Zurich University
of Applied Sciences

School of
Engineering

Ground Speed Measuring System

Authors: Yasmine Sheila Antille

Etienne Gubler

Supervisors: Juan-Mario Gruber

Simon Mathis

Date: 18.12.2020

Contents

Summary **6**

Abstract **7**

1 Introduction **9**
1.1 Formula Student . . . 9
1.2 Aim of this Project . . . 9
1.3 Intended Use of the Final System in the Race Car . . . 10

2 State of the Art **11**

3 Analysis **14**
3.1 General System Requirements . . . 14
3.1.1 Additional Requirements . . . 14
3.2 Compliance with Formula Student Rules . . . 14
3.2.1 Signals . . . 14
3.2.2 Sensor Placement . . . 15
3.2.3 Data Processing and Wireless Communication . . . 15
3.3 Risk Assessment . . . 15

4 Concept **19**
4.1 System Overview . . . 19
4.2 Microcontroller and Sensors . . . 19
4.2.1 Optical Ground Speed Sensor . . . 20
4.2.2 Inertial Measurement Unit . . . 20
4.2.3 Global Positioning System . . . 21
4.2.4 Microcontroller Unit . . . 21
4.3 Sensor Fusion Algorithm . . . 21
4.3.1 Vector Matching to Obtain the Reference Attitude . . . 22
4.3.2 Extended Kalman Filter for the Attitude . . . 23
4.3.3 Determining the Linear Acceleration . . . 23
4.3.4 Converting the Velocity Measurement . . . 23
4.3.5 Kalman Filter for the Velocity . . . 23
4.3.6 Mapping the Velocity into the 2D Plane . . . 24
4.3.7 Converting the Optical Ground Speed Measurement to X and Y coordinates 24
4.3.8 Comparing the Result to the Optical Ground Speed Measurement . . . 24
4.4 Kalman Filter Theory . . . 25
4.4.1 Kalman Filter . . . 25
4.4.2 Extended Kalman Filter . . . 26
4.5 Reference Frames Theory . . . 26
4.5.1 Body Frame . . . 26
4.5.2 Inertial Frame . . . 27
4.5.3 North References . . . 28

5 Implementation **30**
5.1 Adaptations to the Planned System . . . 30
5.2 Overview of the Implemented System . . . 31
5.2.1 Sensors . . . 31
5.2.2 Prototype Setup . . . 33
5.3 Sensor and Host Interfacing . . . 34
5.3.1 Interface to the IMU . . . 35

5.3.2 Interface to the GPS . . . 35
5.3.3 Interface to the Host Computer . . . 36
5.4 Software . . . 36
5.4.1 Software Structure . . . 36
5.4.2 Libraries . . . 37
5.4.3 Microcontroller configuration . . . 37
5.4.4 System Initialization . . . 38
5.4.5 Timing and Buffering . . . 44
5.4.6 Main Execution Flow . . . 49
5.4.7 Debugging . . . 56
5.4.8 MATLAB for Algorithm Design . . . 59
5.5 Data Processing . . . 59
5.5.1 Attitude Kalman Filter . . . 59
5.5.2 Velocity Kalman Filter . . . 64

6 Results **67**
6.1 Test Concept . . . 67
6.2 Test Results . . . 68
6.2.1 Test T1 . . . 68
6.2.2 Test T2 . . . 76
6.2.3 Test T3 . . . 84

7 Conclusion **87**
7.1 Suitability of the GSMS for Autonomous Driving . . . 87
7.2 Outlook . . . 87

Glossary **89**

Bibliography **90**

List of Figures **91**

List of Listings **92**

List of Tables **92**

A Appendix **93**
A.1 Project Management . . . 93
A.1.1 List of Requirements FSG . . . 93
A.1.2 Risk Assessment . . . 96
A.1.3 Timeline . . . 97
A.2 World Magnetic Model . . . 98

B Appendix **99**
B.1 MATLAB Attitude Kalman Filter Algorithm . . . 99
B.1.1 Initialization . . . 99
B.1.2 Prediction Step . . . 99
B.1.3 Measurement Step . . . 100
B.1.4 Propagate Attitude Error . . . 102
B.1.5 Reference Attitude Quaternion . . . 102
B.1.6 Attitude Kalman Filter Test . . . 102
B.2 MATLAB Velocity Kalman Filter Algorithm . . . 104
B.2.1 Initialization . . . 104

B.2.2 Prediction Step . . . 104
B.2.3 Measurement Step . . . 105
B.2.4 Velocity . . . 105
B.2.5 Velocity Kalman Filter Test . . . 105
B.3 Additional MATLAB Functions . . . 110
B.3.1 Quaternion Conjugate . . . 110
B.3.2 Quaternion Multiplication . . . 110
B.3.3 Angle of Rotation Between Two Quaternion . . . 111
B.3.4 Rotate 3 Element Vector by Quaternion . . . 111
B.3.5 Rotate 3 Element Vector Around z Axis by Angle . . . 111
B.3.6 Cross Product Matrix of 3 Element Vector . . . 111

Summary

This project goes in depth into the development of a reliable and accurate system that is capable of determining the ground speed of an autonomous racing vehicle. The race car is set to start at Formula Student competitions around Europe for the first time in the summer of 2021. To be able to drive autonomously, the car depends on precise sensor data and accurate algorithmic calculations. A lot of various kinds of information are required for autonomous driving. One of them is the speed of movement of the vehicle over the ground. To determine the ground speed of an autonomously driving vehicle, multiple sensors can be fused together to form a fail-safe system. The aim of this work is to evaluate the different approaches, design and then build a ground speed sensor that measures the speed over the ground in 2 dimensions and can pass the data on to the control unit of the vehicle.

Initially, related work is researched, analysed and summarised to create an understanding on the current state of the art. Different approaches from other renowned Formula Student teams are discussed. Second, requirements and project risks concerning the to-be-developed system are collected and evaluated. Relevant rules that have to be regarded when developing such a ground speed measuring system were researched and aggregated from the Formula Student Germany rule set, as the race car must be built following the German guidelines for it to be registrable for the competitions.

Following this, a prototype is developed and introduced before the design and implementation are discussed. The algorithms and system architecture are presented, which are designed for robustness, reliability, and extensibility. The ground speed measuring system is designed to accurately measure and uses a set of measurements generated by different sensors. The readings from those sensors must be combined to determine an estimate of the ground speed that is as accurate as possible. The system uses two Kalman filters to compute the final ground speed based on the readings from its various sensors. The proposed solution combines state of the art techniques from different fields of sensor technology and will be incorporated into the high-performance driverless vehicle after completion of this project. Finally, we discuss the findings and learnings of developing this system and present an evaluation of the module. In the end, the system is able to accurately estimate a test vehicle's ground speed during system field tests.

Abstract

Dieses Projekt befasst sich eingehend mit der Entwicklung eines zuverlässigen und genauen Messsystems, mit dem die Fahrgeschwindigkeit eines autonomen Rennfahrzeugs bestimmt werden kann. Der Rennwagen soll im Sommer 2021 erstmals bei Formula Student Wettbewerben in ganz Europa starten. Um autonom fahren zu können, ist das Auto auf präzise Sensordaten und genaue algorithmische Berechnungen angewiesen. Für das autonome Fahren sind viele verschiedene Arten von Informationen erforderlich. Eine davon ist die Bewegungsgeschwindigkeit des Fahrzeugs über dem Boden. Um die Fahrgeschwindigkeit eines autonom fahrenden Fahrzeugs zu bestimmen, können mehrere Sensoren zu einem ausfallsicheren System kombiniert werden. Ziel dieser Arbeit ist es die verschiedenen Ansätze zu evaluieren, einen Fahrgeschwindigkeitssensor zu entwerfen und anschließend zu bauen, welcher die Geschwindigkeit über dem Boden in zwei Dimensionen misst und die Daten an das Steuergerät des Fahrzeugs weitergeben kann.

Zunächst werden verwandte Arbeiten recherchiert, analysiert und zusammengefasst, um ein Verständnis für den aktuellen Stand der Technik zu schaffen. Verschiedene Ansätze anderer renommierter Formula Student Teams werden diskutiert. Danach werden Anforderungen und Projektrisiken bezüglich des zu entwickelnden Systems gesammelt und bewertet. Relevante Regeln, die bei der Entwicklung eines solchen Bodengeschwindigkeitsmesssystems berücksichtigt werden müssen, wurden aus dem Regelsatz der Formula Student Germany recherchiert und aggregiert, da der Rennwagen nach den deutschen Richtlinien gebaut werden muss, damit er für die Wettbewerbe zulässig ist.

Anschließend wird ein Prototyp entwickelt und vorgestellt, bevor das Design und die Implementierung besprochen werden. Die Algorithmen und die Systemarchitektur werden vorgestellt, die auf Robustheit, Zuverlässigkeit und Erweiterbarkeit ausgelegt sind. Das Fahrgeschwindigkeitsmesssystem dient zur genauen Messung und verwendet eine Reihe von Messungen, die von verschiedenen Sensoren generiert werden. Die Messwerte dieser Sensoren müssen kombiniert werden, um eine möglichst genaue Schätzung der Fahrgeschwindigkeit zu ermitteln. Das System verwendet zwei Kalman-Filter, um die endgültige Fahrgeschwindigkeit basierend auf den Messwerten seiner verschiedenen Sensoren zu berechnen. Die vorgeschlagene Lösung kombiniert modernste Techniken aus verschiedenen Bereichen der Sensortechnologie und wird nach Abschluss dieses Projekts in das fahrerlose Rennfahrzeug integriert. Zum Schluss diskutieren wir die Ergebnisse und Erkenntnisse der Entwicklung dieses Systems und präsentieren eine Bewertung des Moduls. Am Ende ist das System in der Lage, die Fahrgeschwindigkeit eines Testfahrzeugs während Systemfeldtests genau zu bestimmen.

List of Abbreviations

AMZ Academic Motorsports Club Zurich

AS Autonomous System

DMA Direct Memory Access

EKF Extended Kalman Filter

ETH Eidgenössische Technische Hochschule

FSG Formula Student Germany

FSZHAW Formula Student ZHAW

GNSS Global Navigation Satellite System

GPS Global Positioning System

GSMS Ground Speed Measuring System

GSS Ground Speed Sensor

HAL Hardware Abstraction Layer

IMU Inertial Measurement Unit

MEKF Multiplicative Extended Kalman Filter

SCS System Critical Signal

TUW Technische Universität Wien

WGS World Geodetic System

ZHAW Zürcher Hochschule für Angewandte Wissenschaften

1 Introduction

This work is carried out in cooperation with the Formula Student ZHAW team and in close contact with the driverless and electrical engineering teams. In this chapter we will give a brief introduction to Formula Student and present the aim of this project as well as go into more detail on the intended use of the final system.

1.1 Formula Student

Competitive sporting events such as traditional motorsports have provided a platform for developing and enhancing vehicle technologies for decades. For example, a modern-day Formula 1 car needs its high-tech sensors and the data they collect to run just as much as it needs high-tech fuel, which makes it the most connected car in the world. To be competitive, F1 teams process a massive amount of data. Hundreds of sensors log thousands of channels of data emitted by the car and power unit - from forces and displacements, temperatures and pressures to control parameters for the power unit and gearbox as well as driver inputs [1].

In 1998, Formula Student was brought to life and offers a similar platform. It is a worldwide design competition for student teams with the aim of building a race car. The different teams compete against each other during various events and in several countries, and compete in different disciplines. In addition to various time trials, technical aspects such as the design and functionality of the vehicle as well as economic efficiency and marketing are assessed. There are three different competition categories: combustion engine vehicles, electric vehicles and driverless vehicles. The driverless category is the most recently added one, having started in 2017.

In 2019, the Formula Student ZHAW (FSZHAW) team was formed by students from different engineering and economical backgrounds at the Zurich University of Applied Sciences (ZHAW). In the first year of the team's existence an electric formula racing car was designed and planned to be built. The completion of the car and participation in the first Formula Student competitions had been planned for July 2020, however, due to the extraordinary situation of the COVID-19 pandemic, these competitions had been cancelled. In 2020 the team created a new division with the goal of developing an autonomous system to convert the electric car to a driverless vehicle. With this upgraded vehicle the team plans to participate and compete in competitions in the summer of 2021, in the electric and driverless categories.

1.2 Aim of this Project

Driverless vehicles are autonomous cars in which human drivers are never required to take control to safely operate the vehicles. They combine sensors and software to control, navigate, and drive the vehicle without any human influence [2]. The concept for the FSZHAW driverless vehicle was designed by a master student at ZHAW in the summer of 2020 and serves as preliminary work. The developed model car will serve as a reference for the sensors needed in the electric car and close attention will be paid to the master student's conclusions. The model car already includes perception[1] and localisation sensors, however, so far the sensors only serve to create a map of the car's surroundings and to localize its position on that map as well as detect nearby objects such as traffic cones[2] by camera. It goes without saying that much more sensor data is needed for the autonomous vehicle to drive. This project will focus on one of those sensor outputs needed to enable a race car to drive autonomously.

The aim of this project is to reliably and to accurately determine the ground speed of an autonomously driving vehicle. This involves designing, developing and verifying a system that consists of a central microcontroller unit and multiple sensors which are combined to solve the

[1] **Perception** refers to the ability of an autonomous system to collect information and extract relevant knowledge from the environment [3].

[2] **Cones**, also referred to as pylons, are used to mark the race path in Formula Student competitions.

problem of measuring the vehicle ground speed resulting in a newly designed sensor fusion. Particular attention shall be given to the reliability of the system. This is regarded as highly important because the ground speed information will be used as an input to the control algorithms of the driverless FSZHAW race car. The system must therefore provide a certain level of redundancy and it must be able to detect faults in individual components. Another consideration regarding our design decisions for this system is the provided accuracy and precision of the output. The goal is to achieve the highest accuracy possible and to design the measuring system in a way that allows further accuracy improvements in the future.

1.3 Intended Use of the Final System in the Race Car

With a ground speed sensor and a suitable sensor fusion algorithm, the vehicle's odometry[3] data will be much more accurate than without such a system. Having an accurate ground speed measurement can therefore help readjust the vehicle's position whilst mapping[4] an environment. In addition, this will allow for longer periods between the corrections the mapping algorithm has to undertake. With a GSMS it will also be better recognizable if one or more wheels lose traction on the ground which could lead to an unfortunate situation known as understeering. Like humans, autonomous vehicles could have difficulties adapting quickly to loss of traction, therefore recognizing such a situation as soon as possible can make all the difference in reaction time.

[3] **Odometry** is the use of data from motion sensors to estimate change in position over time.
[4] **Mapping** takes place in the first lap of a race, where a map of the racetrack is created at a slow pace (3 m/s).

2 State of the Art

In order to better understand the current context in which this project will be set and in order to develop upon existing ideas and concepts it is important to aggregate and organise prior publications and work done by other researchers. In this chapter we go into detail about our findings and discuss some existing solutions to measuring ground speed in a Formula Student race car, where we will focus on publications of related work.

Since 2017, hundreds of Formula Student teams have been working on driverless vehicles. Outstanding results and various publications have been delivered by the Academic Motorsports Club Zurich (AMZ) from ETH Zurich.

In 2019, the AMZ-Racing driverless team published a comprehensive report on the concept of their first driverless racing car for the 2017/2018 racing season [4]. They finished 1st overall in every competition they attended with the car named *gotthard*. The published report covers the hardware (sensors, actuators and computer systems used) and software concepts, the regulation[5] concept and their test concept. The software-hardware architecture of their system can be seen in Figure 1. They divide their software stack in three main modules: *Perception, Motion Estimation and Mapping* and *Control*. Following the architecture design, the velocity estimation is used to compensate the motion distortion in the Lidar[6] pipeline, propagate the state in the SLAM[7] algorithm, as well as input for the control module. AMZ addresses in the report that the velocity estimation needs to combine data from various sensors with a vehicle model in order to be robust against sensor failure and to compensate for model mismatch and sensor inaccuracies. AMZ proposes to use a 9 state Extended Kalman Filter, which fuses data from six different sensors.

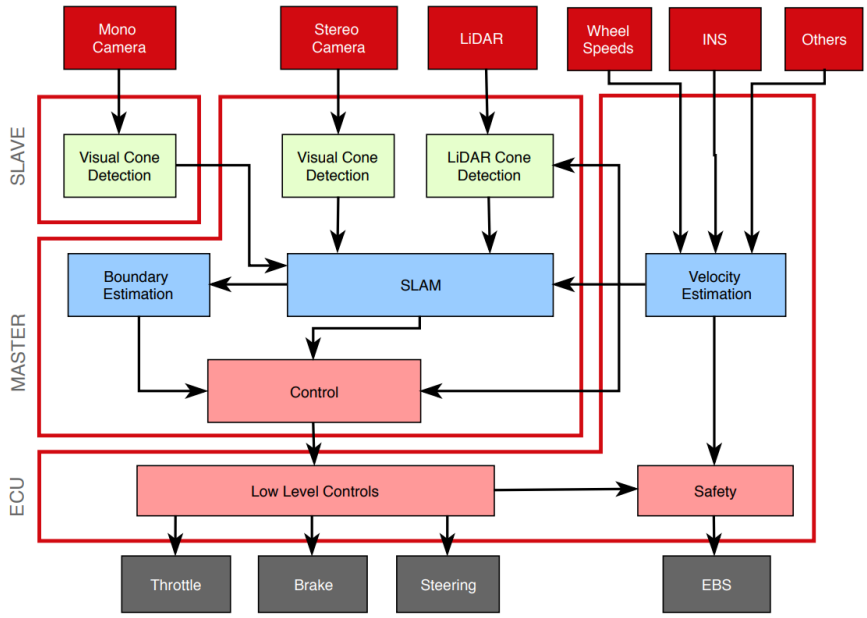


Figure 1: AMZ's software-hardware architecture of the autonomous system [4]

The Extended Kalman Filter (EKF) is the state-of-the-art estimator for fast, mildly non-linear systems. In *Design of an Autonomous Racecar: Perception, State Estimation and System Integration*

[5]Here, **regulation** does not refer to regulations understood as laws governing the testing of an autonomous vehicle, rather it refers to the controls to (among other things) steer the vehicle autonomously.

[6]**Lidar**, which stands for Light Detection and Ranging, is a remote sensing method that uses light in the form of a pulsed laser to measure ranges [5].

[7]**SLAM** stands for Simultaneous Localization And Mapping

AMZ presents the state estimation and system integration for an autonomous race car. They testify that sensor faults are a major factor undermining the robustness of state estimation systems and, therefore, a probabilistic outlier detection method should be used that works with any sensor. Their approach makes use of the innovation covariance calculated in the EKF which intrinsically accounts for the uncertainty of the state and the sensor noise model. Further they determine, that if wheel odometry is the only velocity source, and if the wheels are constantly blocked due to high accelerations, the velocity estimate deteriorates [6]. Generally, bayesian filters provide a statistical tool for dealing with measurement uncertainty, which are described in an easy to follow way in *Bayesian filtering techniques: Kalman and extended Kalman filter basics* [7]. This paper also explains that the probability density function includes all information needed to optimally solve estimation problems doing so recursively, which is why such filter approaches are well suited for velocity estimation.

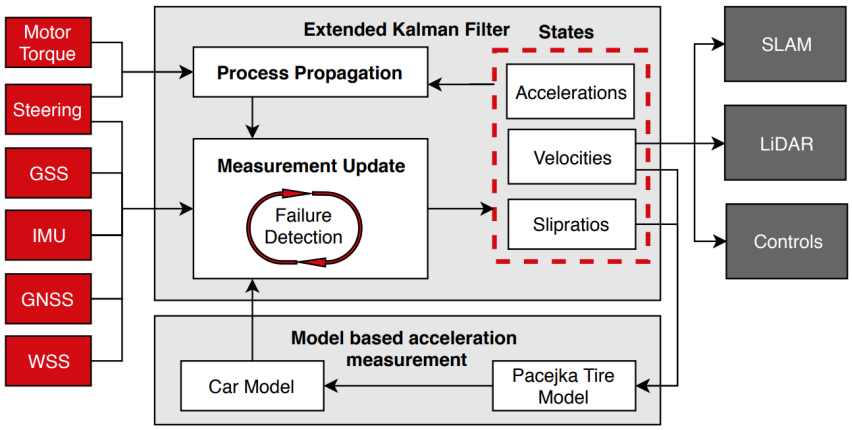


Figure 2: AMZ's velocity estimation architecture design [4]

An overview of AMZ's *gotthard* velocity estimator design can be seen in Figure 2. Highlighted should be that they made use of an Inertial Navigation System (INS), an optical Ground Speed Sensor (GSS), and 4 Wheel Speed Sensors (WSS) which allow for real-time state estimation. In [8] they introduce an interesting measurement model that keeps track of state variables like *Slip ratios* and *Velocity*. The slip ratios are updated using the WSS measurements. Their tests show that high changes in wheel speeds are captured as slip by the process model and the velocity estimate stays reliable. The linear velocities are updated using the GSS, Global Navigation Satellite System (GNSS) and the WSS. The angular velocity is observed using an Inertial Measurement Unit (IMU) in addition to the above three sensors. For *Sensor Failure Detection*, they classify sensor faults as outliers, drifts and nulls. They then analyse the redundancy of the velocity estimator by simulating sensor failures and comparing velocities to ground truth information based on the GSS data. In their paper, they demonstrated that accurate velocity estimation during high wheel slip and under single-sensor failure was correctly calculated by their failure detection module.

In 2020, AMZ presents the sensor setup for their autonomous race car *pilatus*, the successor to *gotthard*, which can be viewed in Figure 3. They equipped *pilatus* with an expensive external velocity sensor (namely the Kistler Ground Speed Sensor) to compare the output of their end-to-end recurrent neural network with. To summarise what can be seen in Figure 3, the sensor setup includes two IMUs measuring accelerations (x,y) and rotational rates (z), four motor encoders measuring the wheel speeds, one steering angle sensor, and two velocity sensors (an optical flow-based velocity sensor and a GNSS velocity sensor). The neural network takes raw sensor inputs from the IMU, wheel odometry and motor currents to calculate the velocity estimates. Their publication shows that their network manages to outperform the state-of-the-art Kalman filter with equivalent input data [9].

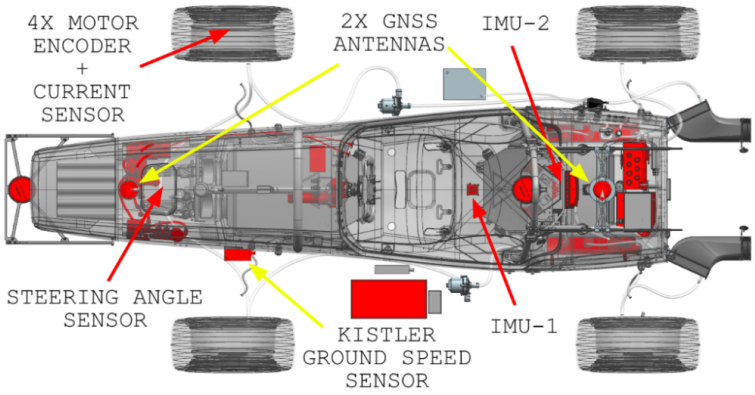


Figure 3: AMZ's autonomous race car *Pilatus*' sensor setup [9]

Of course, AMZ is not the only Formula Student team that has achieved great results with a self-developed measuring system that subsequently results in a ground speed estimation. To name a couple of other remarkable approaches, two teams solved this problem in the following:

- The Vienesse TUW Racing team uses a differential GPS, provided by a Piksi Multi GNSS module along with two beacons placed outside the racetrack. The beacons allow for more precise positioning than a generic GPS system does.[8] To measure the relative movement of the vehicle they included a motorsport-grade IMU [10].
- The Chinese BIT-FSD[9] team relied mainly on wheel speed sensors to calculate their first driverless vehicle's velocity in 2017. Even though their sensor setup also includes GPS, INS, Lidar and camera sensors, those are separately used to determine the vehicle's position and surroundings. Wheel speed sensors are widely used for odometry calculations in wheeled robots, where the team got this idea from [11].

Noteworthy is also the Doppler-based approach which a French research group from the Sorbonne University Pierre and Marie Curie in Paris elaborately discussed in their paper *Doppler-based Ground Speed Sensor Fusion and Slip Control for a Wheeled Rover* [12]. With a low-cost Doppler radar and an accelerometer the ground speed of a vehicle can be obtained as well. The focus of the paper lies on measuring the slip rate, for which an estimation of the true velocity of the vehicle with respect to the ground is necessary. In this paper the authors do not resort to wheel-based methods like optical encoders or resolvers. The chosen Doppler radar has a base frequency of 9.9 GHz (X-band). The Doppler effect principle is: a received electromagnetic wave's frequency is compared to a defined frequency, which changes as the receiver moves with respect to the transmitter. For their sensor fusion, a simple Kalman filter suffices to fuse the Doppler data and the accelerometer data which outputs an estimate of the longitudinal velocity. With these calculations it is possible to measure the slip rate for each wheel.

[8] Having read Formula Student Germany's rulebook (more on this in chapter 3.2) and knowing that even though (D)GPS may be used, there will be no space to securely build up base stations on the competition site and reliable wireless connection is not guaranteed by the officials, TUW's approach seems unstable and risky.

[9] **FSD** stands for Formula Student Driverless

3 Analysis

Based on the project idea, relevant system requirements were gathered and evaluated, which will be discussed in this chapter. We will have a look at the compliance necessary to be able to participate in the Formula Student competitions. Furthermore, a risk assessment was performed in order to detect dangers from an early stage and countermeasures were defined.

3.1 General System Requirements

The general system requirements posed by the CEO of Formula Student ZHAW are:

- Measurement certainty while cornering
- Measurement certainty on wet subsoil

3.1.1 Additional Requirements

The following list contains additional requirements for the developed system regarding its expandability:

- The system can be extended to allow further sensors to be added to the sensor fusion without having to reorganise the software structure.
- The system allows for a simulation of recorded sensor data using tools such as *MATLAB*.
- Additional redundancy checks can be programmed to allow for even more precise system outputs.
- The system can be fully integrated into the future autonomous system of the Formula Student ZHAW driverless car.

3.2 Compliance with Formula Student Rules

In order to comply with the Formula Student regulations, which are published by Formula Student Germany (FSG), the rules were read thoroughly and summarised. The set of rules from 2020, revision 1.0, is used as the basis for this work [13]. A list of requirements is created containing all of the rules that are in any way related to measuring the ground speed (see Appendix A.1.1).

Overall, the rules define the following points:

1. Requirements for the construction and strength of the vehicle.
2. Requirements for the electrical circuits.
3. Definition of mandatory elements (partly with a prescribed type).
4. Requirements for the layout of the software.
5. Definition of prohibited techniques.
6. Administrative requirements for entering competitions.

As can be derived from Appendix A.1.1, there are not a lot of regulations regarding the ground speed sensor. In the following, we limit ourselves to the most important requirements.

3.2.1 Signals

The rules determine that any signal of the Autonomous System (AS) is a System Critical Signal (SCS). Since this applies to the GSMS the SCS single failures must result in a safe state of all connected systems.

3.2.2 Sensor Placement

The sensors must be placed in the area shown in Figure 4. They may be mounted with a maximum distance of 500 mm above the ground and less than 700 mm forward of the front tires. Furthermore, the vehicle is subjected to a rain test at the races. For this, the wheels of the vehicle are dismantled and the vehicle is placed on the team's jack. During the test no driver is sitting in the vehicle for the test, meaning the water can also get into the vehicle through the seat. Then water is sprayed onto the vehicle from all sides for the duration of 120 seconds. During this time the internal insulation monitoring device must not detect any insulation faults. This test is carried out in the state in which the vehicle is ready for use. Therefore, all sensors must be adequately sealed and waterproof.

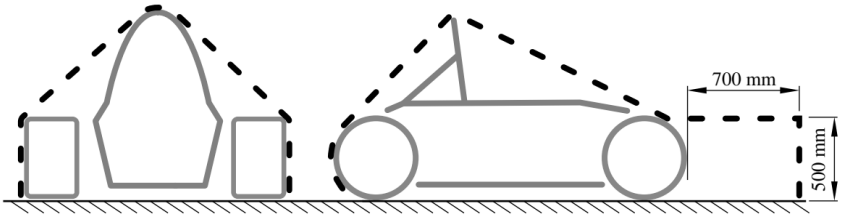


Figure 4: Envelope to mount sensor systems on a formula student race car [13]

3.2.3 Data Processing and Wireless Communication

The rules determine that all data processing must take place in the vehicle itself. Sending data to components outside the vehicle is only permitted in order to follow and record what is happening. Intervention in the process is also not permitted. During dynamic events, wireless communication may be limited and prone to interference. However, reliable wireless connection is not guaranteed by the officials.

3.3 Risk Assessment

A risk analysis was carried out at the beginning of and updated throughout the project (see Appendix A.1.2). Attention was paid to dangers that might have a negative impact on the entire project and would be possible causes of failure. Table 1 contains points that could prove to be potential problem causes and must be addressed early in the project. We monitored the influence of these factors throughout the entire development process. This ensured that we could take appropriate measures in time.

Table 1: Project Risk Assessment and Evaluation

#	Risk	Probable Cause	Impact	Probability	Assessment
1	Delivery difficulties in getting Kistler Sensor	Uncertainty of sponsorship and therefore uncertainty if delivery is even possible for sensor. Time until delivery is uncertain could take until end of project	3	4	12

Continued on next page

Table 1 – continued from previous page

#	Risk	Probable Cause	Impact	Probability	Assessment
2	Hardware defective on delivery	Delivery faces unforeseeable circumstances where items get broken or interfered with in a negative way	1	1	1
3	Hardware defective due to inappropriate behaviour or mishandling	Could happen during prototype setup or in installing sensors into a new environment with untested sensor combinations	2	1	2
4	New Kistler Sensor brings risk of faulty hardware	Because the sensor has not yet been officially released and is brand new it might still have some faulty behaviour given specific unknown environments that we could come in contact with.	2	2	4
5	Computing power is not enough for planned sensor fusion algorithm	The microcontroller might not have enough computing power for the various matrices and vector computations that have to be done for the planned Kalman filter.	2	3	6
6	Kalman filter approach unsatisfactory	Difficulty in arranging custom algorithm too high: getting the measurement and process model exactly right for the kalman filter with unknown error or uncertainty will be very challenging	3	3	9

Continued on next page

Table 1 – continued from previous page

#	Risk	Probable Cause	Impact	Probability	Assessment
7	Delivery difficulties for american sensors due to COVID-19	Since countries are back in lockdown mode, it might be difficult to obtain the planned sensors from the american shipping site Digikey.com	1	2	2
8	Unforeseeable changes made to Formula Student ZHAW car	Other team members make changes on the vehicle without any communication	2	1	2
9	Update frequency of GPS not sufficient	GPS has a lower data output rate compared to other measuring sensors	2	3	6

In Table 1 we assessed each risk on their impact intensity on a scale of 1-3 and the probability of occurrence on a scale of 1-5. Then the final assessment was calculated by multiplying those two factors (Impact $\times$ Probability). To minimize the impact of medium and high probability risks, the consequences should a risk come into effect, were listed and appropriate measures were defined which are listed in Table 2.

Table 2: Risk consequences, mitigation strategies and contingency plans

Risk #	Consequence	Mitigation Strategy	Contingency Plan
1	Delay of project	Frequent communication with Kistler representative. Weekly checkups on updates	-
2	Loss of time, delay of project	Keep in mind that extra orders might need to be place	Reorder defective items
3	Loss of time and work, delay of project	-	-
4	Wrong output data	Thorough research is carried out in advance and professionals are asked when in doubt	-

Continued on next page

Table 2 – continued from previous page

Risk #	Consequence	Mitigation Strategy	Contingency Plan
5	No output or wrong output, Loss of time	Divide mathematical complex computations into smaller separate computations	Connect an external computer with enough computing power to the system to compute complex equations
6	Loss of time, delay of project	Communicate any questions and problems to a master student who has proper Kalman filter knowledge and experience	Simplify algorithms and use pre-implemented libraries instead
7	Loss of time, delay of project	Check that order items are in stock with immediate shipping possibility	Have backup online shops where items can be reordered, in case they are delayed during shipment or at customs
8	Placing and calibrating issue for our sensors	Work closely with the Electrics-Team to insure communication flow and knowledge about all major changes	-
9	The low update rate of the GPS output could lead to very inaccurate data	A GPS with a very high data output rate should be chosen for this project	-

4 Concept

In this chapter we describe our design thoughts and how we initially planned to construct the prototype of the Ground Speed Measuring System (GSMS) for the driverless Formula Student ZHAW race car. In this chapter we refer only to the *initial* architecture and *initially* planned sensor fusion. The design in the implementation chapter will be partially modified, due to issues with obtaining one of the sensors (we go into further detail about this in Chapter 5.1). At the end of this chapter, we also give theoretical insights on how the Kalman Filters work and define the reference frames, which is necessary for understanding the implementation in the upcoming chapter.

4.1 System Overview

The main sensor of the GSMS is a high precision optical ground speed sensor. This sensor produces an accurate reading of the ground speed under all relevant conditions. However, the measuring system will also contain an IMU and a GPS receiver in order to produce a second and independent measurement of the ground speed. This is done by combining the output of the IMU and the GPS in a multistep filtering process. The sensor fusion algorithm, along with all other processing of sensor data, is executed on the ARM Cortex-M4 based microcontroller of the measuring system.

The GSMS measures the ground speed effectively using two distinct methods that rely on different technologies and their own sensors. This allows the central microcontroller to detect if either of the measuring methods fail to provide correct results.

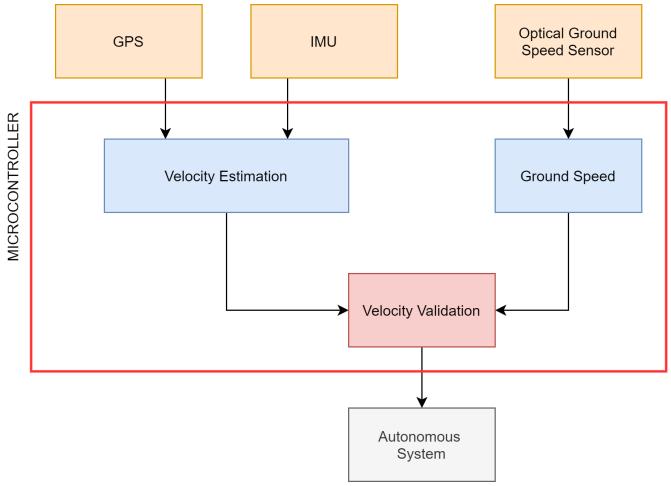


Figure 5: Initial hardware-software architecture of the GSMS

The originally planned hardware-software architecture can be viewed in Figure 5. The Sensors are depicted in orange, velocity estimation and ground speed calculation in blue, velocity validation in red and the autonomous system in grey. The computing platform is bound by the red box.

4.2 Microcontroller and Sensors

This chapter describes the above-mentioned components of the GSMS in more detail. The function of each component and their relation to the other parts of the measuring system is explained. The sensors intended for the implementation of the sensor fusion are listed in Table 3. The table lists the sensors, their output and respective frequency and the interface which can be used to communicate with them.

