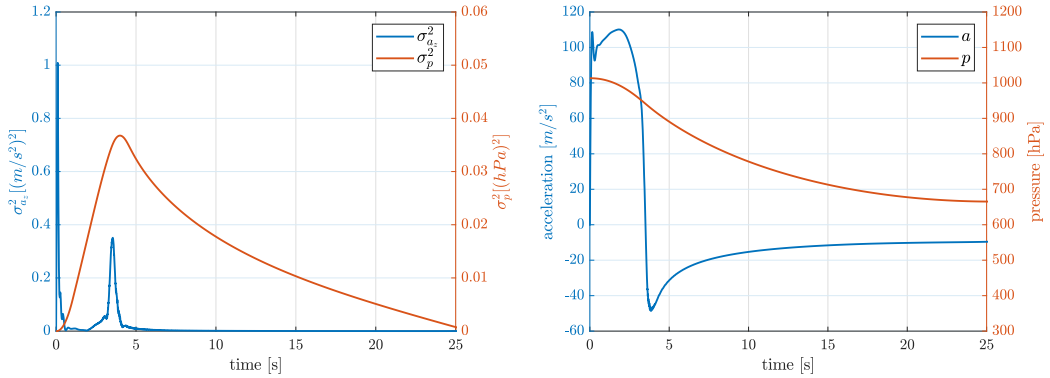
$$\sigma_{a_z}^2 = E\left(w_{a_z}^2\right) - E\left(w_{az}\right)^2 = E\left(w_{az}^2\right) = E(\dot{a}_z^2) \tag{3.10}$$

$$\sigma_{a_z}^2 \approx n\left|\dot{a}_z\right| \rightarrow \sigma_{a_z}^2\left(k\right) \approx n\left|\frac{a_z\left(k\right) - a_z\left(k-1\right)}{T_s}\right| = m\left|a_z\left(k\right) - a_z\left(k-1\right)\right| \tag{3.11}$$
$$m,\ n\ \in \mathbb{R}^+$$

With $a\left(k\right)$ being the acceleration of the simulated rocket trajectory to the target apogee of 10'000 ft., the system noise covariance for the acceleration is dynamic. The same principle was applied on the pressure. This resulted in a dynamic system noise covariance matrix (3.12).

$$Q(k) = \begin{bmatrix} \sigma_{a_z}^2(k) & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & \sigma_p^2(k) \end{bmatrix} \tag{3.12}$$

This property can also be seen by comparing figure 4a and 4b. The curve in figure 4a represents the absolute derivative of the acceleration, representing the system noise covariance, respectively the barometric pressure. Figure 4b on the other hand shows the simulated acceleration and barometric pressure for this 25 second long flight to 10'000 ft.



(a) Derivated dynamic system noise covariance matrix

(b) Simulated acceleration (blue) and pressure (red) during the flight to the target apogee at approx. 25 s

Figure 4: Derivation of system noise covariance matrix Q

3.1.3 Problems With this Model

The Model derived by Kurmann however brings along some problems. These issues need to be resolved prior to the implementation.

State Augmentation

Firstly, from section 2.1.1 can be recalled where the definition was made that the noise on the measurements, as well as the states, must be white with zero mean. Furthermore Haugen [4, p. 116] states that these assumptions only hold for slowly varying, almost constant augmented states. Considering figure 4b, both, the pressure and the acceleration are far away from being almost constant with zero mean and being white.

Secondly, it would be difficult to implement such a dynamic covariance matrix since a large amount of memory would be needed (approximately 200 Kbyte when using single precision for 25 s at 1 kHz state estimation iteration rate). Furthermore, the actual flight trajectory would have to be guessed, which would have led to an additional point of uncertainty.

Multiple Sampling Rates

As seen in table 3, the state estimation algorithm needs to be capable of handling multiple sensor sampling rates. Kurmann pursued the aim of maxing out the respective value in the measurement covariance matrix R [2, p. 28] so that the Kalman gain in equation 2.11 converges to zero as discussed in section 2.1.5.

However, since single precision floating point numbers are used (discussed in section 4.1.1), choosing a value in the R matrix which is too high can result in unintentional rounding effects in the calculation of the Kalman gain (equation 2.11).

Small Noise on GPS

In his Bachelor Thesis, Kurmann generated test data sets by conducting the following steps:

1. Loading test data sets from previous low altitude test flights
2. Second order polynomial curve fitting for the different flight phases
   - Before takeoff
   - During engine burn
   - After engine burnout and before parachute deployment
3. Subtracting the polynomial from the test data to extract the noise
4. Analyzing the noise with the Yule-Walker equation in order to obtain an autoregressive (AR) model
5. Generating new noise data and modulate them on top of a simulated flight trajectory to generate synthetic test data sets

Sensor data from pressure sensors as well as the accelerometer were obtained from EPFL [2]. However, no test flights with a GPS module were conducted. To generate synthetic GPS data, Kurmann analyzed the received signal from a static GPS module and executed the same steps as described above. This procedure led to a GPS signal with a signal to noise ratio of 85.7 dB in average, which would be equivalent to an error of the received GPS of about 10 cm at an altitude of 1000 meters.

Of course, this error is too small since the GPS module on its own has an uncertainty of at least 2.5 m [13]. Additionally, the latency of the GPS module at high velocities might have an influence and the high vibrations during the engine burn may cause the module to lose its satellites, which further increases the inaccuracy.

3.1.4 Conclusion

In conclusion, the following questions can be formulated:

- Is there a way to eliminate the augmentative acceleration state in the state vector and modeling changes in the acceleration with the system noise?
- Is there a better, more efficient solution to handle multiple sampling rates in the Kalman filter?
- Is this so called “best performing system” really the best one or does it just perform well because the modelled sensor errors are too small?

3.2 Improved Model

An improved system model was derived which suits the implementation on an embedded system better and might even outperform the proposed system.

3.2.1 System Dynamics

With equations 3.13 to 3.15, the system equation can be re-formulated. To further reduce the computational effort needed to compute each iteration of the filter, the rocket tilt is neglected in this model. This eliminates the need of the computationally intensive transformation of every IMU reading from the body fixed frame into the Cartesian navigation frame as stated in section 2.2. This neglection assumes that the rocket is flying perfectly vertical. Therefore, the z-component of the accelerometer reading a_z^b can be used directly in the model equations.

$$\dot{h} = v_z \tag{3.13}$$

$$\dot{v}_z = a_{off} + a_z \tag{3.14}$$

$$a_z = a_z^n \approx a_z^b \tag{3.15}$$

With these differential equations, a much simpler state space can be modelled.

First, the measured acceleration in the navigation frame can be used as the system input u. This method is mentioned in literature [10, p. 3] as well as applied in practice [9]. Not only does this eliminate the augmentation of the velocity dynamics, but also does it makes the need for a dynamic system covariance matrix redundant since the system noise now just consists of the difference between the real and the measured vertical acceleration. This is discussed in more detail in section 3.2.2.

In addition, the system is now faster since a change in the acceleration has an immediate influence on the system state. Furthermore, doubts about the lack of filtering concerning the acceleration can be dissolved with the fact that the acceleration is integrated twice which each one acts as a low pass filter.

A second change was made concerning the barometer readings. Instead of taking the atmospheric pressure into the state vector and the calculated altitude from that pressure into the measurement vector, the respective altitude calculated from each barometer can be used in the measurement vector.

The resulting changes in the system vectors can be seen by comparing the proposed model by Kurmann with the improved model in table 4.

Table 4: Changes to the proposed state space model

Model derived by Kurmann		**Improved Model**
$x = \begin{bmatrix} h \\ v_z \\ a_z \\ a_{off} \\ p \end{bmatrix}$	$\Rightarrow$	$x = \begin{bmatrix} h \\ v_z \\ a_{off} \end{bmatrix}$
$u = [0]$	$\Rightarrow$	$u = [a_z] = [a_z^n]$
$y = \begin{bmatrix} h_{GPS} \\ a \\ p_1 \\ p_2 \\ h(p) \end{bmatrix}$	$\Rightarrow$	$y = \begin{bmatrix} h_{GPS} \\ h(p_1) \\ h(p_2) \end{bmatrix}$

With these changes, a new state space representation can be formulated.

$$\dot{x} = \begin{bmatrix} \dot{h} \\ \dot{v}_z \\ \dot{a}_{off} \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} h \\ v_z \\ a_{off} \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} a_z + \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} w \tag{3.16}$$

$$y = \begin{bmatrix} h_{GPS} \\ h(p_1) \\ h(p_2) \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix} x + v \tag{3.17}$$

$$A = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix},\ B = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix},\ G = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, H = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix} \tag{3.18}$$

As can be derived from equations 3.16 to 3.18, the dimensions of the vectors as well as the matrices in the state space could be reduced. Now, considering the status vector $x_{n\times 1}$, the system input $u_{i\times 1}$, the noise vector $w_{j\times 1}$ and the measurement vector $y_{m\times 1}$, one can approximate the needed equations to perform a Kalman filter iteration as described in section 2.1. For this comparison, the assumption was made that a matrix multiplication of an $(n\times m)$ with an $(m \times i)$ matrix has a complexity of $\mathcal{O}(nmi)$ and a matrix inversion of an $(n \times n)$ matrix has the complexity $\mathcal{O}\left(n^3\right)$. Table 5 shows and compares the complexity of each algorithm step.

Table 5: Comparison of the proposed model and the newly derived model in terms of computation complexity

Step	Complexity	Proposed Model	Improved Model
		$n = 5$ $m = 5$ $i = 0$ $j = 3$	$n = 3$ $m = 3$ $i = 1$ $j = 1$
Prediction Step			
$\hat{x}_k$	$\mathcal{O}\left(n^3\right) + \mathcal{O}\left(ni\right)$	125	27
$\hat{P}_k$	$2\mathcal{O}\left(n^3\right) + \mathcal{O}\left(nj^2\right) + \mathcal{O}\left(jn^2\right)$	370	66
Update Step			
K_k	$\mathcal{O}\left(n^2m\right) + \mathcal{O}\left(nm^2\right) + \mathcal{O}\left(m^3\right)$	375	81
$\widetilde{x}_k$	$2\mathcal{O}\left(nm\right)$	50	18
$\widetilde{P}_k$	$\mathcal{O}\left(n^2m\right) + \mathcal{O}\left(n^2\right)$	150	36

From this table, the prediction can be made that the newly derived system model will need around 22% of the computational effort required by the proposed algorithm.

$$100\% * \frac{3^3}{5^3} \approx 22\% \tag{3.19}$$

The changes to the system model are therefore hugely beneficial in terms of computing time. This is important because, as stated in section 1.2, it is a requirement to keep the needed computing resources as low as possible.

This improved model needs to be discretized further with equations 3.20 to 3.21. The term T_s describes the rate at which the state estimation algorithm is iterated. It makes sense to process such iteration whenever a system input arrives, in this case an accelerometer reading. Since the accelerometer is sampled at a rate of 1 kHz (compare table 3), T_s is set to 1 ms.

$$A_d = e^{A*T_s} = \begin{bmatrix} 1 & T_s & T_s^2/2 \\ 0 & 1 & T_s \\ 0 & 0 & 1 \end{bmatrix} \tag{3.20}$$

$$B_d = \int_0^{T_s} e^{Av} B dv = \begin{bmatrix} T_s^2/2 \\ T_s \\ 0 \end{bmatrix} \tag{3.21}$$

$$G_d = \int_0^{T_s} e^{Av} G dv = \begin{bmatrix} T_s^2/2 \\ Ts \\ 0 \end{bmatrix} \tag{3.22}$$

When comparing the output of the proposed algorithm with the output of the new model in figure 5, both fed with the same input data, it can be seen that there is a difference in the range of 1 meter in average. These differences most probably occur due to the fact that the new model is much more reactive since the acceleration has a direct influence on the state.

