A comparison of different approaches to solve the SLAM problem on a Formula Student Driverless race car

Master's Thesis of

Nick Le Large

Institute of Measurement and Control Systems
Karlsruhe Institute of Technology

Reviewer: Prof. Dr.-Ing. Christoph Stiller
Advisor: Frank Bieder, M.Sc.

Karlsruhe, November 2020

Declaration / Erklärung

Ich versichere hiermit wahrheitsgemäß, die Arbeit bis auf die dem Aufgabensteller bereits bekannten Hilfsmittel selbständig angefertigt, alle benutzten Hilfsmittel vollständig angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde.

Nick Le Large
Karlsruhe, 19.11.2020

Acknowledgements

First and foremost, I want to thank Professor Dr.-Ing. Christoph Stiller, head of MRT, for supervising this external thesis.

Further, I wish to express my sincere appreciation to my supervisor, Frank Bieder, for his willingness to support and supervise my work and for the excellent advice he's given me throughout the entire project. This thesis would not have been possible without him.

Next, I want to express my gratitude to the entire team at KA-RaceIng. Even under the extraordinary circumstances of the 2020 season, I will always keep the time I could spend with this great group of people in good memory. A big thank you goes to everyone who helped me by proofreading this thesis. Specifically, I want to thank Luca Wahl and Simon Schäfer for their dedication and for supporting the research I conducted during my time at KA-RaceIng.

An important puzzle piece in the making of this thesis was the ground truth map, and I want to thank Jakob Weisgerber, research associate at the Geodetic Institute, for his help acquiring it.

I want to express my deepest gratitude to my parents and my grandparents for providing me with all the support I needed throughout my entire studies and for always making it possible for me to take not the shortest but the most interesting path.

Last but not least, from the bottom of my heart, a very special thanks goes to my fiancée, Anjana. Her patience, her understanding and her support always kept me going.

Abstract

The Formula Student Driverless competition challenges students all over the world to develop and build self-driving single-seater race cars. One of the greatest challenges of any autonomous vehicle is the problem of "Simultaneous Localization and Mapping", or SLAM in short. There are a number of approaches to solve this problem but unfortunately, there is no silver-bullet solution as an algorithm's performance highly depends on the given use case.

To find the best possible approach for the application on a Formula Student Driverless race car, this thesis presents two very different algorithms - EKF SLAM and GraphSLAM - and their respective implementation on the vehicle.

To form an objective opinion on which algorithm performs better, a tool is presented that allows continuous measurements of the performance of the algorithms. The algorithms are analyzed with respect to their accuracy and their efficiency. The latter is important because to work in the extreme conditions of a racing application, the algorithms have to be real-time capable at high frequencies with limited computational resources.

Simultaneously, there are high demands on the accuracy of the algorithms. A Formula Student track is very narrow compared to the vehicle, leaving little room for error. The accuracy is measured by comparing the SLAM-generated map to a ground truth map which was acquired using high-precision DGPS measurements.

All data sets used in this thesis were recorded on a Formula Student Driverless race car, without the need for simulated input. This requires the algorithms to function in real-world conditions, such as non-linear vehicle behavior or non-Gaussian measurements, that will deviate from assumptions made when implementing the algorithms.

The results of the evaluation show that both algorithms achieve real-time capability thanks to a parallelized architecture, with GraphSLAM draining the computational resources much faster than EKF SLAM. However, the analysis of the maps generated by the algorithms shows that GraphSLAM outperforms EKF SLAM in terms of accuracy.

Kurzfassung

Der Formula Student Driverless Wettbewerb fordert Studenten auf der ganzen Welt dazu heraus, in selbstständing entwickelten und gebauten einsitzigen Rennwagen gegeneinander anzutreten. Eine der größten Herausforderungen eines jeden autonomen Fahrzeugs ist das „Simultaneous Localization and Mapping"-Problem, kurz SLAM-Problem (Engl.: Simultanes Lokalisieren und Kartieren). Es existieren verschiedene Ansätze, um das SLAM-Problem zu lösen. Allerdings gibt es keinen Königsweg, da die Leistungsfähigkeit eines Algorithmus stark vom speziellen Anwendungsfall abhängt.