Table 3: List of sensors and their output data

Sensor	Output	Frequency	Interface
bno055 (IMU)	Acceleration, Angular Velocity, Magnetic Field Strength	100 Hz	I2C
neo-m9n (GPS)	Velocity (3D)	25 Hz	UART
Correvit SF-Motion (GSS)	Velocity (3D)	500 Hz	Ethernet

4.2.1 Optical Ground Speed Sensor

The optical Ground Speed Sensor (GSS) measures the following quantities:

- Velocity (2D): Magnitude and angle (500 Hz)
- Acceleration (2D): X, Y axes
- Angular velocity (3D): X, Y, Z axes

The *Correvit SF-Motion* sensor by Kistler is a high precision sensor specifically designed to measure the ground speed of race cars. The chosen sensor has ideal properties for our application and is therefore the designated main sensor. Under normal operating conditions, the final output of the GSMS will be directly derived from only this sensor. A second and independent measurement is obtained using additional sensors in order to detect errors and also to verify the output of this optical GSS.

The optical GSS also contains two accelerometers that measure acceleration in the X and Y axes and it also measures the angular velocity on the X, Y and Z axes using three gyroscopes. The values from those sensors are never used to produce the final ground speed output signal. They can, however, be used to verify the measurements made by the IMU.

4.2.2 Inertial Measurement Unit

The Inertial Measurement Unit (IMU) measures the following quantities:

- Acceleration (3D): X, Y and Z axes
- Angular velocity (3D): X, Y and Z axes
- Magnetic field strength (3D): X, Y and Z axes

The *bno055* sensor by Bosch-Sensortec is an IMU, containing 3 accelerometers, 3 gyroscopes and 3 magnetic field strength sensors. These sensors are used to measure acceleration and angular velocity on three axes and to measure the direction of the geomagnetic field.

The *bno055* module also features an integrated microcontroller which outputs calibrated sensor values from the accelerometer, gyroscope and magnetometer. The internal microcontroller also runs a proprietary sensor fusion firmware from Bosch. This firmware is capable of calculating and outputting an attitude signal (roll, pitch and yaw) based on the fused measurements of all three sensors.

The GSMS makes direct use of the calibrated accelerometer, gyroscope and magnetometer outputs. Those are processed using our own filtering and sensor fusion algorithm, described later in this chapter. This allows for more customization of the signal processing and therefore make it possible to continuously optimize the sensor fusion algorithm in the future. A custom sensor

fusion algorithm can also be tailored to our specific application and is therefore chosen over the proprietary algorithm from Bosch. One way to improve the custom sensor fusion algorithm would be to take the physical characteristics of the race car into account and to use additional measurements from motor encoders and steering angle sensors. Such customizations would not be possible using a pre-defined proprietary sensor fusion algorithm.

The bno055 is chosen because it allows us to use the built-in sensor fusion algorithm in case the implementation of our customized algorithm does not provide satisfactory results by the end of this project. In this case, the custom algorithm could be improved at a later time and the system would still be functioning and ready on time by relying on the internal sensor fusion algorithm until a better alternative is available. For testing, the attitude signal produced by the internal sensor fusion firmware is used to validate our custom sensor fusion algorithm.

Another important criterion for choosing the bno055 sensor is its ability to output calibrated accelerometer, gyroscope and magnetometer values. This feature can be used, after running a simple calibration procedure. Bosch also provides a software driver which abstracts from the sensor internals to allow for a simpler integration with the central microcontroller.

4.2.3 Global Positioning System

The Global Positioning System (GPS) measures the following quantities:

- Absolute position (3D): Latitude, Longitude, Altitude
- Velocity (3D): Magnitude and angle relative to (true) north, vertical velocity

The GPS receiver provides the GSMS with an absolute value of the speed in three axes. This signal is used in our custom sensor fusion algorithm together with the accelerometer data from the IMU. The different steps of the sensor fusion algorithm are described later in this chapter.

The GPS by u-blox is chosen due to its high accuracy and high data output rate. This module does not allow for the optimization of the position and velocity accuracy by means of a base station. A GPS with this capability might be added at a later time. The position signal of the GPS is currently not used as it is not relevant when determining the ground speed.

4.2.4 Microcontroller Unit

The STM32F429I discovery board by STMicroelectronics is used to quickly create a prototype for the GSMS. This board contains an ARM Cortex-M4 based microcontroller, which is used to perform all the signal processing. The microcontroller also communicates with all sensors and with a host computer or with the control unit of the self-driving car.

4.3 Sensor Fusion Algorithm

The measurements from the sensors mentioned in Chapter 4.2 are processed and combined in a custom sensor fusion algorithm which is described in this section. The custom sensor fusion algorithm consists of multiple stages which perform different filtering and fusing tasks. An overview of the different stages is given in Figure 6, and each stage will be described in more detail on the following pages.

The sensor fusion algorithm mainly processes the different measurements from the IMU and the GPS. The objective of this data processing is to obtain an accurate vehicle ground speed reading by combining the measurements given by the IMU and the GPS. In the very last step of the signal processing, the computed ground speed value will be compared to the measurement obtained through the optical ground speed sensor. This allows the GSMS to detect potential faults in the high precision optical GSS. Under normal operating conditions, the output of the system will be the measurement from the optical ground speed sensor as it delivers the most accurate value for the ground speed at the highest output rate.

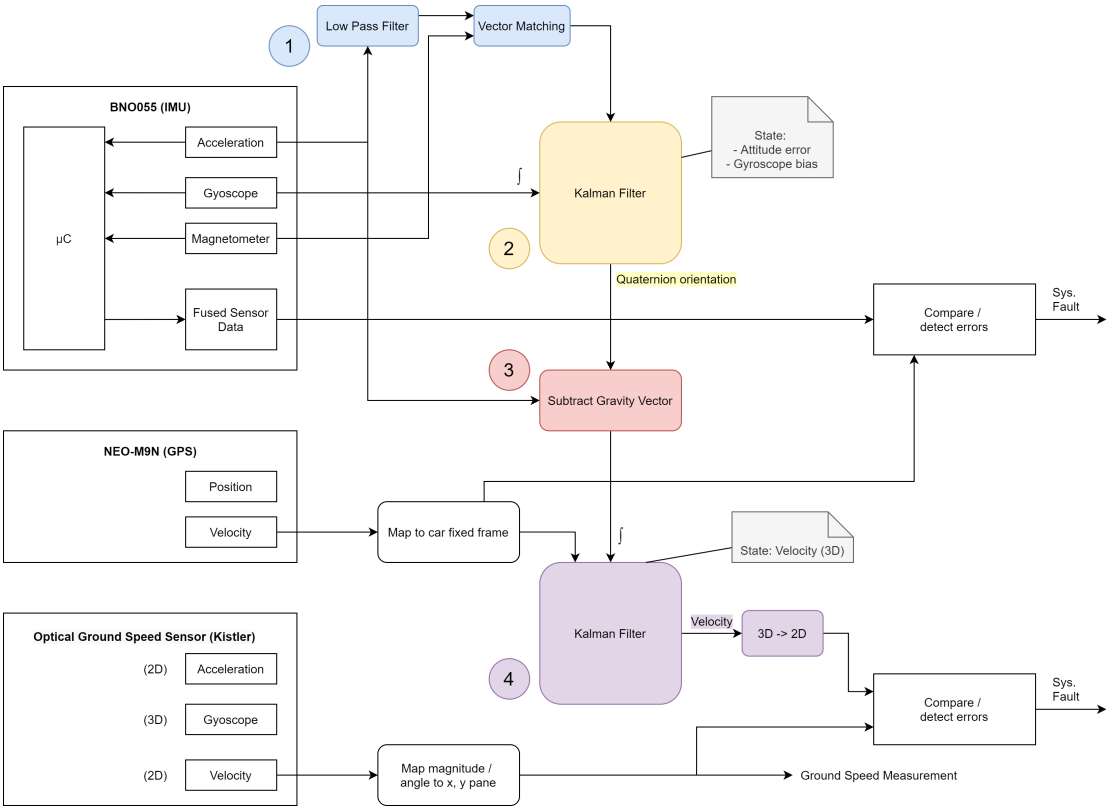


Figure 6: Sensor fusion concept

4.3.1 Vector Matching to Obtain the Reference Attitude

The first step of the sensor fusion algorithm is designed to obtain a reference value for the vehicle attitude. This reference value is computed from the accelerometer and magnetometer measurements of the IMU. To compute the reference attitude, a process called vector matching is applied. This finds an optimal rotation matrix for two or more given vectors if their location in a fixed reference frame is known. The measurements from the accelerometer and the magnetometer can be used to derive two different vectors. The first vector is a north vector obtained by the magnetometer readings on all three axes. The orientation of this vector in the fixed reference frame is known: the vector points northwards and up or down depending on the local inclination. The second vector is a gravity vector, which is given by the accelerometer readings on all three axes. The accelerometer values are first fed through a low pass filter to remove any acceleration caused by the movement of the car. Then, the accelerometer data can be combined into a gravity vector that points down towards the centre of the earth. Assuming that the north and gravity vectors are not colinear, the attitude of the vehicle is fully defined. It is a matter of solving Wahba's Problem[10] to find the optimal rotation matrix that minimizes the loss function proposed by Wahba. Wahba's Problem can also be solved by different algorithms. A common method is Davenport's q-method[11]. This approach involves finding the eigenvectors and corresponding eigenvalues of a small matrix, which could be computationally intensive and difficult to implement, but this method offers the benefit of directly computing the attitude in quaternion form[12]. Once the reference attitude is computed and transformed into quaternion form, it is used

[10] **Wahba's Problem** describes the task of finding a rotation matrix between two coordinate frames from a number of given vector observations.

[11] **Davenport's q-method** is describes a possible solution to Wahba's problem that makes direct use of rotation quaternions. It is therefore the preferred solution if the rotation must be represented in quaternion form.

[12] See Glossary.

as a measurement input to the Kalman filter in the next step.

4.3.2 Extended Kalman Filter for the Attitude

The signal processing step is designed to calculate a more accurate and stable value for the attitude. This is done by combining the previously calculated reference attitude with integrated angular rate measurements from the gyroscopes that are also part of the IMU. The two inputs are fused together by an Extended Kalman filter. The state vector of this EKF will consist of the attitude in quaternion form and of the gyroscope bias for each axis. The bias is part of the state vector and acts as a high pass filter for the angular rate measurements and to compensate for the slowly changing bias of the gyroscopes. The gyroscope readings are integrated in the prediction step of the EKF to produce an estimated attitude value at a high frequency. At a lower frequency the reference attitude will be combined with the predicted attitude in the measurement update step of the EKF. As part of the measurement update step, the bias of the gyroscopes will also be adjusted. This ensures that the attitude, which is calculated by integrating the angular rate measurements, does not drift and that the bias of the gyroscopes is continuously updated. This process results in a reliable and accurate attitude value that is updated at a high frequency. The obtained attitude value is an important part of multiple computations in the following steps of the fusion algorithm.

4.3.3 Determining the Linear Acceleration

This step deals with the accelerometer readings. Accelerometers measure vehicle acceleration, but their measurements are affected by the earth's gravity field. However, the strength and direction of the gravity vector is well known. Using the previously computed attitude, the gravity vector can therefore be subtracted from the accelerometer measurements. This results in a new vector representing the linear acceleration of the vehicle. This is the part of the accelerometer measurement that leads to an increase or decrease in the speed of the vehicle. This value will be used in a later step of the fusion algorithm to predict the vehicle speed.

4.3.4 Converting the Velocity Measurement

The GPS module is capable of measuring the vehicle speed in three dimensions. This speed is measured in an earth fixed frame consisting of the three axes North, East and Down. Those three axes of the earth fixed frame are however not aligned with the X, Y and Z axes of the vehicle fixed measurement frame. Therefore, the velocity measurement obtained by the GPS must be transformed into the vehicle fixed frame. This is done by taking the current vehicle attitude into account. The vehicle attitude was computed in a previous step of the signal processing and it links the earth fixed frame to the vehicle fixed frame. The resulting velocity can be described as values in the X, Y (and Z) axes of the vehicle. In the next step of the signal processing, the transformed velocity obtained through the GPS will be used as a measurement input to a second Kalman filter.

4.3.5 Kalman Filter for the Velocity

In this step of the sensor fusion, the velocity measurement of the GPS is combined with integrated acceleration values. The two measurements are fused together in a Kalman filter to obtain an optimal estimate for the vehicle speed. The state vector of this Kalman filter consists of the vehicle speed on all three axes. In the prediction step of the Kalman filter, the gravity free acceleration values are integrated to obtain an estimate for the vehicle speed at a high frequency. This estimate is then adjusted in the measurement update step of the Kalman filter by combining it with the transformed velocity reading from the GPS. This correction using the GPS measurement ensures, that the integrated acceleration values do not start to drift. The measurement update step is executed at a lower frequency than the integration of the acceleration values in the prediction step. The output here is an accurate velocity of the vehicle in the X, Y and Z axes

and which is updated at a high frequency. In the final step of the signal processing, this value will be compared to the measurement of the high precision optical GSS.

4.3.6 Mapping the Velocity into the 2D Plane

When trying to accurately measure the movement of a vehicle using an IMU and a GPS, it is important to consider all three dimensions. This is because a simpler two-dimensional measurement of the course over ground (speed in two dimensions on the X, Y plane) will not deliver a correct speed value in case the vehicle is driving on a slope. Therefore, all preceding operations of the sensor fusion algorithm are performed in three dimensions on measurements that were obtained on three individual axes.

The optical GSS only measures the vehicle speed in two dimensions. This makes sense, as the optical GSS is fixed to the vehicle and therefore only has to measure the speed on its X and Y axes. There will never be any movement in the direction of the vertical Z axis of the optical GSS. It always measures the correct speed, even if the vehicle is driving up- or downhill, because it is always tilted in the same way as the car.

Due to the fundamental differences in those two methods of measuring vehicle ground speed, the three-dimensional velocity obtained using the IMU and the GPS must be transformed into a two-dimensional measurement. Only then is it possible to compare it to the output from the GSS. This transformation is very simple, as we already transformed the velocity obtained through the GPS from the earth fixed frame into the vehicle fixed frame. Thus, the velocity output of the above mentioned Kalman filter should only show zero values on the vertical Z axis. It might not be exactly zero due to noise and other errors in the measurements, but the Z axis in the velocity measurement can simply be ignored to obtain the actual two-dimensional velocity of the vehicle. This value should then match the measurement obtained through the optical ground speed sensor.

4.3.7 Converting the Optical Ground Speed Measurement to X and Y coordinates

The optical ground speed sensor provides the velocity measurement in the form of a magnitude and an angle. Therefore, this data must first be transformed into X and Y components before it can be compared to the measurement from the IMU and the GPS.

4.3.8 Comparing the Result to the Optical Ground Speed Measurement

In the final step of the sensor fusion and signal processing procedure, the measurement given by the optical ground speed sensor is compared to the measurement obtained by using the IMU and the GPS. The high precision optical ground speed sensor will have a higher accuracy and a higher data output rate compared to the secondary velocity measurement. Therefore, the comparison serves to validate the output of the optical GSS. This allows the GSMS to continuously monitor and check the output of the optical GSS. During normal operation of all components, the final velocity output of the system will be solely based on the GSS. If the system does, however, detect a sensor fault, it is capable of outputting its secondary velocity measurement based on the IMU and the GPS. The control system of the self-driving car is notified about the sensor failure and it can then react accordingly.

The process that is described above ensures that the GSMS is capable of delivering a highly accurate velocity measurement at a high frequency while providing redundancy and the ability to detect sensor failure.

4.4 Kalman Filter Theory

A Kalman filter is an estimator. It is capable of estimating a variable of interest x in an optimal way. The estimate is based on a prediction for the variable x and on an absolute measurement of x. The Kalman filter algorithm can be divided into the following two steps:

- Prediction step (time update)
- Measurement step (measurement update)

The prediction of the variable x is made using a physical model called the process model. It describes how the variable x changes over time and usually also how it changes with respect to some secondary measurement. If the noise in the measurements and the errors introduced by the process model are purely gaussian, then the Kalman filter is the optimal estimator.

To make the optimal combination between the predicted and measured value for the variable x, the Kalman filter always keeps track of the uncertainties involved. This means that the Kalman filter tracks the uncertainty of the current estimate for x and the filter also needs to know the uncertainty in each measurement and in the process model.

4.4.1 Kalman Filter

A regular Kalman filter has a process model (and also a measurement model) that must be linear. The equations for the Kalman filter can therefore be written using matrix and vector operations. Figure 7 shows the key equations for the prediction step and for the measurement step.

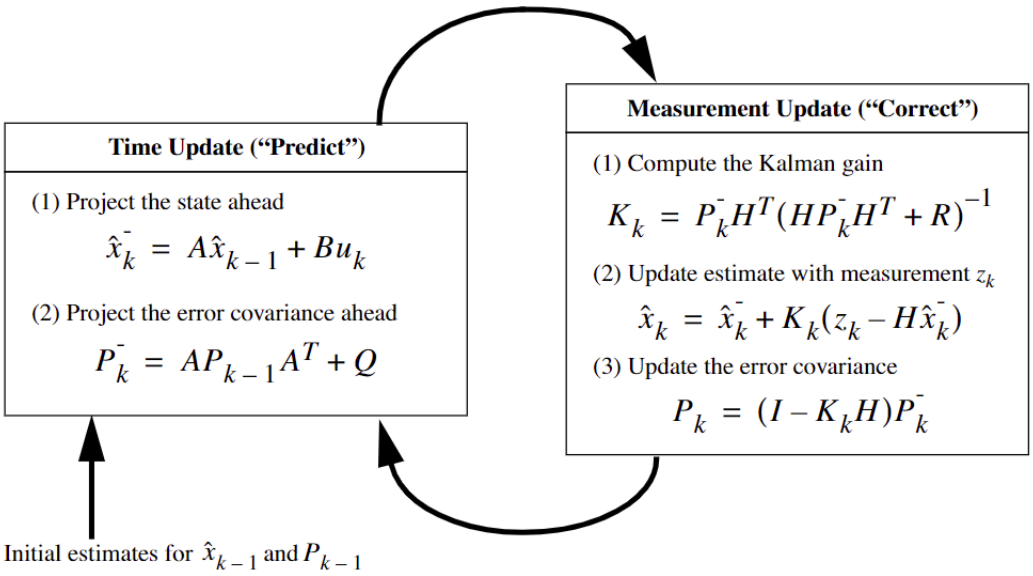


Figure 7: Kalman filter equations and execution flow [14]

The equations and the processes shown in the above diagram will now be explained by following the flow of execution for one iteration of the algorithm.

The Kalman filter starts using an initial estimate for the variable x. x is usually a vector and it is called the state vector of the Kalman filter. The second value that must be given is an initial estimate for the uncertainty of the state x. This uncertainty is expressed using the covariance matrix P.

Prediction step: The equation (1) of the prediction step updates the state vector using the process model. The process model is given by the matrix A. The matrix A must be designed such

that it reflects the physical behaviour of the state x. It usually involves the time t, that has passed since the previous iteration of the algorithm as well as additional indirect observations of the state x. The matrix B can be used to account for additional control inputs.

After updating the state using the process model A, the state covariance matrix P needs to be updated as well. This is done by the equation (2) of the prediction step. It takes into account the uncertainty in the process model, which is represented by the process noise matrix Q.

Measurement step The measurement step combines the prediction of the state that was made in the previous step with an absolute measurement of the state. This combination is done in an optimal way, depending on the uncertainties in the measurement and in the prediction. To achieve an optimal combination, the Kalman gain K is computed. This determines how much weight is given to the measurement and to the predication. Calculating the Kalman gain K therefore involves the covariance matrix P and the measurement uncertainty matrix R, as shown in the equation (1) of the measurement step.

Once the Kalman gain K has been calculated, it is used to combine the measurement z with the prediction of the state x in the next equation (2). This requires that the measurement z can be compared to the state x. Therefore the state x must be mapped onto the measurement z using the measurement model represented by the matrix H.

The term $(z - Hx)$ is called the innovation vector. It represents the error (or difference) between the measured and the predicted state. If the innovation vector is zero, then the prediction was perfect and the measurement step will not perform any correction of the state vector x. Usually the predicted state x does not perfectly equal the measurement. Then the innovation vector will account for that difference. The final value for the state vector x is then derived using this non-zero innovation vector and the Kalman gain K. After this step, the process will be repeated in the next iteration.

4.4.2 Extended Kalman Filter

To implement a Kalman filter all models and state transitions must be linear equations. This is a very strong restriction and it makes it impossible to use the Kalman filter in many situations. The Extended Kalman Filter provides a solution to this problem. It allows non-linear models and equations. The EKF works by linearizing the involved models for every iteration. The GSMS requires the use of an EKF for the attitude estimation, which will be explained in the following chapter.

4.5 Reference Frames Theory

The attitude and the velocity Kalman filter both work with three dimensional measurements made in different reference frames. Before a measurement can be processed it is often necessary to transform (respectively rotate) it from one reference frame to another. To fully understand how the Kalman filters of the GSMS work, we therefore first need to understand the different reference frames used.

4.5.1 Body Frame

The most important reference frame is the body frame. As its name suggests, it is fixed to the body of the race car. This means that the three axes of this reference frame rotate with the body of the car. The axes of the body frame are always labeled X, Y and Z. The following image shows how the body frame is oriented with respect to the race car. It is important to note, that the main axis, facing in the direction of travel, corresponds to the Y axis.

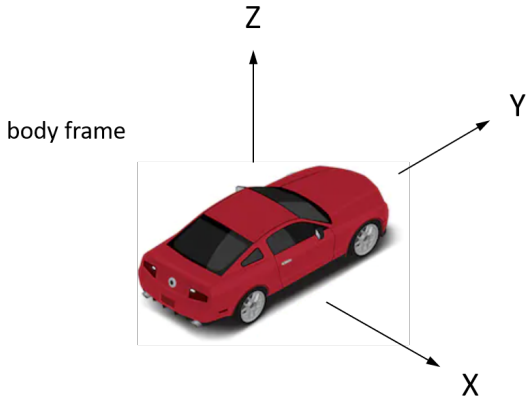


Figure 8: Orientation of the body frame

The orientation of the body frame and the naming of its three axes, as shown in Figure 8, depend on how the IMU is mounted inside the race car. This is because all measurements made by the IMU are made with respect to the body frame. Figure 9 shows how the three axes of the accelerometer, magnetometer and gyroscope measurements are oriented relative to the bno055 chip.

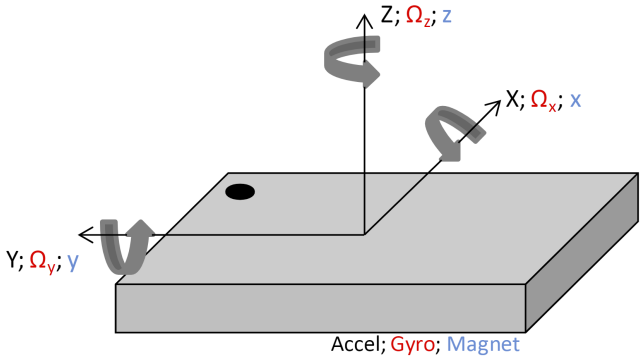


Figure 9: Orientation of the axes from the individual sensors inside the bno055

As mentioned at the beginning of this chapter, the body frame is the most important reference frame for the GSMS. All measurements are converted into this body frame before they are processed further. The system was designed this way because the final output of the system, the estimated ground speed, must also be given relative to the body frame.

4.5.2 Inertial Frame

The second reference frame that plays an important role in the data processing of the GSMS (see Chapter 5.5 is the inertial frame. This reference frame is fixed to the surface of the earth.

There are two different commonly used ways to express the coordinates of a vector in the inertial frame. It is possible to use NED coordinates or ENU coordinates. The data processing performed by the GSMS involves both types.

Both abbreviations (NED and ENU) define the direction and the order of the three axes of the inertial reference frame:

- NED is short for North, East and Down.

- ENU is short for East, North and Up.

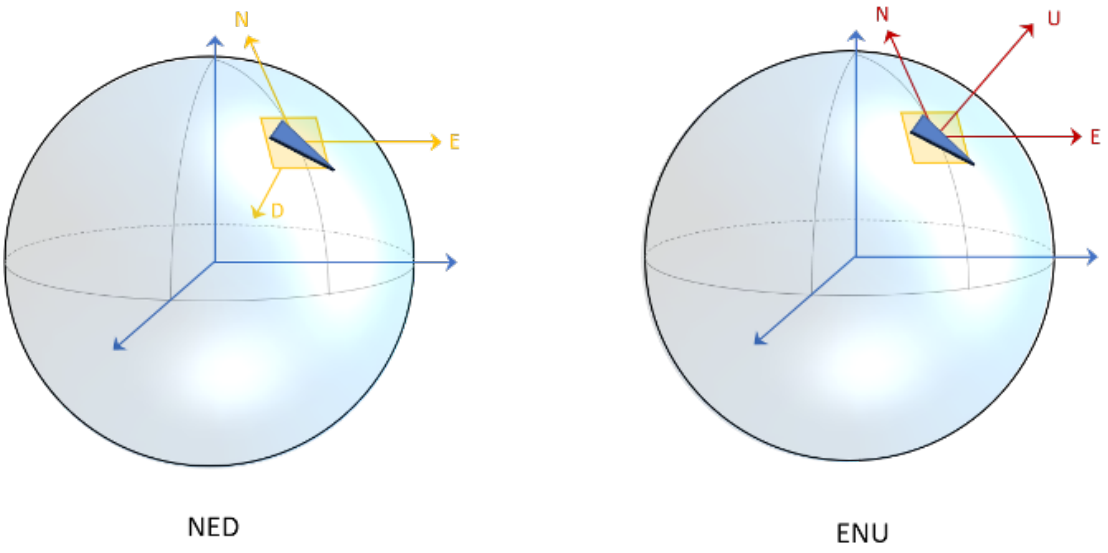


Figure 10: NED and ENU inertial reference frames [15]

The two types of inertial reference frames are illustrated in Figure 10. The inertial NED and ENU reference frames both have one axis that points northwards. It is important to further specify what north reference is used when discussing those inertial reference frames. This will be explained later. Both inertial reference frames also have an axis that point either up and away or down towards the center of the earth. The third axis is oriented perpendicular to the north and up (respectively down) axis. It therefore points eastwards. The NED and the ENU inertial reference frame both follow the right hand rule.

The inertial NED and ENU reference frames are so called local tangent plane reference frames. This means, that their north and east axes lie in a plane, which is tangential to the globe at a certain location. The up, respectively down, axis always goes through the center of the earth and through the point where the tangent plane is touching the globe. Converting three dimensional vectors from NED to ENU coordinates (and vice versa) is done by swapping the first two components and by inverting the third component.

4.5.3 North References

The calculations performed by the GSMS involve two different definitions of the north pole:

- True geodetic north
- Magnetic north

The true geodetic north is defined by the earth's axis of rotation. The true geodetic north corresponds to the location on the globe where the meridians of longitude converge (and the latitude is equal to 90°N). The magnetic north on the other hand is defined by the magnetic field of the earth. This magnetic field slightly changes over time and it is not perfectly aligned with the earth's axis of rotation.

The difference in the alignment of those two north references is called magnetic declination. As already mentioned earlier, the magnetic declination is location dependent (and also slowly changes over time). Figure 11 shows a visual representation of the magnetic declination. It pictures a compass that is aligned to the true geodetic north. This could be achieved by aligning the compass with the help of a map that uses a meridian of longitude as its vertical edge. Regardless

of how the compass is aligned, its needle always points towards the magnetic north pole. The angle between the zero mark on the rim of the compass and the compass needle therefore now corresponds to the local magnetic declination, as shown in the image.

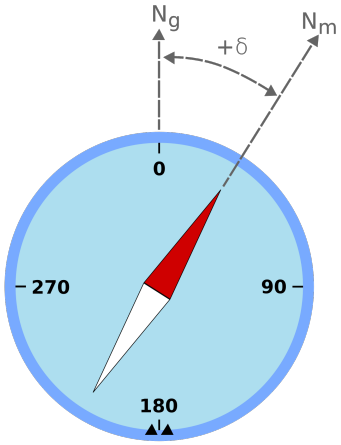


Figure 11: Visual representation of the magnetic declination [16]

The neo-m9n GPS uses the WGS 84 (World Geodetic System) standard to format all measurements. This means that the GPS expresses its velocity in a NED inertial reference frame that uses the true geodetic north as its north reference. The magnetometer in the IMU on the other hand determines where the magnetic north pole is (it can not determine the location of the true geodetic north without additional information). Therefore, when transforming the velocity measurement form the inertial reference frame into the body fixed reference frame, it is important to also account for the magnetic declination.

5 Implementation

In Chapter 4 we describe the main concept and workings of the GSMS as we initially designed it. In this chapter we go into further details about the algorithms and how we implement the different parts of the measuring system. In the chapters, building up to this point we introduced and discussed an optical ground speed sensor which, we were unfortunately not able to obtain. Consequently, we also discuss the adaptations to the initial designs and introduce the new architecture. Our actual implementation closely follows the approach described in the previous chapter, but there are a few important details where our implementation differs from the Concept design. We explain why a different approach was used wherever the actual implementation deviates from the initial design of the GSMS. The software section gives an overview of the different software components of the GSMS. The last section of this chapter explains in full detail the workings of our implemented algorithm and describe the implementation in full detail.

5.1 Adaptations to the Planned System

In section 4.1 we introduced our initial hardware-software architecture design. The adjusted architecture (due to the missing GSS) can be viewed in Figure 12. Furthermore, the algorithm design also had to undergo minor changes. The final version can be viewed in Figure 13.

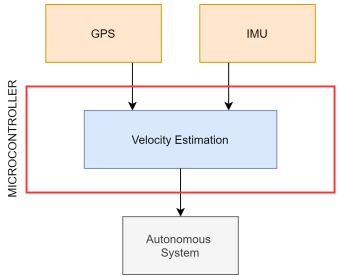


Figure 12: Actual hardware-software architecture of the GSMS

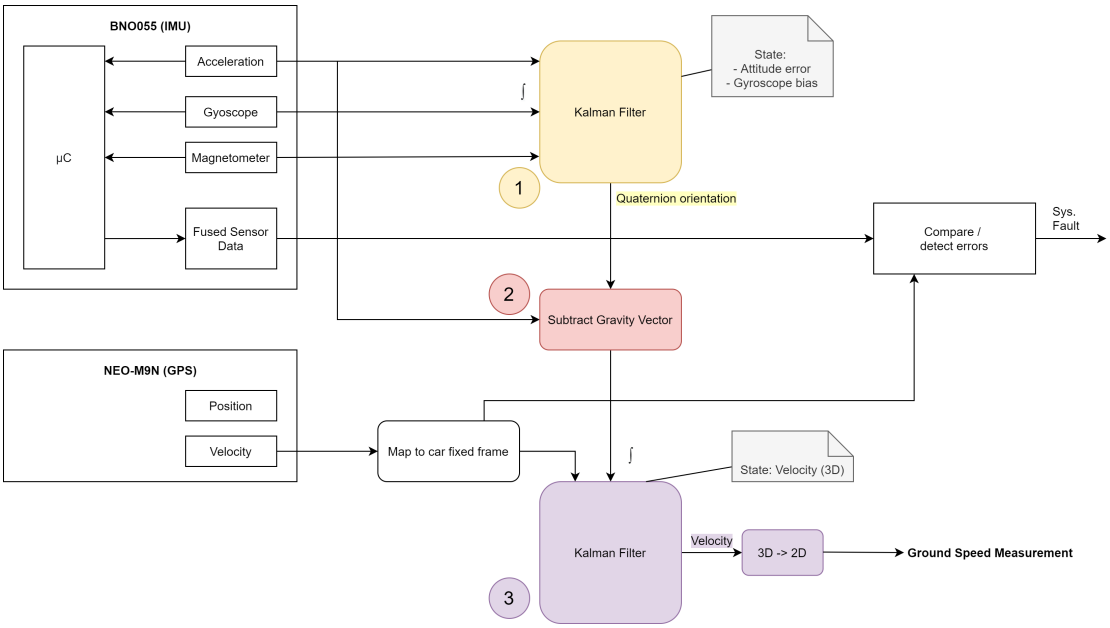


Figure 13: Sensor fusion implementation

5.2 Overview of the Implemented System

The ground speed measuring system is still designed to accurately and reliably measure the ground speed of an autonomous driving vehicle. The final system features an IMU and GPS sensor to measure the ground speed as well as a microcontroller to process the raw sensor readings.

5.2.1 Sensors

The implemented GSMS consists of two main sensors. This is one of the major deviations from the initial design. The original approach consisted of three sensors: IMU, GPS, and a high precision optical ground speed sensor, which the final measuring system lacks (as we were unable to obtain such a sensor within the given time frame). Hence, it is no longer possible to provide a *redundant* ground speed measurement using the implemented system. Nonetheless, the GSMS is still capable of providing an accurate measurement output which is based solely on the two other sensors.

IMU

As outlined in the initial design, the implemented GSMS contains the inertial measurement unit *bno055* from Bosch-Sensortec. This System on a Chip contains a three axis accelerometer, magnetometer and gyroscope as well as an integrated processor which runs the proprietary sensor fusion firmware BSX from Bosch-Sensortec. Figure 14 shows the bno055 chip placed on the prototyping shuttle board.

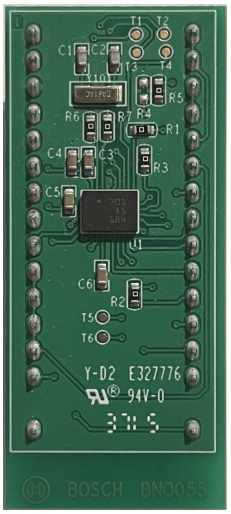


Figure 14: bno055 shuttle board (IMU)

Using the proprietary sensor fusion firmware, the bno055 is capable of directly outputting the device attitude. The device attitude calculated by the BSX sensor fusion firmware was used as a reference to test our custom sensor fusion algorithm (this is described in more detail in Chapters 6.1 and 6.2). An additional benefit of the sensor fusion firmware is that the bno055 can output calibrated values for all integrated sensors. This means that it is possible to read calibrated accelerometer, magnetometer and gyroscope values directly from the IMU without any additional processing. The GSMS makes use of this feature and therefore does not have to calibrate the IMU measurements.

The measurements made by the IMU are required for two different stages of the data processing. The GSMS requires the IMU to obtain:

- a measurement of the current system attitude
- and a measurement of the current linear acceleration[13] applied to the system.

Ultimately, the bno055 was chosen for the following reasons:

- The bno055 has a sufficient measurement frequency and it is intended for use in robotics applications.
- It is capable of directly outputting the system attitude which we can use as a reference for our custom sensor fusion algorithm.
- It can output calibrated measurements for the accelerometer, magnetometer and gyroscope which greatly simplifies our design.
- Bosch-Sensortec provides driver software for the bno055 written in C.

GPS

The GSMS uses the *neo-m9n* GPS module from u-blox. The neo-m9n is a standard precision GPS module with a measurement frequency of up to 25 Hz. Figure 15 shows the neo-m9n GPS module on the development board from MikroE.