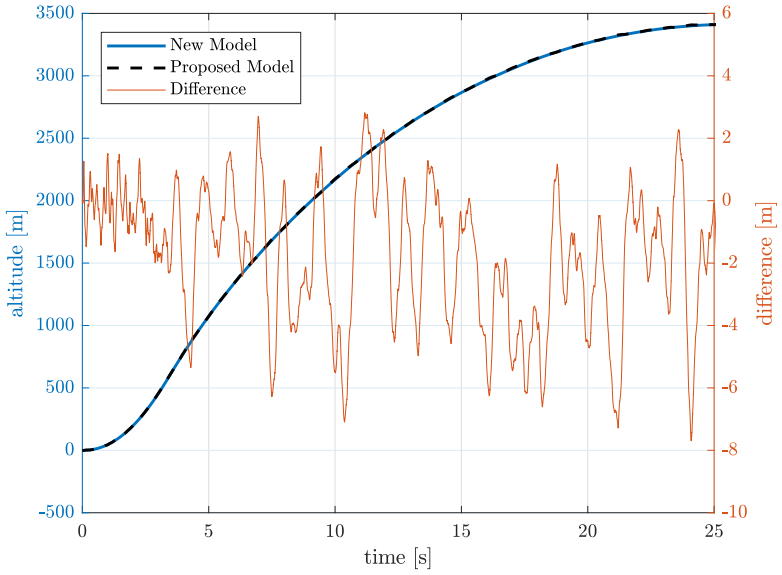


Figure 5: Estimated difference in altitude between the new and the proposed model

3.2.2 Sensor Model

Now, since a system model for the state estimation was derived, a concept on setting the covariance matrices R and Q had to be developed.

Measurement Covariance Matrix R

The measurement covariance matrix R is defined in equation 3.23 as the variance and covariance of the measurement noise vector v_k.

$$v_k \sim \mathcal{N}(0, R_k) \tag{3.23}$$

And v_k is defined as the difference between a real and a perfect measurement.

$$v_k = \hat{y}_k - Hx_k = \hat{y}_k - y_k \tag{3.24}$$

That means v_k can be subdivided into the single measurement noises.

$$v_k = \begin{bmatrix} \delta h_{GPS} \\ \delta h(p_1) \\ \delta h(p_2) \end{bmatrix} \tag{3.25}$$

where:

$\delta h_{GPS}, \delta h\,(p_1)\,,\ h(p_2) =$ noise on each individual sensor reading

Each of these components can be subdivided into further parts, all of which representing random noise. Since an analytical approach to approximate the variances of each of these

components would be quite hard to accomplish, a numerical way was chosen. This process is closely linked to the verification of the algorithm since these parameters are responsible if the filter works well or badly and hence this is described in chapter 5.

However, it was seen that the noise characteristics change for each phase of the flight. This is reasonable since vibrations and the high acceleration during the engine burn phase have an influence on the sensors. To use this knowledge in the state estimation algorithm, it was decided to create multiple noise covariance matrices for each phase of the ascending flight.

- Before takeoff
- During engine burn
- After engine burnout, when braking

These flight phases can be seen in figure 6.

This approach of changing the noise covariance matrices during a rocket flight is also described in literature [8, p. 177]. The resulting state estimator can also be seen as three Kalman filters, each of them having initial conditions matching the last estimation and the last error covariance matrix from the previous filter.

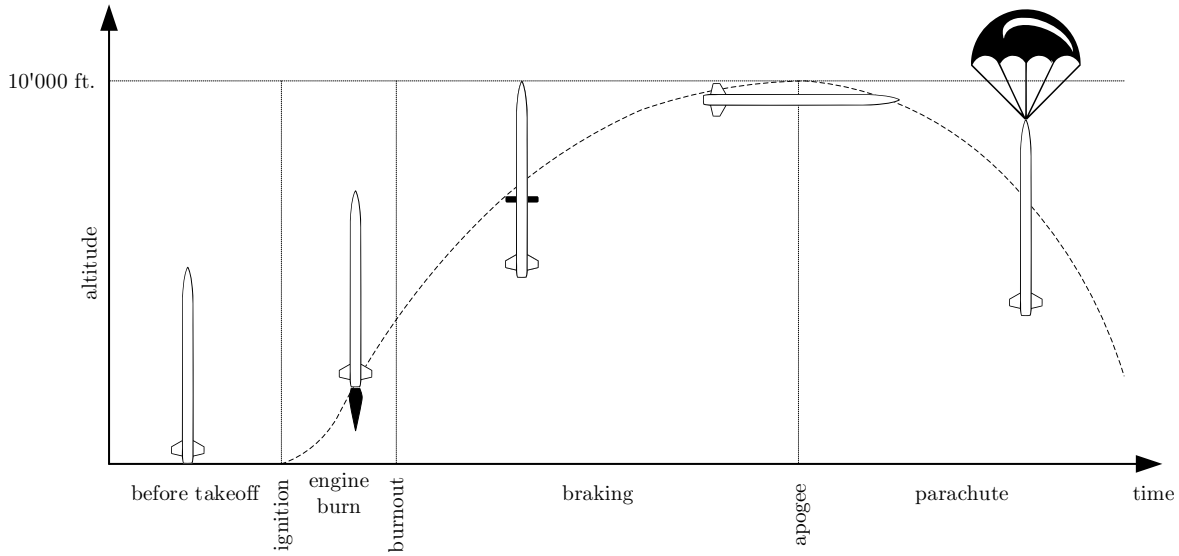


Figure 6: Important phases of the rocket flight

Measurement Covariance Matrix Q

Since the measured acceleration is used as the system input, the system noise now only consists of the difference between the measured and the real acceleration in vertical direction δa_z. This difference is caused by measurement noise and, because the tilt of the rocket is neglected, the error occurring through the fact that the measurement in the body fixed frame is not equal to the acceleration in vertical direction. These two components can be seen in equation 3.26.

$$\begin{aligned} w_k &\sim \mathcal{N}(0, Q_k) \\ w_k &= \delta a_z = \left(a^b_{z_{mes}} - a^b_z\right) + (a^n_z - a^b_z) \end{aligned} \tag{3.26}$$

There are further components adding to the acceleration measurement error. These errors, however, are not discussed in the scope of this work. Non-zero mean noise components, which are coming from the neglected rocket tilt, are taken into account by the a_{off} component in the state vector x as it was discussed in section 3.1.2.

4 Implementation

Having derived a model which is more suitable for the implementation on an embedded system, the implementation itself needed to be conducted.

Since further changes to the used sensors as well as additional optimizations to the algorithm might occur, the implementation on the embedded system must be kept as flexible as possible. This is conducted by executing all changes to the algorithm in Matlab and then using the Matlab functionality Matlab Coder, which can generate C and C++ code from Matlab Code, to quickly port the algorithm to the embedded system. It should then be possible to use the generated C code as it is in the firmware implementation.

On the one hand, this approach enables quick changes to be made in the algorithm as well as quick testing with pre-defined test data sets. On the other hand, the Matlab code must be prepared in a certain way. There are also restrictions in the commands which can be used and the converted code might not be as optimized as one could achieve by just implementing this one specific problem in C.

4.1 Optimization of State Estimation Algorithm

First, the state estimation algorithm was further optimized in Matlab. In literature, multiple methods are stated.

4.1.1 Single Precision Floating Point

Fico et al. [14] mention that the main reduction in execution time, when implementing a Kalman filter on an embedded system, can be obtained by using single precision instead of double precision floating point number representation. This takes advantage of the provided floating point unit on the Processor, which is an ARM Cortex M4 in case of the used STM32F407 MCU. The reason is that double precision floating point calculations are executed in software instead of using the processor hardware. This can also be obtained by comparing the needed clock cycles for computing an addition. One clock is needed in case of using single precision and three clocks are needed when using double precision. Both numbers hold just for STM32 ARM Cortex M processors [15, p. 16].

An additional improvement should be possible in the Stack and Flash usage, since just 32 instead of 64 Bits are needed for all the variables and constants.

One drawback, however, is the loss in dynamic and resolution which will occur when using single precision float. The mantissa will shrink from a 52 bit number to a 23 bit number, which will have around 7 significant figures compared to almost 16 with double precision. In terms of the dynamics, a reduction from 12'318 dB to 1'529 dB must be accepted, which however is still better than using fixed point arithmetic with a dynamic of just 192 dB for uint32 [15, p. 6]. This is also the reason why floating point is preferred since with the available FPU, no advantage would be obtained by using fixed point arithmetic.

Because Fico et al. [14] could achieve a reduction of 68% in execution time for a single filter iteration by using single precision, even though tested with an unscented Kalman filter, similar results are expected. Also when considering the reduction from 3 clocks to 1 clock cycle for a simple addition which was mentioned before.

4.1.2 Sequential Kalman Filtering

As another improvement in reducing the computational effort and stack needed for one state estimation iteration, sequential Kalman filtering can be applied. Simon [6, p. 150] mentions that sequential Kalman filtering "[...] can be a great advantage, especially in an embedded system that may not have matrix routines. [...]". However, to use sequential Kalman filtering, certain criterias must be fulfilled.

- The measurement noise covariance matrix R_k must be diagonal
- The measurement noise covariance matrix R must be constant

Simon [6, p. 150] furthermore mentions that one of these conditions must hold, however, it is beneficial if both conditions hold since less computation is required.

When using sequential Kalman filtering, the matrix inversion, needed to calculate the Kalman gain (recall equation 2.11), degenerates into an inverse scalar. This is achieved by iterating through the measurements in the Kalman filter update step. The reduction in computing time resulting from this can also be seen when comparing the needed additions and multiplications derived by Kettner and Paolone [7, p. 6]. Equation 4.1 shows how many additions and multiplications are needed by the standard discrete Kalman filter to calculate the Kalman gain. Equation 4.2 shows the same for the sequential Kalman filter. Note that D equals the number of measurements and S the number of state variables. The terms $m \in \mathcal{O}(D^3)$ and $n \in \mathcal{O}(D^3)$ denote the method used to calculate the inverse matrix. One let them be D^3.

With $D = S = 3$, for the discrete Kalman filter, this results in the following numbers of operations.

$$\begin{aligned} n_{+|-} &= 2D^2S + D\left(1 - D - S\right) + m = 66 \\ n_{\times|\div} &= 2D^2S + n = 81 \end{aligned} \tag{4.1}$$

For the sequential Kalman filter, the following numbers result.

$$\begin{aligned} n_{+|-} &= DS = 9 \\ n_{\times|\div} &= D\left(2S + 1\right) = 21 \end{aligned} \tag{4.2}$$

where:

$n_{+|-}$ = number of additions and subtractions
$n_{\times|\div}$ = number of multiplications and divisions

Clearly fewer operations are needed to calculate the Kalman gain when using the sequential Kalman filter algorithm. For the other calculations, the same amount of additions and multiplications are needed.

To perform a sequential Kalman Filter update step, firstly, the "a posteriori" estimate and covariance must be initialized.

$$\begin{aligned} \hat{x}_{0k} &= \hat{x}_k \\ \widetilde{P}_{0k} &= \hat{P}_k \end{aligned} \tag{4.3}$$

Now, for $i = 1, \ldots, r$, where r is the size of the measurement vector y_k and R_{ik} is the diagonal element of matrix R_k, perform equations 4.4 to 4.6.

$$K_k = \widetilde{P}_{i-1,k} H_i^T \left(H_i \widetilde{P}_{i-1,k} H_i^T + R_{ik}\right)^{-1} \tag{4.4}$$

$$\widetilde{x}_{ik} = \widetilde{x}_{i-1,k} + K_{ik}\left(y_{ik} - H_i\widetilde{x}_{i-1,k}\right) \tag{4.5}$$

$$\widetilde{P}_{ik} = \left(I - K_{ik}H_i\right)\widetilde{P}_{i-1,k} \tag{4.6}$$

After iterating through the measurements, the new "a posteriori" estimate and covariance can be defined as 4.7.

$$\begin{aligned}\widetilde{x}_k &= \widetilde{x}_{rk}\\ \widetilde{P}_k &= \widetilde{P}_{rk}\end{aligned} \tag{4.7}$$

In theory, when fulfilling the conditions stated above, the results from the sequential implementation and standard Kalman filter implementation should be equivalent. However, comparing the result of both implementations, a difference in the range of a few centimeters can be observed in figure 7. This test was conducted by feeding the algorithm with test data from Kurmann. This difference may arise from numerical effects.