Um den bestmöglichen Ansatz für den Einsatz auf einem Formula Student Driverless Rennwagen zu identifizieren, stellt diese Arbeit zwei sehr unterschiedliche Algorithmen - EKF SLAM und GraphSLAM - und deren jeweilige Implementierung auf dem Fahrzeug vor.

Um objektiv entscheiden zu können, welcher der beiden Algorithmen besser funktioniert, wird ein Werkzeug vorgestellt, welches die Qualität der Ergebnisse kontinuierlich misst. Die Algorithmen werden mit Hinblick auf ihre Präzision und ihre Effizienz evaluiert. Letzteres ist wichtig, da die Algorithmen selbst unter den extremen Bedingungen des Einsatzes auf einem Rennwagen und mit limitierten Ressourcen in Echtzeit arbeiten müssen.

Gleichzeitig ist die Anforderung an die Genauigkeit der Algorithmen sehr hoch. Eine Formula Student Rennstrecke ist im Vergleich zum Fahrzeug sehr schmal, was nur eine geringe Toleranz für Fehler erlaubt. Um die Genauigkeit zu messen, wird die durch den Algorithmus generierte Karte mit einer Referenzkarte verglichen. Diese wurde unter Nutzung eines hochpräzisen DGPS-Messgerätes erstellt.

Alle Datensätze, die in dieser Thesis verwendet werden, wurden auf einem Formula Student Driverless Fahrzeug aufgenommen, was eine Evaluierung ohne simulierte Daten möglich macht. Die Algorithmen müssen daher unter Bedingungen funktionieren, wie sie in der echten Welt zu finden sind, wie z.B. nicht-lineares Fahrzeugverhalten oder nicht-Gauß'sche Sensormessungen. Diese Bedingungen weichen von Annahmen ab, die bei der Implementierung der Algorithmen getroffen wurden.

Die Ergebnisse der Evaluierung zeigen, dass beide Algorithmen dank einer parallelisierten Architektur in Echtzeit arbeiten, wobei GraphSLAM mehr Ressourcen verbraucht als EKF SLAM. Im Hinblick auf die Genauigkeit zeigt die Analyse, dass die Schätzungen des GraphSLAM denen des EKF SLAM überlegen sind.

Contents

Acronyms

ACU Autonomous Computing Unit

DGPS Differential Global Positiong System
DNF Did Not Finish
DOO Down or Out
DQ Disqualified

EBS Emergency Brake System
EKF Extended Kalman Filter

FS Formula Student
FSD Formula Student Driverless
FSG Formula Student Germany
FSO Formula Student Online

GPU Graphics Processing Unit

ICP Iterative Closest Point
IMU Inertia Measurement Unit

JCBB Joint Compatibility Branch and Bound

KIT Karlsruhe Institute of Technology

MPC Model Predictive Controller

OC Off-course

RES Remote Emergency System
ROS Robot Operating System

SLAM Simultaneous Localization and Mapping

USS Unsafe Stop
UTM Universal Transverse Mercator

1 Introduction

1.1 Motivation

Driver Assistance and Safety

The automotive industry is changing rapidly due to technological progress in the field of automation. Automation in road vehicles has come a long way since the introduction of features like cruise control or ABS. Today's vehicles come equipped with an increasing number of automation systems that improve comfort and safety. These systems play a significant role in the reduction of deadly road accidents. Figure 1.1 shows the drop of fatal accidents in Germany during the past decades, even though the number of registered vehicles increased tenfold over the past 60 years [43].