Figure 15: neo-m9n develoment board (GPS) [17]

The GPS is used to periodically obtain an absolute measurement of the system velocity in all three dimensions. This signal is only available at a lower data rate than the intended system output data rate. Therefore, the GSMS uses data from the accelerometer to estimate the velocity between absolute measurements given by the GPS.

The neo-m9n was chosen for the following reasons:

- The neo-m9n has a high measurement frequency of up to 25 Hz.
- The proprietary protocol UBX from u-blox allows easy access to the three dimensional velocity measurements of the GPS module.

Microcontroller

[13] See section "Velocity processing" in Chapter 5.4.6 for an explanation of the term "linear acceleration".

The GSMS's central microcontroller is an *STM32F429ZI*. All data processing, as described in Chapter 5.5, is performed on this microcontroller. Both sensors (IMU and GPS) as well as the host computer are connected to the microcontroller.

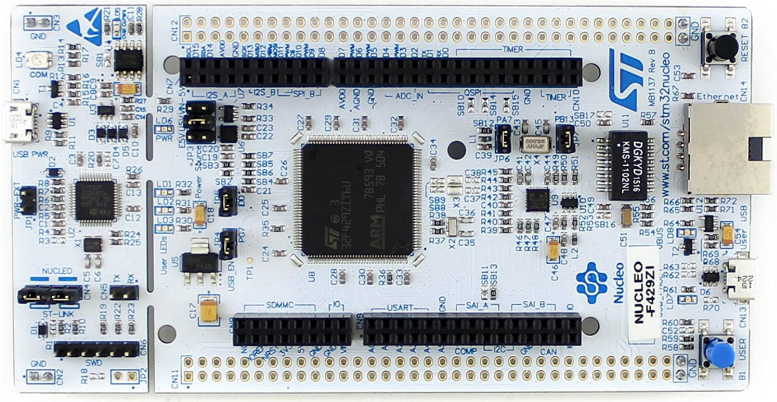


Figure 16: STM32F429ZI on the Nucleo-144 development board [18]

Figure 16 shows the STM32F429ZI placed on the Nucleo-144 development board. This microcontroller was chosen for the following reasons:

- High performance
- Development boards available

5.2.2 Prototype Setup

Initial Setup

We created a simple prototype setup (Figure 17 shows the initial GSMS prototype) for first testings of the system. This prototype was built with the following goals:

- Quick and easy setup
- Maximum adjustability
- Testing connections between system components
- Simple tests of the sensor fusion algorithms (this required that at least the IMU could be rotated around all three axes)

Final Setup

The initial prototyping setup was suitable for quickly testing the compatibility of different components but it was not adequate to make more accurate tests. We therefore created a second prototype of the GSMS (Figure 18 shows the final GSMS prototype). This final prototype was created with the following goals in mind:

- Allow for accurate dynamic tests in a car
- Minimize disturbance of the magnetometer of the IMU caused by electrical components
- Mechanically robust setup

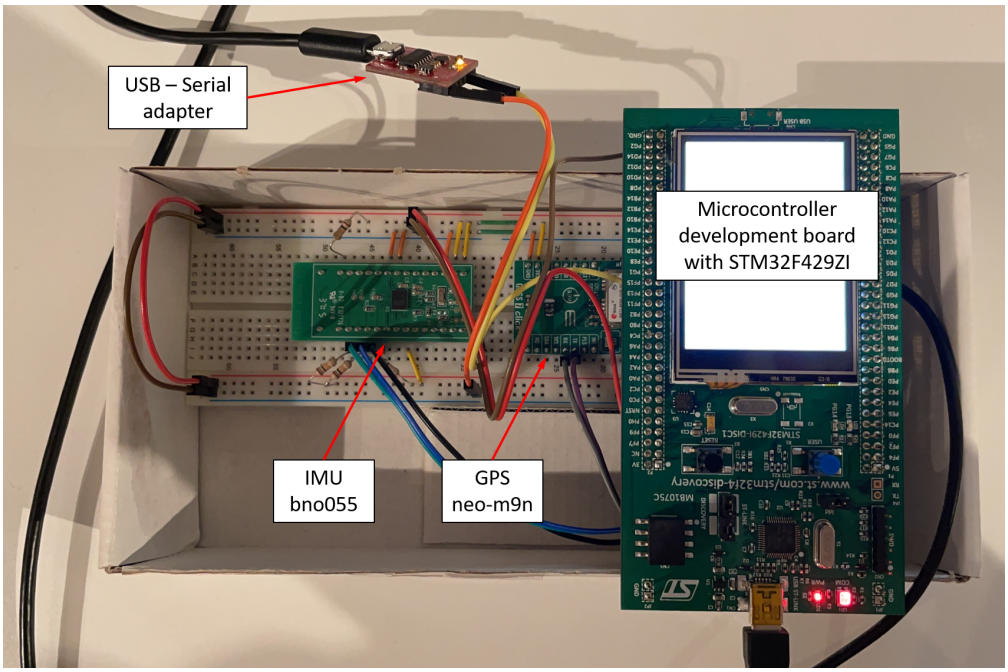


Figure 17: Initial prototype setup for the Ground Speed Measuring System

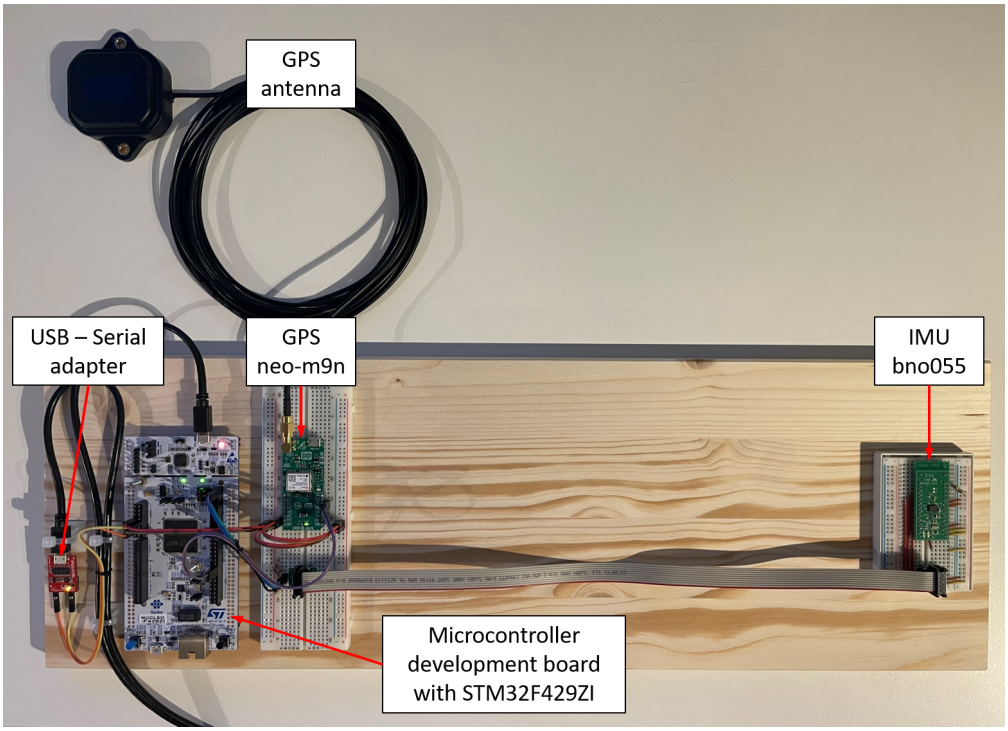


Figure 18: Final prototype of the Ground Speed Measuring System

5.3 Sensor and Host Interfacing

The GSMS is designed in such a way that all data processing is performed on the central microcontroller. The microcontroller can transmit the computed measurement of the ground speed to

a control unit of the autonomous vehicle [14]. For testing purposes, the GSMS has a simple debugging interface to an external computer (the host computer). This connection to the host is made using UART and a serial-to-USB adapter. In addition to the host connection, the microcontroller is also directly connected to the IMU and the GPS. Figure 19 shows the electrical connections between the different components of the GSMS.

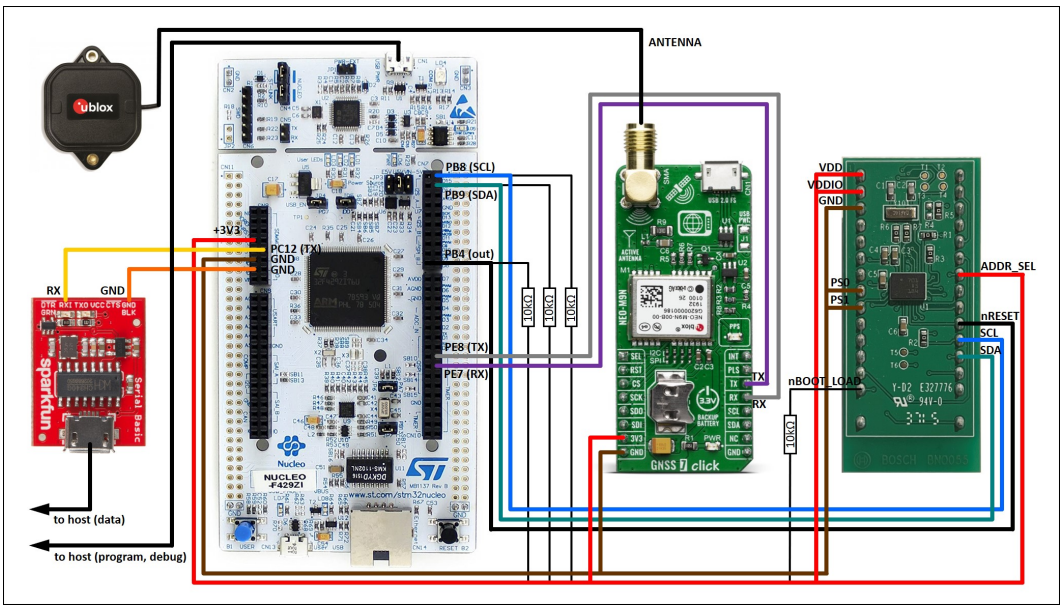


Figure 19: Schematic of the GSMS

The central microcontroller is connected to the sensors and to the host computer as follows:

- The BNO055 (IMU) sensor is connected using I2C at 400 kHz (fast mode).
- The NEO-M9N (GPS) sensor is connected using UART at 115200 baud.
- The host computer is also connected using UART at 115200 baud.

5.3.1 Interface to the IMU

The IMU provides a set of registers that can be read and written over I2C. The GSMS continuously reads the calibrated sensor values (accelerometer, magnetometer and gyroscope) from the IMU and processes them using the custom sensor fusion algorithm. In addition to the sensor values, the GSMS also reads the system attitude in quaternion format and the linear acceleration from the IMU. Those values are used as a reference and to compare them to the results of the custom sensor fusion algorithm.

5.3.2 Interface to the GPS

The GPS offers a large variety of different messages that can be enabled for output through UART. One possibility for reading the GPS data is using the standard NMEA[15] protocol. This does however make it difficult to access the three dimensional velocity data. As a solution, the GSMS configures the GPS to output a specific message that is part of the proprietary UBX protocol from u-blox. The message is called UBX-NAV-PVT and indicates that the message is part of the **NAV**igation class and that it contains a combination of **P**osition **V**elocity and **T**ime data.

[14] the interface to the control unit lies outside the scope of this project and is not yet defined, therefore the GSMS was designed so that the interface to the control unit can be defined and implemented at a later time

[15] **NMEA** is a commonly used protocol in GPS receivers developed by the National Marine Electronics Association.

The UBX-NAV-PVT message is continuously read by the microcontroller using UART and stored into a buffer. The message is processed by the GSMS once an entire message is received.

5.3.3 Interface to the Host Computer

The host computer is used to test, analyse and optimize the GSMS and the implemented sensor fusion algorithms. Thus, the GSMS offers the possibility to transmit raw data and intermediate signal values as well as the final algorithm output to the host computer. This transmission is done using UART.

5.4 Software

The entire software for the GSMS is developed using Keil μVision IDE. The programming language used throughout the whole project is C. Certain libraries were written in C++ and therefore required the use of some C++ code to be integrated into the project. The initial configuration of the microcontroller (STM32F429ZI) was created using the CubeMX configuration tool provided by STMicroelectronics. All algorithms were designed and implemented using MATLAB. Figure 20 gives an overview of the tools used and the most important components of the software for GSMS.

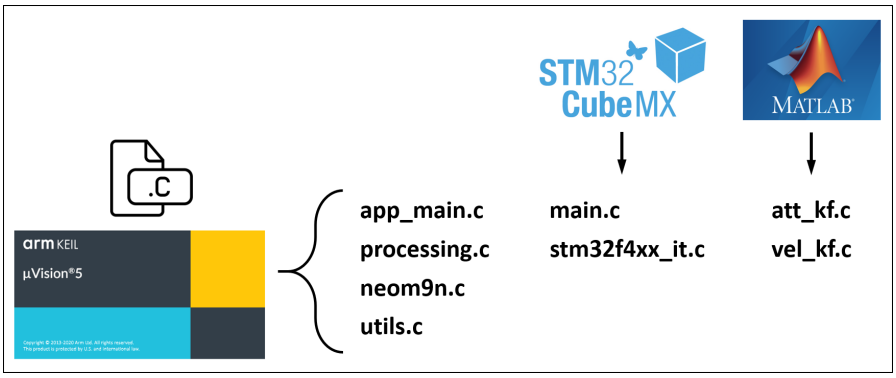


Figure 20: Tools used to develop the GSMS

5.4.1 Software Structure

The software of the GSMS is divided into the following main components:

- Program entry point and initialization of the microcontroller (main.c)
- Application initialization and interrupt handlers for timing and communication (app_main.c)
- Main data processing loop (processing.c)
- bno055 driver (bno055.c)
- neom9n initialization and configuration (neom9n.cpp)
- neom9n driver and UBX message processing (ublox.cpp)
- utility and debug functions (utils.c)

Figure 21 shows the software structure.

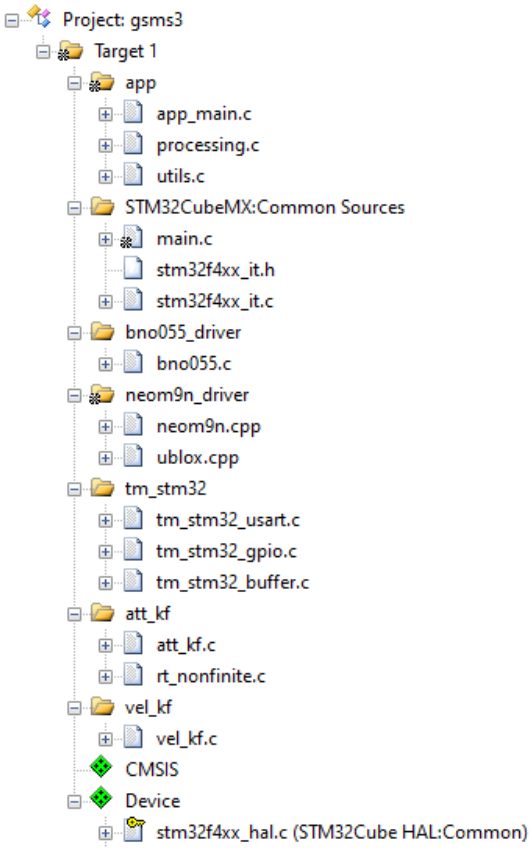


Figure 21: Components of the software for the GSMS

5.4.2 Libraries

We developed the entire software of the GSMS ourselves. This includes the algorithms used for our custom sensor fusion. Nonetheless, we made use of some additional libraries which are listed and described here:

- **bno055 driver** - Bosch-Sensortec offers a software driver written in C for their bno055 IMU. This driver abstracts from the device hardware and allows the user to configure the IMU without manually accessing the individual sensor registers. The driver can be configured by simply providing an I2C read and write function pointer. Using this driver was straight forward and did not cause any issues.
- **u-blox library** - The neo-m9n GPS module is rather complicated to configure and it does not come with a software driver written by the manufacturer. Instead, we used a generic driver for u-blox GPS receivers that was developed by sparkfun electronics. This driver needed considerable modifications to fit our requirements.
- **UART library** - To continuously read data arriving over UART from the neo-m9n GPS we used an additional library. This library makes use of the HAL functions for UART and provides a buffer to store data from a specific UART port.

5.4.3 Microcontroller configuration

The configuration created using CubeMX makes use of the HAL (Hardware Abstraction Layer) libraries from STM. This was especially useful for implementing the various connections to the sensors and to the host computer using I2C and UART.

As the name suggests, the HAL libraries are designed to make the code hardware independent (to some extent) and to provide a standardized interface to the different peripherals of the microcontroller. This allows a developer to focus on implementing the actual application rather than having to deal with setting the correct bits of different configuration registers.

5.4.4 System Initialization

After power up, the software runs through an initialization routine. Figure 22 shows a sequence diagram of the complete system initialization.

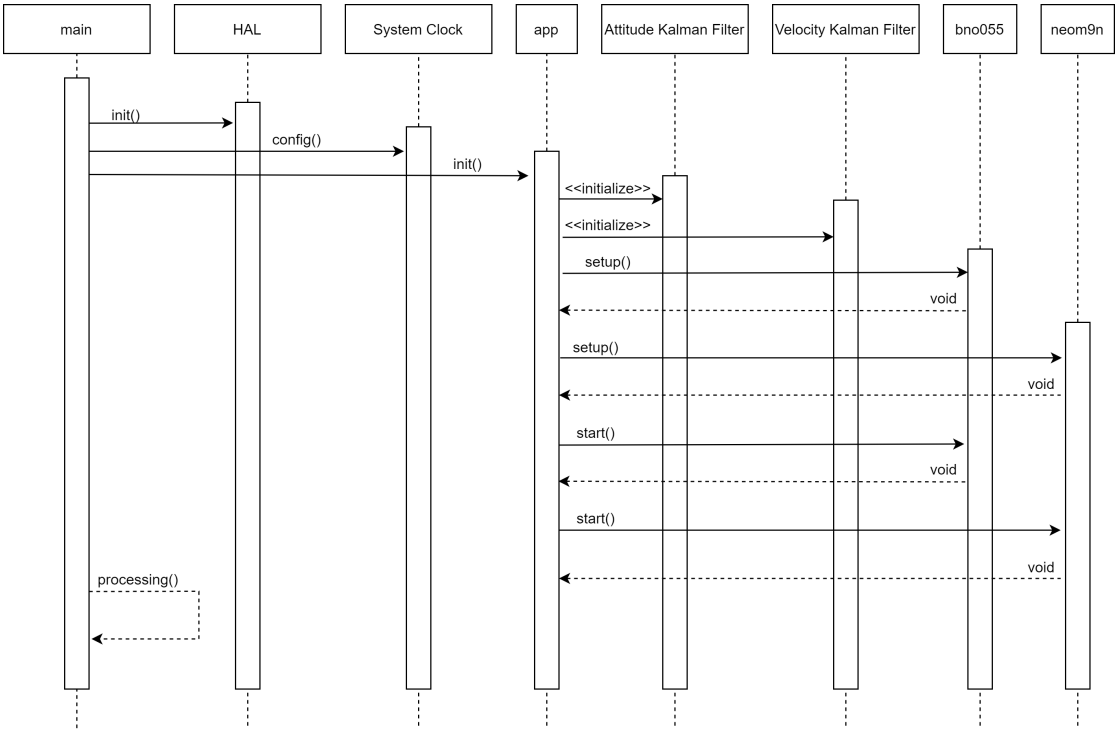


Figure 22: Sequence diagram of initialization routine

Peripherals and Clock

The first thing that is executed is the initialization of the HAL library using HAL_Init() proceeded by the setup of the system clock which configures the STM32F429ZI to run at 180 MHz. Once the clock is configured, all peripherals are initialized. This includes GPIO pins, DMA[16] controllers I2C and UART peripherals as well as different timers. This can be recognized in Listing 1.

Listings 1: The main function of the software system

```
1  /**
2    * @brief The application entry point
3    * @retval int
4    */
5  int main(void)
6  {
7      HAL_Init();
8      /* Configure the system clock */
9      SystemClock_Config();
10     /* Initialize all configured peripherals */
```

[16] **DMA**: Direct Memory Access

```
11      MX_GPIO_Init();
12      MX_DMA_Init();
13      MX_I2C1_Init();
14      MX_TIM3_Init();
15      MX_TIM4_Init();
16      MX_UART5_Init();
17      // initialiize LEDs
18      LED_init();
19      // initialize app
20      app_init();
21      /* Infinite loop */
22      while (1)
23      {
24          // perform data processing
25          processing();
26      }
27  }
```

The initialization code shown in the above listing was generated using the tool CubeMX from ST Microelectronics. Figure 23 shows a screenshot of the configuration interface displayed in CubeMX.

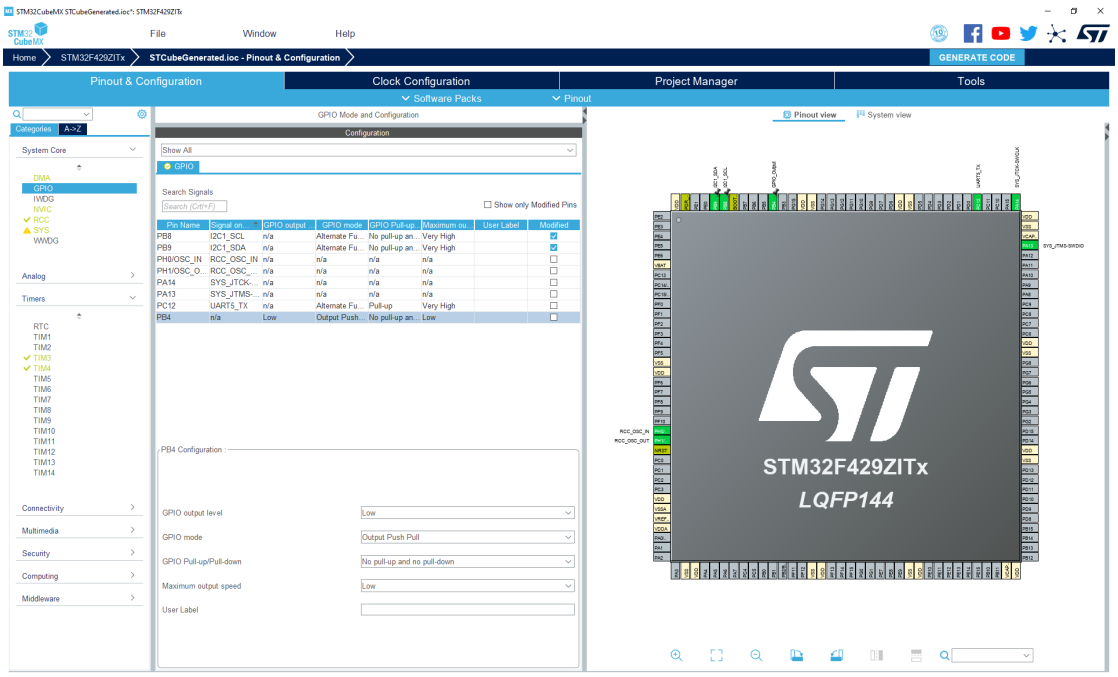


Figure 23: CubeMX pin configuration interface

A separate interface exists to configure the various clock settings of the microcontroller. Figure 24 shows this interface.

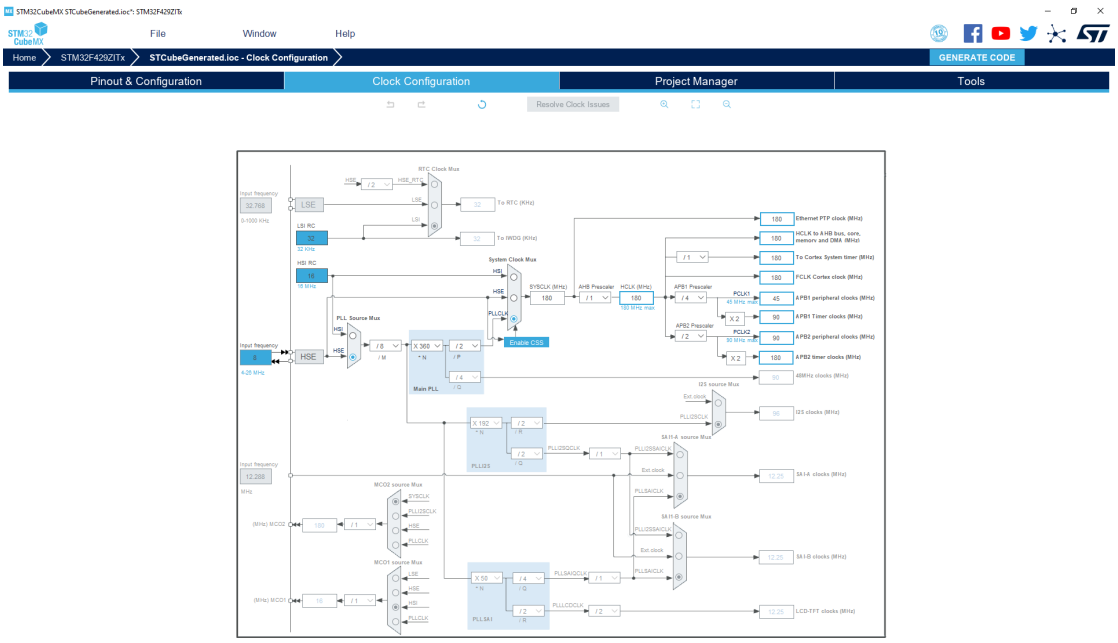


Figure 24: CubeMX clock configuration interface

CubeMX can be integrated into Keil μVision IDE[17]. This allows to easily switch between adjusting the microcontroller configuration and writing code in Keil μVision IDE. Figure 25 shows how CubeMX can be opened from within a Keil μVision IDE project once the project has been setup to use CubeMX.

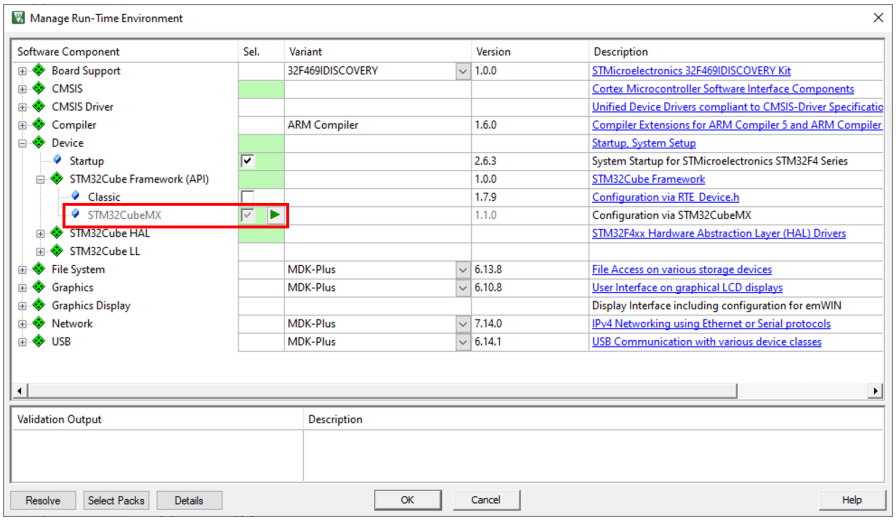


Figure 25: CubeMX integration with Keil μVision IDE

CubeMX makes use of the HAL (or alternatively the Low Layer LL) libraries from STMicroelectronics. The generated code calls the HAL functions, which makes it very readable and easy to understand.

[17] See reference [19].

App Initialization

Listing 1 above declared a call to app_init() on line 20, which we will look at in more detail now. All application specific initialization code is located in the app_main.c file. It is executed by calling the app_init() function inside that file. Listing 2 shows the app_init() function.

Listings 2: App initialization function

```
1  void app_init(void)
2  {
3      if (GSMS_DEBUG)
4      {
5          // initialize debug buffer
6          multi_buf_init(debug_buf, BUF_DEPTH_DOUBLE, DEBUG_BUF_SIZE);
7      }
8      // initialize kalman filters
9      att_kf();
10     vel_kf();
11     // setup bno055 and neo-m9n
12     bno055_setup();
13     neom9n_setup();
14     // start reading bno055 at 100Hz
15     bno055_start();
16     // start reading neo-m9n at 25Hz
17     neom9n_start();
18     // enable data processing
19     processing_enabled = 1u;
20 }
```

The essential hardware and software components of the GSMS are initialized in this function. First the following software components are initialized:

- Attitude Kalman filter to calculate the system attitude
- Velocity Kalman filter to calculate the system velocity

In a second step, the hardware components of the GSMS are initialized. This inculdes the bno055 IMU and the neo-m9n GPS module. The following paragraphs explain the necessary initialization that is performed to configure those two main components for their operation in the GSMS.

After configuring both sensors, the processing of the sensor data is started by calling bno055_start() and neom9n_start() and by setting the processing_enabled flag. The actual data processing is performed by repeatedly calling the processing() function located inside the processing.c file. The above picture of the main loop shows that the processing() function is indeed called inside an infinite while loop. The workings of this function are described in Chapter 5.4.6.

bno055 Setup: The bno055 IMU can be configured by setting individual bits in its configuration registers. The bno055 software driver abstracts from having to access the registers manually. The bno055_setup() function first creates the data buffer for the bno055, which will be explained in Chapter 5.4.5. Then, a handle for the bno055 chip is created and the device is resetted by pulling the nRESET pin low for 1ms. After the bno055 has booted up again, the following initialization steps are performed:

- Configuration of the device to use the external clock signal provided by the oscillator on the bno055 shuttle board.

- Selection of the NDOF[18] operation mode. In this mode, the firmware uses the sensor values from all nine axes (3 sensors with 3 axes each) to compute the system attitude.
- Setting the output unit for the gyroscope to rad/s.

Listing 3 shows how those steps were implemented.

Listings 3: Setup function for bno055

```
1  static void bno055_setup(void)
2  {
3      // initialize buffer
4      multi_buf_init(bno055_buf, BUF_DEPTH_DOUBLE, BNOO055_BUF_SIZE);
5      // initialize handle
6      bno055.bus_read = bno055_i2c_bus_read;
7      bno055.bus_write = bno055_i2c_bus_write;
8      bno055.delay_msec = HAL_Delay;
9      bno055.dev_addr = BNO055_I2C_ADDR2;
10     // reset device
11     HAL_GPIO_WritePin(GPIOB, GPIO_PIN_4, GPIO_PIN_RESET);
12     HAL_Delay(1u);
13     HAL_GPIO_WritePin(GPIOB, GPIO_PIN_4, GPIO_PIN_SET);
14     HAL_Delay(1000u);
15     // initialize driver
16     if (bno055_init(&bno055) != BNO055_SUCCESS)
17     {
18         Error_Handler();
19     }
20     // set clock source to external
21     if (bno055_set_clk_src(BNO055_BIT_ENABLE) != BNO055_SUCCESS)
22     {
23         Error_Handler();
24     }
25     // set operation mode to NDOF
26     if (bno055_set_operation_mode(BNO055_OPERATION_MODE_NDOF)
27         != BNO055_SUCCESS)
28     {
29         Error_Handler();
30     }
31     // set gyroscope unit to rad/s
32     uint8_t gyro_unit = BNO055_INIT_VALUE;
33     if (bno055_get_gyro_unit(&gyro_unit) != BNO055_SUCCESS)
34     {
35         Error_Handler();
36     }
37     if (gyro_unit != BNO055_GYRO_UNIT_RPS)
38     {
39         if (bno055_set_gyro_unit(BNO055_GYRO_UNIT_RPS)
40             != BNO055_SUCCESS)
41         {
42             Error_Handler();
```

[18] **NDOF** stands for Nine Degrees of Freedom which indicates that the integrated firmware is running in fusion mode.

```
43          }
44      }
45  }
```

neo-m9n Setup: The neo-m9n GPS module can be configured by sending specific UBX protocol messages. The configuration messages are all part of the UBX-CFG class. The driver for u-blox GPS receivers abstracts from having to compose the UBX-CFG messages manually.

To configure the GPS module a connection must first be established using the u-blox GPS driver. This is done by calling the connect() function of the neom9n.cpp file which makes use of the u-blox GPS driver. At powerup, the UART interface of the neo-m9n GPS module is configured to run at 38400 baud. To successfully connect to the GPS after it has been powered down, but also if it has previously been configured, the connect() function attempts to connect using the default as well as the configured baud rate. Once a connection is established, the baud rate is set to 115200 baud if necessary. Listing 4 shows the steps described in this paragraph.

Listings 4: Connect function for neo-m9n

```
1  static uint8_t connect(void)
2  {
3      uint32_t t0 = HAL_GetTick();
4      // timeout after 2s
5      while(HAL_GetTick() - t0 < 2000u)
6      {
7          // initialize UART7 (default baud rate)
8          TM_USART_Init(UART7, TM_USART_PinsPack_1,
9              NEOM9N_DEFAULT_BAUD_RATE);
10         // try to connect
11         if (neom0n.begin())
12         {
13             // successfully connected at default baud rate,
14             // set configured baud rate
15             neom9n.setSerialRate(NEOM9N_BAUD_RATE);
16             HAL_Delay(100);
17         }
18         // initialize UART7 (configured baud rate)
19         TM_USART_Init(UART/, TM_USART_PinsPack_1, NEOM9N_BAUD_RATE);
20         // try to connect
21         if (neom9n.begin())
22         {
23             // successfully connected at configured baud rate
24             return 0u;
25         }
26         // retry after 100ms
27         HAL_Delay(100);
28     }
29     // initialization failed due to timeout
30     return 1u;
31 }
```

Once a connection has been established, the configure() function of the neom9n.cpp file is called. This function enables the neo-m9n GPS module to periodically output the UBX-NAV-PVT message on its UART port. It also sets the output rate to 25 Hz. Listing 5 shows these steps.

Listings 5: Configure function for neo-m9n

```
static uint8_t configure(void)
{
    // activate UBX protocol on UART1 port
    if (!neom9n.setUART1Output(COM_TYPE_UBX))
        return 1u;
    // activate UBX-NAV-PCT message on UART1 port
    if (!neom9n.configureMessage(UBX_CLASS_NAV, UBX_NAV_PVT,
            COM_PORT_UART1, 1))
        return 1u;
    // configure output rate
    if (!neom9n.setNavigationFrequency(NEOM0N_OUTPUT_RATE))
        return 1u;
    // enable auto PVT without implicit update
    neom9n.assumeAutoPVT(true, false);
    // configuration successful
    return 0u;
}
```

5.4.5 Timing and Buffering

The GSMS runs two main sensor fusion algorithms that are both dependent on each other (the two algorithms being the Attitude Kalman Filter and the Velocity Kalman Filter). Both algorithms process sensor data at a specific frequency.

Attitude Kalman Filter Timing

The base frequency of the entire GSMS is given by the rate at which the bno055 IMU outputs its measurements. This output data rate of the IMU is exactly 100 Hz. So the attitude Kalman filter, which processes the output of the IMU, runs at 100 Hz. This means that the attitude Kalman filter processes a new sample of the bno055 IMU every 10ms to calculate an updated value for the system attitude. The 100 Hz frequency is provided by TIM3 of the microcontroller. It is setup to create repeated interrupts at the desired 100 Hz frequency. Listing 6 shows the interrupt handler for TIM3.