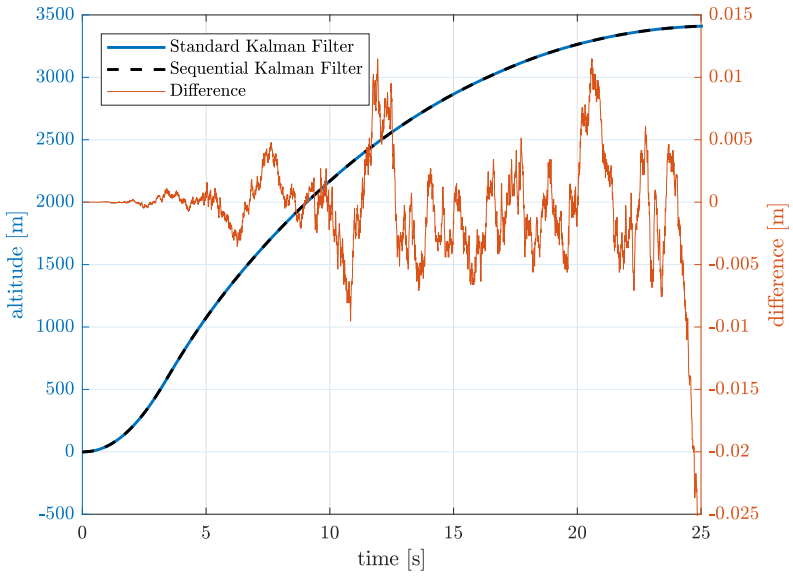


Figure 7: Estimated altitude of the standard Kalman filter minus the sequential Kalman filter

An additional aspect of using sequential Kalman filtering is that the measurements in the measurement vector y, which did not arrive, can simply be skipped when iterating. This does not only solve the problem stated in section 3.1.3, concerning multiple sampling rates, but it also can reduce the needed computing time. Since only three measurements are present in the measurement vector y (compare equ. 3.17), and the fastest one of them is sampled 10 times slower than the state estimation is executed, nine out of ten times there will be no update step at all.

In Matlab such an implementation can look as follows.

```
1 for i = 1:length(y)
2     if bitget(avbl, i) > 0
3         K_i = P*H(i,:)'/(H(i,:)*P*H(i,:)'+R(i,i));
4         x = x + K_i*(y(i)-H(i,:)*x);
5         P = (eye(3)-K_i*H(i,:))*P;
6     end
7 end
```

The variable "avbl" contains flags which are set at the position i if the respective measurement is available. Furthermore, it can be seen how the variable i runs from 1 to the length of the y vector which was previously called r.

4.2 Software Porting

To efficiently port a Matlab Code to C, Chou [16] recommends in his article this three-step iterative workflow:

1. Preparing the algorithm in Matlab for the code generation
2. Testing the implemented Matlab algorithm
3. Generate the C code, test it again and change the settings in order to improve the performance of the C implementation

Step 2 and 3 could easily be conducted within the Matlab Coder functionality, the algorithm needed to be prepared manually first.

4.2.1 Preparing Algorithm in Matlab

To port the Matlab code to C, the Kalman filter algorithm had to be placed into a function. The corresponding C implementation will then consist of this function, as well as a test file where this function is called.

It was decided to define the measurement vector as well as the system input vector as function parameters since these values arrive outside of the function's scope. Furthermore, it was decided to use the state vector as a parameter as well since in that way, initial x values can be given to the state estimator. To tell the function, which measurements are available and which are not, respective flags are set in the "avbl" parameter. And finally, pointers to the Q and R vector are passed to the function so that they are changeable when entering a new flight phase.

Considering the measurement vector y, it was further decided to do the calculation of the altitude from the atmospheric pressure before the function call. In that way, the algorithm became more generic.

Considering the decisions above, the following Matlab code header was defined.

```
1 function [x, y] = Kalman(x, u, y, avbl, Q, R)
2     ...
3 end
```

By using Matlab Coder, the following C code resulted.

```
1 void Kalman(float x[3], float u, const float y[3], unsigned char avbl, ...
      float Q, const float R[9]){
2     ...
3 }
```

In addition to the need of a Matlab function, certain limitations in the range of supported functionalities which can be ported by Matlab Coder exist. However, none of these limitations affected this implementation.

4.2.2 Matlab Coder

In Matlab Coder, multiple settings concerning optimization, target device, including comments, etc., can be specified.

Concerning speed settings, saturation on integer overflow as well as the support of non-finite numbers, were disabled. Both of these functionalities are not needed since mostly floating point numbers are used.

Concerning memory, no variable sizing is needed because the used memory should be known beforehand to prevent stack overflow during runtime. Furthermore, the option for generating reentrant code was disabled since the state estimation function does not share any resources with any other task running.

Concerning the hardware, the coder could be set such that the generated code is ARM Cortex compatible.

In table 6, all the files generated by Matlab coder as well as a short description for each one of them can be seen.

Table 6: Generated files by Matlab Coder

File	**Description**
Kalman.c Kalman.h	Contains the function "Kalman" which needs to be called to execute one state estimation iteration. Also contains Kalman_init(void) function which needs to be called first to initialize the variables used by the algorithm.
Kalman_initialize.c Kalman_initialize.h	Kalman_initialize function which just calls Kalman_init. Kalman_init can be called directly instead. These files are not needed.
Kalman_terminate.c Kalman_terminate.h	Empty function. Would be used if there was dynamic allocation to free the allocated memory. These files are also not needed.
Kalman_types.h	Includes rtwtypes.h.
rtwtypes.h	Matches platform dependent types to the types used by the algorithm.
main.c main.h	Main function contains the initialization of empty parameters (filled with 0) and a call of Kalman_initialize, Kalman and Kalman_terminate. These files are also not needed, but they can serve as a reference how to call the Kalman function.

4.3 Further Changes to the System Model

If in the future the system model changes, the following adaptations would have to be made to the implementation.

1. Changing the state space vectors in the Kalman function according to the new model.
2. Increasing or decreasing the dimension of the covariance matrix P in the Kalman function.
3. Changing the iteration time Ts in the Kalman function such that it is equal to the period in which the function is called.
4. Change the test bench file according to the new system model and setting the newly derived parameters of the Kalman filter.
5. Port the algorithm to C by using Matlab Coder
6. Change the part in the firmware where the sensor data is received, checked and the state estimator is called.

4.4 Firmware Structure

Having converted the C implementation from Matlab to C, embedding the state estimation function in the firmware framework was the next task to complete. In the last competition, FreeRTOS was used by ARIS as a real time operation system on the STM32F407 MCU and hence it made sense to wrap the sensor fusion algorithm in an own task.

4.4.1 Interfaces

Firstly, the firmware interfaces needed to be defined. Since the framework for the competition in summer 2019 did not exist during the course of this work, assumptions based on the firmware from project TELL needed to be made.

In the planned firmware structure for project HEIDI in section 1.3.2, it was seen that the sensor fusion algorithm just needs to communicate with the data aggregation. For this data exchange, message queues were used since they are thread safe and commonly used for inter-task communication in the context of FreeRTOS.

On the one hand, the only inputs needed by the sensor fusion algorithm are the sensor readings and hence one message queue, owned by the sensor fusion task, is used to receive this data. The decision was made to use a structure with the sensor type, as well as the sensor reading as members.

On the other hand, the sensor fusion task has two outputs. The current state of the rocket, calculated by the state estimation algorithm, as well as event information concerning the current flight phase of the rocket or the detection of a sensor outfall.

The data aggregation task is set to pass these data further to the telemetry, the control algorithm and the data logging.

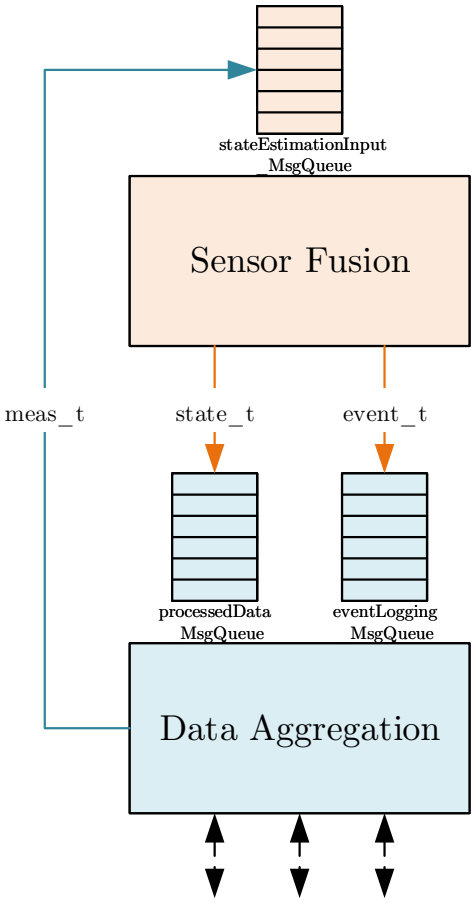


Figure 8: Firmware framework by using FreeRTOS

4.4.2 Sensor Fusion Routine

The routine for the state estimation algorithm, which is shown in figure 9, can be described as follows.

Firstly, the task must be created. Initial values are passed as parameters to initialize the algorithm. In a next step, the sensor fusion task will enter its main loop.

The state estimation input queue is read until an acceleration measurement is received. Since the accelerometer is sampled at 1 kHz, which is 10 times faster than the fastest of the other sensors, the order in which the sensors are sampled and put into the queue should not matter. Meanwhile, until the acceleration is received, the measurements are saved in an array of length 3, representing the measurement vector.

After receiving the measurements, a check on the reliability of each reading can be made to detect potential faulty sensors. A possible approach to this problem is described in section 4.4.3. The case of a missing accelerometer reading is discussed in section 4.4.5.

Having conducted a check on the sensor readings, the flight phase can be determined. This procedure is discussed in section 4.4.4. With the determined flight phase, the covariance matrices R and Q can be set accordingly.

In a next step, the Kalman function, which was described in section 4.2.1, can be called. The green colored flow chart blocks in figure 9 equal the process described.

The results from the state estimation algorithm are then put into the processed data queue. Additionally, events are logged at any time events occur. This applies for changes in the flight phase as well as detected faulty sensors.

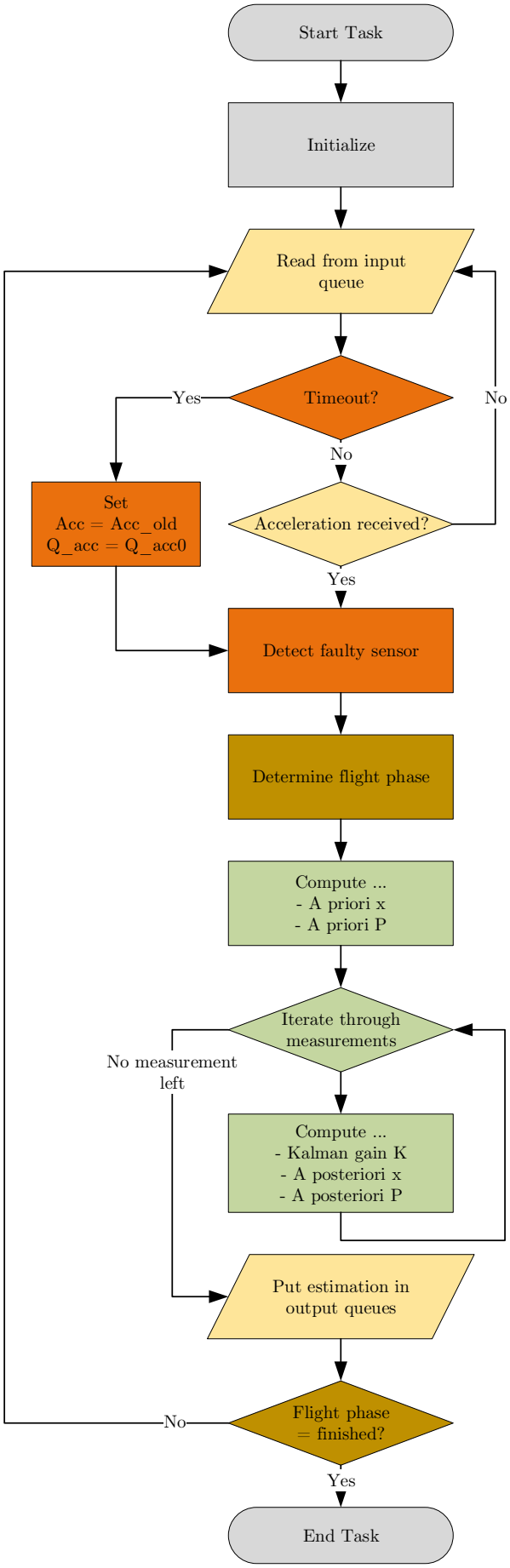


Figure 9: Flowchart of the sensor fusion task; green: State estimation, brown: Flight phase detection, yellow: Input/output (i/o) control, red: Faulty sensor detection, grey: Task management

4.4.3 Faulty Sensor Detection

The detection of faulty sensors is an important part in increasing the robustness of the whole system. Such a detection could use the prediction $\hat{x}_k$ and compare it to the measurements in the measurement vector y_k. If one recalls equation 2.10 from section 2.1.5, this difference equals the innovation r_k.

Probability theory could now be used by taking the uncertainty of $H\hat{x}_k$, which is $H\hat{P}_k H^T$, as well as the uncertainty of y_k, which is R, to test the innovation of each sensor. However, it must be said that developing and implementing a state of the art error detection is out of the scope of this work. Therefore, an implementation is omitted in this project.