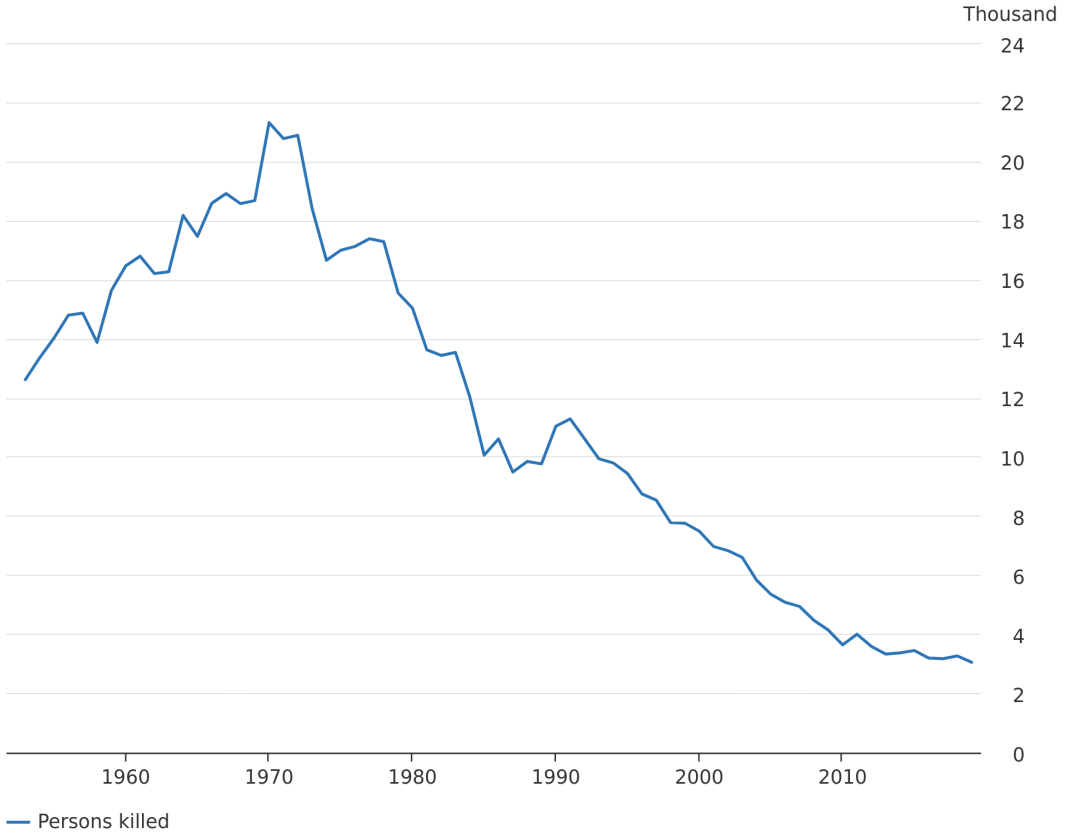


Figure 1.1: Number of traffic accident fatalities since 1950 [40].

The next step from driver assistance systems is to remove the need for a driver altogether. Going from assisted driving to autonomous driving is a huge challenge for different reasons. Aside from legal and ethical questions that need to be answered, the consumer must accept the new technology and must be ready to hand over control to a machine. However, the benefits an autonomously driven vehicle has over a manually driven car are significant. In 2018, the number of accidents in

Germany caused by driver mistake was more than 6 times larger than all other causes combined, including technical faults, road surface conditions or obstacles [32] [33] [34] [35].

The increased safety in addition to an improved comfort promised by autonomous vehicles leads the automotive industry to use a big portion of their investments for the development of autonomous driving. In 2017, this portion was only surpassed by the budget spent on the drivetrain [38]. These numbers are explained by the fact that autonomous driving is challenging from a technological standpoint as well. When taking the human out of the equation, the machine must be able to respond to unknown situations in order to ensure a reliable system. Therefore, it needs to accomplish all the tasks a human would. This includes perceiving the environment and the objects in it, estimating the ego position, predicting what other road users will do, planning a trajectory accordingly and finally giving the correct acceleration and steering input to follow that trajectory.

Racing and Technological Advancement

When it comes to new technology in the automotive industry, racing has quite often played a key role in fueling innovation and pushing cars to the limit of what is possible. Technologies like all-wheel drive, semi-automatic gearboxes or materials like carbon fiber have made their way into production vehicles through their application in race cars. Racing series, like Formula E or Roborace have been founded specifically with this in mind. Formula Student (FS) is another competition with this purpose but the race cars are developed exclusively by students. The cars are competing in various, loosely connected events all over the world. Following the technological trend, various Formula Student competitions, including Formula Student Germany (FSG), introduced the Formula Student Driverless (FSD) competition. In slightly adjusted disciplines compared to the counterpart competitions Formula Student Combustion and Formula Student Electric, the students are required to overcome the technological challenges mentioned above to build a fully autonomous race car.