Listings 6: Period elapsed callback function for TIM3

```
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
    if (htim->Instance == TIM3)
    {
        // instance is TIM3
        // request bno055 samples at 100Hz (algorithm base timer)
        // check if i2c is ready
        if (HAL_I2C_GetState(&hi2c1) != HAL_I2C_STATE_READY)
        {
            gsms_error += 0x01;
            Error_Handler();
        }
        // check if next bno055 buffer is empty
        if (!bno055_buf[bno055_next_buf_index].empty)
        {
            gsms_error += 0x02;
```

```
17              Error_Handler();
18          }
19          // i2c read parameters
20          uint16_t dev_addr = bno055.dev_addr;
21          uint8_t reg_addr = BNO055_BUF_BASE_REG_ADDR;
22          uint8_t *reg_data = bno055_buf[bno055_next_buf_index].data;
23          uint16_t cnt = BNO055_BUF_SIZE;
24          // start DMA transfer
25          if (HAL_I2C_Mem_Read_DMA(&hi2c1, (uint16_t) (dev_addr << 1u),
26              reg_addr, 1u, reg_data, cnt) != HAL_OK)
27          {
28              Error_Handler();
29          }
30      }
31  }
```

Every time the HAL timer interrupt handler is called for the timer instance TIM3, a new sample is read from the bno055 IMU. This data transfer is only initiated in the interrupt handler and then performed independently by the DMA controller.

bno055 Sample Double Buffering: The data transfer performed by the DMA controller happens while the previously received sample is still being processed by the attitude Kalman filter algorithm. This requires the bno055 sample data to be stored using a double buffer. One buffer always contains a complete bno055 sample which can be processed by the algorithm while the other buffer is being filled with new data. Once the new sample is received completely, and when the processing of the previous sample has finished, the buffers are swapped. In this process, data is not actually copied. The only thing that changes are pointers to the respective buffers. This double buffering mechanism is facilitated by a simple data structure that is shown in Listing 7.

Listings 7: Buffer type definition

```
1  // multi buffer defines
2  #define BUF_DEPTH_DOUBLE 2
3
4  // multi buffer type
5  typedef struct {
6      uint8_t* data;
7      uint8_t empty;
8  } multi_buf_t;
```

As the name "multi buffer" suggests, the data structure could also be used to create more than just two buffers. However, the GSMS only requires the use of double buffering and therefore always initializes exactly two buffers.

In addition to the type definition, we implemented an initialization function to allocate the required memory and to properly set the additional fields of the struct. Listing 8 shows both utility functions for the buffer type.

Listings 8: Utility functions for the buffer type

```
1  void multi_buf_init(multi_buf_t* multi_buf, uint8_t depth,
2      uint8_t size)
3  {
4      // check depth and size
```

```
5       if (depth == 0 || size == 0)
6       {
7           Error_Handler();
8       }
9       // initialize buffers
10      for (int i = 0; i < depth; i++)
11      {
12          // allocate memory
13          multi_buf[i].data = (uint8_t*) malloc(size * sizeof(uint8_t));
14          // set empty flag
15          multi_buf[i].empty = 1u;
16      }
17  }
18
19  void index_inc_wrap(uint8_t* index, uint8_t size)
20  {
21      // increment the index and wrap to zero if size is reached
22      *index = (*index + 1u) % size;
23  }
```

As previously described, the bno055 sample buffer makes use of this type definition and the related utility functions as shown in Listing 9.

Listings 9: bno055 buffer defines

```
1  // bno055 buffer defines
2  #define BNO055_BUF_SIZE              (46u)
3  #define BNO055_BUF_BASE_REG_ADDR     (BNO055_ACCEL_DATA_X_LSB_ADDR)
4
5  // bno055 buffer
6  extern uint8_t bno055_curr_buf_index;
7  extern uint8_5 bno055_next_buf_index;
8  extern multi_buf_t bno055_buf[];
```

The bno055 buffer stores 46 bytes of data from the output registers of the device. Due to the design of the bno055, the time required for the transfer of those 46 bytes varies between 1ms and 7ms. This is caused by the internal microcontroller of the bno055 which slows down the I2C data transfer during certain intervals while it is busy running its internal sensor fusion algorithms. The I2C data transfer is slowed down by means of clock stretching[19] from the bno055 slave. This unpredictable and changing transfer duration is another reason for implementing the double buffering of the bno055 sample data. The double buffering ensures that every 10ms a new sample is ready for processing by the attitude Kalman filter independent of how long the transfer of the data actually takes.

Listing 10 shows the handler that is being called once a DMA transfer of a bno055 sample is completed. It shows that the bno055 buffer, which now contains a new and complete bno055 sample, is marked as full. The last line of code in this handler changes the next buffer pointer so that the next data transfer writes into the second buffer instead of immediately overwriting the data that must firstly be processed.

[19] **Clock Stretching** is used by an I2C slave if it requires more time before sending the next byte of data. It forces the I2C master to wait by pulling the CLK signal low.

Listings 10: I2C DMA RX complete callback function for bno055

```
1  void HAL_I2C_MemRxCpltCallback(I2C_HandleTypeDef *hi2c)
2  {
3      if (hi2c->Instance == I2C1)
4      {
5          // instance is I2C1
6          // bno055 sample received
7          // transfer into next bno055 buffer is complete,
8          // clear the empty flag
9          bno055_buf[bno055_next_buf_index].empty = 0u;
10         // increment next buffer index
11         index_inc_wrap(&bno055_next_buf_index, BUF_DEPTH_DOUBLE);
12     }
13 }
```

Velocity Kalman Filter Timing

As mentioned above, the second algorithm that runs on the microcontroller of the GSMS is the velocity Kalman filter. This Kalman filter combines a velocity estimate generated by integrating the accelerometer data with a velocity measurement obtained by the GPS. The velocity estimate is calculated and updated at the frequency of the attitude Kalman filter, which is 100 Hz. The velocity measurement however, is not available at such a high data rate. The neo-m9n GPS provides a new velocity measurement every 40ms which means it has an output data rate of only 25 Hz.

The timing for the velocity Kalman filter depends on the timing of the attitude Kalman filter. There is no separate timer for the execution of the velocity Kalman filter. After every execution of the attitude Kalman filter, the velocity Kalman filter is executed as well. Unquestionable, the velocity Kalman filter behaves slightly different than the attitude Kalman filter. This difference is caused by the lower output data rate of the GPS module. The velocity Kalman filter can incorporate a new velocity estimate based on the accelerometer readings every 10ms. But a new GPS velocity measurement is only available every 40ms. The part of the velocity Kalman filter that incorporates the GPS velocity measurement into the calculated velocity is therefore only executed in every fourth iteration.

The velocity measurement required by the attitude Kalman filter is read from the GPS module using UART. The u-blox library is designed in a way that it reads and processes individual characters from the UBX protocol messages transferred via UART. This makes it impossible to use a DMA controller for the UART transfer. Instead, we used an UART library that reads individual characters from the UART peripheral into a buffer. This library makes use of interrupts to avoid polling. The buffer must be emptied regularly to avoid overflow and the loss of data from the GPS. This is done by the timer instance TIM4 of the microcontroller. This timer is configured to repeatedly cause an interrupt at a frequency of 625 Hz. The rate of 625 Hz was chosen based on how quickly the 32 byte UART buffer would overflow if the GPS is transmitting data at 115200 baud.

Listing 11 shows the interrupt handler for the timer instance TIM4. The last line of the handler shows the call to neom9n_check() which causes all characters currently present in the UART buffer to be processed. The characters are part of a UBX-NAV-PVT message that will be parsed once all characters of a message have been received.

Listings 11: Period elapsed callback function for TIM4

```
1  void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
2  {
3      if (htim->Instance == TIM4)
4      {
5          // instance is TIM4
6          // check if UART transmission of debug buffer is not ongoing
7          if (!debug_tx_busy)
8          {
9              // check if current debug buffer is not empty
10             if (!debug_buf[debug_curr_buf_index].empty)
11             {
12                 // start transmission of current debug buffer
13                 transmit_debug();
14                 // set the debug tx busy flag
15                 debug_tx_busy = 0x01;
16             }
17         }
18         // check for new neo-m9n data in UART buffer at 625Hz
19         // Note: This will process received characters and
20         //       update internal neo-m9n state variables
21         //       if a full packet has been received.
22         neom9n_check();
23     }
24 }
```

Debug Buffer

The GSMS transfers debug data to a host computer after every iteration of the data processing step. This means that data is transferred at the base frequency of 100 Hz. The data is transferred via UART using the DMA controller of the STM32F429ZI. The debug data is therefore also stored in a double buffer. This allows data to be transferred to the host, while a new set of debug data is stored in the secondary buffer at the same time. Listing 11 also shows that the timer instance TIM4 is additionally used to initiate the DMA transfer of the current debug buffer once it has been filled and is ready to be transmitted.

Listing 12 shows the interrupt handler, which is called once the DMA transfer is completed. It marks the current debug buffer as empty and adjusts the pointer for the current debug buffer.

Listings 12: UART DMA TX complete callback function for debug buffer

```
1  void HAL_UART_TxCpltCallback(UART_HandleTypeDef *huart)
2  {
3      if (huart->Instance == UART5)
4      {
5          // instance is UART5
6          // transfer of current debug buffer is complete,
7          // set the empty flag
8          debug_buf[debug_curr_buf_index].empty = 1u;
9          // increment current buffer index
10         index_inc_wrap(&debug_curr_buf_index, BUF_DEPTH_DOUBLE);
11         // clear the debug tx busy flag
12         debug_tx_busy = 0x00;
```

```
13      }
14  }
```

5.4.6 Main Execution Flow

Once the initialization of the GSMS has been completed, the system enters the data processing mode. This mode is enabled by setting the processing_enabled flag, which is done as the very last step of the application initialization routine shown in Listing 2.

Listing 13 shows that the main function of the GSMS repeatedly calls the processing() function of the processing.c file. This function is displayed in Listing 13. The processing() function first checks if the data processing has been enabled as described above. The function then continues to check if new data from the bno055 IMU is available. If this check returns true, the new sample will be processed by the attitude Kalman filter. This is initiated by the call to attitude_processing(). Once the attitude Kalman filter has determined an updated system attitude, the velocity_processing() function is called. This function causes an updated velocity estimate to be calculated by the velocity Kalman filter. The attitude as well as the velocity processing functions are explained in more detail below.

Listings 13: Main data processing function

```
1  void processing(void)
2  {
3      if (processing_enabled)
4      {
5          // data processing is enabled
6          if (!bno055_buf[bno055_curr_buf_index].empty)
7          {
8              // unread bno055 raw data is available
9              // update bno055 sample
10             update_bno055_sample(bno055_buf[bno055_curr_buf_index].data);
11             // display bno055 calibration status
12             display_calib_stat();
13
14             // attitude processing
15             attitude_processing();
16
17             // processing of current bno055 buffer is complete,
18             // set the empty flag
19             bno055_buf[bno055_curr_buf_index].empty = 1u;
20             // increment current buffer index
21             index_inc_wrap(&bno055_curr_buf_index, BUF_DEPTH_DOUBLE);
22
23             // velocity processing
24             velocity_processing();
25         }
26     }
27 }
```

Attitude Processing

The first of the two main algorithms to be executeed in each iteration is the attitude Kalman filter. It consists of different components, which all get invoked by the attitude_processing() function. Listing 14 shows the attitude_processing() function.

Listings 14: attitude processing function

```
1  static void attitude_processing(void)
2  {
3      // prediction step (time update)
4      att_kf_predict(gyro);
5      // measurement step (measurement update)
6      // measure gravity vector
7      double gravity_i[VEC_3_SIZE] = {0.0, 0.0, 1.0};
8      double accel_in[VEC_3_SIZE];
9      vec_copy(accel_in, accel, VEC_3_SIZE);
10     att_kf_measure(gravity_i, accel_in);
11     // measure north vector
12     double north_i[VEC_3_SIZE] = {0.0, 0.446, -0.895};
13     double mag_in[VEC_3_SIZE];
14     vec_copy(mag_in, mag, VEC_3_SIZE);
15     att_kf_measure(north_i, mag_in);
16     // propagate attitude error
17     att_kf_propagate();
18     // get attitude kalman filter output
19     att_kf_q_ref(quat_kf);
20 }
```

The attitude Kalman filter consists of the following three main parts:

- Prediction step (att_kf_predict() function)
- Measurement step (att_kf_measure() function)
- Error propagation step (att_kf_propagate() function)

Implementation details and the theoretical foundations of the attitude Kalman filter are discussed in Chapters 5.5.1 and 4.4. Here, we will focus on explaining how the different components of the attitude Kalman filter are called by the attitude_processing() function and how this leads to an updated value for the system attitude.

Prediction step: The first step of the attitude_processing() function is a call to att_pf_predict() which corresponds to the prediction step of the attitude Kalman filter. The att_pf_predict() function takes the gyroscope readings from all three axes as an argument. This data is then used by the Kalman filter to predict the new system attitude by integrating the rotation velocity obtained using the gyroscopes.

Measurement steps: The prediction for the new system attitude must be combined with absolute measurements to increase the accuracy of the output and to avoid drift. The attitude Kalman filter uses two different measurements of the system attitude. Therefore, the measurement step of the attitude Kalman filter is performed twice for each iteration. This is unique to the attitude Kalman filter.

The attitude Kalman filter uses the following two vectors as an absolute measurement for the system attitude:

- A gravity vector measured by the accelerometer
- A north vector measured by the magnetometer

Both measurements are provided by sensors that are part of the bno055 IMU and therefore those measurements are available at the base frequency of 100 Hz.

The measurement of the gravity vector and of the north vector are both processed by a separate call to the att_kf_measure() function. For each call to att_kf_measure(), the actual measurement must be provided as an argument. This corresponds to either the accelerometer readings for the gravity vector or to the magnetometer readings for the north vector.

The att_kf_measure() function requires a second argument to process the given measurement. The measurement given to the att_kf_measure() function always comes in the shape of a vector. It is either a gravity vector which points down towards the center of the earth or it is a north vector which points towards the magnetic north pole. The coordinates of those vectors change, depending on the orientation of the bno055 IMU. This is because the gravity vector and the north vector are measured relative to a frame that is attached to the body of the bno055 IMU. But both of those vectors have fixed coordinates if we choose an inertial reference frame that is fixed to the surface of the earth instead of to the bno055 IMU. Those fixed coordinates that indicate the direction of the measured vector in an earth fixed frame must also be passed to the att_kf_measure() function. This allows the att_kf_measure() function to process the given measurement made in a body fixed frame with respect to the corresponding direction of the measured vector in an inertial frame that is fixed to the earth.

A more detailed explanation of the different measurement frames involved in the data processing is given in Chapter 4.5.

Gravity vector (1st Measurement step): The first of the two measurement steps that is performed by the attitude_processing() function makes use of the gravity vector as an absolute measurement of the system attitude. Herefore, the accelerometer data is passed to the att_kf_measure() function. In this step, the accelerometer readings constitute the measurement of the gravity vector. As described above, we also have to provide a second argument, which indicates the direction of the gravity vector in an earth fixed frame. The earth fixed frame which the attitude Kalman filter uses is an ENU frame (see Chapter 4.5 for details). In an ENU frame, the gravity vector has a very simple representation as illustrated in the graph that can be seen in Figure 26.

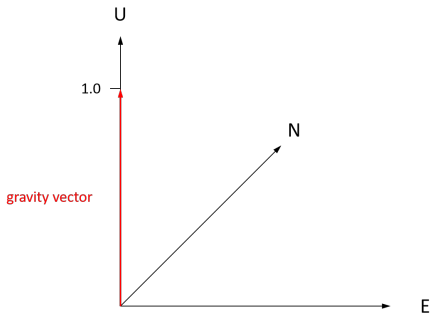


Figure 26: Direction of the gravity vector in the inertial ENU reference frame

The gravity vector always has the coordinates (0, 0, 1) in the ENU frame. The length of the vector has no meaning, therefore the vector is normalized to have a length of 1. The important aspect of this gravity vector in the inertial frame is the direction in which it points. As shown in the above graph, the gravity vector points upwards, away from the center of the earth. This is counter intuitive at first, as gravity actually pulls down towards the center of the earth. In order to understand why the gravity vector truly points upwards, we need to understand how the bno055 IMU represents the measured gravity or, more precisely, the measured acceleration. This is explained in Chapter 4.5. When the att_kf_measure() function is called with the two arguments as described above, the attitude Kalman filter uses the given gravity vector measurement and its corresponding orientation in the inertial frame to update the calculated system attitude.

North vector (2nd Measurement step): The second measurement step which the attitude_processing() function performs, makes use of the north vector as an absolute measurement of the system attitude. Herefore, the magnetometer data is passed to the att_kf_measure() function. In this step, the magnetometer readings constitute the measurement of the north vector. This measurement step for the north vector works in the same way as the previous measurement step for the gravity vector. In addition to the magnetometer readings, we must again provide a second argument, which indicates the direction of the north vector in an earth fixed frame. The earth fixed frame is again an ENU frame. The representation of the north vector in an ENU frame is more complicated than the one for the previous gravity vector. The following graph in Figure 27 illustrates the representation of the north vector in an ENU frame.

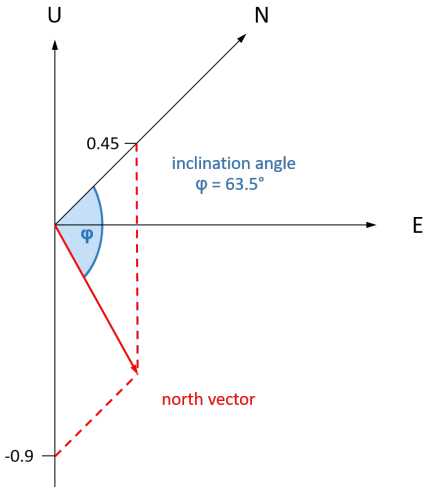


Figure 27: direction of the north vector in the inertial ENU reference frame

The north vector always has the coordinates (0, 0.446, -0.895) in the ENU frame. As with the gravity vector, the length of the north vector has no meaning, therefore it is normalized to have a length of 1. The important aspect of this north vector in the inertial frame is the direction in which it points. As shown in the above graph the north vector points northwards. This is intuitive, as the magnetic north pole is located in that direction. But in Figure 27, we can clearly see, that the north vector is not exactly (0, 1, 0). Instead it has a second non-zero component which makes it not only point northwards but also down into the earth. This is caused by magnetic inclination[20].

The magnetic inclination is location dependent (and also slowly changes over time). This means that the direction of the north vector, as shown in the graph above, can only be determined for a specific location. The entire GSMS was designed and tested in Winterthur, Switzerland. For that reason the software and all data processing algorithms are currently configured to use geographical parameters that are valid in Winterthur. The exact location that was used is shown on the map in Figure 28.

[20] **Magnetic inclination** describes the angle between the magnetic field vector (north vector) and the horizontal plane (the plane is tangent to the surface of the earth at the point of interest). The magnetic inclination is positive when the magnetic field points downwards into the earth and negative when it points upwards. [20]

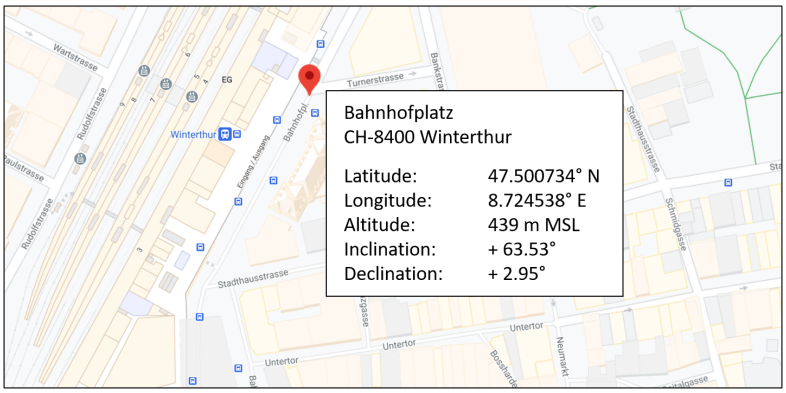


Figure 28: Geographical location with corresponding constants used in the GSMS

As we can see in the image above, the magnetic inclination in Winterthur is 63.53°. Again, a positive inclination value means that the north vector points downwards into the earth. This is exactly what we can see in the above illustration of the north vector.

The values for the magnetic inclination and declination were determined using a magnetic field calculator[21]. This tool allows to calculate the properties of the magnetic field for any given location and date. Figure 29 shows the output of the tool displaying the current parameters in Winterthur.

Magnetic Field

Model Used:	WMM-2020
Latitude:	47° 29' 54" N
Longitude:	8° 43' 58" E
Elevation:	439.0 m Mean Sea Level

Date	Declination (+ E | - W)	Inclination (+ D | - U)	Horizontal Intensity	North Comp (+ N | - S)	East Comp (+ E | - W)	Vertical Comp (+ D | - U)	Total Field
2020-12-11	2° 57' 9"	63° 31' 41"	21,475.8 nT	21,447.3 nT	1,106.2 nT	43,126.6 nT	48,177.9 nT
Change/year	0° 9' 43"/yr	0° 1' 11"/yr	8.4 nT/yr	5.2 nT/yr	61.0 nT/yr	53.9 nT/yr	51.9 nT/yr
Uncertainty	0° 22'	0° 13'	128 nT	131 nT	94 nT	157 nT	145 nT

Figure 29: Output of the magnetic field calculator for Winterthur

When the att_kf_measure() function is called with the two arguments as described above, the attitude Kalman filter uses the given north vector measurement and its corresponding orientation in the inertial frame to update the calculated system attitude.

Velocity Processing

After the attitude Kalman filter has calculated an updated system attitude using the prediction and measurement steps, the velocity processing is performed. The velocity processing is also based on a Kalman filter consists again of a prediction and a measurement step. The GSMS uses only one absolute measurement of the velocity as an input. This means that there will only be one measurement step in the velocity Kalman filter as opposed to the attitude Kalman filter which performed two measurement steps. The velocity processing is performed by the function velocity_processing().

Prediction step: The first part of the velocity processing consists of the prediction step. This step predicts the system velocity by calculating an integral over the current linear acceleration. The linear acceleration must first be determined based on the raw accelerometer measurements. This works by substracting the gravity vector from the accelerometer measurements. The remaining

[21] The calculator is provided by the National Centers for Environmental Information [21]

vector after this subtraction is equal to the linear acceleration of the system. The linear acceleration is the part of the acceleration that actually causes a change of velocity in the system. Figure 30 visualizes how the linear acceleration vector, the acceleration vector and the gravity vector are related to each other.

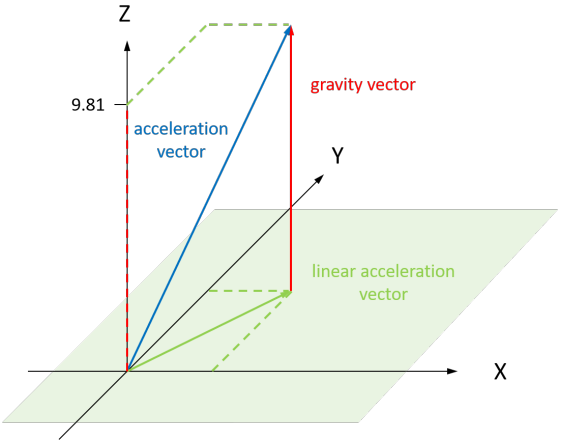


Figure 30: Components measured by the accelerometer (gravity vector, linear acceleration)

As shown in the image above, a body might be exposed to acceleration caused by the gravitational field of the earth and by additional external forces applied to the body. The term linear acceleration refers to the part of the acceleration that is caused by additional external forces. Therefore, linear acceleration is the part of acceleration, that actually results in a change of velocity of the observed body. To obtain the linear acceleration from measurements made by an accelerometer, the acceleration caused by the gravitational field of the earth must be removed first.

After the gravity vector (depicted in red) has been subtracted from the acceleration vector (depicted in blue), the linear acceleration vector (depicted in green) remains. The acceleration is measured in a frame that is fixed to the body of the car. This means that the X, Y and Z axes shown in the above graph are fixed to the car (they rotate with the car). Therefore the linear acceleration vector always lies in the X, Y plane. This is intuitive as the car never accelerates up- or downwards relative to its own Z axis.

The IMU is placed in the car with the Y axis pointing in the direction of travel. This means that the most significant component of the linear acceleration vector will always be on the Y axis. This is where the actual acceleration, caused by the motors of the car, will be measured. Acceleration is only measured on the X axis if the wheels of the car are slipping (refer to Chapter 4.5 for more details).

The first few lines of Listing 15 declare the steps performed to calculate the linear acceleration. In addition to the accelerometer measurement, this calculation also requires the previously determined system attitude. This is necessary as the gravity vector is described in the inertial frame and must therefore first be rotated into the body frame before subtracting it from the accelerometer measurements, which are also measured in the body frame. After the linear acceleration has been determined, the function vel_kf_predict() is called with the linear acceleration as an argument. This function then performs the prediction step by integrating the given linear acceleration. The result of the prediction step is an updated value for the system velocity.

Listings 15: Velocity processing function

```
1 static void velocity_processing(void)
```

```
2  {
3      // calculate linear acceleration
4      double quat_kf_conj[QUAT_SIZE];
5      quat_conj(quat_kf, quat_kf_conj);
6      double gravity_i[VEC_3_SIZE] = {0.0, 0.0, 9.8053};
7      double gravity_b[VEC_3_SIZE];
8      // rotate gravity vector from the inertial frame into
9      // the body frame
10     rotate_by_quat(gravity_i, quat_kf_conj, VEC_3_SIZE);
11
12     // prediction step (time update)
13     vel_kf_predict(lin_accel_kf);
14
15     // check if unread neo-m9n raw data is available
16     if (neom9n_data_ready(NEOM9N_UPDATE_BUF)
17     {
18         // unread neo-m9n raw data is available
19         // update neo-m9n sample
20         update_neom9n_sample(&neom9n_buf);
21
22         // change velocity reference frame from NED to ENU
23         double vel_ENU[VEC_3_SIZE];
24         vel_ENU[VEC_ENU_E] = vel[VEC_NED_E];
25         vel_ENU[VEC_ENU_N] = vel[VEC_NED_N];
26         vel_ENU[VEC_ENU_U] = -vel[VEC_NED_D];
27         // correct declination
28         double ang = 2.9425 / 180.0 * M_PI;
29         double measured_i[VEC_3_SIZE];
30         rotate_z_by_ang(vel_ENU, ang, measured_i);
31         // rotate measurement from the inertial frame into
32         // the body frame
33         double measured_b[VEC_3_SIZE];
34         rotate_by_quat(measured_i, quat_kf_conj, measured_b);
35
36         // measurement step (measurement update)
37         // measurement velocity vector
38         vel_kf_measure(measured_b);
39     }
40     // get velocity kalman filter output
41     vel_kf_vel(velocity_kf);
42 }
```

Measurement step: After the prediction step is completed, the system velocity must be combined with an absolute measurement. This is required to prevent drift, which is caused by integrating the accelerometer data in the prediction step.

The absolute measurement used in the velocity Kalman filter is provided by the GPS module. This module has an output data rate of 25 Hz, which means that the velocity measurement is only available in every fourth iteration of the velocity_processing() function. As shown in Listing 15, the measurement step of the velocity Kalman filter is therefore only executed if a new GPS sample is available.

The GPS sample consists of a three dimensional measurement of the system velocity in the ENU frame. This velocity measurement must first be converted into the ENU frame. After this conversion, the magnetic declination must be taken into account by rotating the velocity measurement around the U (respectively Z) axis (refer to Chapter 4.5 for more details). The velocity measurement in the ENU frame is rotated around the U axis by the declination angle. The magnetic declination angle is location dependent and was determined using the same method as for the magnetic inclination angle, described in Chapter 5.4.6. The map in Appendix A.2 shows a visual representation of the declination angle in different parts of the world. It is clearly visible that Switzerland has a declination angle of around 3 degrees.

The velocity measurement made by the GPS is relative to the inertial frame. This means that the velocity measurement must first be rotated into the body frame. For this rotation, the previously calculated system attitude is needed again. The vel_kf_measure() function is then called with the velocity measurement in the body frame as an argument. This measurement step again updates the computed system velocity. It is the last step performed in the velocity_processing() function. After this step, the GSMS is ready to output the final value of the system velocity for the current iteration. In the next iteration all values will be updated or recalculated using the same process.

5.4.7 Debugging

The GSMS is designed to be easily debuggable. The most simple method for debugging and observing the behaviour of the GSMS is using the ST-LINK debugger that is part of the Nucleo-144 development board. The screenshot in Figure 31 shows how intermediate system variables as well as the final output of the data processing algorithm can be observed when the system is running. In the screenshot, the variable watch window of the Keil μVision IDE is used to display the attitude quaternion and the computed estimate for the system velocity.

Name	Value	Type
accel	0x20000530 accel	double[3]
mag	0x20000A50 mag	double[3]
gyro	0x20000848 gyro	double[3]
quat	0x20000A80 quat	double[4]
[0]	0.3963012695313	double
[1]	0.0074462890625	double
[2]	-0.0023193359375	double
[3]	-0.9180908203125	double
vel	0x20000AC8 vel	double[3]
quat_kf	0x20000AA0 quat_kf	double[4]
velocity_kf	0x20000BD0 velocity_kf	double[3]
[0]	-0.185273302423	double
[1]	-0.1575248360785	double
[2]	0.02236956125804	double
gsms_error	0x00	uchar
dt_proc	0x04	uchar
att_error	1.733474192196	double

Figure 31: Variable watch window of the Keil μVision IDE when debugging the GSMS

The GSMS also offers the possibility to transmit different signals and system variables to a host computer for further analysis and processing. The system transmits a buffer containing debug data (debug buffer) to the host system after every iteration of the main processing() function. This transfer is done over UART using a DMA controller of the STM32F429ZI. Listing 16 shows the different fields (and their offset values) of the debug buffer. This buffer could be expanded to contain any number of additional variables.

Listings 16: Debug buffer defines

```
1 // debug buffer defines
2 #define DEBUG_BUF_SIZE 68
3 #define DEBUG_BUF_OFFSET_BNO055 0
```

```
4  #define DEBUG_BUF_OFFSET_NEOM9N 38
5  #define DEBUG_BUF_OFFSET_QUAT_KF 50
6  #define DEBUG_BUF_OFFSET_TICK 58
7  #define DEBUG_BUF_OFFSET_VEL_KF 62
8  #define DEBUG_BUF_NEOM9N_PADDING_VAL 0X00000080
9  #define DEBUG_BUF_QUAT_SCALE ((DOUBLE) (1u << 14u))
10 #define DEBUG_BUF_VEL_SCALE 1000.0
```

This data is intended to be recorded on the host computer so that it can then be processed further. This offers a number of different possibilities to perform experiments and to analyse the behaviour of the GSMS:

- Raw measurements can be recorded and the data processing can be tested on the host computer.
- The resulting values from the data processing of the GSMS can be visualized.
- Different methods and algorithms for computing the system attitude and the system velocity can be compared to each other.

We created different tools and scripts to process the data that is periodically transmitted to the host computer. The following paragraphs describe one possible workflow that makes use of these tools for conducting an experiment with the GSMS.

The transmitted data is recorded into a single file (capture.txt) on the host computer using a serial monitor. The interface of the serial monitor is shown in Figure 32.

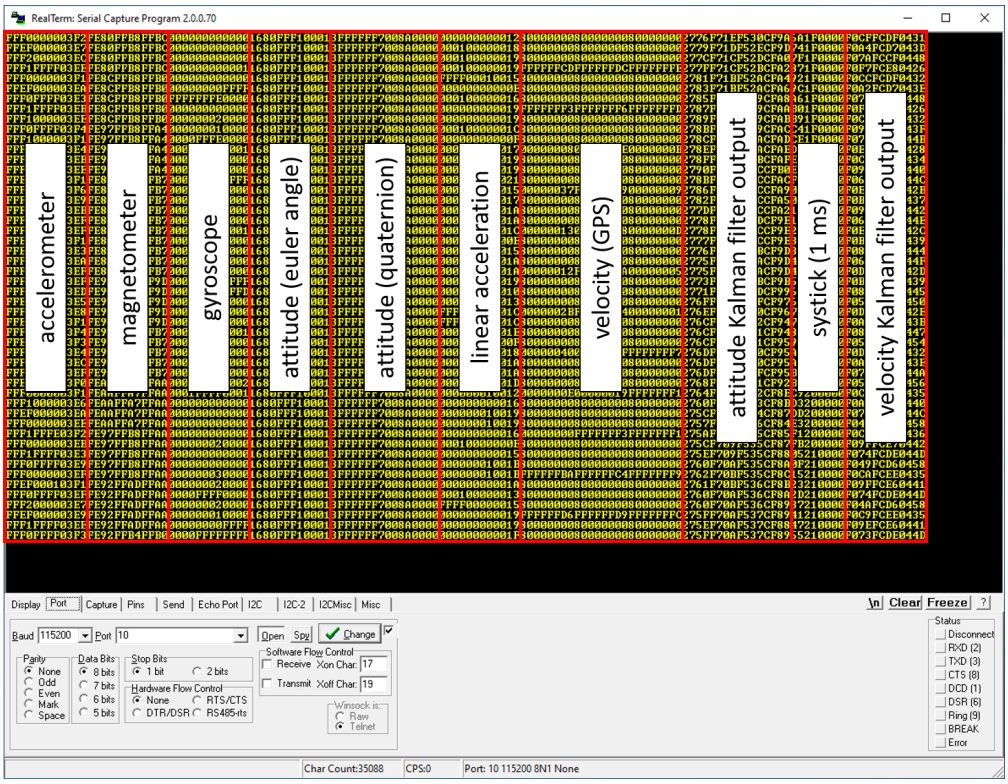


Figure 32: Serial monitor interface displaying data from the GSMS

The serial monitor program shown above is configured to display the complete contents of the debug buffer on one line. Each line is a newly transmitted sample of the debug buffer. The

different variables contained in the debug buffer are labeled on the screenshot. Once this data has been recorded into a file it can be split up into its different components by using a short script that we created for this specific task. The script is show in Listing 17.

Listings 17: Script to preprocess captured debug data from the GSMS

```
# !/bin/bash
# process raw data from capture
echo "Processing␣raw␣data␣from␣capture..."
# insert line break after each sample
sed 's/.\{136\}/&\n/g' capture.txt > samples.txt
# prompt to check for incomplete samples
echo "Check␣samples.txt␣for␣incomplete␣or␣misaligned␣samples."
read -p "Press␣enter␣to␣continue..."
# separate raw bno055 data
sed 's/.\{60\}$//' samples.txt > bno055_raw.txt
# separate raw neo-m9n data
sed 's/.\{36\}$//' samples.txt > temp.txt
sed 's/^.\{76\}//' temp.txt > neom9n_raw.txt
# separate raw attitude kalman filter output data
sed 's/.\{20\}$//' samples.txt > temp.txt
sed 's/^.\{100\}//' temp.txt > attkf_raw.txt
# separate raw velocity kalman filter output data
sed 's/.\{124\}//' samples.txt > velkf_raw.txt
# output number of processed samples
count=$(wc -l < samples.txt)
echo "Successfully␣processed␣${count}␣samples."
# remove temporary files
rm temp.txt
```

Figures 33 and 34 show the output generated by the script and the resulting files when it is used to process data that was previously captured using the serial monitor.