4.4.4 Flight Phase Detection

As explained in chapter 3.2.2, the current flight phase needs to be known to set the covariance matrices R and Q appropriately. This detection is done by comparing the acceleration as well as the estimated velocity with thresholds stated in table 7. These thresholds are in no way set definitely. Instead, they can be changed in the C implementation.

To reduce the risk of detecting certain events too early, the respective thresholds have to be exceeded or undershot at least a number of times.

Table 7: Flight state transitions

Event	Threshold		No. of Times	Current State	Next State
Launch	a_{mes}	$> 20m/^2$	4	PRELAUNCH	TAKING_OFF
Burnout	a_{mes}	$< 10m/s^2$	4	TAKING_OFF	BRAKING
Apogee	v_{est}	$< 0m/s$	1	BRAKING	APOGEE

A structure was implemented so that changes to the thresholds can be executed quickly. Each event consists of the threshold itself, the current and the next flight phase, a boolean which has to be set true, if the value must be greater than the threshold, a reference to the value which needs to be checked as well as the number of times the threshold needs to be exceeded. Multiple thresholds can be set by adding entries to the "thresholds" array.

```
1  typedef struct{
2    phase_t   currentPhase;
3    phase_t nextPhase;
4    float threshold;
5    bool greaterThan;
6    float* value;
7    uint8 noTimes;
8  } threshold_t;
9
10 threshold_t thresholds[N_THRESHOLDS];
11 thresholds[0] = (threshold_t) {.currentPhase = PRELAUNCH,
12                                 .nextPhase = TAKING_OFF,
13                                 .threshold = 20,
14                                 .greaterThan = true,
15                                 .value = &u,
16                                 .noTimes = 4};
```

4.4.5 Sensor Failure

What needs to be discussed last is what happens when a sensor failure occurs. A possible solution to this problem is stated. However, this approach was neither tested nor implemented since a final answer can only be derived when the actual implementation of the sensor sampling task as well as the data aggregation task is known.

Concerning the sensor readings which are represented in the measurement vector, a sensor failure can just be handled by stopping to put them into the state estimation input queue. The sensor fusion task, in simple terms, does not care whether certain measurements arrive or not. The respective flag in the "avbl" variable is then just set to zero as explained in section 4.2.1.

A bigger problem occurs if no more or just infrequently acceleration readings are put into the state estimation input queue since the whole algorithm is depending on that sampling frequency. The data aggregation task therefore should guarantee that in a rate of 1 kHz acceleration measurements are sent to the sensor fusion algorithm, even if they are wrong. If that fails as well for whatever reason, a queue receive timeout can be set at $T_{timeout} = 1.5T_s$. If such a timeout occurs, the last nominal acceleration received could be used in the algorithm. Furthermore, the system noise covariance matrix Q could be set to the same value as if there was a faulty sensor. After processing the filter algorithm, the task would have to be delayed for $0.5T_s$. The timeout would now be set to $T_{timeout} = T_s$. If now the sensor was sampled again, the process could proceed as normal. If, however, the sensor anomaly still continued, the queue receive function would exit and the filter algorithm would have to be called twice.

Having a look at figure 10, the variable k equals the tick time of the RTOS. The tick frequency would have to be set at $T_{tick} = 0.5T_s$. Every two ticks, the acceleration sensors is tried to be sampled. The green arrow at tick number k and $k+12$ illustrate a successful sampling. The red flashes show a failed sampling. The blue boxes represent a filter iteration each. Note that after tick k, a measurement is received from the queue. After processing, the queue is called again with a timeout of $1.5T_s$. Note that the sampling at $k+2$ failed and therefore, a timeout occurs at $k+3$ (marked with three red exclamation marks). After one filter iteration as described above, it is waited until $k+4$, where the queue receive function is called again. This process continues as described above until at tick $k+12$ the sampling succeeded and the nominal case continues. Note that with this solution, the Kalman filter function would be called exactly as many times as a sensor sampling attempt was made.

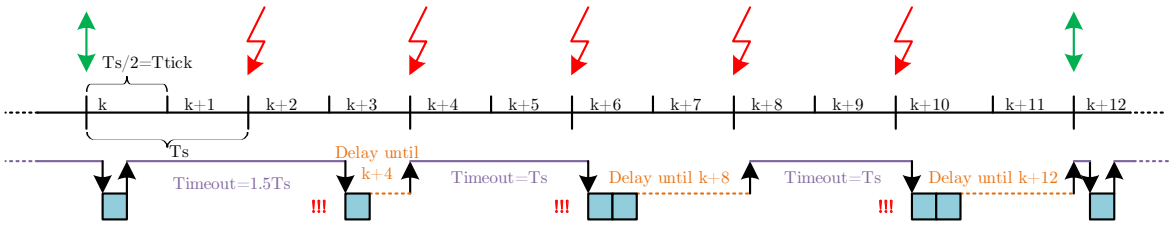


Figure 10: Possible solution on how to handle faulty accelerometer.

This approach, however, is not ideal since it is quite complicated. It furthermore must be guaranteed that the tick period of the FreeRTOS, which defines the interval in which the blocked and sleeping tasks are checked and therefore defines the smallest interval in which a timer can be set asleep, is equal or a rational fraction of $0.5T_s$. It must be further noted that FreeRTOS normally does not support tick frequencies above 1 kHz. This stays in contradiction to the fact that at least 2 kHz tick frequency would be needed to support an acceleration sampling period of 1 ms.

The following example C implementation shows how this problem could be implemented.

```
1  start_time = FRTOS1_xTaskGetTickCount();
2  res = FRTOS1_xQueueReceive(stateEstimationInput_MsgQueue, &raw_data, ...
       timeout);
3  timeout = (portTickType) (1.5*(TS_MS/portTICK_RATE_MS)+0.5);
4
5  // flight phase detection
6  // faulty sensor detection
7
8  if(res == pdFALSE){
9    Q = A_VAR_FAILURE;
10   u = u_old;
11   switch(acc_state){
12   case NOMINAL:  // first failure
13     delay = (portTickType) (0.5*(TS_MS/portTICK_RATE_MS)+0.5);
14     timeout = (portTickType) (1*(TS_MS/portTICK_RATE_MS)+0.5);
15     acc_state = FAILURE;
16     break;
17   case FAILURE:  // second, ... failure
18     delay = (portTickType) (1*(TS_MS/portTICK_RATE_MS)+0.5);
19     timeout = (portTickType) (1*(TS_MS/portTICK_RATE_MS)+0.5);
20     // call kalman function so that the function is called twice
21     break;
22   }
23 }
24 // call kalman function and put in queue
25 FRTOS1_vTaskDelayUntil(&start_time, delay);
```

4.4.6 Implementation of Barometric Pressure Model

Something which was not addressed before was the implementation of the barometric pressure model. Since the calculation of the altitude from the barometric pressure is performed outside of the algorithm function, it is not included into the measurement of computational effort and stack usage. However, it was seen that the standard implementation, as it is mentioned in section 3.1.1 and used in the firmware implementation of project TELL, uses up to 16'000 clock cycles per calculation. Since two barometers are used, two function calls would be needed which sums up to 32'000 clock cycles.

Such a standard implementation in C by using the "pow" function provided by the "math.h" library would look as following.

```
1      float bar_mod_exp = init_data.T_grad*R_CONST/(M_CONST*G_CONST);
2      float bar_mod_factor = (init_data.T_0+273.15F) / init_data.T_grad;
3      y[1] = (1.0F - (float)pow(raw_data.data / init_data.P_0, ...
           bar_mod_exp)) * bar_mod_factor;
```

However, tests have shown that an implementation by using the powf (power float) or the exponential function in combination with the natural logarithm (eq. 4.8) is far more efficient.

$$h = *\frac{T_0}{T_{grad}}\left(1 - e^{ln\left(\frac{p}{p_0}\right)\frac{R*T_{grad}}{M*g}}\right) \tag{4.8}$$

Figure 11 shows the measured clock cycles needed for every barometer reading by using these different calculating methods over time. It can be seen that the needed clock cycles when using the powf function and the exponential / logarithmic function is with 1028 and 728 cycles quite low compared to almost 20'000 used by the calculation with the pow function.

There is no certainty in explaining the spikes which can be seen in all curves. However, one possible explanation could be that the calculation is interrupted by another task which adds additional clock cycles. The fact that there are much more spikes on the pow measurement adds to this assumption because the calculation is more likely to be interrupted since it has a longer execution time than the others.

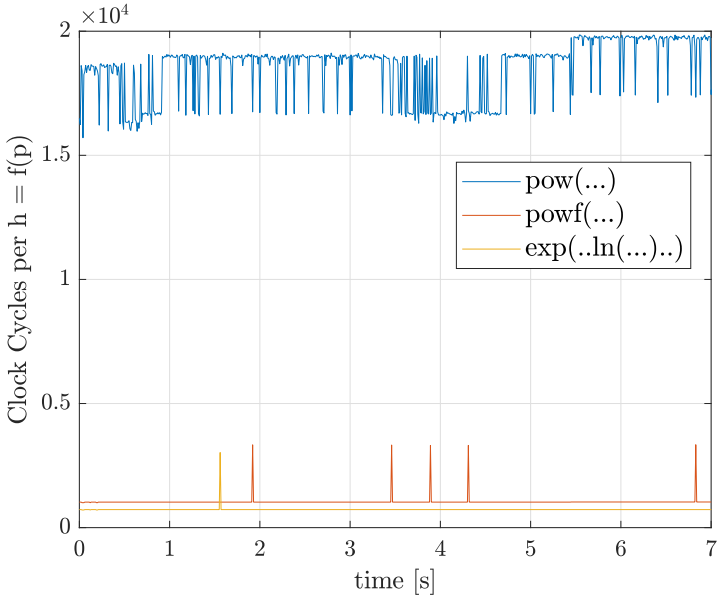


Figure 11: Clock cycles needed per barometric pressure to altitude transformation.

It is a logical conclusion to use the formula stated in equation 4.8 as a base for the implementation of this problem.

5 Verification and Testing

Having derived a new system model as well as a C implementation of that algorithm, choosing proper parameters, verifying the algorithm with the chosen parameters as well as testing the firmware implementation are the next steps to conduct.

5.1 System Model

Verifying the system model and determine the filter parameter are closely linked together. The problem is: How can the Kalman filter be verified, without knowing ...

- ... the ground truth or in other terms the real trajectory of the rocket?
- ... the characteristics of the sensors used?

Simon [6, p. 298] states that a Kalman filter can be verified by looking at the innovation r_k which is, as previously shown:

$$r_k = y_k - H_k\hat{x}_k = (H_k x_k + v_k) - H_k\hat{x}_k = H_k\left(x_k - \hat{x}_k\right) + v_k = H_k w_k + v_k \tag{5.1}$$

As stated in section 2.1.1, the system noise w as well as the measurement noise v are both white with zero mean. He further states that if this assumption holds, the innovation r must also be white with zero mean. Therefore, checking the resulting innovation can be a way of declaring whether the parameters are set correctly and whether the system model holds. Furthermore, it is recommended to test multiple parameters and choose the one that matches theory the best. This process is also called filter tuning.

Mohan M et al. [3] mention that the problem of filter tuning and verifying is present in all variations of Kalman filtering. They also have noted that knowing whether the estimates are correctly or not is a matter of observing the innovation. Concepts of automated iterative algorithm to choose the best performing parameters by minimizing respective cost functions are introduced. However, implementing these algorithms would go beyond the scope of this report.

Marchthaler and Dingler [5, p. 118] state that the covariance matrices R and Q can be obtained by analyzing measurements from the respective sensors. Since no data from test flights with the sensors for the competition in summer 2019 were available, a Matlab script was implemented to extract and analyze test data sets such that this script can be used in the future to quickly obtain parameters from such data. This script was based on the methods used by Kurmann [2], where he loaded data from low altitude test flights, fitted it with second order polynomials and extracted the noise. Furthermore, the noise was analyzed with the Yule Walker equations to generate an AR model with which newly synthetized noise could be made. The implemented script also gives a recommendation for the R and Q matrices.

A second script with the name "verifier.m" was implemented. With this script, different test data sets can be loaded and tested. Every previously analyzed test data can be used to feed the simulation. Furthermore, a simulated flight trajectory, modulated with noise from each respective test data set can be used as a test input. The R and Q matrices also can be chosen independently. The script will generate multiple plots. One of them, the change in innovation over time, can be checked if it is white noise with zero mean. Furthermore, the estimation can be compared to the measurements as well as to the simulated trajectory, if the simulated sensor data are used as the input.