The SLAM Problem

One of these challenges is localizing the car in an environment whilst at the same time mapping that environment. This task is often referred to as a hen-or-egg problem, as localization requires a map and vice versa. In robotics, this dilemma is called Simultaneous Localization and Mapping (SLAM).

SLAM is not restricted to autonomous cars but has a broad field of use cases, from autonomous submarines [29] and drones [26] to robot vacuums [24]. Because of this, it has been widely researched since the 1980s [17] but is still an extremely relevant topic today. Years of research resulted in different mathematical approaches in the attempt of solving the SLAM problem, each having a specific set of advantages and disadvantages. Choosing the correct approach highly depends on the use case scenario.

1.2 Problem Statement

Localization and mapping can be one of the bottlenecks of the overall system. First of all, bad localization performance leads to inaccurate maps and vice versa. But additionally, in autonomous racing, if the map quality is not sufficient, the trajectory planning needs to diverge from the ideal racing line in order to leave big enough safety margins to assure the vehicle will not leave the track. The controller highly depends on the quality of the localization because it will try to reduce the difference between the estimated state and the planned state. If the localization is not accurate enough, the controller will not be able to reduce the error and the vehicle might leave the track. In the worst case, the SLAM algorithm might even diverge, which will inevitably lead to an emergency break manoeuver and no points scored in that particular run.

In order to assure high map and localization quality, it is essential that the best possible algorithm for the specific use case is chosen. With this in mind, this thesis aims to answer the following question:

Which approach to the SLAM problem results in the best performance in the context of an autonomous Formula Student race car?

To answer this question, it must be established what ”best performance” actually means and how it can be measured. For this purpose a benchmarking tool will be implemented.

As for the SLAM algorithm itself, two of the most prominent approaches will be implemented. A more classical approach based on the Kalman Filter and a more recent solution, a graph based approach. It must be ensured that the algorithms are performing in real-time and have the proper interfaces to communicate and work together with the other modules of the autonomous pipeline.

If these prerequisites have been fulfilled, the algorithms can then be tested and evaluated in order to answer the question above.

1.3 Outline

In **Chapter 2**, related work that this thesis is built upon is presented. This includes publications that address topics such as the SLAM approaches used in this thesis or related publications by other Formula Student teams.

Chapter 3 presents the necessary theoretical background.

Boundary conditions that influence the SLAM algorithms, such as the various disciplines of Formula Student Driverless, the base car used to record the analyzed data sets, or the autonomous pipeline, are presented in **Chapter 4**.

Chapter 5 addresses the modeling of the system, relevant to both of the SLAM algorithms, such as the used reference frames or the motion and measurement models.

Chapters 6 and **7** discuss the SLAM algorithms - EKF SLAM and GraphSLAM - and their respective implementation.

The method used to evaluate the algorithm's performance is presented in **Chapter 8**, followed by the results from evaluating the algorithms presented in Chapters 6 and 7.

Finally, in **Chapter 9**, the conclusion from these results is drawn and future work on the subject is discussed.

2 Related Work

2.1 SLAM in General

SLAM is not a new problem. It dates back to the late 1980s, when Durrant-Whyte [2] and Smith and Cheesman [1] started laying the foundation for this field of research by introducing methods to describe robots in a probabilistic way. A few years later, Smith, Cheesman and Self presented representations for maps containing spatial, uncertain information in the form of landmarks that are revised incrementally [3]. In 1991, Leonard and Durrant-Whyte identified mapping and localization as a fundamental problem of mobile robotics [4]. It was recognized that the vehicle state and all landmark positions would indeed be correlated. Therefore, localization and mapping would have to be formulated as a combined problem.

Representing this break-through, the term *SLAM* was established by Durrant-Whyte et al. in 1995 [5]. However, because all early approaches to the SLAM problem were based on a Kalman Filter, combining the problem meant that the state vector would have to include all landmarks as well as the vehicle state. Various approaches aim at reducing the computational complexity of filter based approaches resulting from such a big state vector. Bailey and Durrant-Whyte give a great overview of the methods used to do so [16]. State augmentation, for example, is a technique where the structure of the problem is used to only compute small blocks of the state vector and covariance matrix when applying the motion model [11]. State augmentation will deliver optimal results and is used in one of the algorithms presented in this thesis. Another method to reduce the algorithm's complexity is presented by Thrun et al., who take advantage of the sparse structure of the problem when it is transformed into information space by neglecting close to zero elements of the information matrix [13]. However, the problems the algorithms in this thesis are faced with consist of less than 1,000 landmarks. Techniques that deviate from the optimal solution are not needed, so they are not further discussed hereafter.