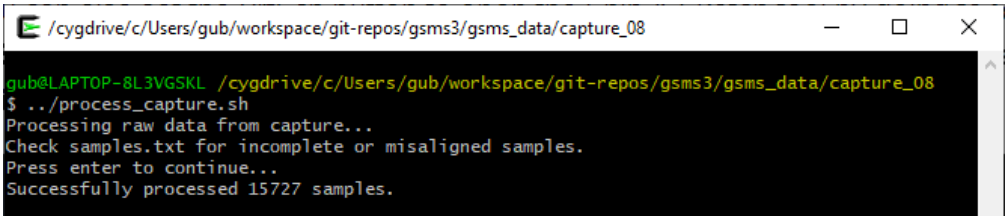


Figure 33: Output of the script used to preprocess the captured data from the GSMS

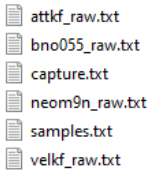


Figure 34: Files generated after preprocessing the captured data from the GSMS

Those files now contain the debug data in a format that is suitable as an input to our MATLAB

plot script. This script creates various plots for visualizing the behaviour of the GSMS. Refer to Chapter 6 for a discussion of the plots created by this MATLAB script.

5.4.8 MATLAB for Algorithm Design

The attitude Kalman filer and the velocity Kalman filter algorithms of the GSMS were designed in MATLAB. This allowed for faster development and easier testing of the algorithms compared to directly implementing the algorithms in C code. MATLAB offers a tool called *MATLAB Coder* to automatically generate C code from existing MATLAB code[22]. This C code could then be integrated into the existing μVision project.

5.5 Data Processing

Readings from the implemented sensors must be combined to determine an estimate of the ground speed that is as accurate as possible, which the GSMS uses two Kalman filters for. The first Kalman filter is only used to determine the system attitude while the second Kalman filter actually estimates the final system velocity. Both algorithms are explained in this section. The source code for those algorithms is written in MATLAB and can be found in Appendix B.1 and B.2.

Figure 35 gives an overview of the two Kalman filters and their input values to the prediction and measurement steps. The table also shows at what frequency each part of the algorithm is executed.

		Attitude Kalman filter	Velocity Kalman filter
Estimated variable		System attitude	System velocity
Prediction step	**input**	Angular velocity (gyroscope)	Linear acceleration (accelerometer)
	frequency	100 Hz	100 Hz
Measurement step	**input (1st)**	Gravity vector (accelerometer)	Velocity (GPS)
	input (2nd)	North vector (magnetometer)	n/a
	frequency	100 Hz	25 Hz

Figure 35: Kalman filters and their input values

Chapter 4.4 gives an overview of how Kalman filters work and it builds the theoretical foundation to understand the descriptions of the attitude and velocity Kalman filter that we implemented for the GSMS. The attitude and the velocity Kalman filter both work with three dimensional measurements made in different reference frames, which are described in Chapter 4.5.

5.5.1 Attitude Kalman Filter

The attitude Kalman filter is implemented in MATLAB. The code of the entire algorithm can be found in Appendix B.1. The following description of the attitude Kalman filter refers to different sections of the code at various points. The MATLAB code contains extensive explanations for each step performed. We therefore recommend to also refer to the comments in the code in addition to the explanations in this chapter.

The attitude Kalman filter[23] is divided into multiple files as shown in Figure 36. In addition to the those files, the algorithm requires additional functions that are depicted in Figure 37. The contents of these files above can be found in aforementioned Appendix B.1.

[22] The MATLAB Coder plugin for MATLAB allows to generate platform independent C code from MATLAB code.

[23] The attitude Kalman filter of the GSMS is based on the approach described in [22] and our implementation closely follows the implementation found in the GitHub repository with the following url: `https://github.com/Stapelzeiger/adcs`.

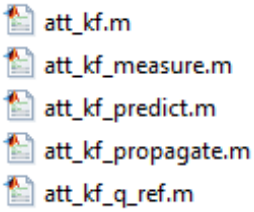


Figure 36: Attitude Kalman filter functions

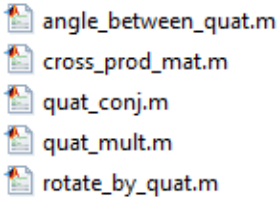


Figure 37: Additional functions

There is also a test file, which can process recorded measurements on the host computer and generate various plots to analyse the behaviour of the attitude Kalman filter. This test file was used to adjust and verify the attitude Kalman filter algorithm. The source code of the test file can be found in Appendix B.1.6.

Overview

The attitude Kalman filter is an EKF as the equations required to describe the physical properties of the system attitude are non-linear. The attitude Kalman filter is used to track the system attitude, which is represented using a quaternion.

The quaternion q_ref (see Appendix B.1.1, line 3) is the main variable that stores the absolute system attitude. It describes the system attitude as a rotation between the body frame and the inertial frame. This means that the attitude of the race car is always described relative to the earth fixed inertial reference frame. If the quaternion q_ref represents no rotation (if q_ref is the identity quaternion), then the body frame is aligned with the inertial frame. In that state, the two reference frames are aligned as shown in Figure 38.

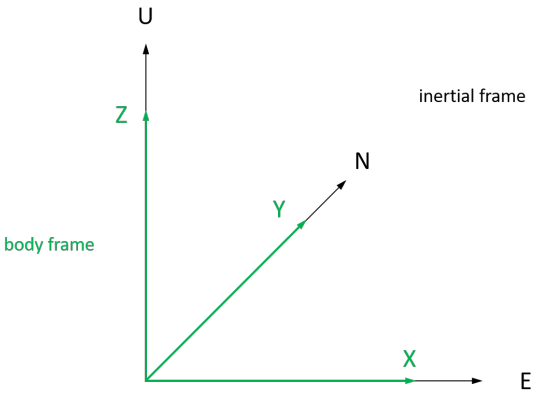


Figure 38: Alignment of body frame and inertial frame

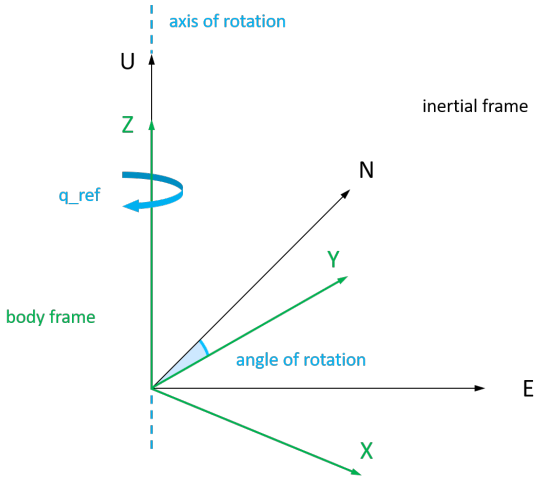


Figure 39: Rotation q_ref between body frame and inertial frame

If q_ref represents a non-zero rotation, then the body frame is no longer aligned with the inertial frame as shown in Figure 39. This picture shows a situation where the difference in alignment between the inertial and the body frame can be expressed as a rotation around the U axis by a certain angle. The axis of rotation does not have to be the U axis, it can be an arbitrary axis. The last three components of a quaternion are used to describe the axis of rotation while the first component of a quaternion describes the angle of rotation.

The attitude Kalman filter has a state vector x that tracks the following variables:

- x, y and z components of the attitude error a
- x, y and z components of the gyroscope bias

The state vector is a column vector with 6 elements. It is important to note, that the attitude Kalman filter does not directly track the quaternion q_ref as part of its state vector. Instead, it tracks an attitude error. At the beginning of each iteration, this error is assumed to be zero. And at the end of each iteration, the attitude error is added to the reference attitude quaternion q_ref and then reset to zero. The system attitude is therefore expressed using two parts as described in the following equation:

$$q(t) = \delta q(a(t)) \oplus q_{ref}(t)$$

$q(t)$ is the current system attitude represented as a quaternion. $a(t)$ is the attitude error tracked using the state vector x of the attitude Kalman filter. $a(t)$ is a three element vector that represents a rotation. The direction of the vector $a(t)$ represents the axis of rotation and the length of the vector $a(t)$ determines the angle of rotation. This representation is more suitable to be used in the Kalman filter than a quaternion. The representation used for the attitude error $a(t)$ is especially well suited to represent small angles of rotation. This is convenient as the attitude error is usually only a very small angle of rotation. The attitude quaternion q_ref however might represent any angle between 0°and 180°of rotation, depending on the current attitude of the race car.

The above approach of tracking an attitude error instead of directly tracking the attitude is unique to the attitude Kalman filter. The EKF used to track the attitude in the GSMS has therefore a special name. It is called *Multiplicative Extended Kalman Filter* (MEKF). The term "multiplicative" refers to the fact, that the attitude error $a(t)$ tracked by the attitude Kalman filter is propagated to the reference attitude quaternion q_ref using a quaternion multiplication operation. To perform this quaternion multiplication at the end of each iteration, $a(t)$ is first converted

into its quaternion representation and then multiplied with q_ref.

In the following pages we will explain in detail how the attitude Kalman filter works by following the flow of execution for one iteration of the algorithm.

Initialization

Before the first iteration, the attitude Kalman filter is initialized using the code shown in Appendix B.1.1.

Initial values: The initialization function sets the initial values for the state vector x and for the error covariance matrix P.

The initial value for the state vector x is set to the zero vector (see Appendix B.1.1, line 19). The state vector tracks both the attitude error $a(t)$ and the gyroscope bias. Setting the state vector x to the zero vector therefore results in the following two initial conditions:

- The attitude error $a(t)$ represents a rotation of zero degrees. This is a required state at the beginning of each iteration.
- The gyroscope bias is set to zero for all three axes. This value is chosen because the actual gyroscope bias is initially unknown. The gyroscope bias tracked by the state vector x will change over time.

The actual system attitude is also unknown when the attitude Kalman filter is initialized. Therefore the variable q_ref is set to the identity quaternion (1, 0, 0, 0) as shown in Appendix B.1.1 on line 18.

The values set for q_ref and for the gyroscope bias by the initialization vector have a very high uncertainty (they were chosen at random because the actual values are initially unknown). This high amount of uncertainty must be reflected by the initial value of the error covariance matrix P. As shown in Appendix B.1.1 on line 22, the diagonal elements of the matrix P are set to very high values.

Parameters: In addition to setting the above initial values, the initialization function also defines the following two constant matrices:

- the process noise covariance matrix Q
- the measurement noise covariance matrix R

Those covariance matrices describe the uncertainties in the process model respectively in the measurements. The initialization function uses the constants defined on lines 10 to 17 in Appendix B.1.1 to set the values for the matrix Q (on line 22 of Appendix B.1.1) and for the matrix R (on line 23 of Appendix B.1.1). The values used in both covariance matrices were chosen experimentally.

After this initialization routine, the prediction and measurement steps will be executed in every iteration of the algorithm. The steps of one complete iteration are explained in the following chapters, starting with the prediction step.

Prediction step

The attitude Kalman filter starts each iteration with an attitude error $a(t)$ that has been reset to zero. This is intuitive, as the Kalman filter is designed in a way that its previous estimate is free of any error. Therefore the attitude error $a(t)$ is zero at the beginning of each iteration. During this prediction step, the attitude error $a(t)$ will remain zero.

Instead of changing the attitude error $a(t)$, the prediction step directly adjusts the reference quaternion q_ref. This is done by integrating the measurements made from the gyroscope. As

shown in Appendix B.1.2 on line 1, the gyroscope measurements w must therefore be passed to the att_kf_predict() function as an argument.

The integral over the angular velocity w (gyroscope measurements) is computed using the time dt that has passed since the last iteration of the algorithm. The GSMS is designed such that dt is always 10 ms.

Line 12 in Appendix B.1.2 shows, that the gyroscope bias is removed form the gyroscope measurements w before calculating the angle of rotation (line 14). Depending on the angle of rotation, a different method is used to compute a delta quaternion $delta\_q\_ref$ that represents the rotation since the last iteration of the algorithm (lines 15 to 23). This delta quaternion ($delta\_q\_ref$) is then combined with the reference attitude quaternion q_ref using quaternion multiplication (line 25). The resulting value is the new reference attitude quaternion q_ref.

After q_ref has been updated, the error covariance matrix P must be updated as well. This is required as the process model (which made use of the gyroscope measurements) introduced new errors. Those errors are represented by the process noise covariance matrix Q. The steps to update P therefore involve the matrix Q. The required equations have been derived in [22] and they are shown in Appendix B.1.2 from line 27 to line 41.

Measurement step: The measurement step of the attitude Kalman filter will correct the previously calculated estimate using absolute measurements of the system attitude. The measurement step is executed twice, as the GSMS uses the following two absolute measurements for the system attitude:

- Gravity vector given by the accelerometer
- North vector given by the magnetometer

The following explanations are valid for both calls to the att_kf_meaure() function. Appendix B.1.3 shows on line 1 that each call to the function requires the following two arguments:

- a vector $expected\_i$ expressed in the inertial frame that represents the expected direction of a measured vector in the inertial frame (the length of this vector has no meaning)
- the coordinates of a measured vector $measured\_b$ expressed in the body frame (the length of this vector has no meaning)

The measurement step uses those two arguments to determine the attitude error $a(t)$. This is done by executing the following steps:

1. A rotation matrix is calculated which rotates any vector from the body frame into the inertial frame and then onto a vector m which is perpendicular to an arbitrary projection plane (lines 18 - 23).

   The projection plane is required to obtain a measurement that consists of only two components instead of three as given by the north or gravity vector. This is caused by the fact that the attitude of an object in 3D space can not be fully defined using only the direction given by a single vector such as a gravity or north vector.

2. The vector $expected\_i$ is rotated into the body frame by using the reference attitude quaternion q_ref (line 23).

3. The vector $measured\_b$ is rotated using the rotation matrix calculated in step 1. The rotated vector is the projected onto the arbitrary projection plane to determine the measurement z (lines 24 - 31)

4. To compare the measurement z with the current state x, the observation matrix H must be calculated (lines 32 - 36). This matrix is used to map the state x onto the measurement.

5. The measurement z is then used to calculate the innovation vector y (lines 37 - 39).
6. To combine the measurement with the predicted state in an optimal way, the Kalman gain K must be calculated (lines 40 - 42). The Kalman gain depends on the measurement noise covariance matrix R.
7. The innovation vector y and the Kalman gain matrix K are then used to update the state vector x. This step changes the attitude error $a(t)$ depending on how far the measurement differs from the previous prediction.
8. After the state vector x has been updated, the error covariance matrix P needs to be updated accordingly.

The equations for steps 4 to 8 are derived in [22]. After executing those steps, the attitude error $a(t)$, which is part of the state vector x might be non-zero. Before the next iteration, the attitude error $a(t)$ must be transferred to the reference attitude quaternion q_ref and then reset to zero. This is done by the error propagation step explained in the next section

Error propagation step

The error propagation step must always be called before a new iteration of the attitude Kalman filter algorithm is started. This error propagation step ensures the following two things:

- The reference attitude quaternion q_ref is updated according to the attitude error $a(t)$ determined by the previous iteration.
- The attitude error $a(t)$ is reset to zero for the next iteration.

On line 7 in Appendix B.1.4 $a(t)$ is converted into the rotation quaternion $delta\_q\_of\_a$. The derivation of this equation is given by [22]. The resulting delta quaternion $delta\_q\_of\_a$ is then combined with the reference attitude quaternion q_ref using quaternion multiplication (line 10). This quaternion multiplication to transfer the attitude error $a(t)$ of the current iteration over to the reference attitude quaternion q_ref is the reason, why this type of algorithm is called an MEKF. Line 14 shows how the attitude error $a(t)$ is reset to zero for the next iteration.

After executing this att_kf_propagate() function, the current iteration is completed and the next iteration of the attitude Kalman filter can start. To obtain the current system attitude that was computed by the attitude Kalman filter, the function att_kf_q_ref() must be called (see Appendix B.1.5). This function returns the current system attitude in quaternion form.

5.5.2 Velocity Kalman Filter

The velocity Kalman filter is also implemented in MATLAB. The code of the entire algorithm can be found in Appendix B.2. The following description of the velocity Kalman filter refers to different sections of the code at various points. The MATLAB code contains extensive explanations for each step performed. We again recommend to also refer to the comments in the code in addition to the explanations in this chapter.

The velocity Kalman filter is divided into multiple files as shown in figure 40. In addition to those files, the algorithm requires additional functions that are found in the files shown in figure 41.

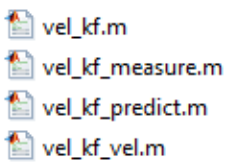


Figure 40: Velocity Kalman filter functions

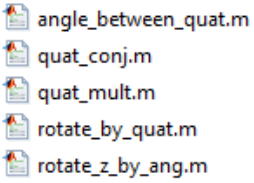


Figure 41: Additional functions

Similar to the attitude Kalman filter files, there is again a test file, which can process recorded measurements on the host computer and generate various plots to analyse the behaviour of the velocity Kalman filter. This test file was used to adjust and verify the velocity Kalman filter algorithm. The source code of the test file can also be found in Appendix B.2.5. The velocity Kalman filter of the GSMS was entirely developed by us and is not based on any preexisting work.

Overview

The velocity Kalman filter is a regular Kalman filter, as the equations required to describe the system velocity are all linear. This makes the velocity Kalman filter significantly simpler than the attitude Kalman filter. The velocity Kalman filter can also directly track the velocity of the system in its state vector x. The state vector x for the velocity Kalman filter contains the following elements:

- x, y and z components of the system velocity

The state vector is therefore a column vector with 3 elements.

The next paragraphs will explain how the velocity Kalman filter works by following the flow of execution for one iteration of the algorithm.

Initialization

Before the first iteration, the velocity Kalman filter is initialized using the code shown in Appendix B.2.1.

Initial values: The initialization function sets the initial values for the state vector x and for the error covariance matrix P. The initial value for the state vector x is set to the zero vector (Appendix B.2.1, line 14). The state vector directly tracks the system velocity. Setting the state vector x to the zero vector therefore results in the following initial condition:

- The system velocity vel is set to the zero vector.

The system is usually initialized while the race car is not moving, therefore the above initial value for the system velocity vel is a reasonable choice. The system might already be moving, in which case the initial value would be wrong. This scenario is taken into account by assuming a very high uncertainty for the initial value. This high amount of uncertainty must be reflected by the initial value of the error covariance matrix P. As shown in Appendix B.2.1 on line 17, the diagonal elements of the matrix P are set to very high values.

Parameters: In addition to setting the above initial values, the initialization function also defines the following two constant matrices:

- the process noise covariance matrix Q
- the measurement noise covariance matrix R

Those covariance matrices describe the uncertainties in the process model respectively in the measurements. The initialization function uses the constants defined on lines 10 to 12 in Appendix B.2.1 to set the values for the matrix Q (on line 15 in Appendix B.2.1) and for the matrix R (on line 16 in Appendix B.2.1). The values used in both covariance matrices were chosen experimentally.

After this initialization routine, the prediction and measurement steps will be executed in every iteration of the algorithm. The steps of one complete iteration are explained in next, starting with the prediction step.

Prediction step

The velocity Kalman filter predicts the system velocity by integrating the linear acceleration measured using the accelerometer. As shown in Appendix B.2.2 on line 1, the linear acceleration lin_accel must be passed to the vel_kf_predict() function as an argument. The integral over the linear acceleration lin_accel is computed using the time dt that has passed since the last iteration of the algorithm (line 10). The GSMS is designed such that dt is always 10 ms.

After the state vector x has been updated, the error covariance matrix P must be updated as well (line 12). This is required as the process model (which made use of the accelerometer measurements) introduced new errors. Those errors are represented by the process noise covariance matrix Q. The steps to update P therefore involve the matrix Q. The used equation is shown in Chapter 4.4.

Measurement step

The measurement step of the velocity Kalman filter will correct the previously calculated estimate using an absolute measurement of the system velocity. The GSMS uses the following absolute measurements for the system velocity:

- velocity measured in 3 dimensions using a GPS

Appendix B.2.3 shows on line 1 that each call to the function requires the following argument:

- a vector $measured\_b$ that represents the system velocity expressed in the body frame.

The measurement step combines this measurement with the previously calculated prediction for the system velocity. This is done with the following steps:

1. The innovation vector y is calculated as the difference between the predicted and the measured system velocity (line 8).
2. The Kalman gain matrix K is calculated to optimally combine the measurement with the predicted value (lines 10 - 11). K depends on the measurement noise covariance matrix R.
3. The innovation vector y and the Kalman gain matrix K are then used to update the state vector x (line 13).
4. After the state vector x has been updated, the error covariance matrix P needs to be updated accordingly.

After executing this att_kf_measure() function, the current iteration is completed and the next iteration of the velocity Kalman filter can start. To obtain the current system velocity that was computed by the velocity Kalman filter, the function vel_kf_vel() must be called (refer to Appendix B.2). This function returns the current system velocity as a three dimensional vector.

6 Results

In this chapter we introduce our test concept and the test procedure to prove the validity of our designed system. We also describe the tests performed and discuss the results.

6.1 Test Concept

In order to test the functionality of the measurement system, the proprietary algorithm by Bosch is used as a baseline to compare to the self-developed algorithms implemented as a result of this work. It is possible to compare these algorithms using MATLAB test scripts and obtain differences in measurements and magnitude of the errors. To collect measurements and data for the test, the sensors can be attached to a passenger car and driven in a variety of traffic scenarios. To maximise the amount of measurements taken, these scenarios should include driving at constant speed, up and down inclines, and accelerating and braking with varied force. Testing close to the scenarios encountered while racing helps detect and mitigate edge cases early, and avoid surprises when used together with the Autonomous System. The sensor data is captured over a serial interface on a host computer, allowing for later offline evaluation.

The developed system must closely follow the predefined specification, which means that the test concept must be designed to ensure that this is indeed the case. This also provides a guarantee on the accuracy of the system, provided the specification is tight. To this end, the requirement of the test concept is to perform acceptance tests[24] in the form of performance and field tests (as illustrated in Figure 42).

Testing the system in the field is vital, as it captures its reaction and sensitivity to variations of the surrounding conditions. The ability to perform tests under different conditions indicates the resilience of the system to its environment. The input to the field test is the chosen route, actual day, time and weather conditions, as well as the system under test. The output is the measurement data along with the environmental conditions along the route. This data can be stored in a database as historical data, as well as to aid in data processing tasks.

The measurements collected during the tests are assessed based on pass/fail criteria, and evaluated after the test drive. The results from the 2 discussed algorithms are compared and if they are within some absolute error from each other, then the test passes.

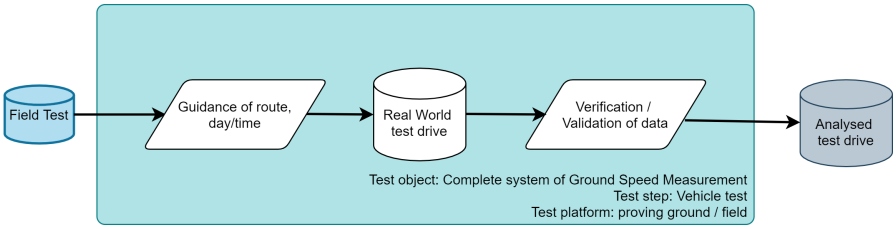


Figure 42: Detailed test concept illustration

As described in Table 4, a total of three different tests are performed. The first two tests are intended to prove the correctness of our custom sensor fusion algorithms and to qualitatively assess the accuracy of the system. The third test focuses on demonstrating the behaviour of the GSMS when used on tracks with a significant slope.

[24] **Acceptance Tests** are tests conducted to determine if the requirements of a specification are met (in engineering). The delivered system is checked for applicability / usability.

Table 4: Test Scenarios Overview

Test #	Description	Test Goal
T1	Drive a figure 8 twice in a residential area	Test performance of attitude Kalman filter and velocity Kalman filter at low speeds
T2	Drive along a section a main road	Test velocity Kalman filter performance at medium speeds
T3	Drive on an inclined road section	Test system performance on inclined roads

All tests are designed to prove the correctness of the algorithms implemented in the GSMS. However, the tests are not designed to make a quantitative analysis of the accuracy that is achieved using those algorithms. This is due to the lack of an accurate and independent second ground speed measuring device, which is required to perform such tests.

6.2 Test Results

Instead of comparing the measurements made by the GSMS to a secondary and independent measuring device (which would have been more exact), we made the following comparisons in our tests:

- The output of our custom attitude Kalman filter algorithm was compared to the output of the Bosch-Sensortec BSX sensor fusion firmware, which runs on the bno055 IMU.
- The output of our custom velocity Kalman filter algorithm was compared to the neo-m9n GPS velocity measurement (which is only available at a lower frequency than the output of the velocity Kalman filter).

The analysis and discussion of each test is divided into three sections. The first two sections correspond to the two main algorithms of the GSMS and the third section evaluates the final system output:

- Attitude Kalman filter
- Velocity Kalman filter
- Final system output

The test results are presented in the form of plots that visualize different measurements, which were recorded for the duration of the entire test set.

6.2.1 Test T1

Test Description

The first test was carried out in a residential area by driving around two blocks in a figure 8. This test was performed at low speeds of around 30 km/h. The goal of this test was to demonstrate that the algorithms of the GSMS are processing the sensor data correctly. The accuracy is only evaluated in a qualitative manner.

Attitude Kalman Filter Performance

The performance of the attitude Kalman filter is demonstrated very well using this test. Driving a figure 8 involved turns in both directions and it ensured that the vehicle was oriented in a variety of different directions throughout the test.

The attitude is represented as a rotation quaternion which consists of four components. All attitude plots display the w, x, y and z components of the quaternion separately. Due to the trigonometric functions involved in the quaternion representation of a rotation, the meaning of individual components of a quaternion is often difficult to interpret. Certain individual components displayed in the attitude plots manage to show specific properties of the tests, which will be explained for each plot.

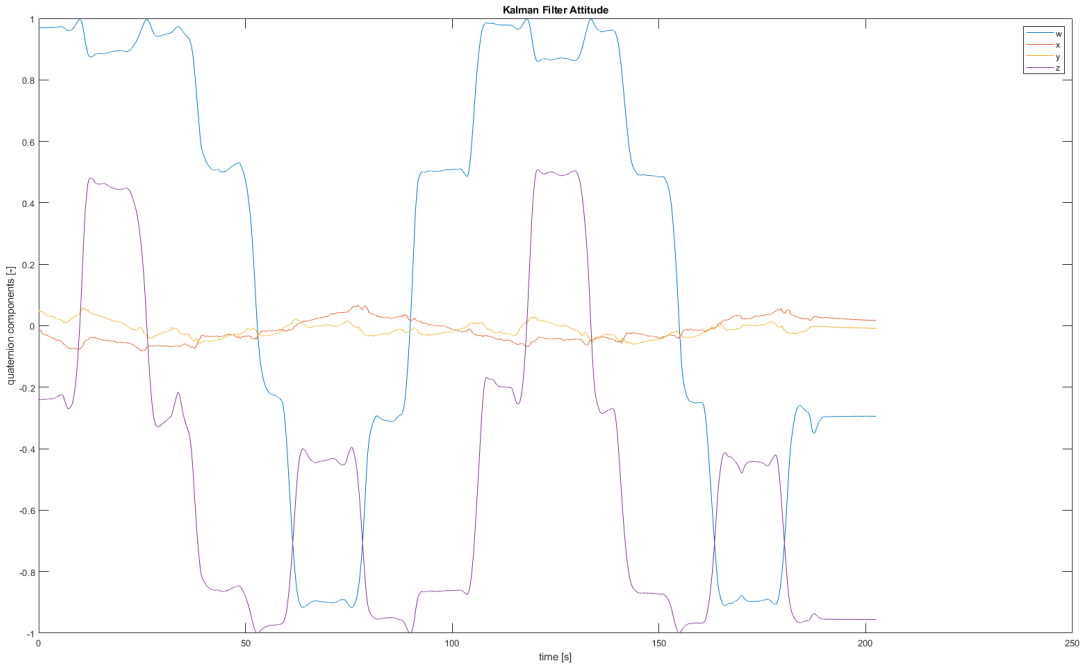


Figure 43: System attitude (as computed by our custom attitude Kalman filter)

Figure 43 clearly demonstrates that our custom attitude Kalman filter is actually calculating a meaningful system attitude. This claim is supported by the following observations:

- The graph shows that the x and y components of the plot remain roughly constant around zero. This is the expected behaviour if the vehicle is only rotated around its z axis.
- The graph also shows that the z component of the quaternion continuously changes its value throughout the test. This is the expected behaviour if the vehicle is driving a figure 8 on a horizontal plane.

The attitude computed by our custom sensor fusion algorithm can now be compared to the attitude, that is computed by the bno055 sensor fusion firmware. The output of the Bosch-Sensortec BSX sensor fusion firmware is displayed in Figure 44.

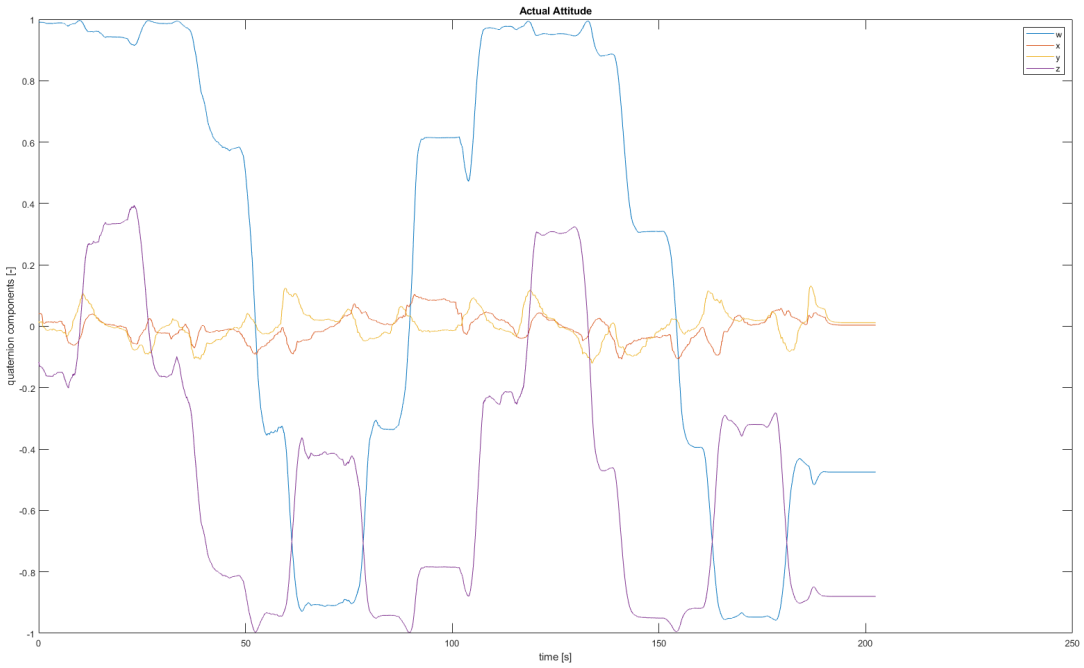


Figure 44: Actual system attitude (as computed by the bno055 firmware)

This graph allows us to prove that the GSMS does not only calculate a meaningful system attitude but that it is also inline with the output of an independent algorithm. The graph of the actual system attitude (calculated by the bno055 firmware) is qualitatively identical to the graph of the Kalman filter attitude (calculated by our custom sensor fusion algorithm). When comparing the two graphs we observe the following:

- Our custom sensor fusion algorithm implemented in the GSMS computes (qualitatively) the same attitude as the bno055 sensor fusion firmware.
- The custom sensor fusion algorithm correctly processes the input of all three sensors (accelerometer, gyroscope and magnetometer).
- The x and y axis of the attitude quaternion measured by our custom sensor fusion algorithm shows higher fluctuation than the output computed by the bno055 sensor fusion firmware. The reason for this is unknown.

Figure 45 shows the angle of the difference rotation between the attitude computed by the GSMS and by the bno055 sensor fusion firmware. This plot effectively visualizes the difference between the two plots shown before.

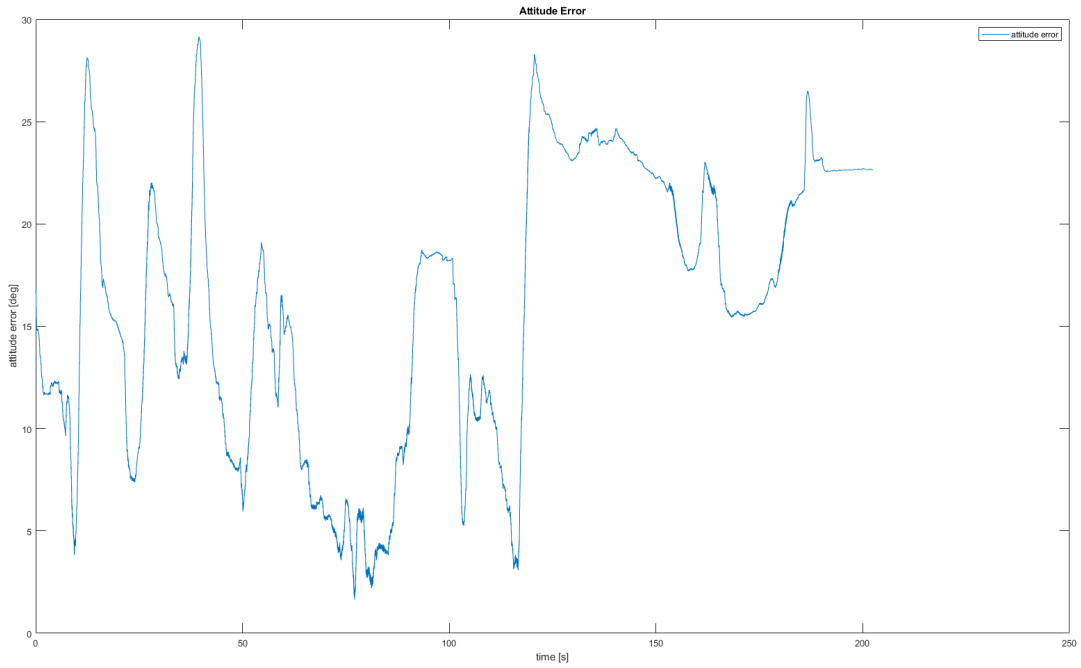


Figure 45: Attitude error

The plot shows that the difference between the two attitude measurements is usually between 5 and 25 degrees. Our experiments showed, that this is mostly caused by magnetic disturbances which influence the measurement made by the magnetometer inside the IMU. Our custom sensor fusion firmware starts to output inaccurate attitude values if the magnetometer is not in an ideal environment. The bno055 sensor fusion firmware on the other hand seems to have a mechanism that detects situations where magnetometer measurements are unreliable. It then most likely stops relying on the magnetometer input until it was able to re-calibrate the magnetometer.