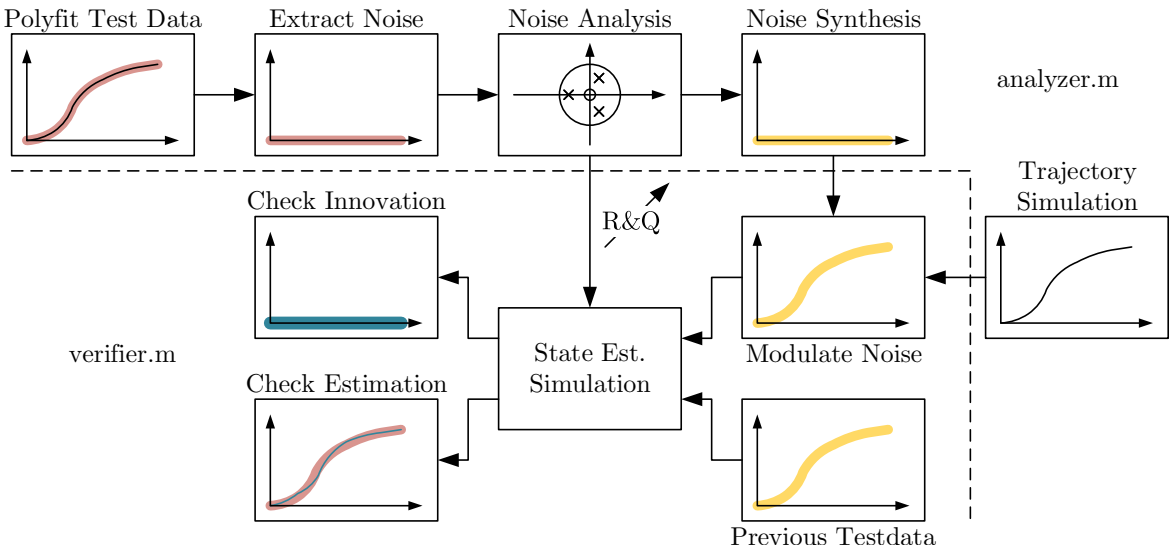


Figure 12: Block scheme of the two Matlab scripts "analyzer.m" and "verifier.m" which can be used to find values for the R and Q matrices as well as for verifying the state estimation algorithm.

With these two scripts, it should be possible to choose and experiment different values for R and Q and verify the state estimator.

Now, the previously stated questions can be answered.

In order to verify the algorithm without knowing the ground truth as well as the sensor characteristics, the innovation and the estimation compared to the measurements can be observed (compare "Check Innovation" and "Estimation" in figure 12). This process can then be used to either tune the parameters to achieve better filtering results or to choose parameters for newly chosen sensors after low or high-altitude test flights were conducted. However, in the scope of this work, it is needless to tune the parameters since changes in terms of the actually used sensors will occur. Therefore, only the verification of the model will be conducted.

5.1.1 Verification with simulated data

Two kinds of test data sets were used to feed the algorithm. Firstly, test data generated by Kurmann which consist of a simulated trajectory with modulated noise was used as the algorithm input. This contained two pressure readings, one GPS measurement as well as the measured acceleration. Furthermore, the covariance matrices were set according to the variances of the modulated noise during engine burn and the braking phase. These values can be found in table 8.

Table 8: Used parameters for state estimator verification

R_{ii}	**Burn**	**Brake**	Q_{ii}	**Burn**	**Brake**
$\sigma^2_{h_{GPS}}$	1.5290	2.1497	σ^2_a	5.6293	0.4342
$\sigma^2_{hp_1}$	2.9702	9.4840			
$\sigma^2_{hp_2}$	2.7017	6.2824			

Furthermore, the initial estimation error covariance matrix was chosen to be a diagonal matrix with each entry being 0.5. The initial state was chosen to be a zero vector since the simulation starts at an altitude of 0 m above sea level.

As previously stated, the innovation as well as the estimation are recorded. It is expected that the innovation will be almost white with zero mean. Almost, because the noise on the GPS and the pressure sensor are coloured [2, p. 19]. The "whiteness" will be observed by looking at the power spectrum.

Furthermore, the progress of the estimation error covariance matrix is recorded. It is expected that the values of this matrix converge, how long it takes, however, cannot be predicted since it really depends on the chosen algorithm parameters.

5.1.2 Verification with Real Data

Since the testing with the simulated data in some respects just removes noise which previously was synthesised and modulated, also real measurements are tested. This should further show the robustness of the filter.

To do so, raw data from low altitude flights, conducted by the EPFL with regards to the competition in summer 2018, were used. These data sets consist of one altimeter reading, which was used for both, the p_1 and p_2 measurement, as well as one accelerometer reading. To test the algorithm like that, the availability flags, as introduced in section 4.2.1, were simply set to zero for the respective missing sensors.

Both, the altimeter as well as the barometer were sampled at roughly 67 Hz. Furthermore, the data set stretches over all phases of the flight, from "before ignition" until the landing of the rocket by parachutes. This will further be of interest since the response of the algorithm to these previously not considered flight phases can be seen.

To be more specific, the data from the first EPFL flight from 18.11.2017 were used for this test. Concerning the algorithm parameters, they were obtained by using the analyzer.m script on the EPFL flight conducted on the 24.02.2018.

The script resulted in the following values:

Table 9: Generated parameters by verification script

R_{ii}	**Burn**	**Brake**	Q_{ii}	**Burn**	**Brake**
$\sigma^2_{hp_1}$	1.1677	2.1881	σ^2_a	0.9938	0.0590
$\sigma^2_{hp_2}$	1.1677	2.1881			

For the initial parameters, the altitude in the initial state was set to 412.8 meters, since that was the altitude above sea level at the place the rocket was launched. The initial error covariance matrix was also set to a diagonal matrix with each entry being 0.5.

Further, the algorithm interval period T_s had to be set to $1/(67Hz)$ to calculate the predictions properly.

The same output is recorded in this test as in the test with the simulated data. Furthermore, it is expected that the algorithm is working in similar ways as it did for the data described in the previous section.

5.2 Firmware

Having developed, verified and ported a state estimation algorithm onto an embedded system, tests of both, the reliability as well as the consumption of resources on the MCU can be made.

5.2.1 Test Environment

As it was stated in section 1.3, an STM32F407 ARM Cortex M4 microcontroller is used by ARIS for the competition in summer 2019. Furthermore, the Keil uVision5 IDE (integrated development environment) is applied. The decision that this IDE is used was made fairly late, the freeware license of the IDE has a limitation on compiling 32 Kbyte code at maximum and no professional license was available at the time of development. This, with the fact that no experience with this software was present, the decision of using products from NXP was made. Since the algorithm does not use any periphery, any other MCU equipped with an ARM Cortex M4 processor with FPU could be used instead without expecting any differences in the test results. Therefore, an NXP Freedom Development Platform for Kinetis (FRDM-K22F) was used to test the implementation. The Kinetis IDE, in combination with the GNU C Compiler, is easy to use and well documented. Furthermore, the FRDM-K22F development kit includes an SD card slot which can be used for loading big data sets and saving the respective results. It also should not be difficult to take the source code over to the STM platform since on both FreeRTOS is used.

Speaking of the test environment concerning the firmware, FreeRTOS was set up with multiple dummy tasks in order to simulate a possible use of the algorithm. This structure can be seen in figure 13.

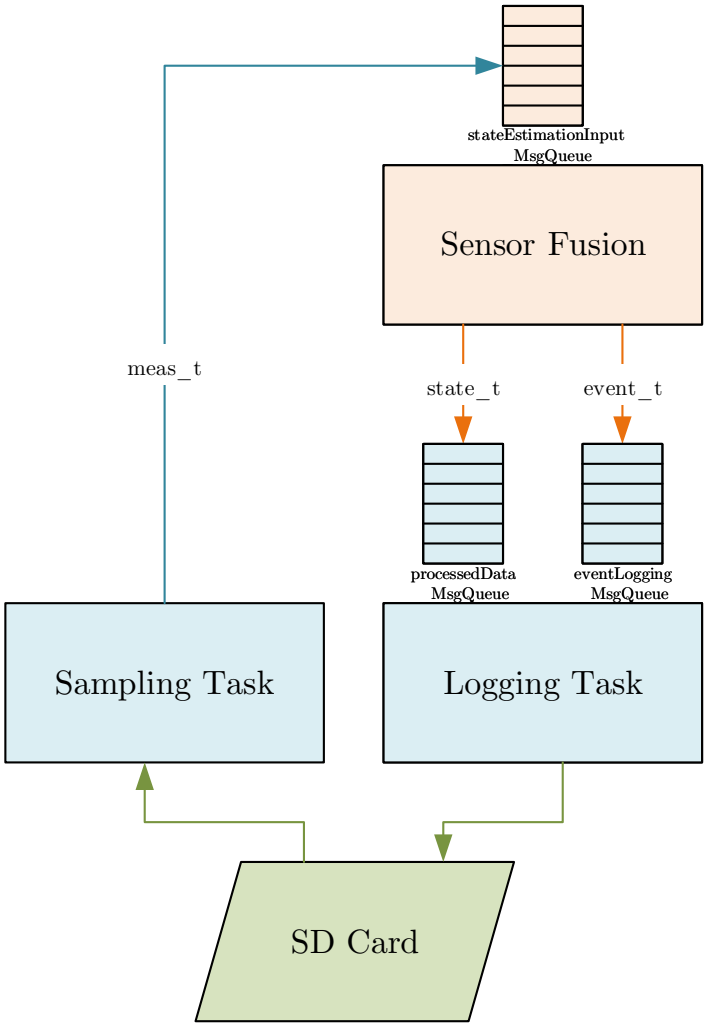


Figure 13: Test environment for firmware tests

5.2.2 Test Procedure

The reliability of the implemented firmware, the added parts described in section 4.3 and the used resources in terms of computational time and stack usage can be measured and tested in order to evaluate the stated requirements from section 1.2. The computing time and stack usage is also measured for the multiple design iterations performed such that the improvements can be verified. This includes the proposed algorithm by Kurmann, the newly introduced model, the usage of floating point number representation and sub sequentially the use of sequential Kalman filtering.

Accuracy

The accuracy describes in this context the difference in the results of the algorithm running in Matlab and running on the embedded system. In order to perform this step, the same data set which was used to verify the filter in Matlab was put on the SD card as described in section 13 and then fed into the running algorithm on the embedded system. The resulting state estimates for each discrete point in time are then saved on the SD card. Afterwards, these results are compared with the results in Matlab. In theory, it should not make any difference in what kind of input sequence is chosen for this test if all the parameters are equally set since both implementations should perform identically well or bad; hence, it was chosen to use the simulated flight trajectory from Kurmann [2] with modulated noise as an input for this test.

Additionally to the estimation, the innovation is logged on the embedded system in order to verify the system model on the MCU. Equal results are expected for this test compared to the results from section 5.1.1.

Flight Phase Detection

The flight phase detection algorithm is tested on the embedded system as well. The events generated by the implementation are logged and later analyzed. The resulting points in time, where the events which are described in section 4.4.4 occur, should be equal to the indices which resulted when modulating the noise onto the simulation.

Computational Time

Styger [17] mentions that the unsigned integer value at address *0xE0001004* in RAM of an ARM Cortex M4 processor can be read which represents a counting value, incremented at every clock cycle. While debugging, by looking at the deltas between these values, the needed cycles between two lines in the source code can be determined.

This principal is used to determine the execution time of the Kalman function.

Stack Usage

The FreeRTOS Task Aware Debugger [18] plugin for the Eclipse based Kinetis Design Studio was used to obtain the effectively used stack by the different algorithms. The needed stack is expected to stay the same at each iteration.

6 Results

This chapter shows the results from the tests presented in chapter 5. As described, tests on the algorithm in Matlab to verify the algorithm as well as on the embedded system, to test the firmware implementation were conducted.

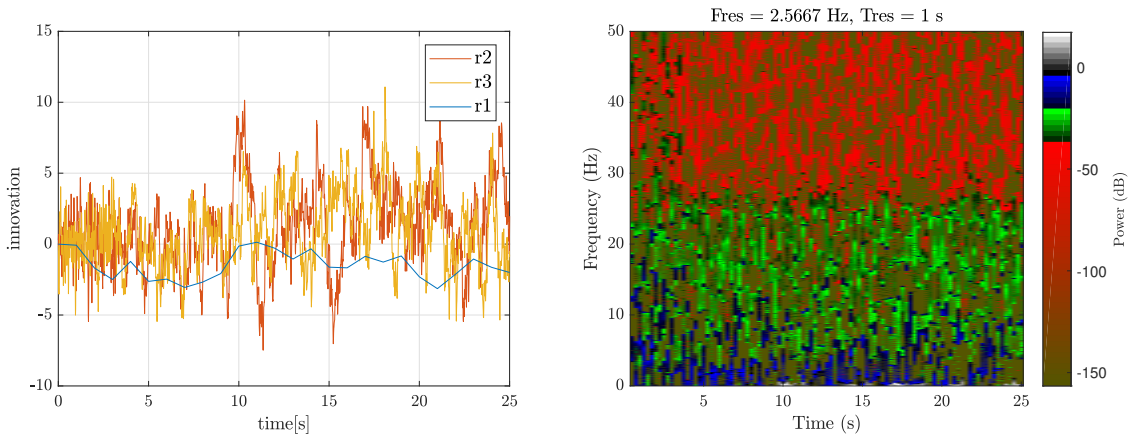
6.1 Verification of System Model

First, tests with a simulated trajectory to verify the performance of the algorithm when being fed with sensor data which actually reaches an altitude of 10'000 ft. were conducted. Furthermore real sensor data, obtained by EPFL from a low altitude test flight, were used to test the implementation and to gather information about the reaction of the algorithm to authentic sensor data.

6.1.1 Simulated Sensor Data

When having a look at the innovation, it must be noted that r_1 equals the innovation of the first element in the measurement vector, which is the GPS reading, r_2 and r_3 on the other hand equal the innovation regarding the two altitudes calculated from the barometric pressures. In the first 3 seconds, which covers the engine burn phase, the innovation caused by the pressure sensors is fairly white with zero mean, which is consistent with the criteria established in section 5.1. This can also be seen in the spectrum in figure 14b, where in the first couple of seconds, the colors are uniformly distributed. However, when the simulated rocket is in free flight, the innovation tends to be on the positive side which indicates that the prediction tends to underestimate the altitude more than overestimating it. This can occur when the acceleration in the system input is constantly too low. Since the acceleration is in fact a composite of the measured acceleration and the acceleration offset, which was estimated by the algorithm at around 2 meters (figure 15b), it is also possible that the acceleration offset is estimated wrongly.

In figure 15c, the fact that the error covariance matrix converges is visible. The variances of the estimated velocity and the acceleration offset reach a very small value. The reason is that both states are not directly coupled to a noisy measurement. In every algorithm iteration, the variance of the estimated velocity is just increased by T_s times the variance of the system input. With T_s being 1 millisecond, the variance of the velocity is not increased significantly.

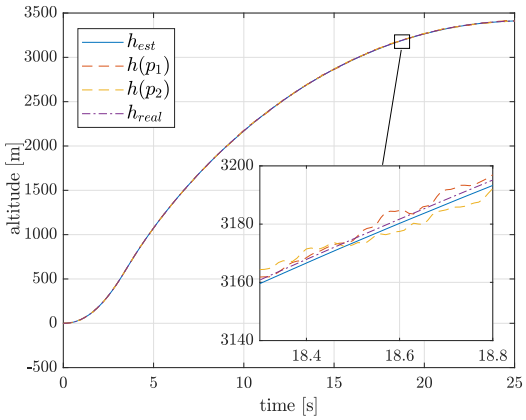


(a) Innovation r

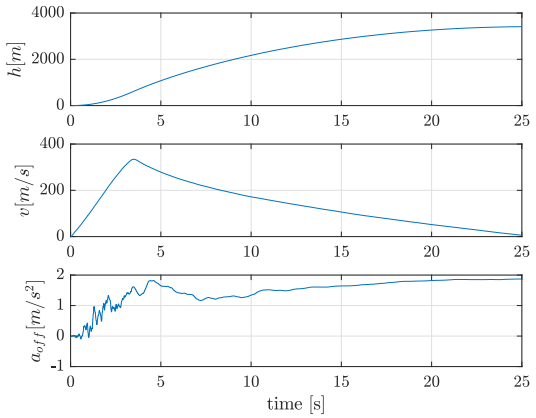
(b) Spectrum of innovation $r2$

Figure 14: Verification signals from simulated sensor data

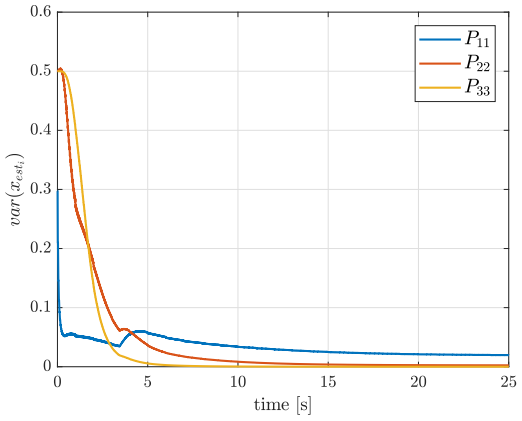
Having said that the innovation is not perfect and looking at the estimation in figure 15a, it can be obtained that it is smooth and within the realms of possibility compared to the measured altitude from the simulated barometric sensor data and the "ground truth".



(a) Estimated vs. measured altitude



(b) Estimated altitude, velocity and acceleration offset



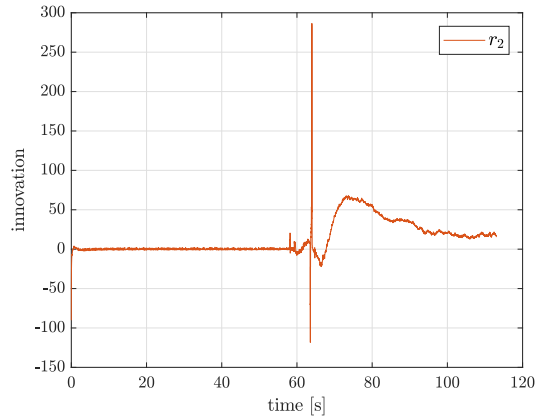
(c) Error covariance matrix P

Figure 15: In and output of algorithm with simulated sensor data

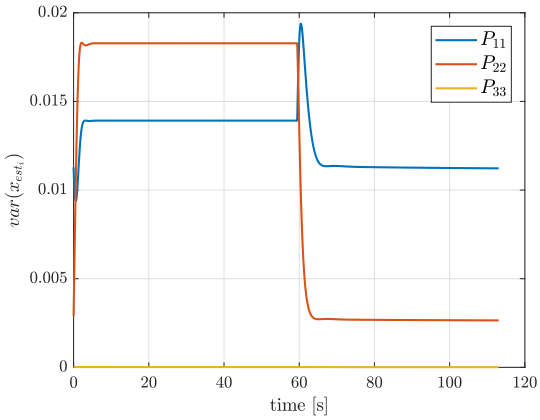
In conclusion it can be said that even though the results from the innovation are not perfect, the algorithm works and has an output which appears to be useable. In terms of the algorithm parameters, a verdict cannot be made. They must be determined with the provided tools when low altitude test flights have been conducted with the sensors from the final concept.

6.1.2 Real Sensor Data

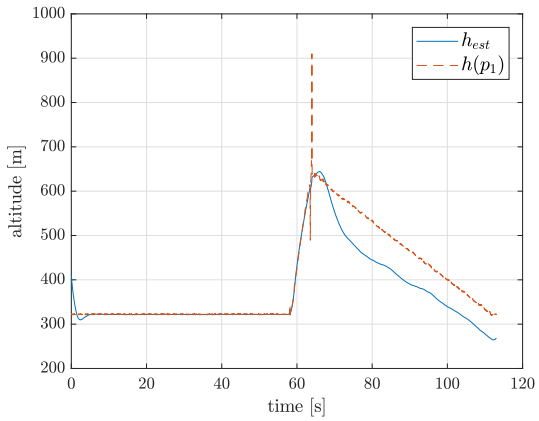
When simulating the data from the low altitude test obtained by EPFL, from figure 16c, it can be seen immediately that after the parachute deployment at around 65 seconds, the altitude estimation is above and then under the barometer measurements. Until impact on the ground, the state estimation never recovers from this error. This can also be seen in the innovation in figure 16a. The filter tries to correct its mistake, but it does not since it sees the altitude estimation with a variance of around 0.01 (figure 16b) much more accurate than the variance of the measurements, which are around 1 (compare table 9). However, it does increase the acceleration offset which can be seen in figure 16d.



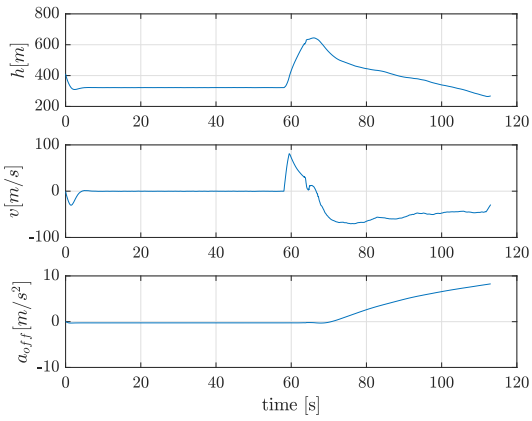
(a) Innovation r



(b) Error covariance matrix P



(c) Estimated vs. measured altitude



(d) Estimated altitude, velocity and acceleration offset

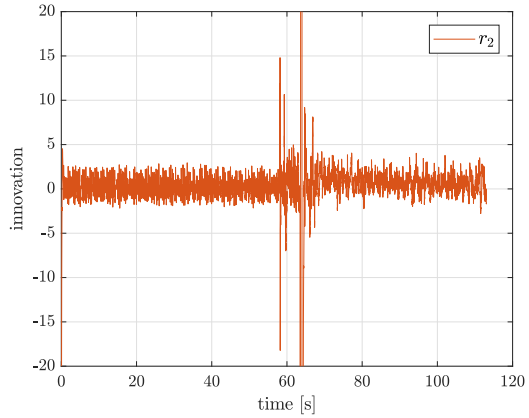
Figure 16: In and output of algorithm with real sensor data and with first method of determing G_d

The question is why this effect did not occur in the previous test. There, the acceleration is sampled 10 times faster than the barometers and hence, the uncertainty of the acceleration is accumulated 10 times before it is compared to a measurement. Furthermore, with the much higher sampling rate, the error can recover much faster. And lastly, the simulation reaches just until apogee. Until there, the estimation with the real flight data also works.

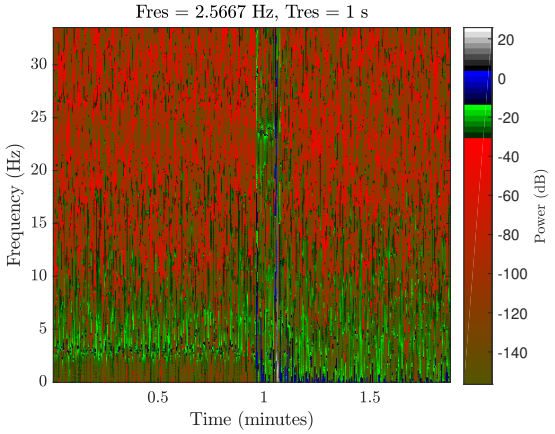
One way of resolving this issue is changing the method of obtaining the matrix G_d which was introduced in section 2.1.2.

$$G_d = \begin{bmatrix} T_s \\ 1 \\ 0 \end{bmatrix} \tag{6.1}$$

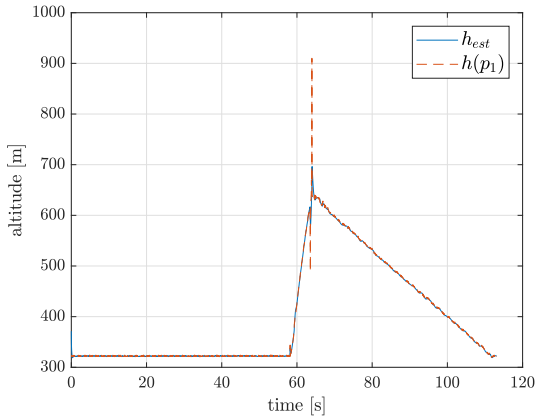
This would result in a much higher influence of the system noise covariance matrix Q on the estimation error covariance matrix P. Simulation with this G_d then leads to an innovation (figure 17a) which matches theory much better. In figure 17b can be seen that the power is not perfectly, but quite well distributed over the spectrum, except for the high power disturbance which stands out at around 60 s. The problem after parachute deployment does not exist anymore (figure 17c). However, one trade-off for this tactic is that the estimated velocity is much noisier than before.