The strong magnetic disturbances were mostly caused by the residential area in which this test was performed. The second test was performed in an area with less buildings which lead to an improved performance of our custom attitude Kalman filter.

Velocity Kalman Filter Performance

This test is also well suited to demonstrate the performance of the velocity Kalman filter. Figure 46 shows the velocity as it was recorded by the GPS. The plot visualizes all three axes which correspond to the North (x), East (y) and Up (z) axes of the inertial frame. It is important to note that the following plot shows the GPS measurement, which is always made in the inertial frame.

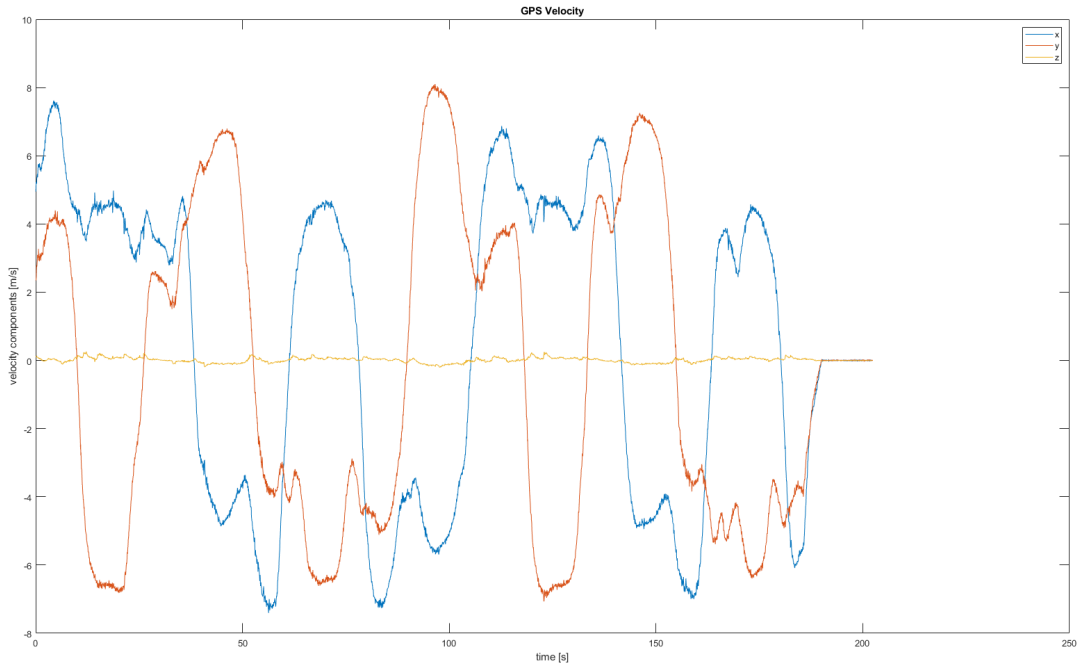


Figure 46: GPS velocity

The plot (Figure 46) shows that the car was moving in different directions on a horizontal plane. Therefore, the GPS measured non-zero values on the x and y axis while the z axis remained constant at zero velocity. This is the expected behaviour.

The next plot in Figure 47 shows how the previously calculated system attitude is used to transform the GPS velocity from the inertial frame into the body frame.

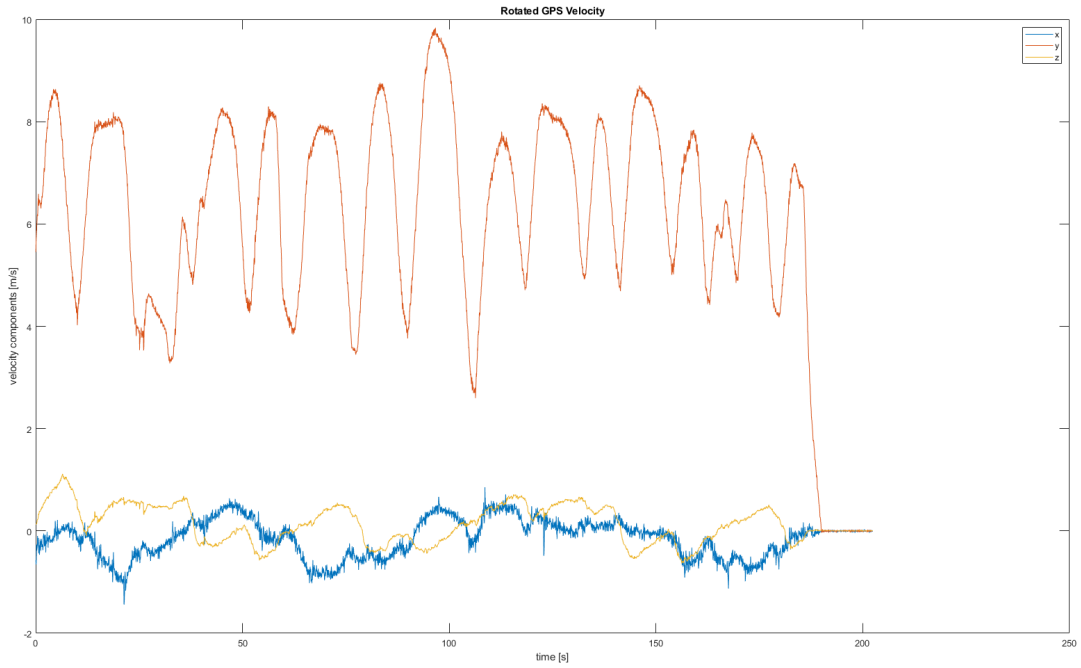


Figure 47: Rotated GPS velocity

This plot displays the same velocity as in Figure 46 but after it was rotated into the body frame by using the system attitude computed by the GSMS. Thus, the plot now shows the three axes x, y and z corresponding to the body frame. We can clearly see, that the plot only shows a significant velocity for the y axis. This is the expected behaviour because, as shown in the Chapter 4.5, the y axis is facing in the direction of travel. This is another proof that the attitude Kalman filter is actually computing the correct attitude. Because the transformation from Figure 46 to Figure 47 is solely based on this attitude. An incorrect attitude would have caused high non-zero values on the x and z components in figure 47.

Figure 48 displays the linear acceleration calculated by the GSMS, after the gravity vector has been subtracted. Subtracting the gravity vector again involves the previously calculated system attitude.

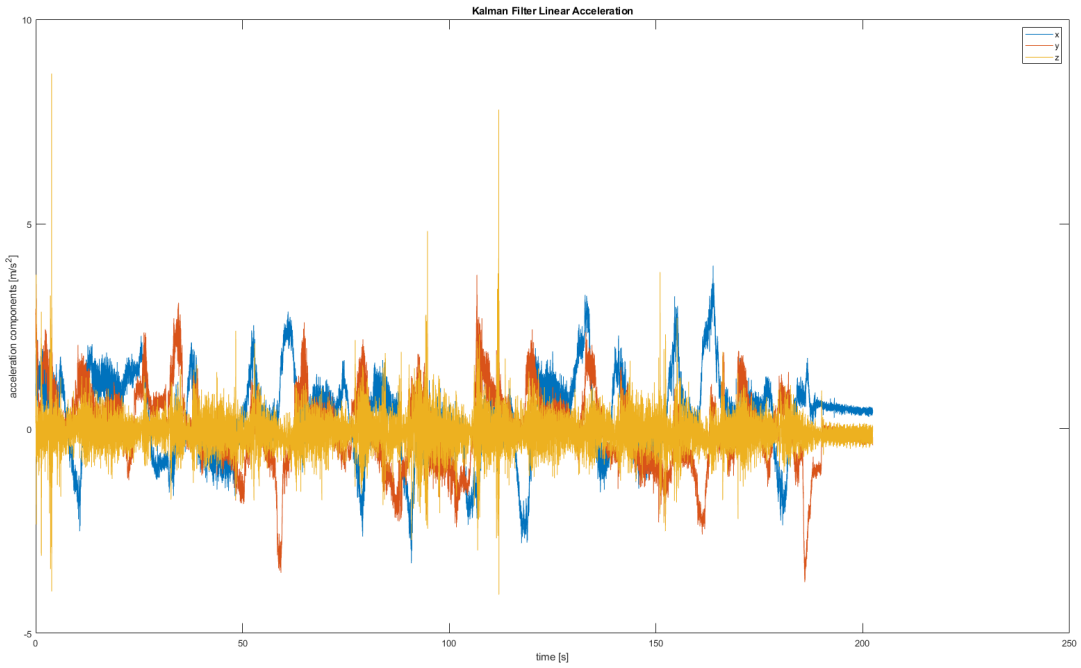


Figure 48: Kalman filter linear acceleration

Figure 49 displays the linear acceleration as it was calculated by the bno055 sensor fusion firmware.

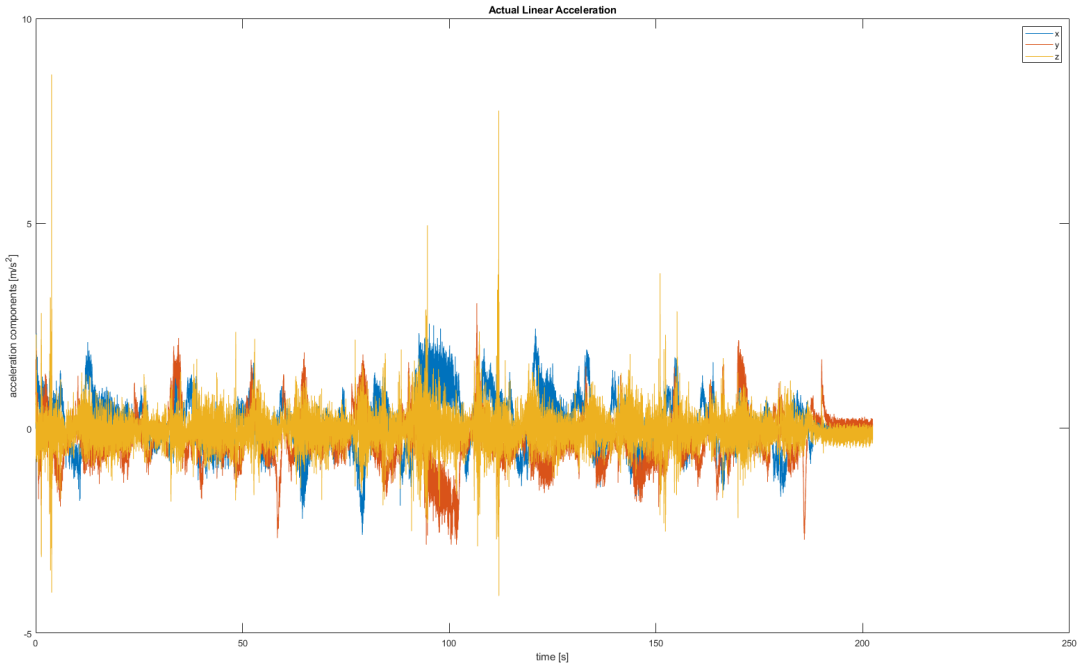


Figure 49: Actual linear acceleration

These two plots show a similar linear acceleration. Especially high peaks of acceleration are found in both plots. The plots do however not meet the expected behaviour. The expected behaviour would be that both plots continuously show the same values on all three axes. The reason for this unknown.

Final system output

This part of the test compares the final output of our custom velocity Kalman filter implemented in the GSMS with the rotated velocity measurement that was recorded by the GPS.

Figure 50 visualizes the error between the velocity Kalman filter output and the rotated GPS velocity. The plot shows two different properties:

- The angle between the two velocity vectors in degrees (velocity angle error)
- The difference in length between the two velocity vectors in m/s (velocity norm error)

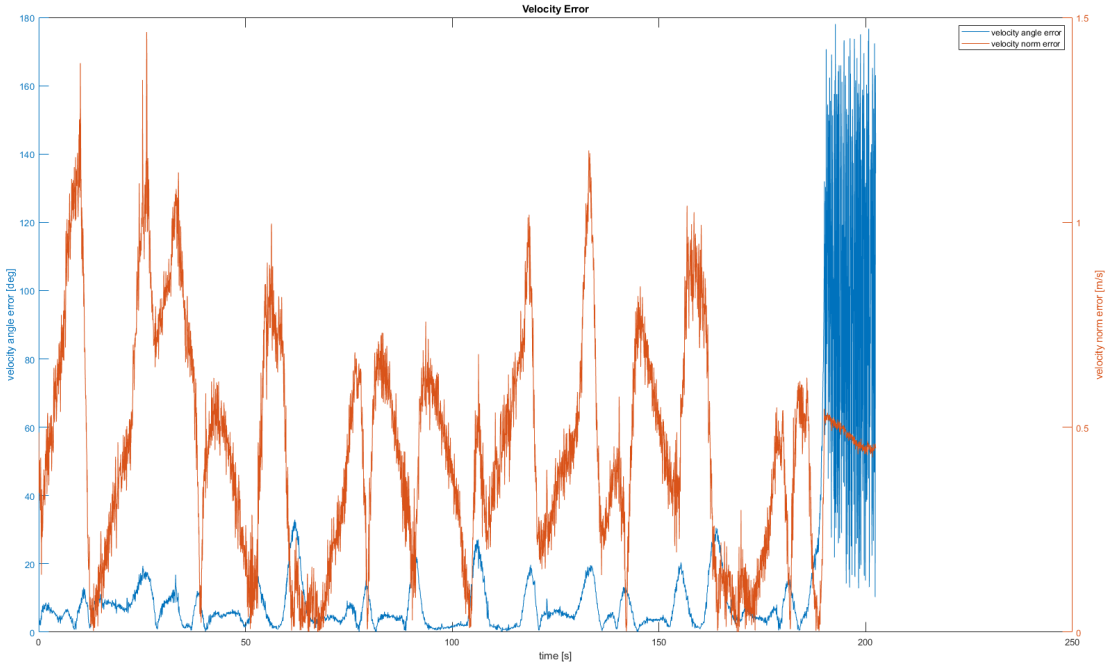


Figure 50: Velocity error

The graph can be interpreted as follows:

- The velocity angle error is usually below 20 degrees and often below 10 degrees. This is the expected behaviour. It proves that the direction of the velocity measured by the GSMS is correct.
- The velocity norm error is usually below 1 m/s. This is the expected behaviour. It proves that the absolute value of the velocity measured by the GSMS is correct.
- The velocity angle error increases to values of up to 180 degrees at the very end of the test. This is the expected behaviour for situations where the actual velocity is zero. At the very end of the test, the vehicle was not moving anymore, as it came to a halt, and therefore the GSMS as well as the GPS are measuring velocity vectors which are very close to the zero

vector. It is expected that the angle between those two vectors can assume any angle in the range of 0 to 180 degrees, as seen in the graph.

Figure 51 shows the final output of the GSMS. This is the signal which will be transmitted to the control unit of the self driving car. It shows the three components of the vehicle ground speed in the body frame. It clearly shows, that the main part of the velocity is measured on the y axis which is facing in the direction of travel. This is the expected behaviour.

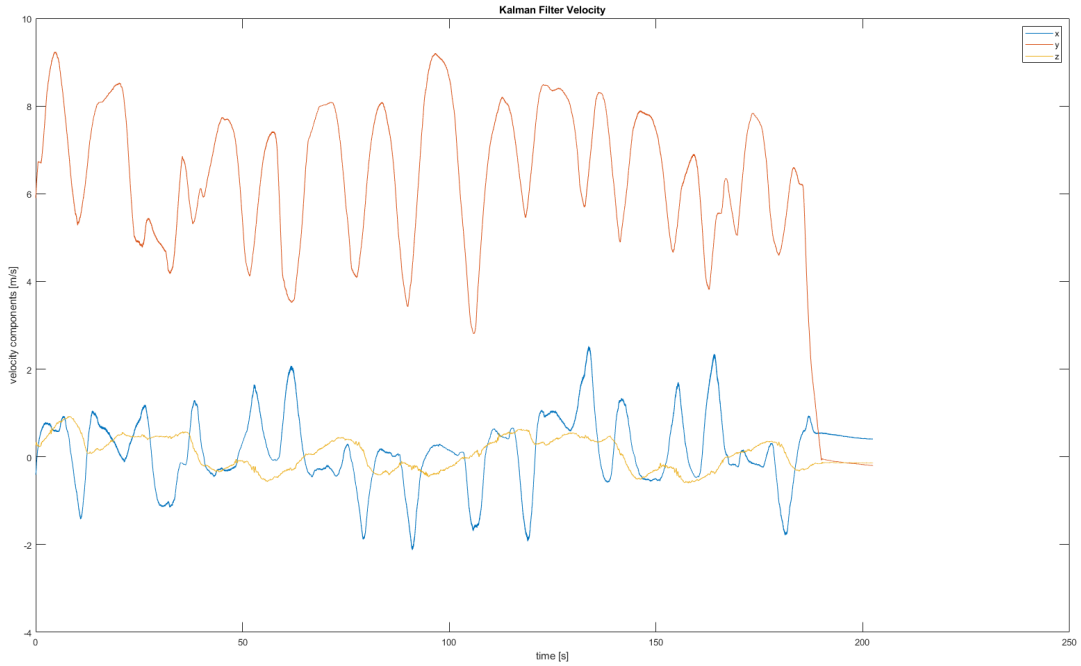


Figure 51: Kalman filter velocity output

6.2.2 Test T2

Test Description

The second test was carried out while driving along a main road. This test was performed at medium speeds of around 50 km/h. The goal of this test was to demonstrate that the algorithms of the GSMS are processing the sensor data correctly. The accuracy is only evaluated in a qualitative manner.

Attitude Kalman Filter Performance

This test shows, that the performance of the attitude Kalman filter is significantly improved, when less magnetic disturbance is present. Figure 52 shows the system attitude as it was computed by our custom attitude Kalman filter.

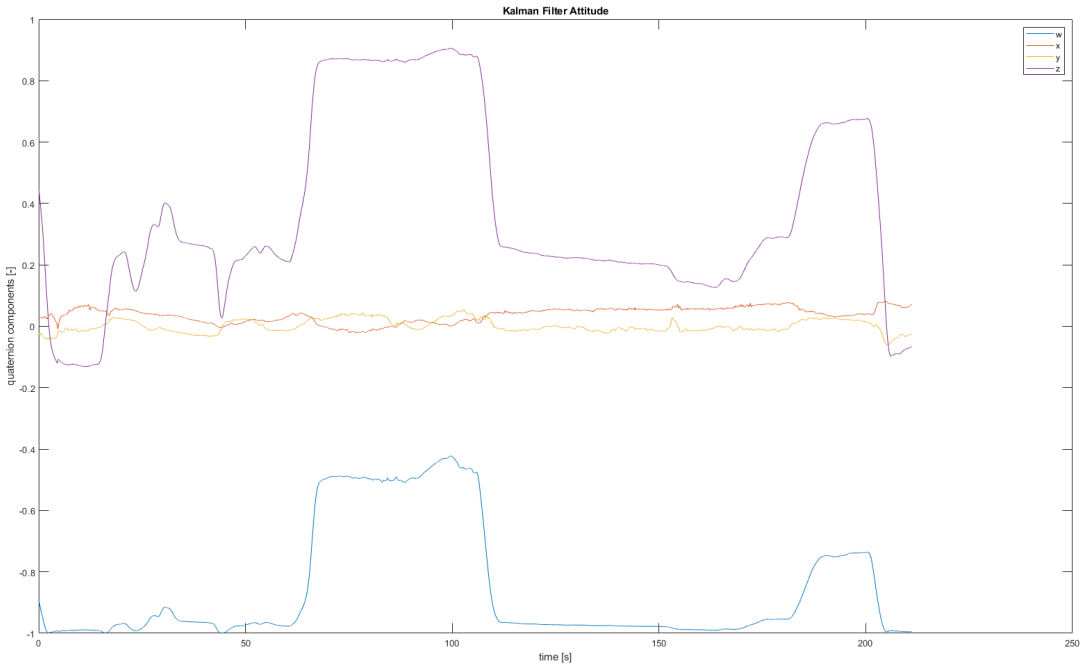


Figure 52: System attitude (as computed by our custom attitude Kalman filter)

The graph clearly demonstrates that our custom attitude Kalman filter is actually calculating a meaningful system attitude. This claim is supported by the following observations:

- The graph shows that the x and y components of the plot remain roughly constant around zero. This is the expected behaviour if the vehicle is only rotated around its z axis.
- The graph also shows that the z component of the quaternion changed its value whenever the vehicle took a turn. The z component remained constant when the vehicle was not changing its orientation. This is the expected behaviour.

The attitude computed by our custom sensor fusion algorithm can now be compared to the attitude computed by the bno055 sensor fusion firmware. The output of the Bosch-Sensortec BSX sensor fusion firmware is displayed in Figure 53.

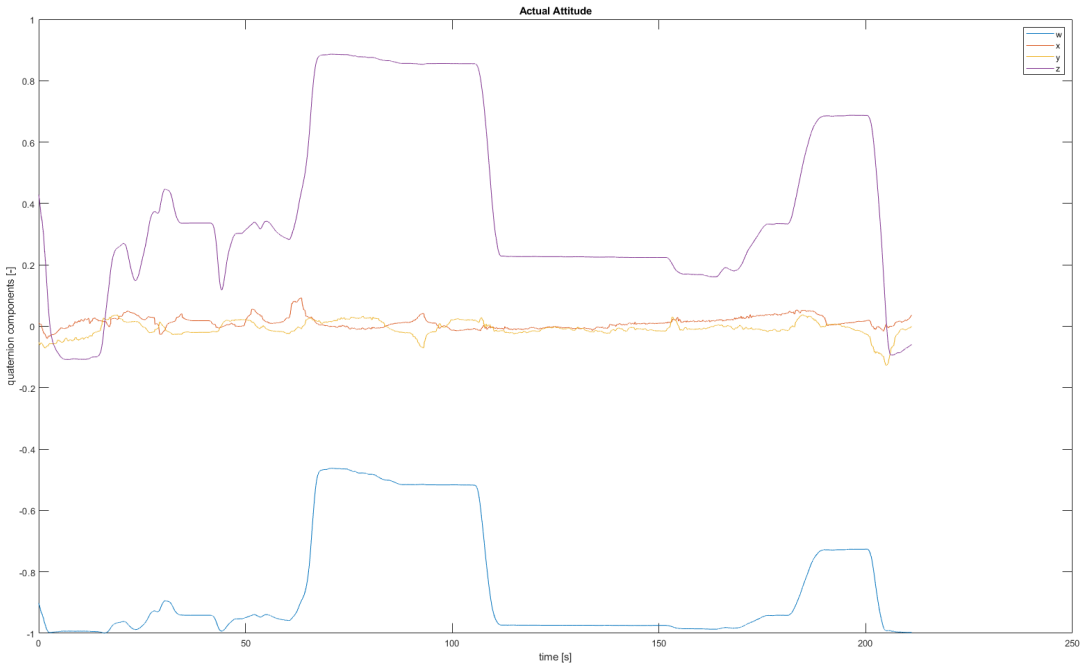


Figure 53: Actual system attitude (as computed by the bno055 firmware)

The above graph allows us to prove that the GSMS not only calculates a meaningful system attitude but that it also inlines with the output of an independent algorithm. The graph of the actual system attitude (calculated by the bno055 firmware) is qualitatively identical to the graph of the Kalman filter attitude (calculated by our custom sensor fusion algorithm). When comparing the two graphs we observe the following:

- Our custom sensor fusion algorithm implemented in the GSMS computes (qualitatively) the same attitude as the bno055 sensor fusion firmware.
- The custom sensor fusion algorithm correctly processes the input of all three sensors (accelerometer, gyroscope and magnetometer).
- The x and y axis of the attitude quaternion measured by our custom sensor fusion algorithm shows higher drift than the output computed by the bno055 sensor fusion firmware. The reason for this is again unknown.

Figure 54 shows the angle of the difference rotation between the attitude computed by the GSMS and by the bno055 sensor fusion firmware. This plot effectively visualizes the difference between the two plots shown above.

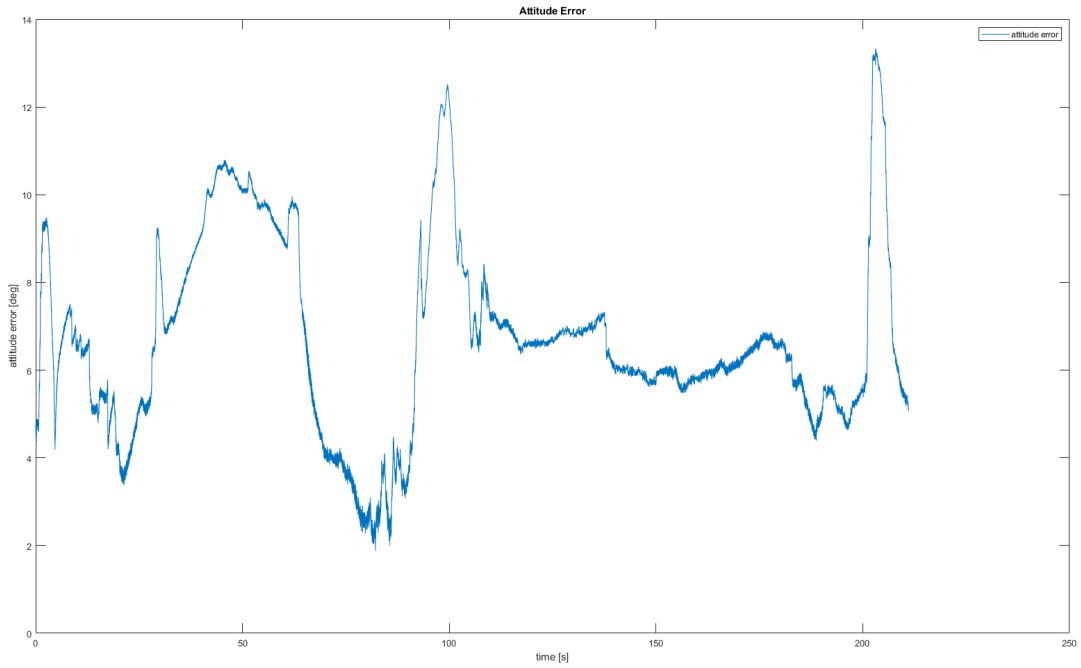


Figure 54: Attitude error

The plot shows that the difference between the two attitude measurements is usually between 2 and 12 degrees. This test showed, that the accuracy of the attitude Kalman filter implemented in the GSMS can be significantly improved in areas with less magnetic disturbance. Comparing this plot to Figure 45 from the first test shows that the accuracy was highly improved in the second test. This test was performed in an area with less buildings which lead to the improved performance of our custom attitude Kalman filter.

Velocity Kalman Filter Performance

This test is used to demonstrate the performance of the velocity Kalman filter at speeds of around 50 km/h. Figure 55 shows the velocity as it was recorded by the GPS. The plot visualizes all three axes which correspond to the North (x), East (y) and Up (z) axes of the inertial frame. It is importatnt to note that the following plot shows the GPS measurement, which is always made in the inertial frame.

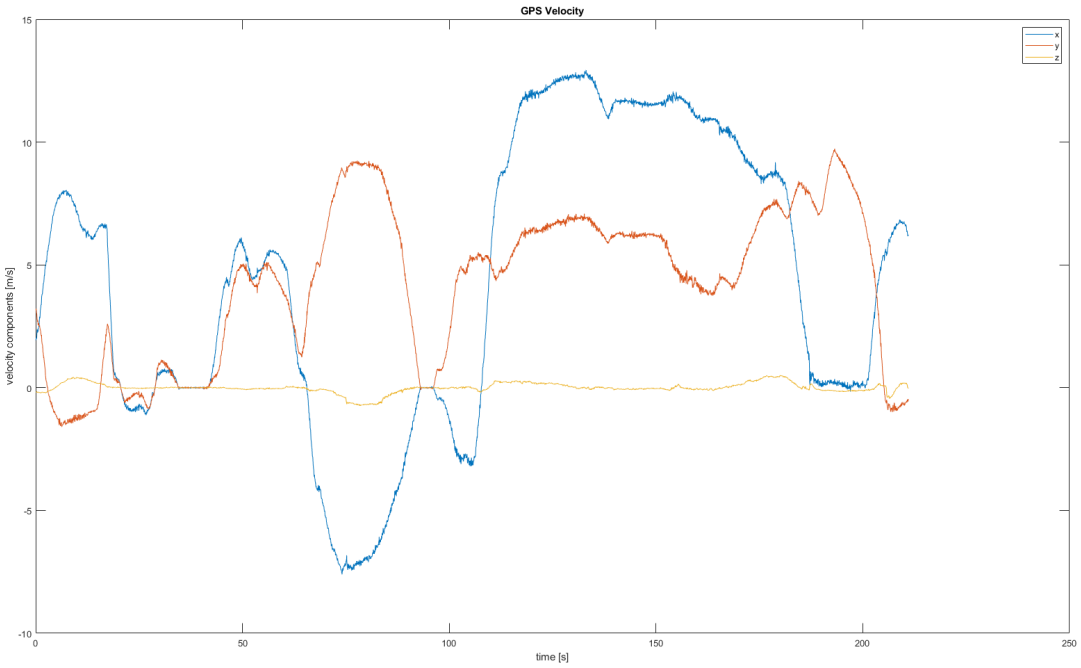


Figure 55: GPS velocity

The plot shows that the car was moving in different directions on a mostly horizontal plane. Therefore, the GPS measured non-zero values on the x and y axis while the z axis remained almost constant at zero velocity. Only when driving on a part of the road that was inclined we can see a non-zero measurement on the z axis. This is the expected behaviour.

Figure 56 shows how the previously calculated system attitude is used to transform the GPS velocity from the inertial frame into the body frame.

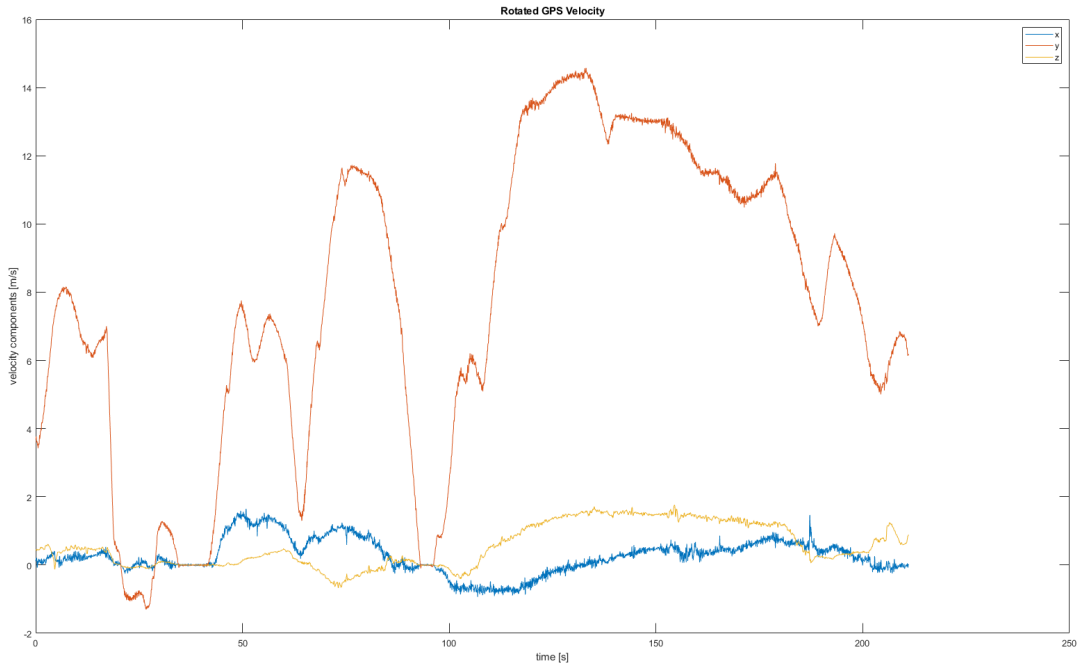


Figure 56: Rotated GPS velocity

This plot displays the same velocity as in figure 55 but after it was rotated into the body frame by using the system attitude computed by the GSMS. The plot therefore now shows the three axes x, y and z corresponding to the body frame. We can clearly see, that the above plot only shows a significant velocity for the y axis. This is the expected behaviour because, as shown in the Chapter 4.5, the y axis is facing in the direction of travel. This is another proof that the attitude Kalman filter is actually computing the correct attitude. Because the transformation from Figure 55 to Figure 56 is solely based on this attitude. An incorrect attitude would have caused high non-zero values on the x and z components in figure 56.

Figure 57 displays the linear acceleration calculated by the GSMS, after the gravity vector has been subtracted. Subtracting the gravity vector again involves the previously calculated system attitude.

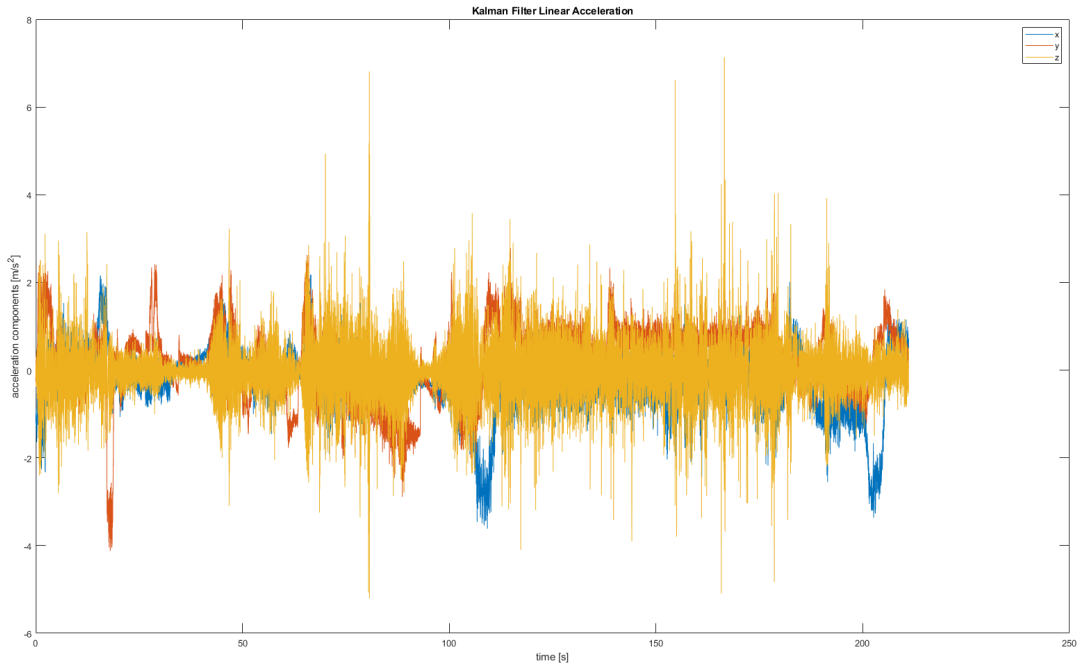


Figure 57: Kalman filter linear acceleration

Figure 58 displays the linear acceleration as it was calculated by the bno055 sensor fusion firmware.