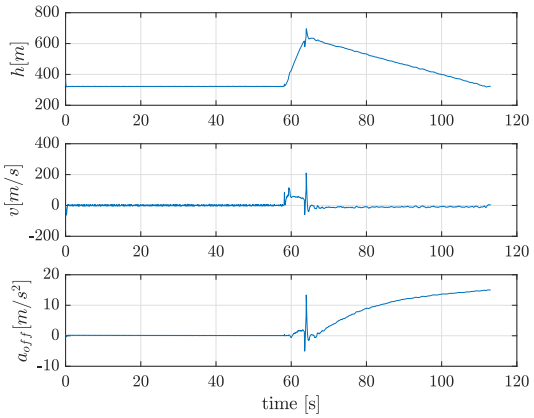
(a) Innovation r



(b) Spectogram of innovation $r2$



(c) Estimated vs. measured altitude



(d) Estimated altitude, velocity and acceleration offset

Figure 17: In and output of algorithm with real sensor data and with second method of determing G_d

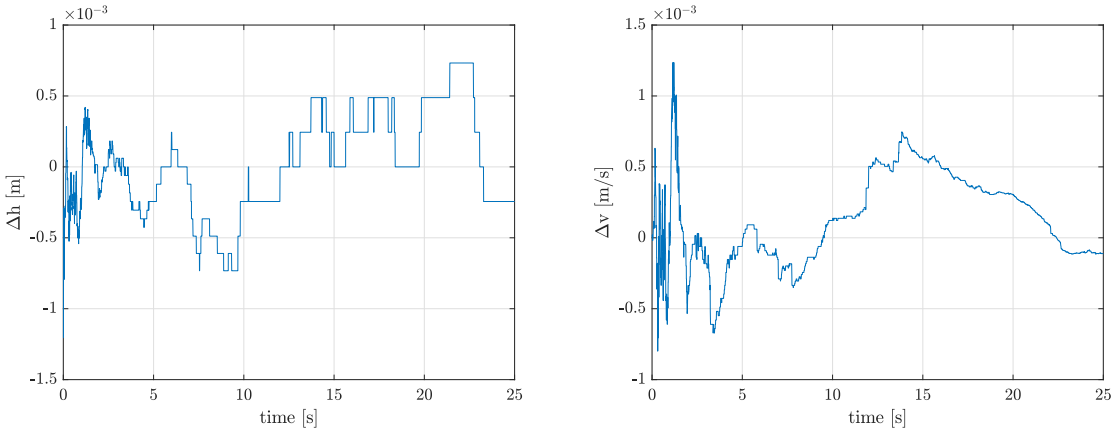
Concluding, it needs to be said that tests with low altitude flight data must be obtained which will need to be sampled at comparable sampling rates to those of the final system. Then, a verdict on what method should be used to discretize the G matrix can be made.

6.2 Verification of Firmware

In this section, the firmware is verified by testing the accuracy of the C implementation by comparing it to the results generated by Matlab. Furthermore, the consumption of resources was measured for each design iteration.

6.2.1 Accuracy

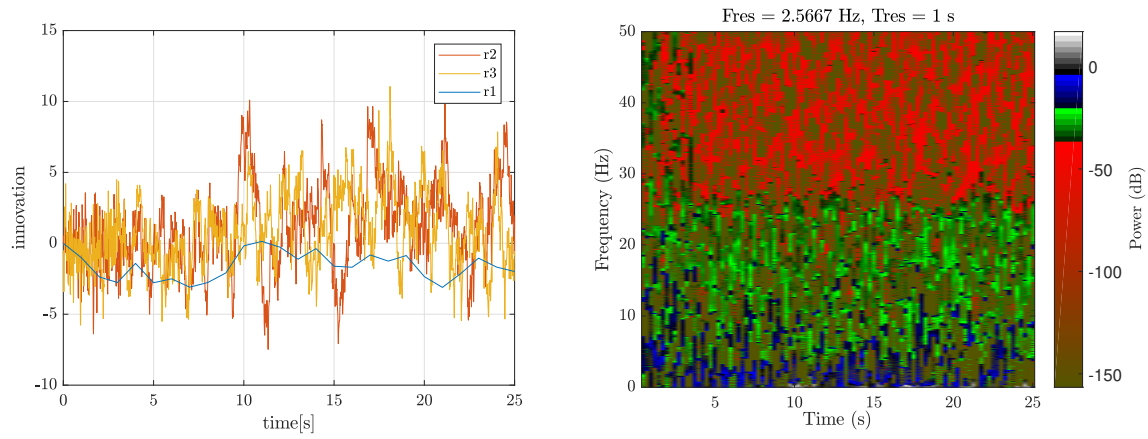
As can be seen in figure 18, there is a difference in the algorithm that is resulting from Matlab and the C implementation running on the embedded system. During engine burn, which happens in the first three seconds, both, the difference in the estimated velocity as well as the estimated altitude are quite noisy. Afterwards, the error seems to be much more deterministic. The reason for this is most likely the fact that the data had to be converted to strings of characters which then again had to be converted back to the floating-point representations. The differences are therefore most likely the outcomes of numerical effects. The root mean square error is 0.34 mm for the difference in altitude and 3.6 mm/s for the difference in velocity.



(a) Difference in estimated altitude (b) Difference in estimated velocity

Figure 18: Comparison of results from ARM Cortex M4 and Matlab ($\tilde{x}_{k_C} - \tilde{x}_{k_{Matlab}}$)

Concerning the innovation, the results from figure 19 seem to be almost exactly the same as it was seen in section 6.1.1 and thus as expected.



(a) Innovation r (b) Spectogram of the innovation r_2

Figure 19: Innovation on the ARM Cortex M4 processor

6.2.2 Flight Phase Detection

Logging of the events when changes to the flight state occur resulted in the times stated in table 10. There is a delay of around 16 ms after launch and 63 ms after burnout. These numbers show that a detection of the current flight phase with the stated implementation is possible. The thresholds, as well as the number of times, which need to be exceeded until the event occurs, can be changed in the future.

Table 10: Comparison: Event time given by simulation and obtained by testing

	Simulation	**Testing**
Launch Time [s]	0	0.016
Burnout Time [s]	3.450	3.513

6.2.3 Consumption of Resources

By having a look at the pie charts in figure 20, where 100% equals the consumed resources by Kurmann's proposed algorithm, it can be seen that both, the computational effort needed for one algorithm iteration as well as the used stack by the algorithm could be reduced by over 90% respectively 80%.

The greatest improvement could be achieved by using an improved model with smaller vectors and matrices. When recalling section 3.2, it was expected that a reduction of the computing time by 78% should be possible. This is really close to the measured 75%, which further confirms the choice of this model.

Using single precision representation in the algorithm computation also caused an improvement by further 19%. This is an additional reduction in computing time of 73% compared to the improved model. This also almost matches the expected 68% stated in section 4.1.1.

Sequential Kalman filtering further reduced the needed clock cycles by around 1% compared to the proposed model. This, however, is just the improvement referring to the worst-case scenario (WCS), namely, when all measurements (GPS, barometer 1 and 2) arrive at the same time. At best case (BCS), which means only an accelerometer reading arrives, 9'585 clock cycles are needed to compute the estimated state. Since the barometer is sampled 10 times and the GPS 200 times slower than the accelerometer (compare table 3), most of the times, the best-case scenario holds.

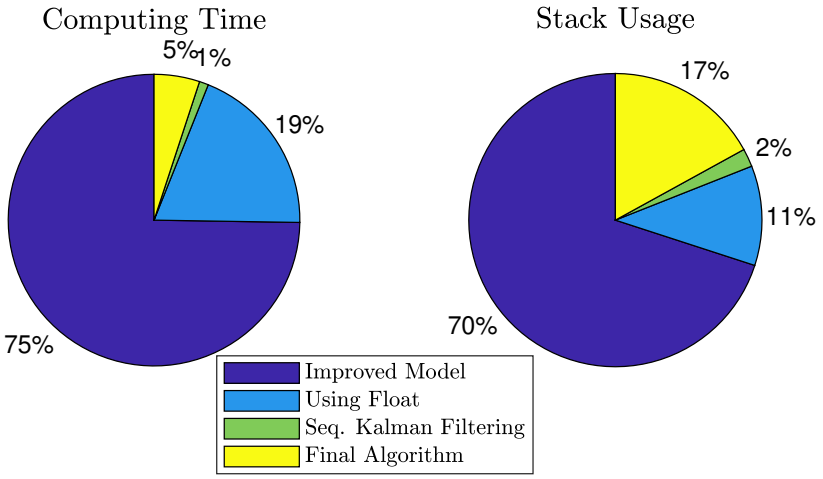


Figure 20: Improvement in computing time and stack usage

In terms of stack usage, the needed stack size of 3'160 bytes by the proposed algorithm could be reduced to 528 bytes which equals an improvement of 83%. The reasons for these numbers are the same as they were for the improvement in computing time. Smaller matrices and vectors meant automatically smaller arrays for storing these values. Using single instead of double precision also meant that 4 bytes instead of 8 were used for each individual number. 528 byte is furthermore an acceptable number since it is, even with a safety margin of 100%, less than 1% of the total 192 Kbyte RAM provided by the STM32F407 MCU. These numbers can also be verified by looking at table 11.

Table 11: Improvement of resources needed by the single iteration steps

	Clock Cycles per Iteration	**Bytes of Stack**
Proposed Model	395'713	3'160
Improved Model	104'035	956
Improved Model with single precision	26'575	608
Sequential Kalman Filter	24'002 (WCS) 9'585 (BCS)	528

7 Conclusions

As stated in section 1.1, the aim of this term project was to port a proposed sensor fusion algorithm onto an embedded system. This sensor fusion algorithm is going to be used on a sounding rocket to estimate the current altitude and vertical speed. This is then applied to control so called air-brakes to reach an altitude of exactly 10'000 ft. AGL.

Certain issues were found in this proposed algorithm. To meet requirements, a new system model was introduced. The improved system model then contained the current altitude and vertical speed of the rocket as well as the accelerometer offset in its state vector. The accelerometer readings were used as the system input. The GPS altitude, as well as the altitude, calculated from two barometric pressure sensors were used in the measurement vector. The tilt of the rocket during the flight was neglected, which means that the sensor readings from the accelerometer are used directly in the system model which is represented in the Cartesian coordinate navigation frame.

To further decrease the resource consumption of the algorithm in terms of computational power and stack usage, the algorithm was changed such that it uses single precision floating point numbers which run more efficiently on a processor with a floating-point unit. Furthermore, sequential Kalman filtering was implemented which eliminates the need of matrix inversion and enables a more efficient way of dealing with multiple sampling rates.

In Matlab the improved algorithm was prepared for converting it to C. Matlab Coder was used as a tool to quickly and efficiently perform this step. This approach was chosen to enable changes being implemented quickly since the concept of hardware and firmware was not set during the execution of this term project.

The C implementation of the algorithm was then embedded into a firmware framework using FreeRTOS. A concept on how to detect the current flight phase was implemented. Furthermore, possible approaches on how to detect faulty sensors as well as how to deal with measurements which do not arrive were derived but not implemented.

To determine the parameters for the state estimation algorithm, a Matlab script was implemented with which sensor data from test flights can be loaded and analyzed. The tool also allows to simulate the algorithm with different parameters. This script is also used to verify the correctness of the algorithm.

Using this tool, it was found that the algorithm model is correct and produces reliable results if the parameters are chosen correctly. However, the accuracy of the algorithm cannot be quantified by a number.

To test the firmware, a test framework in C was implemented. Test data were loaded onto the embedded system by using an SD-card. The results derived by the algorithm running on the embedded system were then compared to the results from Matlab. Furthermore, the consumption of resources was measured.

What resulted was an implementation which needs 24'000 clock cycles per iteration in worst-case and 10'000 clock cycles in a best-case scenario. Furthermore, the algorithm, running as a FreeRTOS task, needs 528 bytes of stack. Compared to the proposed algorithm, this is a reduction of more than 80% in both cases. Furthermore, the C implementation produces almost equal results as the Matlab implementation.

7.1 Evaluation

The results can now be checked with the requirements stated in section 1.2.

System Load

The system load can be summarized with the measured clock cycles per second and stack usage.

24'000 clock cycles are needed in the worst-case scenario for one algorithm iteration. Assuming the iteration period is set at $T_s = 1ms$, 24 million clock cycles are needed per second. This equals around 14% of the whole system capacity and is below the aim of 56 million at maximum.

Concerning the needed stack, with only 528 bytes, this is well below the set constrain at 10 Kbytes.

Precision

In terms of precision, the comparison of the results from Matlab and the C implementation gave a difference which peaks at roughly 0.8 mm. The root mean square error is at 0.36 mm which is clearly below the stated requirement. Furthermore, the innovation has the same characteristics on the embedded system as in the matlab environment. This is a good indication that the algorithm works in Matlab as well as on the embedded system.

Flexibility

In terms of flexibility, with Matlab Coder a tool was used that provides the requirement that changes in the algorithm can quickly be adapted to the target device.

Summary

The table from section 1.2 can be recalled and extended with the findings from this chapter.

Table 12: Summary of fulfilled requirements

Requirement	**Aim**	**Achieved**	**Conclusion**
Computational time	<56 M/s	24 M/s	✓
Stack usage	<10 Kbyte	528 bytes	✓
Precision	<1 cm	0.8 mm	✓

7.2 Outlook

Even though feasible results could be achieved, there are still open points which need to be tackled in the near and distant future. The topics mentioned in the near future section are meant to be resolved until the competition in summer 2019 since they are critical for the success of the project. The distant future refers to the future generations of ARIS.

Near Future

Firstly, changes concerning the used sensors must be adapted to the algorithm. It is now known that most likely, contrary to previous assumptions, two accelerometers are used by ARIS. Since the acceleration is used as a system input, including a second accelerometer reading is not that simple. The second accelerometer, however, could be used as a backup to guarantee the sampling rate of $T_s = 0.001s$ as it was stated in section 4.4.5. Furthermore, if the sampling rate of the accelerometer changes, it must be considered in the algorithm by setting T_s correspondingly.

Secondly, as soon as low altitude test data is available, the covariance matrices Q and R must be newly defined or at least checked with the provided tools.

Thirdly, the firmware needs to be embedded into the firmware framework used by ARIS. This includes mainly the interfaces between the tasks. Since the implementation was done with the eclipse based Kinetis Design Studio, small changes to some function calls must be made. Besides these two points, it should be possible to adapt as much as possible from the provided C implementation.

Lastly, the concepts of flight phase detection, faulty sensor detection as well as sensor failure detection must be improved and implemented. Especially the case of a failure of one or even both accelerometers must be studied. With the current implementation, an acceleration reading in every T_s interval is vital. This dependency could be resolved at least similarly to the described approach in section 4.4.5.

Distant Future

To further decrease the computational effort, steady state Kalman filtering could be applied as described in [6, p. 193]. When using steady state Kalman filtering, the fact that the Kalman gain K converges to a steady value is used. The calculation of the Kalman gain as well as the calculation of the error covariance matrix P are hence omitted. One drawback, however, is that steady state Kalman filtering is not optimal.

Since the noise on the measurements is just approximately white, a better approach to this colored noise could be made in the future. Simon [6, p. 312] for example mentions robust Kalman filtering when dealing with non-gaussian noise and uncertainties at the identification of the noise covariance matrices R and Q. Marchthaler and Dingler [5] on the other hand recommend using a so-called ROSE Filter (Rapid Ongoing Stochastic Estimator) to adaptively estimate the stochastic processes. However, the focus should first be set on getting simple algorithms running before considering more advanced algorithms.

Another point which could be considered in the future is the attitude of the rocket. Not only would this be beneficial for the accuracy of the state estimation, since the transformation from the body fixed frame to the navigation frame could be done correctly, but also would it be a chance to enhance the control algorithm by considering the attitude of the rocket.

References

[1] S. A. Cup, "Spaceport America Cup - Intercollegiate Rocket Engineering Competition - Rules & Requirements Document," 2018. 1, 9

[2] M. Kurmann, *ARIS - Data Fusion for a Sounding Rocket*. Bachelor thesis, Lucerne University of Applied Sciences and Arts, 2018. 1, 8, 11, 12, 15, 33, 35, 37

[3] S. Mohan M, N. Naik, R. M. O. Gemson, and M. R. Ananthasayanam, "Introduction to the Kalman Filter and Tuning its Statistics for Near Optimal Estimates and Cramer Rao Bound," tech. rep., Department of Electrical Engineering Indian Institute of Technoligy Kanpur, Kanpur, 2015. 5, 8, 12, 33

[4] F. Haugen, "State estimation with Kalman Filter," in *Kompendium for Kyb. 2 ved Høgskolen i Oslo*, ch. 8, pp. 101–127, 2001. 5, 7, 8, 12, 13, 14

[5] R. Marchthaler and S. Dingler, *Kalman-Filter: Einführung in die Zustandsschätzung und ihre Anwendung für eingebettete Systeme*. Wiesbaden: Springer Vieweg, 2017. 5, 6, 8, 12, 33, 47

[6] D. Simon, *Optimal state estimation : Kalman, H∞, and nonlinear approaches*. New Jersey: John Wiley & Sons, Inc., 2006. 6, 8, 22, 33, 47

[7] A. M. Kettner and M. Paolone, "Sequential Discrete Kalman Filter for Real-Time State Estimation in Power Distribution Systems: Theory and Implementation," *IEEE Transactions on Instrumentation and Measurement*, vol. 66, no. 9, pp. 2358–2370, 2017. 8, 22

[8] B. Tong Minh, *Real-time position and attitude determination of the Stratos II sounding rocket*. Master of science thesis, Delft University of Technology, 2012. 8, 20

[9] C. Eck, *Navigation Algorithm with Applications to Unmanned Helicopters*. PhD thesis, ETH Zürich, 2001. 8, 10, 12, 16

[10] M. B. Rhudy, R. A. Salguero, and K. Holappa, "A Kalman Filtering Tutorial for Undergraduate Students," *International Journal of Computer Science & Engineering Survey (IJCSES)*, vol. 8, no. 1, 2017. 8, 16

[11] J. Wendel, *Integrierte Navigationssysteme. Sensordatenfusion, GPS und Inertiale Navigation*. Berlin, Boston: Oldenbourg Wissenschaftsverlag, 2 ed., 2011. 10

[12] N. Hall, "Earth Atmospheric Model," 2015. [Online]. Available: `https://www.grc.nasa.gov/www/k-12/airplane/atmosmet.html` [Accessed Dec. 02, 2018]. 12

[13] U-blox, "NEO / LEA-M8T Datasheet," 2016. 15

[14] V. M. Fico, C. P. Arribas, A. R. Soaje, M. A. M. Prats, S. R. Utrera, A. L. R. Vazquez, and L. M. P. Casquet, "Implementing the Unscented Kalman Filter on an embedded system: A lesson learnt," *Proceedings of the IEEE International Conference on Industrial Technology*, vol. 2015-June, no. June, pp. 2010–2014, 2015. 21

[15] STMicroelectronics, "Application note Floating point unit demonstration on STM32 microcontrollers," 2016. 21

[16] B. Chou, "The Joy of Generating C Code from MATLAB," 2016. [Online]. Available: `https://ch.mathworks.com/de/company/newsletters/articles/the-joy-of-generating-c-code-from-matlab.html` [Accessed Dec. 04, 2018]. 24

[17] E. Styger, "Cycle Counting on ARM Cortex-M with DWT," 2017. Available: `https://mcuoneclipse.com/2017/01/30/cycle-counting-on-arm-cortex-m-with-dwt/` [Accessed Dec. 06, 2018]. 37

[18] E. Styger, "NXP FTF Hands-On with FreeRTOS Task Aware Debugger," 2016. Available: https://mcuoneclipse.com/2016/05/18/nxp-ftf-hands-on-with-freertos-t [Accessed Dec. 08, 2018]. 37

Appendices

The entire appendix which includes the Matlab source code, the firmware implementation and the definition of the project, can be found on the attached CD.

A Digital Appendix

1 Matlab	All important Matlab files
2 C	All important C files
3 Kinetis Project.zip	Project, exported from Kinetis Design Studio
4 PAIND Aron Schmied.pdf	Entire report
5 Assignment.pdf	Contains assignment of task
6 Mid Term Presentation.pdf	Mid term presentation slides
7 Final Presentation.pdf	Final presentation slides
8 Poster.pdf	

B Assignment

Lucerne University of
Applied Sciences and Arts

HOCHSCHULE
LUZERN

Technik & Architektur

Horw, 17. September 2018
Seite 1/2

Industrieprojekt im Fachbereich Elektrotechnik & Informationstechnologie

Aufgabe für Herrn Aron Schmied

ARIS: Data Fusion für Sounding Rocket

Fachliche Schwerpunkte
Signalverarbeitung & Kommunikation

Einleitung
Der Verein ARIS – Akademische Raumfahrt Initiative Schweiz – nimmt 2019 zum zweiten Mal am Spaceport Amercia Cup (www.spaceportamericacup.com) teil. Dazu wird eine Sounding Rocket konzipiert, gebaut und getestet, die mit einem eigen gebauten Hybrid Motor, Fallschirm System, Flugcomputer (Avionik) auf 10'000 ft (3km) fliegen soll. Um die Regelung der Höhe zu verbessern, wurde im FS 18 im Rahmen einer Bachelor Thesis Data Fusion Algorithmus als Matlab Modell entwickelt [1]. Dieser Algorithmus fusioniert zurzeit die Messdaten von 2 Barometer, 1 Accelerometer, 1 Gyrometer sowie eines GNSS Moduls.

Dieser soll nun von der Matlab Umgebung in C portiert werden und für das Funktionieren auf einem Embedded System angepasst werden.

Aufgabenstellung
Der vorhandene Algorithmus für die Data Fusion soll auf einem Embedded System implementiert werden. Insbesondere gilt es folgende Punkte zu bearbeiten:

- Einarbeiten in Themenbereiche Trajektorien-Simulation einer Rakete, Fusion von Sensordaten, Kalman Filter, etc.
- Die Identifikation der Parameter allfälliger neuer Sensoren mit Hilfe der Trajektorien-Simulation
- Optimierung des Fusion-Algorithmus
- Portierung und Test des Algorithmus auf eine Embedded Plattform (ARM Prozessor).
- Technischer Bericht.

Termine

Start der Arbeit:	Montag 17.9.2018
Zwischenpräsentation:	Zu vereinbaren im Zeitraum 29.10.-23.11.2018
Abgabe Schlussbericht:	Freitag 21. Dezember 2018, 16:00 im D311, an Oberassistent
Abgabe Poster-File:	Montag 28. Januar 2019 per Mail an Betreuer und Oberassistent
Abschlusspräsentation:	Zu vereinbaren im Zeitraum 17.12.2018 - 25.1.2019

Hochschule Luzern
Technik & Architektur

Horw, 17.9.2018
Seite 2/2
Industrieprojekt im Fachbereich
Elektrotechnik & Informationstechnologie

Dokumentation

Es ist ein gebundener Schlussbericht (nicht Ordner) mit CD in 3-facher Ausführung zu erstellen. Der Schlussbericht enthält zudem zwingend

- die folgende Selbstständigkeitserklärung auf der Rückseite des Titelblattes:
  „Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig angefertigt und keine anderen als die angegebenen Hilfsmittel verwendet habe. Sämtliche verwendeten Textausschnitte, Zitate oder Inhalte anderer Verfasser wurden ausdrücklich als solche gekennzeichnet.
  Horw, Datum, eigenhändige Unterschrift"
- einen englischen Abstract mit maximal 2000 Zeichen
- Ein Titelblatt mit: Name des Studierenden, Titel der Arbeit, Abgabedatum, Dozent, Experte, Abteilung, Klassifikation (Einsicht/Rücksprache/Sperre)
- Eine CD-Hülle, innen, auf der Rückseite des Berichtes

Alle Exemplare des Schlussberichtes müssen termingerecht abgeben werden. Es muss zu jedem Exemplar eine CD mit dem Bericht (inkl. Anhänge), dem Poster und den Präsentationen, Messdaten, Programmen, Auswertungen, usw. unmittelbar nach der Präsentation abgeben werden.

Ein Poster ist gemäss den offiziellen Layout-Vorgaben zu erstellen.

Fachliteratur/Web-Links/Hilfsmittel

[1] Kurmann, Michael (2018). *ARIS – Data Fusion for a sounding rocket*.
Unveröffentlichte Bachelor Thesis. Institut für Elektrotechnik der Hochschule Luzern, Technik & Architektur.

Geheimhaltungsstufe: Einsicht

Verantwortlicher Dozent/Betreuungsteam, Industriepartner

Dozent	Prof. Marcel Joss	marcel.joss@hslu.ch
Industriepartner	intern	

Experte / Zweiter Dozent
Prof. Thomas Hunziker
Thomas.Hunziker@hslu.ch

Hochschule Luzern
Technik & Architektur

Prof. Marcel Joss

Appendix on CD