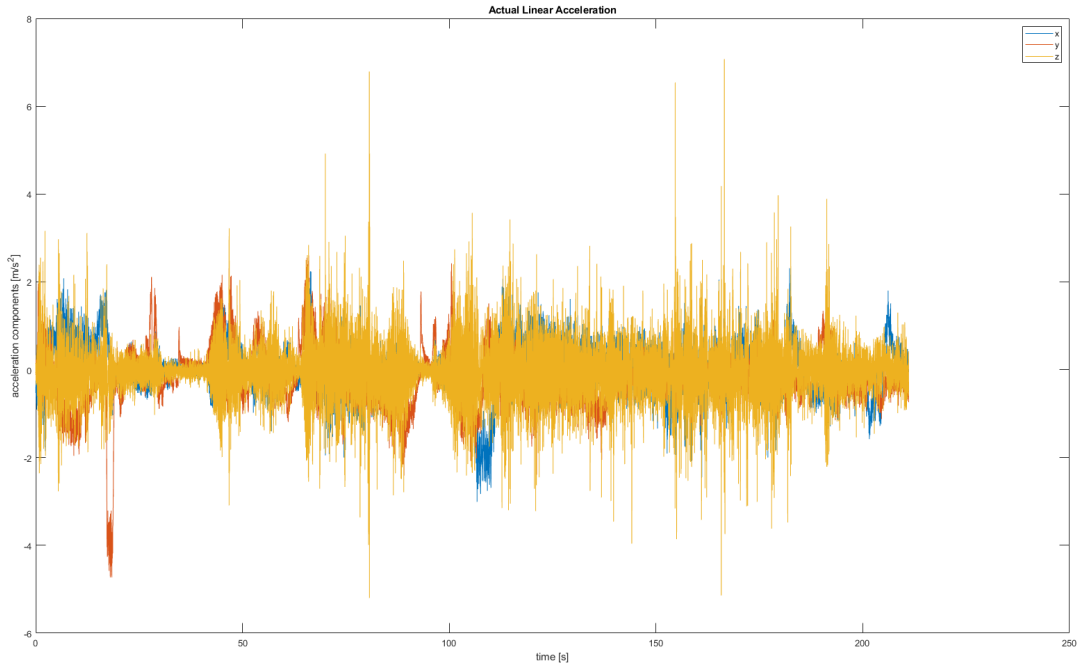


Figure 58: Actual linear acceleration

These two plots show a similar linear acceleration. Especially high peaks of acceleration are found in both plots. The plots do however not meet the expected behaviour. The expected behaviour would be that both plots continuously show the same values on all three axes. The reason for this unknown.

Final system output

This test compares the final output of our custom velocity Kalman filter implemented in the GSMS with the rotated velocity measurement that was recorded by the GPS. Figure 59 visualizes the error between the velocity Kalman filter output and the rotated GPS velocity. The plot shows two different properties:

- The angle between the two velocity vectors in degrees (velocity angle error)
- The difference in length between the two velocity vectors in m/s (velocity norm error)

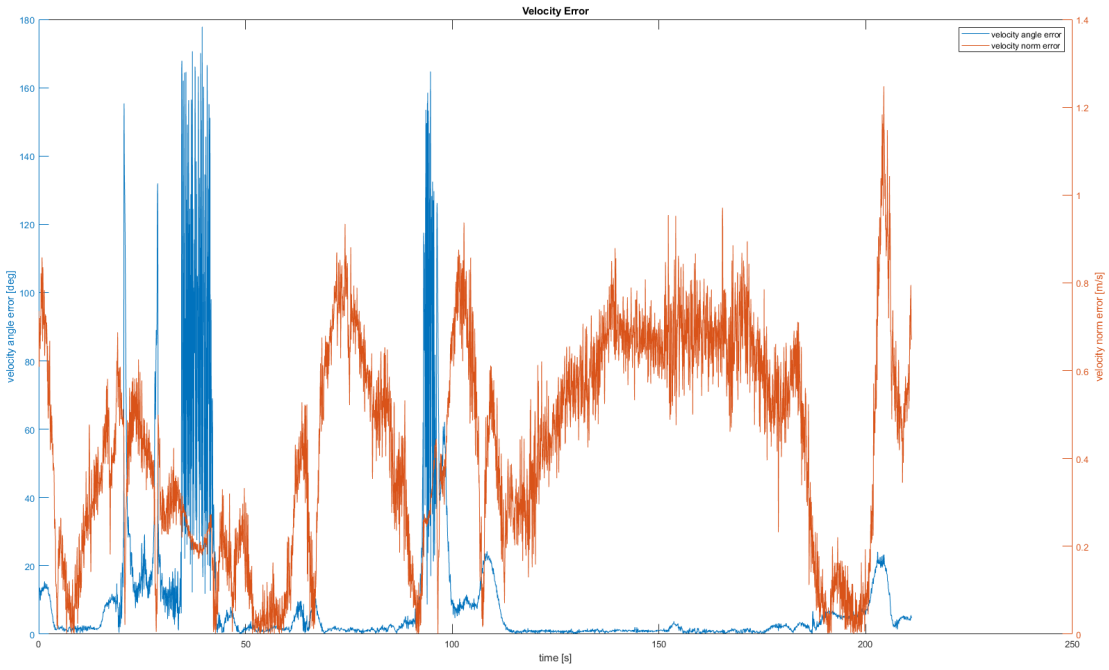


Figure 59: Velocity error

Figure 59 can be interpreted as follows:

- The velocity angle error is usually below 20 degrees and often below 5 degrees. This is the expected behaviour. It proves that the direction of the velocity measured by the GSMS is correct.
- The velocity norm error is usually below 1 m/s. This is the expected behaviour. It proves that the absolute value of the velocity measured by the GSMS is correct.
- The velocity angle error increases to values of up to 180 degrees in situation where the vehicle was not moving. This is the expected behaviour for situations where the actual velocity is zero. If the vehicle is not moving, the GSMS as well as the GPS are measuring velocity vectors which are very close to the zero vector. It is expected that the angle between those two vectors can then assume any angle in the range of 0 to 180 degrees, as

seen in the above plot for t ≈ 35s and t ≈ 90s.

Figure 60 shows the final output of the GSMS. This is the signal which will be transmitted to the control unit of the self driving car. It shows the three components of the vehicle ground speed in the body frame.

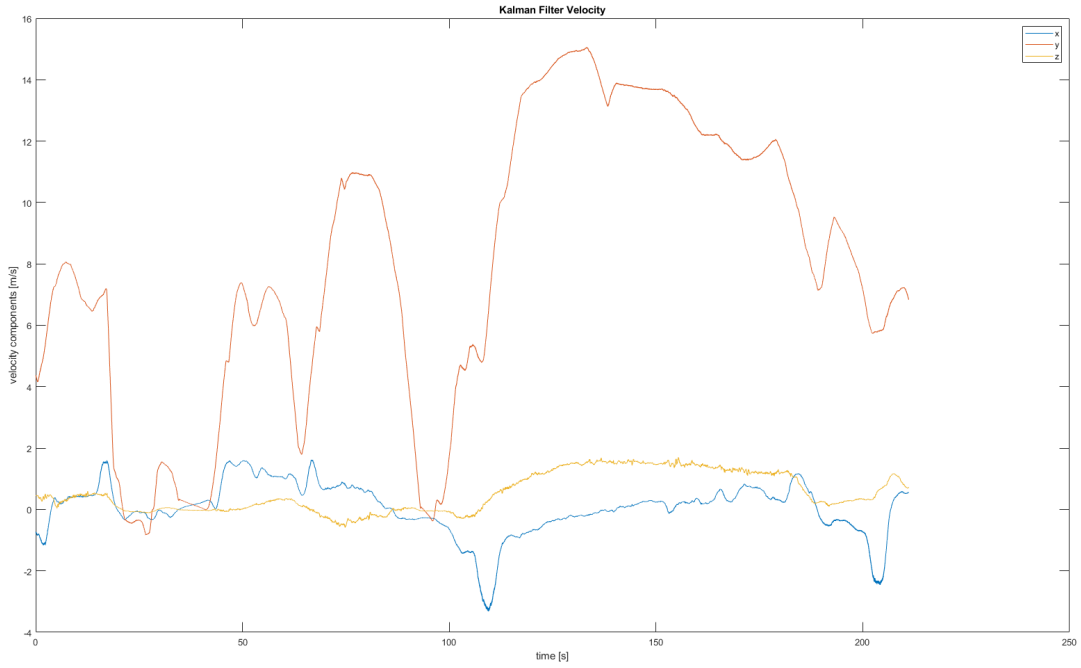


Figure 60: Kalman filter velocity output

6.2.3 Test T3

Test Description

This last test is intended to demonstrate the importance of measuring the velocity in three dimensions as opposed to simply measuring a two dimensional course over ground. This test was therefore performed while driving up a hill. When driving on an inclined road, there is a significant part of the velocity that is measured on the vertical z (up or down) axis. This part of the velocity must also be considered when trying to determine the actual ground speed. For this test we will only discuss the performance of the velocity Kalman filter.

Velocity Kalman Filter Performance

Figure 61 shows the velocity as it was measured by the GPS in the inertial frame. It is clearly visible that the down (z) component continuously assumes values below zero while driving up the hill. This is intuitive as the down (z) axis of the inertial frame points down towards the center of the earth, which leads to negative values when driving upwards.

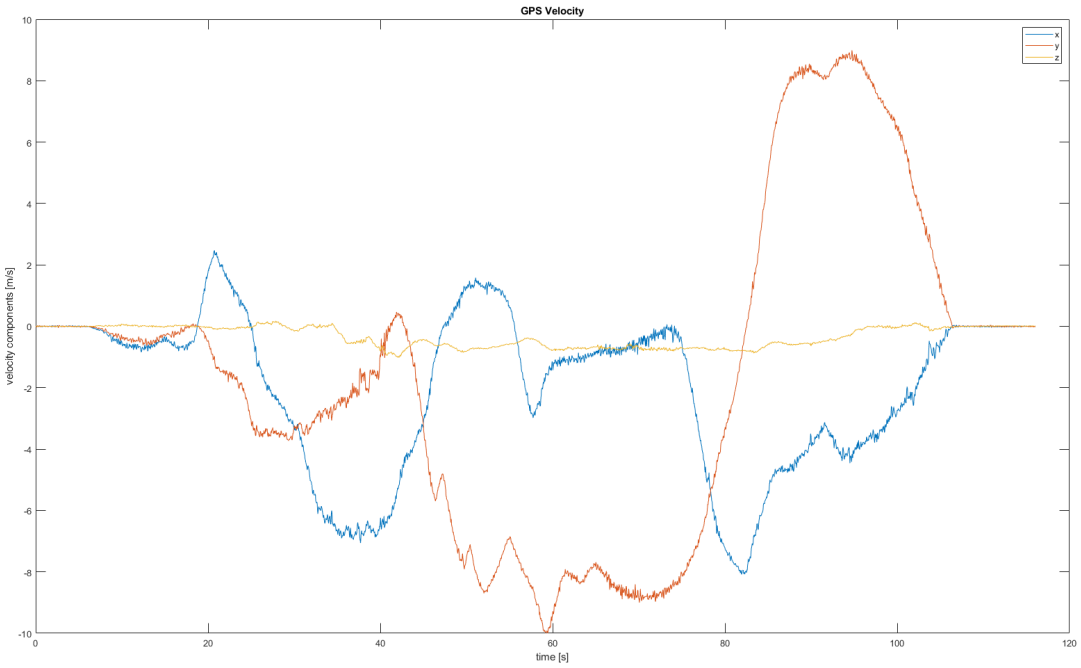


Figure 61: GPS velocity

The velocity measured on the z axis, as shown above, is an important component of the vehicle ground speed. Ignoring this axis would result in an inaccurate ground speed measurement.

Figure 62 shows the above GPS velocity after it has been rotated into the body frame. This is done by taking the computed system attitude into account. The plot shows, that after the GPS velocity is rotated into the body frame, the values measured on the z axis become part of the y axis. This is intuitive, as the vehicle is only moving in the direction of its y axis, even if it is driving up a hill.

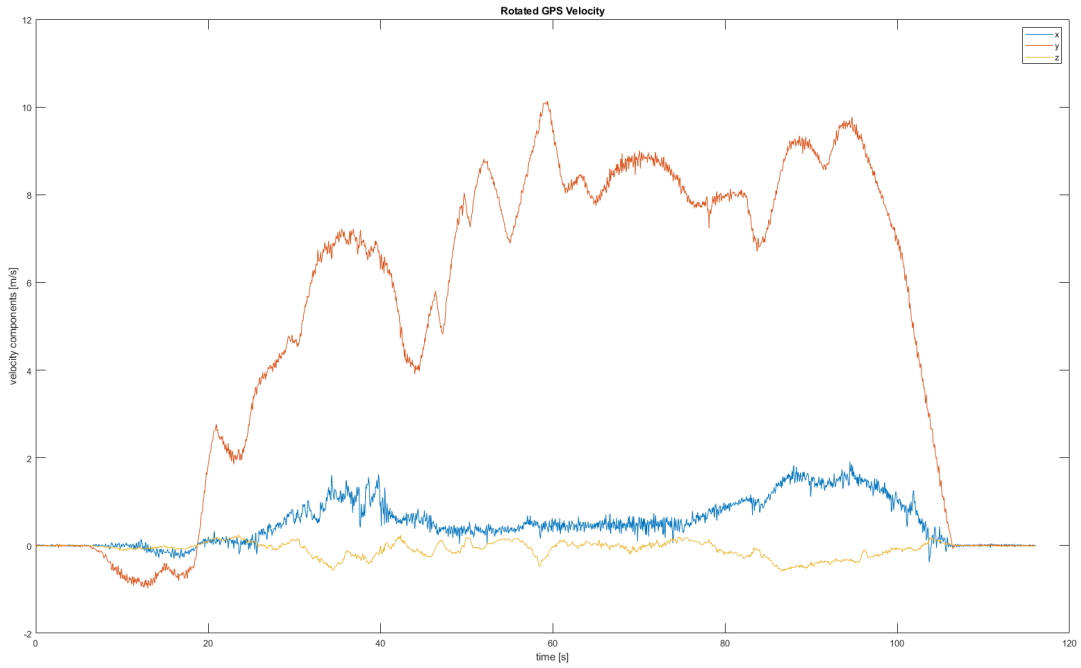


Figure 62: Rotated GPS velocity

This test proves the following:

- It is important to determine a complete attitude of the vehicle in three dimensional space. Only determining the heading (compass direction) is not sufficient.
- It is important to measure the velocity in three dimensions. Only determining a two dimensional course over ground is not sufficient.

7 Conclusion

One of the main aspects of motivation for developing a ground speed measuring system was to take part in the development of an autonomous race car. The idea was to have a reliable and redundant measuring system support the driverless vehicle in various aspects with this sensor fusion output, in order to be eligible for competing with the current world best teams. During our research we came to the conclusion that nearly all driverless teams have a similar sensor fusion set up to calculate the vehicle's velocity. The differences mostly only vary in the chosen GPS modules or algorithm implementations. Therefore our self-developed system closely follows the other teams' ideas and designs albeit with some deviations. The advantage of the other teams currently lies in owning the optical ground speed sensor and being able to add that sensor to their sensor fusions. Even though we are a little bit disappointed that we weren't able to implement that sensor as well, we are nonetheless positively surprised and happy about the outcome of our algorithmic implementation.

During the course of this project, we faced multiple difficulties related to the use of the HAL libraries. We were always able to eventually resolve the issues, but it was usually quit difficult to debug functions related to the HAL, as it takes a considerable amount of time to understand the inner workings of the HAL functions.

Our ground speed measurement system proved its functionality during testing. The tests performed on the GSMS clearly showed that our custom sensor fusion algorithms are mathematically correct. They produced a qualitatively acceptable and usable output in all tests. However, our sensor fusion algorithms start to produce inaccurate outputs if the environment does not provide ideal conditions. This could be improved by adding more features to the currently implemented algorithms of the GSMS such as:

- detection of incorrect sensor measurements (especially for the magnetometer)
- additional filtering of sensor measurements prior to executing the Kalman filters (especially a high pass filter for the gyroscope and a low pass filter for the accelerometer when determining the gravity vector)

The custom sensor fusion algorithms could also be further improved by optimizing the parameters that are currently used in the attitude and velocity Kalman filters. This would however require further tests and experiments with a secondary measurement device as an accurate reference.

7.1 Suitability of the GSMS for Autonomous Driving

The autonomous race car is currently still built and it is not yet clear where exactly the final GSMS will be installed. Therefore, this project focused on building a prototype of the GSMS. This prototype can be used to perform experiments and to optimize the algorithms as well as the hardware of the measuring system. In a future step, which is outside of the scope of this work, the optimized GSMS can then be adjusted to fit into a particular location of the race car. The GSMS can successfully determine the vehicle ground speed. It is therefore suitable as an additional input to an autonomous driving system. This input can be used for navigational and path planning tasks. The accuracy of the GSMS is however not yet sufficient to use its measurements for traction control and to prevent wheel slip.

7.2 Outlook

Following the end of this project, this ground speed measurement system shall be implemented in the Formula Student ZHAW race car for the competitions in the summer of 2021. The system must undergo extensive testing before use. At the time of writing, the FSZHAW driverless team is still in talks about obtaining an additional optical ground speed sensor to add to the vehicle's

sensor setup. This would mean that the system's reliability would be greater and the measures more accurate. The system was built to keep interfaces available for additional sensors that can be added at any time, so the team can easily implement this step in the future.

An interesting yet different approach to implement the sensor fusion, other than with the Kalman Filter, would be to make use of neural networks, which tend to treat every signal as its own entity and then integrate the resulting representations into a new neural network for a high-level fusion. For future driverless cars the Formula Student ZHAW team should consider this approach and try implementing this with the knowledge and experience they gather from the first car, for optimal success.

Another project that is being planned at the time of writing is the development of a simulation tool. This tool should offer an optimal testing environment, where single sensors and sensor fusions should be extensively tested on their accuracy. A simulation environment could test our system in different scenarios that are similar to the competition surroundings and thus predict the system's stability. However, real-world testing in the autonomous race car should not be underestimated or neglected.

At the time of implementation, the GSMS does not make use of the absolute position determined by the GPS. This should be implemented in future projects to determine the driverless vehicles' position as additional input to the control system.

Glossary

Attitude Kalman Filter:	The term Attitude Kalman Filter refers to a part of the custom algorithm that is implemented in the ground speed measuring system. The attitude Kalman filter processes the sensor data from the IMU to compute the system attitude.
Driverless:	Driverless vehicles are cars or trucks in which human drivers are never required to take control to safely operate the vehicle. Also known as autonomous or self-driving cars, they combine sensors and software to control, navigate, and drive the vehicle.
Ground Speed Measuring System:	The term Ground Speed Measuring System refers to the entire hardware and software that makes up the device to measure the vehicle ground speed.
Linear Acceleration:	A physical body might be exposed to acceleration caused by the gravitational field of the earth or by additional external forces applied to the body. The term linear acceleration refers to the part of the acceleration that is caused by additional external forces. Therefore, linear acceleration is the part of acceleration, that actually results in a change of velocity of the observed body. To obtain the linear acceleration from measurements made by an accelerometer, the acceleration caused by the gravitational field of the earth must be removed first.
Rotation Quaternion:	A (rotation) quaternion can be used to describe rotations in three dimensional space. A rotation quaternion that describes a valid rotation is always a unit quaternion. The first component of the rotation quaternion describes the angle of rotation and the remaining three components describe the axis of rotation.
Sensor Fusion:	Sensor fusion refers to the process of combining measurements obtained through different sensors using data processing algorithms with the goal of computing a specific quantity.
Velocity Kalman Filter:	The term Velocity Kalman Filter refers to a part of the custom algorithm that is implemented in the Ground Speed Measuring System. The velocity Kalman filter processes the sensor data from the IMU and the GPS to compute the system velocity.

Bibliography

References

[1] Mercedes. *INSIGHT: Five Examples Why F1 Is Accelerating the Future*. 2018. URL: https://www.mercedesamgf1.com/en/news/2018/10/insight-five-examples-why-f1-is-accelerating-the-future/ (visited on 11/26/2020).

[2] *Self-Driving Cars Explained*. Jan. 2017. URL: https://www.ucsusa.org/resources/self-driving-cars-101.

[3] Marcelo H. Ang Jr. et al. *Perception, Planning, Control, and Coordination for Autonomous Vehicles*. Feb. 2017. URL: https://www.mdpi.com/2075-1702/5/1/6.

[4] Juraj Kabzan et al. *AMZ Driverless: The Full Autonomous Racing System*. 2019.

[5] National Oceanic US Department of Commerce and Atmospheric Administration. *What is LIDAR*. Oct. 2012. URL: https://oceanservice.noaa.gov/facts/lidar.html#:~:text=Lidar,%20which%20stands%20for%20Light,variable%20distances)%20to%20the%20Earth..

[6] Miguel I. Valls et al. "Design of an Autonomous Racecar: Perception, State Estimation and System Integration". In: *2018 IEEE International Conference on Robotics and Automation (ICRA)* (2018). DOI: 10.1109/icra.2018.8462829.

[7] Jan Mochnac, Stanislav Marchevsky, and Pavol Kocan. "Bayesian filtering techniques: Kalman and extended Kalman filter basics". In: *2009 19th International Conference Radioelektronika* (2009). DOI: 10.1109/radioelek.2009.5158765.

[8] N. Gosala et al. "Redundant Perception and State Estimation for Reliable Autonomous Racing". In: *2019 International Conference on Robotics and Automation (ICRA)*. 2019, pp. 6561–6567. DOI: 10.1109/ICRA.2019.8794155.

[9] Sirish Srinivasan et al. "End-to-End Velocity Estimation for Autonomous Racing". In: *IEEE Robotics and Automation Letters* 5.4 (2020), pp. 6869–6875. DOI: 10.1109/lra.2020.3016929.

[10] Marcel Zeilinger et al. *Design of an Autonomous Race Car for the Formula Student Driverless (FSD)*. May 2017.

[11] Hanqing Tian, Jun Ni, and Jibin Hu. "Autonomous Driving System Design for Formula Student Driverless Racecar". In: *2018 IEEE Intelligent Vehicles Symposium (IV)* (2018). DOI: 10.1109/ivs.2018.8500471.

[12] Damien Lhomme-Desages et al. "Doppler-Based Ground Speed Sensor Fusion and Slip Control for a Wheeled Rover". In: *IEEE/ASME Transactions on Mechatronics* 14 (4) (2009), pp. 484–492.

[13] Formula Student Germany. *Formula Student Rules 2020*. Sept. 2019.

[14] Greg Welch and Gary Bishop. *An Introduction to the Kalman Filter*. 2001.

[15] MathWorks. *Orientation, Position, and Coordinate*. URL: https://www.mathworks.com/help/fusion/gs/spatial-representation-coordinate-systems-and-conventions.html (visited on 12/09/2020).

[16] Wikipedia. *Magnetic Declination*. URL: https://en.wikipedia.org/wiki/Magnetic_declination (visited on 12/09/2020).

[17] Mikro-E. *GNSS 7 CLICK*. 2020. URL: https://www.mikroe.com/gnss-7-click (visited on 12/17/2020).

[18] ST Microelectronics. *STM32 Nucleo-144 development board with STM32F429ZI MCU*. 2020. URL: https://www.st.com/en/evaluation-tools/nucleo-f429zi.html (visited on 12/17/2020).

[19] arm KEIL. *Create Projects with STM32Cube HAL and STM32CubeMX*. URL: https://www.keil.com/pack/doc/stm32cube/html/index.html (visited on 11/12/2020).

[20] *Help: Magnetic Field Calculator*. URL: https://www.ngdc.noaa.gov/geomag/calculators/help/igrfwmmHelp.html#interpretingresults (visited on 12/10/2020).

[21] National Centers for Environmental Information (NCEI). *Geomagnetic Field Calculator*. URL: https://www.ngdc.noaa.gov/geomag/calculators/magcalc.shtml#igrfwmm (visited on 12/10/2020).

[22] F. Landis Markley. "Attitude Error Representations for Kalman Filtering". In: *Journal of Guidance, Control, and Dynamics* 26.2 (2003), pp. 311–317. DOI: 10.2514/2.5048.

List of Figures

1 AMZ's software-hardware architecture of the autonomous system [4] 11
2 AMZ's velocity estimation architecture design [4] 12
3 AMZ's autonomous race car *Pilatus*' sensor setup [9] 13
4 Envelope to mount sensor systems on a formula student race car [13] 15
5 Initial hardware-software architecture of the GSMS 19
6 Sensor fusion concept 22
7 Kalman filter equations and execution flow [14] 25
8 Orientation of the body frame 27
9 Orientation of the axes from the individual sensors inside the bno055 27
10 NED and ENU inertial reference frames [15] 28
11 Visual representation of the magnetic declination [16] 29
12 Actual hardware-software architecture of the GSMS 30
13 Sensor fusion implementation 30
14 bno055 shuttle board (IMU) 31
15 neo-m9n develoment board (GPS) [17] 32
16 STM32F429ZI on the Nucleo-144 development board [18] 33
17 Initial prototype setup for the Ground Speed Measuring System 34
18 Final prototype of the Ground Speed Measuring System 34
19 Schematic of the GSMS 35
20 Tools used to develop the GSMS 36
21 Components of the software for the GSMS 37
22 Sequence diagram of initialization routine 38
23 CubeMX pin configuration interface 39
24 CubeMX clock configuration interface 40
25 CubeMX integration with Keil μVision IDE 40
26 Direction of the gravity vector in the inertial ENU reference frame 51
27 direction of the north vector in the inertial ENU reference frame 52
28 Geographical location with corresponding constants used in the GSMS 53
29 Output of the magnetic field calculator for Winterthur 53
30 Components measured by the accelerometer (gravity vector, linear acceleration) . 54
31 Variable watch window of the Keil μVision IDE when debugging the GSMS 56
32 Serial monitor interface displaying data from the GSMS 57
33 Output of the script used to preprocess the captured data from the GSMS 58
34 Files generated after preprocessing the captured data from the GSMS 58
35 Kalman filters and their input values 59
36 Attitude Kalman filter functions 60
37 Additional functions 60
38 Alignment of body frame and inertial frame 60
39 Rotation q_ref between body frame and inertial frame 61
40 Velocity Kalman filter functions 64
41 Additional functions 64
42 Detailed test concept illustration 67
43 System attitude (as computed by our custom attitude Kalman filter) 69

44 Actual system attitude (as computed by the bno055 firmware) . . . 70
45 Attitude error . . . 71
46 GPS velocity . . . 72
47 Rotated GPS velocity . . . 73
48 Kalman filter linear acceleration . . . 74
49 Actual linear acceleration . . . 74
50 Velocity error . . . 75
51 Kalman filter velocity output . . . 76
52 System attitude (as computed by our custom attitude Kalman filter) . . . 77
53 Actual system attitude (as computed by the bno055 firmware) . . . 78
54 Attitude error . . . 79
55 GPS velocity . . . 80
56 Rotated GPS velocity . . . 81
57 Kalman filter linear acceleration . . . 82
58 Actual linear acceleration . . . 82
59 Velocity error . . . 83
60 Kalman filter velocity output . . . 84
61 GPS velocity . . . 85
62 Rotated GPS velocity . . . 86

List of Listings

1 The main function of the software system . . . 38
2 App initialization function . . . 41
3 Setup function for bno055 . . . 42
4 Connect function for neo-m9n . . . 43
5 Configure function for neo-m9n . . . 44
6 Period elapsed callback function for TIM3 . . . 44
7 Buffer type definition . . . 45
8 Utility functions for the buffer type . . . 45
9 bno055 buffer defines . . . 46
10 I2C DMA RX complete callback function for bno055 . . . 47
11 Period elapsed callback function for TIM4 . . . 48
12 UART DMA TX complete callback function for debug buffer . . . 48
13 Main data processing function . . . 49
14 attitude processing function . . . 50
15 Velocity processing function . . . 54
16 Debug buffer defines . . . 56
17 Script to preprocess captured debug data from the GSMS . . . 58

List of Tables

1 Project Risk Assessment and Evaluation . . . 15
2 Risk consequences, mitigation strategies and contingency plans . . . 17
3 List of sensors and their output data . . . 20
4 Test Scenarios Overview . . . 68

A Appendix

A.1 Project Management

A.1.1 List of Requirements FSG

Formula Student Germany Rules that may apply to GSS

Page #	Topic	Rule	Importance	Ruletext	Comments
58	System Critical Signals (SCSs)	T 11.9.1	medium	SCS are defined as all electrical signals which • influence actions on the shutdown circuit, see CV4.1 and EV 6.1 • influence the wheel torque. • [EV ONLY] influence indicators according to EV 5.8.8, EV 4.10 or EV 6.3.7 • [DV ONLY] influence indicator according to DV 3.2.7	General definition
58	System Critical Signals (SCSs)	T 11.9.2	high	Any of the following SCS single failures must result in a safe state of all connected systems: (a) Failures of signals transmitted by cable: • Open circuit • Short circuit to ground (b) Failures of analog sensor signals transmitted by cable: • Short circuit to supply voltage (c) Failures of sensor signals used in programmable devices: • Implausibility due to out of range signals, e.g. mechanically impossible angle of an angle sensor. (d) Failures of digitally transmitted signals by cable or wireless: • Data corruption (e.g. checked by a checksum) • Loss and delay of messages (e.g. checked by transmission time outs) Signals might be a member of multiple signal classes, e.g. analog signals transmitted by cable might be a member of T 11.9.2.a, T 11.9.2.b and T 11.9.2.c. If a signal failure is correctable, e.g. due to redundancy or worst case values, the safe state must be entered as soon as an additional non correctable failure occurs.	
58	System Critical Signals (SCSs)	T 11.9.3	high	The maximum allowed delay of messages according to T 11.9.2.d must be chosen depending on the impact of delayed messages to the connected system, but must not exceed 500 ms.	
58	System Critical Signals (SCSs)	T 11.9.4	high	Safe state is defined depending on the signals as follows: • signals only influencing indicators – Indicating a failure of its own function or of the connected system • low voltage battery signals – At least one pole is electrically disconnected from the rest of the vehicle • [EV ONLY] For all others signals – opened shutdown circuit and opened AIRs [...]	
58	System Critical Signals (SCSs)	T 11.9.5	low	Indicators according to T 11.9.1 with safe state "illuminated" (e.g. absence of failures is not actively indicated) must be illuminated for 1 s to 3 s for visible check after power cycling the LVMS.	
90	Wireless Communication	DV 1.2.1	High	It is prohibited to change parameters, send commands or make any software changes by wireless communication. Receiving information from the vehicle via one-way-telemetry is allowed. During dynamic events, wireless communication may be limited and an uninterfered and reliable wireless connection is not guaranteed by the officials.	
90	Wireless Communication	DV 1.2.3	High	(D)GPS may be used, but there will be no space to securely build up base stations on the competition site	
90	Data logger	DV 1.3.1	medium	The officials will provide a standardized data logger that must be installed in any DV during the competition. Further specifications for the data logger and required hardware and software interfaces can be found in the competition handbook.	

90	Data logger	DV 1.3.2	low	The intention of the data logger is to understand and reproduce the system state in case of failure. This includes a basic set of signals defined in the competition handbook and the set of vehicle-individual signals that have to be monitored by the Emergency Brake System (EBS) to ensure redundancy and fault detection.		
91	Signals	DV 2.1.1	High	Any signal of the AS is a SCS	SCS = System Critical Signal	
92	ASMS	DV 2.2.5	low	When the ASMS is in "Off" position, the following must be fulfilled: • No steering, braking and propulsion actuation can be performed by request of the autonomous system. **• The sensors and the processing units can stay operational.** [...]	ASMS = Autonomous System Master Switch	
94	Autonomous Missions	DV 2.6.1	medium	The AS must at least implement the following missions: • Acceleration • Skidpad • Autocross • Trackdrive • EBS test • Inspection • Manual driving		
95	Autonomous System Form	DV 2.7.1	low	Prior to the competition, all teams must submit a clearly structured documentation of their entire AS (including EBS and steering system) called ASF.	Not important during PA, will be needed for competition in 2021	
95	Autonomous System Form	DV 2.7.2	low	The ASF must at least contain the following items: **• All applied sensors (see also DV 4.2)** • A clearly structured documentation of the entire EBS. • A dbc file defining the supervised signals of the EBS monitoring. • A clearly structured documentation of the entire steering system.		
96	Mounting	DV 4.1.1	medium	All sensors and components must be securely mounted. For all mounts, T 8.3.1 applies.	Requirement for construction and montage of sensors T 8.3.1: All forward facing edges of aerodynamic devices that could contact a pedestrian must have a minimum radius of 5 mm for all horizontal edges and 3 mm for vertical edges.	
96	Mounting	DV 4.1.2	medium	Sensors and components may not come into contact with the driver's helmet under any circumstances.	restriction of placement of sensors	
96	Mounting	DV 4.1.3	low	All sensors and components must be positioned within the surface envelope (see T 1.1.16).	T 1.1.16: Surface envelope – The surface envelope is the surface defined by the top of the roll bar and the outside edges of the four tires. (See Figure 2)	
96	Mounting	DV 4.1.4	low	Antennas that are exclusively acting as such with the longest side <100 mm may protrude from the envelope. For components behind the driver's compartment an overhang by 25 % of their bounding box volume is accepted.		
96	Mounting	DV 4.1.5	high	Additionally, sensors may be mounted with a maximum distance of 500 mm above the ground and less than 700 mm forward of the front of the front tires (see figure 22). They must not exceed the width of the front axle (measured at the height of the hubs).		
96	Legal & Work Safety	DV 4.2.1	High	All sensors must fulfill the local legislative specifications (i.e. eye-protection classification for laser sensors, power limitation for radar sensors, etc.) in the country of competition.		
97	Legal & Work Safety	DV 4.2.2	low	This must be demonstrated by submitting the datasheets for the implemented sensors prior to the competition as an ASF Add Item Request (AAIR).		
103	Driverless Inspection Objective	IN 6.1.1	low	The objective of the DV inspection is to prove that: • All implemented sensors, including their mounting and location, are compliant with the rules [...]		

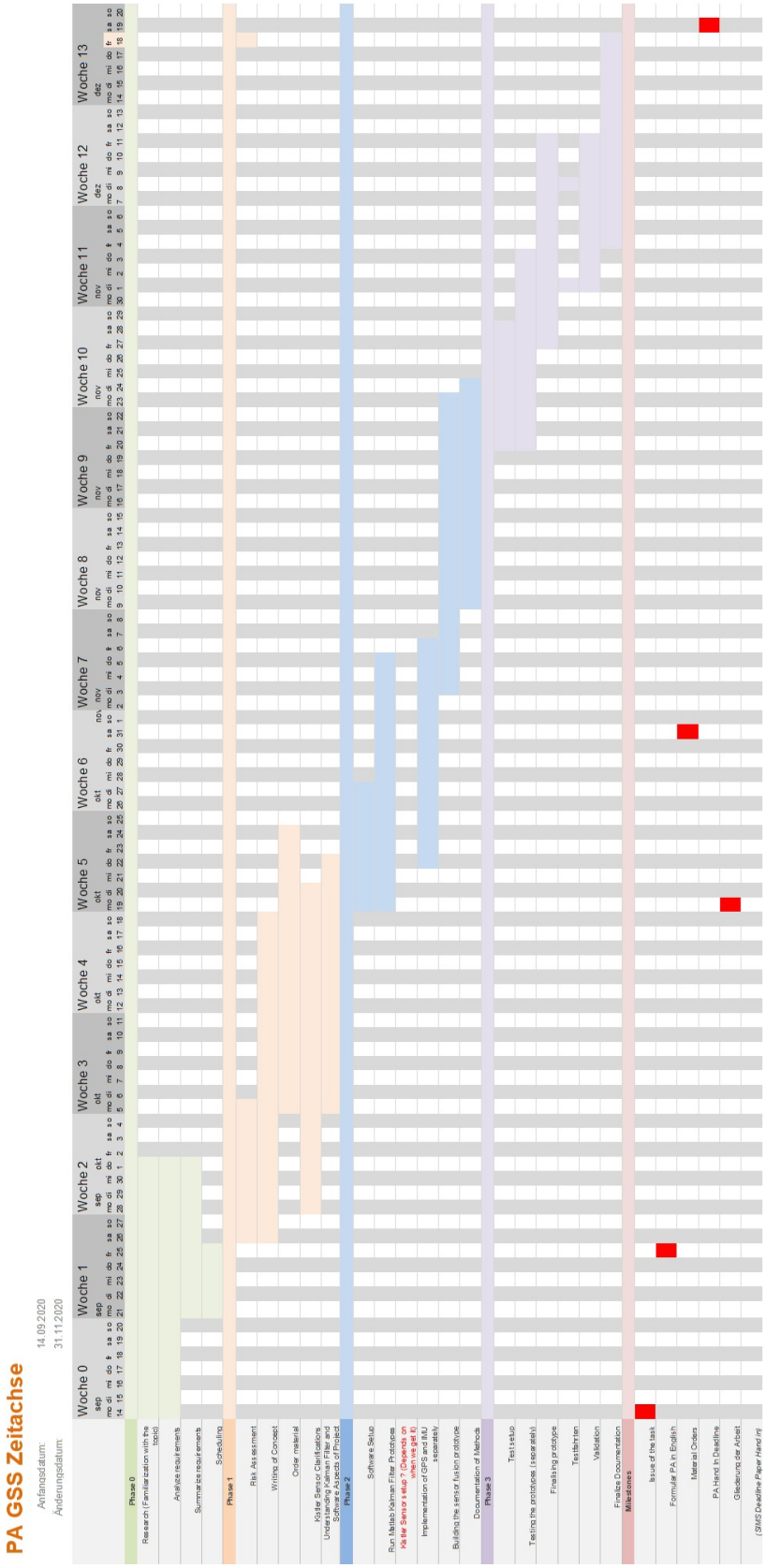
103	Driverless Inspection Required Items	IN 6.2.1	High	The following items are required: • One ASR • The vehicle (in fully assembled , ready-to-race condition including mounted datalogger (see DV 1.3) • **Data sheets for all perception sensors** • **Documents which proof that all perception sensors meet local legislation** • RES remote control • ASF • Tools needed for the (dis)assembly of parts for DV inspection • Print-outs of rule questions (if applicable)	
103	Driverless Inspection EBS Test	IN 6.3.3	medium	During the brake test, the vehicle must accelerate in autonomous mode up to at least 40 km/h within 20 m. From the point where the RES is triggered, the vehicle must come to a safe stop within a maximum distance of 10 m.	Good to know for speed
103	Driverless Inspection EBS Test	IN 6.3.4	medium	In case of wet track conditions, the stopping distance will be scaled by the officials dependent on the friction level of the track.	Track conditions
104	Rain Test Procedure	IN 9.2.6	medium	Water will be sprayed at the vehicle from any possible direction. The water spray is similar to a vehicle driving in rain and not a direct high-pressure stream of water.	Track conditions
106	Post Event Inspection Procedure	IN 12.1.10	low	[EV OR DV ONLY] Directly after trackdrive or endurance and leaving parc fermé, the data logger, see EV 4.6 or DV 1.3, will be disassembled from the vehicle.	
110	Bill of Material (BOM)	S 2.4.2	medium	The BOM must list all parts and equipment fitted to the prototype vehicle at any time during the competition.	Rule concerning competition: Keep track of finances
117	Ground Clearance	D 2.3.1	medium	Sliding skirts or other aerodynamic devices that by design, fabrication or as a consequence of moving, contact the track surface are prohibited. Any violation may be penalized by a mechanical black flag.	
118	[DV only] Vehicle Break Downs and Usage of RES	D 2.7.4	medium	The ASR or the officials may stop the vehicle using the RES in any of the following cases: • The average speed of the first three laps in trackdrive (after completing the third lap) is below 2.5 m/s or the average speed of any of the following laps is below 3.5 m/s.	ASR = Autonomous System Responsible
119	Operating Conditions	D 3.1.1	medium	The following track conditions are recognized: • Dry • Damp • Wet	
95	Functional Safety	DV 3.2.1	low	Due to the safety critical character of the EBS, the system must either remain fully functional, or the vehicle must automatically transition to the safe state in case of a single failure mode.	sensors have to be planned

A.1.2 Risk Assessment

Risk Assessment Form

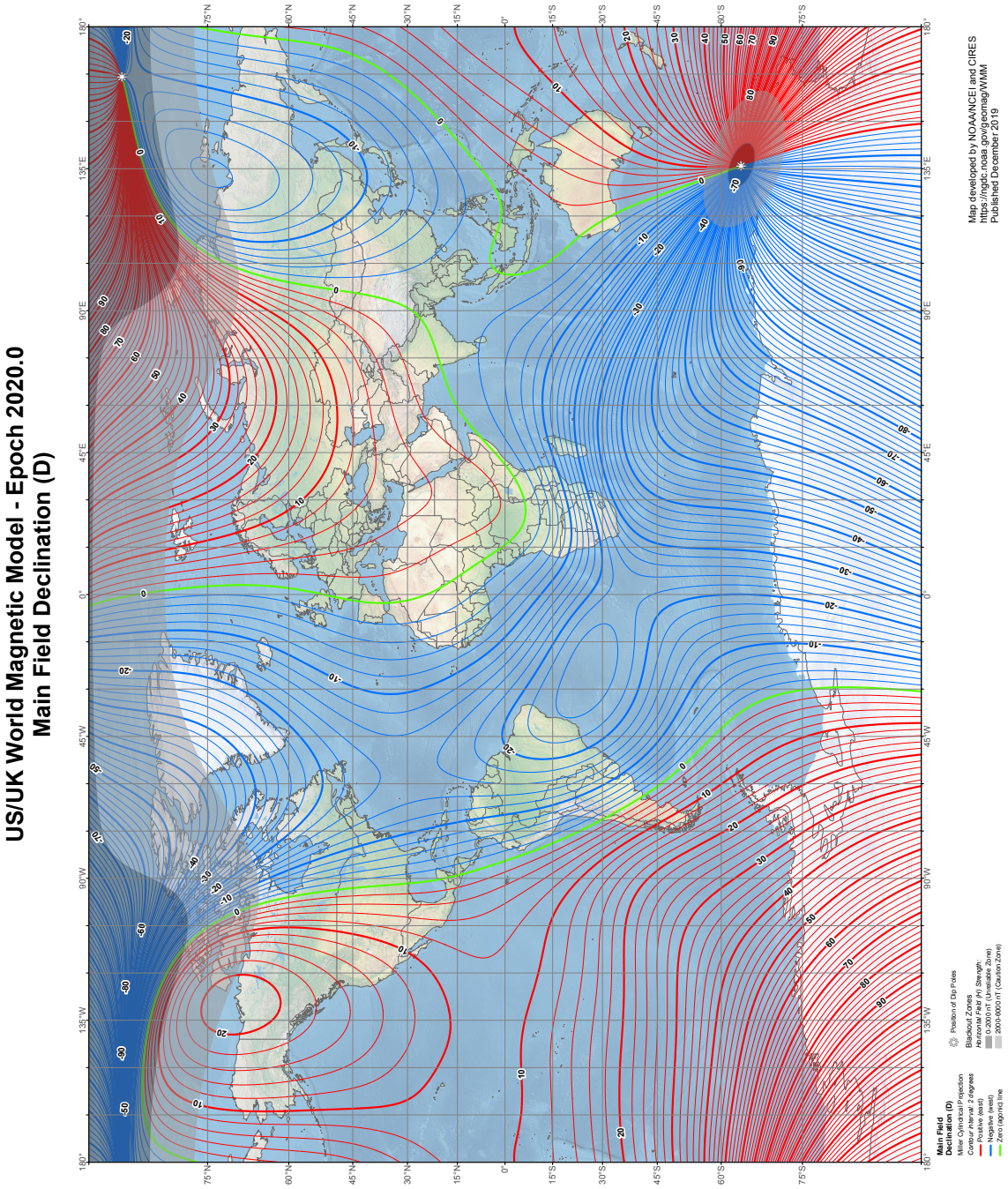
Risk #	Risk Description	Probable Cause	Consequence	Probability of Occurrence (1-5)	Impact Intensity (1-3)	Assessment (Probability * Impact)	Existing Measures / Mitigation Strategy	Contingency Plan
1	Delivery Difficulties in getting Kistler Sensor	Uncertainty of Sponsorship and therefore uncertainty if delivery is even possible for sensor. Time until delivery is certain could take until end of project	Delay of project	4	3	12	Frequent Communication with Kistler Representative. Weekly checkups on updates	
2	Hardware deffective on delivery	Delivery faces unforeseeable circumstances where items get broken or interfered with in a negative way	Loss of time, delay of project	1	1	1	Keep in mind that extra orders might be needed to place	Reorder deffective items
3	Hardware deffective due to inappropriate behaviour	Could happen during prototype setup or in inputting sensors into a new environment with untested sensor combinations	Loss of time and work, delay of project	1	2	2		
4	New Kistler Sensor brings risk of faulty hardware	Because the sensor is not officially released yet and brandnew it might still have some faulty behaviour given specific unknown environments that we could come in contact with.	Wrong output data	2	2	4	Thorough research is carried out in advance and professionals are asked when in doubt	
5	Computing power is not enough for planned sensor fusion algorithm	The microcontroller might now have enough computing power for the various matrices and vector computations that have to be done for the planned Kalman filter.	No output or wrong output, Loss of time	3	2	6	Divide mathematical complex computations into smaller separate computations	Connect a host computer to the system to compute complex equations with enough computing power there instead
6	Kalman Filter Approach Unsatisfactory	Difficulty in arranging custom algorithm too high: getting the measurement and process model exactly right for the kalman filter with unknown error or uncertainty will be very challenging	Loss of time, delay of project	3	3	9	Communicate any questions and problems to a master student who has proper kalman filter knowledge and experience	Simplify algorithms and use implementation libraries instead
7	Delivery Difficulties for American Sensors due to Corona	Since Countries are back in lockdown mode, it might be difficult to obtain the planned sensors from the american shipping site Digikey.com	Loss of time, delay of project	2	1	2	Check that order items are in stock and that they should have immediate shipping possibility	Have backup online shops where items can be reordered, in case they get stuck in shipment or at customs
8	Unforeseeable changes on Formula Student ZHAW car	Other team members made changes on the vehicle without any communication	Placing and Cabling issue for our sensors	1	2	2	We work closely with the Electrics-Team to insure communication flow and knowledge about all major changes	
9	Update frequency of GPS not sufficient	GPS has a lower data output rate compared to other measuring sensors	The low update rate of the GPS output could lead to very inaccurate data	3	2	6	A GPS with a very high data output rate should be chosen for this project	

A.1.3 Timeline



A.2 World Magnetic Model

Image source: `https://www.ngdc.noaa.gov/geomag/WMM/image.shtml`



B Appendix

B.1 MATLAB Attitude Kalman Filter Algorithm

B.1.1 Initialization

```
function att_kf() %#codegen
    % ATT_KF initialize attitude kalman filter

    global att_kf_q_ref; % reference attitude quaternion q_ref
    global att_kf_x; % state vector x
    global att_kf_Q; % process noise covariance matrix Q
    global att_kf_R; % measurement noise covariance matrix R
    global att_kf_P; % error covariance matrix P

    % attitude kalman filter constants
    gyro_white_noise = 0.03 / 180 * pi; % [rad/s/sqrt(Hz)]
    % [rad/s^2/sqrt(Hz)] :
    gyro_bias_white_noise = 0.003 / sqrt(200) / 180 * pi;
    % [uT/s/sqrt(Hz)], [m/s^3/sqrt(Hz)] :
    measurement_white_noise = 4 / 180 * pi;
    init_attitude_error_stddev = sqrt(1000); % [-]
    init_gyro_bias_stddev = 2 / 180 * pi; % [rad/s]

    % initialize kalman filter variables
    att_kf_q_ref = [1; 0; 0; 0];
    att_kf_x = zeros(6, 1);
    att_kf_Q = diag([ones(1, 3) * gyro_white_noise^2, ones(1, 3) *
        gyro_bias_white_noise^2]);
    att_kf_R = eye(2) * measurement_white_noise^2;
    att_kf_P = diag([ones(1, 3) * init_attitude_error_stddev^2,
        ones(1, 3) * init_gyro_bias_stddev^2]);
end
```

B.1.2 Prediction Step

```
function att_kf_predict(w) %#codegen
    % ATT_KF_PREDICT prediction step (time update)

    global att_kf_q_ref; % reference attitude quaternion q_ref
    global att_kf_x; % state vector x
    global att_kf_Q; % process noise covariance matrix Q
    global att_kf_P; % error covariance matrix P

    % calculate delta time
    dt = 10 / 1000; % [s]
    % remove gyroscope bias
    omega = w - att_kf_x(4:6);
    % calculate angle of rotation
    ang = norm(omega) * dt;
    % calculate delta quaternion
    if ang > 0.000001
        % calculate delata quaternion from angle and axis of rotation
```

```
18          axis = omega / norm(omega);
19          delta_q_ref = [cos(ang/2); axis * sin(ang/2)];
20      else
21          % calculate delta quaternion using small angle approximation
22          delta_q_ref = [1; omega * dt / 2];
23      end
24      % predict q_ref
25      att_kf_q_ref = quat_mult(att_kf_q_ref, delta_q_ref);
26      % calculate linearized state transition matrix Phi
27      % Note: The EKF linearizes the underlying model to produce matrix
            Phi.
28      F = [cross_prod_mat(-omega), eye(3) * -1;
29           zeros(3, 6)];
30      A = [-F, att_kf_Q';
31           zeros(6,6), F'];
32      B = expm(A * dt);
33      Phi = B(7:12, 7:12)';
34      % calcualte linearized process noise covariance matrix Qs
35      % Note: The EKF linearizes the underlying model to produce matrix
            Qs.
36      Qs = Phi * B(1:6, 7:12);
37      % Note: The attitude error is transferred to q_ref after every
            iteration. This ensures, that the predicted (apriori) state
            vector x(-) is equal to the state vector x of the previous
            iteration.
38      % calculate predicted (apriori) error covariance matrix P(-)
39      att_kf_P = Phi * att_kf_P * Phi' + Qs;
40  end
```

B.1.3 Measurement Step

```
1   function att_kf_measure(expected_i, measured_b) %#codegen
2       % ATT_KF_MEASURE measurement step (measurement update)
3
4       global att_kf_q_ref; % reference attitude quaternion q_ref
5       global att_kf_x; % state vector x
6       global att_kf_R; % measurement noise covariance matrix R
7       global att_kf_P; % error covariance matrix P
8
9       % skip measurement step if vector measured_b is close to zero
10      if norm(measured_b) < 0.001
11          return;
12      end
13
14      % normalize expected vector in the inertial frame
15      expected_i = expected_i / norm(expected_i);
16      % normalize measured vector in the body frame
17      measured_b = measured_b / norm(measured_b);
18      % Note: The following steps calculate a rotation matrix to
            transform the vector measured_b from the body frame into the
            inertial frame and to rotate it onto the vector m (normal
            vector of the projection plane).
```

```
19      i_to_m = quat_from_vect(expected_i, [1; 0; 0]);
20      b_to_m = quat_mult(i_to_m, att_kf_q_ref);
21      b_to_m = rotm_from_quat(b_to_m);
22      % rotate vector expected_i from the inertial frame into the body
            frame
23      expected_b = rotate_by_quat(expected_i, quat_conj(att_kf_q_ref));
24      % Note: The following steps rotate the vector measured_b using the
            rotation matrix b_to_m and then calculate the measurement
            vector z by projecting the rotated vector measured_b onto the y
             and z plane.
25      m = b_to_m * measured_b + [1; 0; 0];
26      % skip the measurement step if the expected and the measured vector
             are antiparallel
27      if norm(m) < 0.001
28          return;
29      end
30      m = m / norm(m);
31      z = m(2:3) / m(1) * 2;
32      % calculate observation matrix H
33      Proj = [0, 1, 0;
34              0, 0, 1];
35      Ha = Proj * b_to_m * cross_prod_mat(expected_b);
36      H = [Ha, zeros(2, 3)];
37      % calculate measurement innovation vector y
38      h = [0; 0];
39      y = z - h;
40      % calculate kalman gain matrix K
41      S = H * att_kf_P * H' + att_kf_R;
42      K = att_kf_P * H' / S;
43      % calculate corrected (aposteriori) state vector x(+)
44      att_kf_x = att_kf_x + K * y;
45      % calculate corrected (aposteriori) error covariance matrix P(+)
46      att_kf_P = (eye(6) - K * H) * att_kf_P;
47  end
48
49  function q = quat_from_vect(v1, v2) %#codegen
50      %QUAT_FROM_VECT quaternion from two 3 element vectors
51
52      v1 = v1 / norm(v1);
53      v2 = v2 / norm(v2);
54      c = v1' * v2;
55      if (c > 0.99999)
56          q = [1; 0; 0; 0];
57      elseif (c < -0.99999)
58          w = cross(v1, [1, 1, 1])';
59          q = [0; w/norm(w)];
60      else
61          ax = cross(v1, v2);
62          s = sqrt((1+c)*2);
63          q = [s*0.5; ax/s];
64      end
```

```
end

function m = rotm_from_quat(q) %#codegen
    %ROTM_FROM_QUAT rotation matrix from quaternion

    m1 = rotate_by_quat([1; 0; 0], q);
    m2 = rotate_by_quat([0; 1; 0], q);
    m3 = rotate_by_quat([0; 0; 1], q);
    m = [m1, m2, m3];
end
```

B.1.4 Propagate Attitude Error

```
function att_kf_propagate() %#codegen
    % ATT_KF_PROPAGATE propagate attitude error
    global att_kf_q_ref; % reference attitude quaternion q_ref
    global att_kf_x; % state vector x

    % calculate rotation quaternion from attitude error vector a
    delta_q_of_a = [2; att_kf_x(1); att_kf_x(2); att_kf_x(3)];
    % propagate attitude error to reference attitude quaternion q_ref
    att_kf_q_ref = quat_mult(att_kf_q_ref, delta_q_of_a);
    % normalize reference attitude quaternion q_ref
    att_kf_q_ref = att_kf_q_ref / norm(att_kf_q_ref);
    % reset attitude error vector a
    att_kf_x(1:3) = zeros(3, 1);
end
```

B.1.5 Reference Attitude Quaternion

```
function q_ref = att_kf_q_ref() %#codegen
    % ATT_KF_Q_REF reference attitude quaternion
    global att_kf_q_ref; % reference attitude quaternion q_ref

    % return reference attitude quaternion
    q_ref = att_kf_q_ref;
end
```

B.1.6 Attitude Kalman Filter Test

```
% test attitude kalman filter

% read raw bno055 data
fid = fopen("bno055_raw.txt");
data = textscan(fid, '%4xs16');
data = data{1,1};
data = data';
data = reshape(data, 16, []);
data = double(data);
count = length(data);
% accelerometer
accel = data(1:3,:) / 100; % [m/s^2]
% magnetometer
```

```
mag = data(4:6,:) / 16; % [uT]
% gyroscope
gyro = data(7:9,:) / 900; % [rad/s]
% quaternion data
quat = data(13:16,:) / 2^14; % [-]

% delta t
dt = ones(1, count) * 10 / 1000; % [s]
% calculate absolute time
t = cumsum(dt);

% plot actual attitude
figure;
plot(t, quat(1,:));
hold on
plot(t, quat(2,:));
plot(t, quat(3,:));
plot(t, quat(4,:));
hold off
title("Actual Attitude");
xlabel("time [s]");
ylabel("quaternion components [-]");
legend("w", "x", "y", "z");

% kalman filter attitude
quat_kf = zeros(4, count);
% attitude error
diff = zeros(1, count); % [deg]

% initialize attitude kalman filter
att_kf();
% run attitude kalman filter
for i = 1:count
    % prediction step (time update)
    att_kf_predict(gyro(:,i));
    % measurement step (measurement update)
    % measure gravity vector
    att_kf_measure([0; 0; 1], accel(:,i));
    % measure north vector
    att_kf_measure([0; 0.446; -0.895], mag(:,i));
    % propagate attitude error
    att_kf_propagate();
    % store kalman filter attitude
    quat_kf(:,i) = att_kf_q_ref();
    % calculate attitude error
    diff(i) = angle_between_quat(quat_kf(:,i), quat(:,i)) / pi * 180;
end

% plot attitude error
figure;
plot(t, diff);
```

```
title("Attitude Error");
xlabel("time [s]");
ylabel("attitude error [deg]");
legend("attitude error");

% plot kalman filter attitude
figure;
plot(t, quat_kf(1,:));
hold on
plot(t, quat_kf(2,:));
plot(t, quat_kf(3,:));
plot(t, quat_kf(4,:));
hold off
title("Kalman Filter Attitude");
xlabel("time [s]");
ylabel("quaternion components [-]");
legend("w", "x", "y", "z");
```

B.2 MATLAB Velocity Kalman Filter Algorithm

B.2.1 Initialization

```
function vel_kf() %#codegen
    % VEL_KF initialize velocity kalman filter
    global vel_kf_x; % state vector x
    global vel_kf_Q; % process noise covariance matrix Q
    global vel_kf_R; % measurement noise covariance matrix R
    global vel_kf_P; % error covariance matrix P

    % velocity kalman filter constants
    accel_white_noise = 150 / 1000000 * 9.8053; % [m/s^2/sqrt(Hz)]
    measurement_white_noise = 4 / 180 * pi; % [m/s/sqrt(Hz)]
    init_velocity_error_stddev = sqrt(1000); % [m/s]

    % initialize kalman filter variables
    vel_kf_x = zeros(3, 1);
    vel_kf_Q = eye(3) * accel_white_noise^2;
    vel_kf_R = eye(3) * measurement_white_noise^2;
    vel_kf_P = eye(3) * init_velocity_error_stddev^2;
end
```

B.2.2 Prediction Step

```
function vel_kf_predict(lin_accel) %#codegen
    % VEL_KF_PREDICT prediction step (time update)
    global vel_kf_x; % state vector x
    global vel_kf_Q; % process noise covariance matrix Q
    global vel_kf_P; % error covariance matrix P

    % calculate delta time
    dt = 10 / 1000; % [s]
    % calculate predicted (apriori) state vector x(-)
    vel_kf_x = vel_kf_x + (lin_accel * dt);
```

```
    % calculate predicted (apriori) error covariance matrix P(-)
    vel_kf_P = vel_kf_P + vel_kf_Q;
end
```

B.2.3 Measurement Step

```
function vel_kf_measure(measured_b) %#codegen
    % VEL_KF_MEASURE measurement step (measurement update)
    global vel_kf_x; % state vector x
    global vel_kf_R; % measurement noise covariance matrix R
    global vel_kf_P; % error covariance matrix P

    % calculate measurement innovation vector y
    y = measured_b - vel_kf_x;
    % calculate kalman gain matrix K
    S = vel_kf_P + vel_kf_R;
    K = vel_kf_P / S;
    % calculate corrected (aposteriori) state vector x(+)
    vel_kf_x = vel_kf_x + K * y;
    % calculate corrected (aposteriori) error covariance matrix P(+)
    vel_kf_P = (eye(3) - K) * vel_kf_P;
end
```

B.2.4 Velocity

```
function vel = vel_kf_vel() %#codegen
    % VEL_KF_VEL velocity
    global vel_kf_x; % state vector x

    % return velocity
    vel = vel_kf_x;
end
```

B.2.5 Velocity Kalman Filter Test

```
% test velocity kalman filter

% read raw bno055 data
fid = fopen("bno055_raw.txt");
data = textscan(fid, '%4xs16');
fclose(fid);
data = data{1,1};
data = data';
data = reshape(data, 19, []);
data = double(data);
count = length(data);
% accelerometer
accel = data(1:3,:) / 100; % [m/s^2]
% magnetometer
mag = data(4:6,:) / 16; % [uT]
% gyroscope
gyro = data(7:9,:) / 900; % [rad/s]
% quaternion data
```

```
quat = data(13:16,:) / 2^14; % [-]
% linear acceleration data
lin_accel = data(17:19,:) / 100; % [m/s^2]

% read raw neo-m9n data
fid = fopen("neom9n_raw.txt");
data = textscan(fid, '%8xs32');
fclose(fid);
data = data{1,1};
data = data';
data = reshape(data, 3, []);
data = double(data);
% mark empty velocity samples using NaN
vel_empty = -2147483648; % 0x80000000
vel = data(1:3,:);
vel(vel == vel_empty) = NaN;
% velocity
vel = vel / 1000; % [m/s]

% read raw attitude kalman filter output data
fid = fopen("attkf_raw.txt");
data = textscan(fid, '%4xs16');
fclose(fid);
data = data{1,1};
data = data';
data = reshape(data, 4, []);
data = double(data);
% kalman filter attitude
quat_kf = data(1:4,:) / 2^14; % [-]

% delta t
dt = ones(1, count) * 10 / 1000; % [s]
% calculate absolute time
t = cumsum(dt);

% ------------------------------------------------------------------------
% attitude
% ------------------------------------------------------------------------

% plot actual attitude
figure;
plot(t, quat(1,:));
hold on
plot(t, quat(2,:));
plot(t, quat(3,:));
plot(t, quat(4,:));
hold off
title("Actual Attitude");
xlabel("time [s]");
ylabel("quaternion components [-]");
legend("w", "x", "y", "z");
```

```
70
71 % plot kalman filter attitude
72 figure;
73 plot(t, quat_kf(1,:));
74 hold on
75 plot(t, quat_kf(2,:));
76 plot(t, quat_kf(3,:));
77 plot(t, quat_kf(4,:));
78 hold off
79 title("Kalman Filter Attitude");
80 xlabel("time [s]");
81 ylabel("quaternion components [-]");
82 legend("w", "x", "y", "z");
83
84 % attitude error
85 diff = zeros(1, count); % [deg]
86 % calculate attitude error
87 for i = 1:count
88     diff(i) = angle_between_quat(quat_kf(:,i), quat(:,i)) / pi * 180;
89 end
90
91 % plot attitude error
92 figure;
93 plot(t, diff);
94 title("Attitude Error");
95 xlabel("time [s]");
96 ylabel("attitude error [deg]");
97 legend("attitude error");
98
99 % ------------------------------------------------------------------------
100 % linear acceleration
101 % ------------------------------------------------------------------------
102
103 % plot actual linear acceleration
104 figure;
105 plot(t, lin_accel(1,:));
106 hold on
107 plot(t, lin_accel(2,:));
108 plot(t, lin_accel(3,:));
109 hold off
110 title("Actual Linear Acceleration");
111 xlabel("time [s]");
112 ylabel("acceleration components [m/s^2]");
113 legend("x", "y", "z");
114
115 % kalman filter linear acceleration
116 lin_accel_kf = zeros(3, count); % [m/s^2]
117 % calculate linear acceleration
118 gravity = [0; 0; 9.8053]; % [m/s^2]
119 for i = 1:count
120     lin_accel_kf(:,i) = accel(:,i) -
```

```
121     rotate_by_quat(gravity, quat_conj(quat_kf(:,i)));
122 end
123
124 % plot kalman filter linear acceleration
125 figure;
126 plot(t, lin_accel_kf(1,:));
127 hold on
128 plot(t, lin_accel_kf(2,:));
129 plot(t, lin_accel_kf(3,:));
130 hold off
131 title("Kalman Filter Linear Acceleration");
132 xlabel("time [s]");
133 ylabel("acceleration components [m/s^2]");
134 legend("x", "y", "z");
135
136 % linear acceleration error
137 diff_lin_accel_ang = zeros(1, count); % [deg]
138 diff_lin_accel_norm = zeros(1, count); % [m/s^2]
139 % calculate linear acceleration error
140 for i = 1:count
141     s = (lin_accel_kf(:,i)' * lin_accel(:,i)) /
142         (norm(lin_accel_kf(:,i)) * norm(lin_accel(:,i)));
143     diff_lin_accel_ang(i) = real(acos(s)) / pi * 180;
144     diff_lin_accel_norm(i) = abs(norm(lin_accel_kf(:,i))
145         - norm(lin_accel(:,i)));
146 end
147
148 % plot linear acceleration error
149 figure;
150 yyaxis left
151 plot(t, diff_lin_accel_ang);
152 ylabel("linear acceleration angle error [deg]");
153 hold on
154 yyaxis right
155 plot(t, diff_lin_accel_norm);
156 ylabel("linear acceleration norm error [m/s^2]");
157 hold off
158 title("Linear Acceleration Error");
159 xlabel("time [s]");
160 legend("linear acceleration angle error",
161     "linear acceleration norm error");
162
163 % ------------------------------------------------------------------------
164 % velocity
165 % ------------------------------------------------------------------------
166
167 % kalman filter velocity
168 velocity_kf = zeros(3, count); % [m/s]
169 % rotated gps velocity
170 vel_rot = NaN(3, count); % [m/s]
171 % velocity error
```

```
172 diff_vel_ang = NaN(1, count); % [deg]
173 diff_vel_norm = NaN(1, count); % [m/s]
174 vel_sample = false(1, count);
175
176 % initialize velocity kalman filter
177 vel_kf();
178 % run velocity kalman filter
179 for i = 1:count
180     % prediction step (time update)
181     vel_kf_predict(lin_accel_kf(:,i));
182     % check if the velocity sample is not empty
183     if not(isnan(vel(:,i)))
184         % valid velocity sample
185         vel_sample(i) = true;
186         % measurement step (measurement update)
187         vel_NED = vel(:,i);
188         vel_ENU = [vel_NED(2); vel_NED(1); -vel_NED(3)];
189         % correct declination
190         phi = 2.9425 / 180 * pi;
191         measured_i = rotate_z_by_ang(vel_ENU, phi);
192         % rotate measurement from the inertial frame
193             into the body frame
194         measured_b = rotate_by_quat(measured_i,
195             quat_conj(quat_kf(:,i)));
196         % measure velocity vector
197         vel_kf_measure(measured_b);
198     end
199     % store kalman filter velocity
200     velocity_kf(:,i) = vel_kf_vel();
201     % check if the velocity sample is not empty
202     if vel_sample(i)
203         % calculate velocity error
204         s = (velocity_kf(:,i)' * measured_b) /
205             (norm(velocity_kf(:,i)) * norm(measured_b));
206         diff_vel_ang(i) = real(acos(s)) / pi * 180;
207         diff_vel_norm(i) = abs(norm(velocity_kf(:,i)) -
208             norm(measured_b));
209         % store rotated gps velocity
210         vel_rot(:,i) = measured_b;
211     end
212 end
213
214 % plot gps velocity
215 figure;
216 plot(t(vel_sample), vel(1,vel_sample));
217 hold on
218 plot(t(vel_sample), vel(2,vel_sample));
219 plot(t(vel_sample), vel(3,vel_sample));
220 hold off
221 title("GPS Velocity");
222 xlabel("time [s]");
```

```
ylabel("velocity components [m/s]");
legend("x", "y", "z");

% plot rotated gps velocity
figure;
plot(t(vel_sample), vel_rot(1,vel_sample));
hold on
plot(t(vel_sample), vel_rot(2,vel_sample));
plot(t(vel_sample), vel_rot(3,vel_sample));
hold off
title("Rotated GPS Velocity");
xlabel("time [s]");
ylabel("velocity components [m/s]");
legend("x", "y", "z");

% plot kalman filter velocity
figure;
plot(t, velocity_kf(1,:));
hold on
plot(t, velocity_kf(2,:));
plot(t, velocity_kf(3,:));
hold off
title("Kalman Filter Velocity");
xlabel("time [s]");
ylabel("velocity components [m/s]");
legend("x", "y", "z");

% plot velocity error
figure;
yyaxis left
plot(t(vel_sample), diff_vel_ang(vel_sample));
ylabel("velocity angle error [deg]");
hold on
yyaxis right
plot(t(vel_sample), diff_vel_norm(vel_sample));
ylabel("velocity norm error [m/s]");
hold off
title("Velocity Error");
xlabel("time [s]");
legend("velocity angle error", "velocity norm error");
```

B.3 Additional MATLAB Functions

B.3.1 Quaternion Conjugate

```
function qc = quat_conj(q) %#codegen
    % QUAT_CONJ quaternion conjugate
    qc = [q(1); -q(2); -q(3); -q(4)];
end
```

B.3.2 Quaternion Multiplication

```
function q = quat_mult(q1, q2) %#codegen
```

```
    % QUAT_MULT quaternion multiplication
    q = [q1(1)*q2(1) - q1(2)*q2(2) - q1(3)*q2(3) - q1(4)*q2(4);
         q1(1)*q2(2) + q1(2)*q2(1) + q1(3)*q2(4) - q1(4)*q2(3);
         q1(1)*q2(3) - q1(2)*q2(4) + q1(3)*q2(1) + q1(4)*q2(2);
         q1(1)*q2(4) + q1(2)*q2(3) - q1(3)*q2(2) + q1(4)*q2(1)];
end
```

B.3.3 Angle of Rotation Between Two Quaternion

```
function r = angle_between_quat(q1, q2) %#codegen
    % ANGLE_BETWEEN_QUAT angle of rotation between two quaternions
    u = q1(1)*q2(1) + q1(2)*q2(2) + q1(3)*q2(3) + q1(4)*q2(4);
    v = (u.^2).*2-1;
    if v > 1
        v = 1;
    elseif v < -1
        v = -1;
    end
    r = acos(v);
end
```

B.3.4 Rotate 3 Element Vector by Quaternion

```
function v_rot = rotate_by_quat(v, q) %#codegen
    % ROTATE_BY_QUAT rotate 3 element vector by quaternion
    qv = [0; v];
    qv_rot = quat_mult(quat_mult(q, qv), quat_conj(q));
    v_rot = qv_rot(2:4);
end
```

B.3.5 Rotate 3 Element Vector Around z Axis by Angle

```
function v_rot = rotate_z_by_ang(v, ang) %#codegen
    % ROTATE_Z_BY_ANG rotate 3 element vector around z axis by angle
    Rz = [cos(ang) -sin(ang) 0; sin(ang) cos(ang) 0; 0 0 1];
    v_rot = Rz * v;
end
```

B.3.6 Cross Product Matrix of 3 Element Vector

```
function m = cross_prod_mat(v) %#codegen
    % CROSS_PROD_MAT cross product matrix of 3 element vector
    m = [0, -v(3), v(2);
         v(3), 0, -v(1);
        -v(2), v(1), 0];
end
```