A different approach to the SLAM problem is based on smoothing instead of filtering. Lu and Milios were the first to formulate the SLAM problem using constraints to connect robot poses and to formulate a global optimization problem from these constraints in 1997 [7]. However, the computational resources available at the time limited the use for such algorithms because of the high complexity they introduced. Since then, impressive progress has been made in modern computing as well as on the field of sparse linear algebra, which is why approaches based on smoothing are popular more than ever nowadays and are referred to as state-of-the-art techniques for solving the SLAM problem [20]. In 2006 Thrun and Montemerlo introduced the GraphSLAM algorithm [18] based on the work of Lu and Milios. One of the algorithms presented in this thesis is largely based on this approach.

2.2 SLAM in Driverless Racing

Choosing the best possible approach to solve the SLAM problem is very dependent on context. External factors, such as the number and type of available sensors or the size of the environment that is going to be mapped, may rule out certain algorithms or favor others. This thesis aims at finding the best approach for a Formula Student Driverless vehicle. This implies some extreme requirement due to the fact that it is a racing application and at the same time relaxes other requirements because the setting is very well defined.

Formula Student Driverless is a fairly new competition. In Germany it is was introduced in 2017. But since it is a sport performed exclusively by students, there are a number of publications discussing various parts of the autonomous system of these race cars. Zeilinger et al. [27] and Valls et al. [30] present the overall autonomous system of their respective vehicles. Both teams utilize SLAM algorithms based on an Extended Kalman Filter (EKF) and while they give a good overview over the building blocks needed for autonomous racing, they only touch SLAM briefly and do not evaluate different approaches. Nekkah et al. present the software stack of the autonomous vehicle that this thesis is based on [42]. However, the presented SLAM algorithm has since been improved. For the purposes if this thesis, it serves as a base line for the experimental evaluations made in Chapter 8.

SLAM evaluations have been performed in the context of Formula Student Driverless: Kabzan et al. use a fastSLAM algorithm and provide data of an evaluation of the algorithm [36]. Anderson and Baerveldt chose and discussed another algorithm, ORB-SLAM2 [28]. Wahlqvist implemented an EKF SLAM algorithm on a modified remote controlled car to test different motion priors with an application on a Formula Student Driverless car in mind [39]. Similarly, Brunnbauer and Bader implemented and evaluated an EKF based localization algorithm on a 1:10 car [31]. However, no concrete comparison to a different SLAM algorithm is made in any of these references.

Outside of Formula Student, SLAM approaches are evaluated, for example, in the context of Roborace [37] or the DARPA challenge [19]. However, these settings differ quite significantly from the Formula Student application that this thesis covers.

To the knowledge of the author, no comparison of fundamentally different approaches to the SLAM approach have been made in the context of a Formula Student Driverless application. This thesis presents an evaluation of both a filter-based and a smoothing-based SLAM algorithm, based on real-world data collected by a Formula Student Driverless race car. For this, the used algorithms and their respective implementation are introduced in Chapters 6 and 7.

3 Fundamentals

In this chapter, the theoretical background needed to solve the SLAM problem is established. If not stated otherwise, the fundamentals of the SLAM problem and the approaches to solve it are taken from *Probabilistic robotics* [14].

3.1 Basic Concepts

3.1.1 Uncertainty

All algorithms presented in this thesis are based on probabilistic approaches. These approaches have proven to be more robust and to have weaker requirements on the accuracy of a robot's[1] model or the robot's sensors.

This is due to the fact that any real world application of a robot is faced with uncertainties. For example, sensor input will never be completely accurate and even the robot's movement is uncertain. A control input to make the robot move somewhere can only be as accurate as the motors that perform this movement. Additionally, external influences such as slip or dynamically changing tire diameters will increase the uncertainty of the motion model that is being used. Because of this, many robotics applications rely on estimating a probability density function, which allows the robot to use the received input over time as well as implemented models to continuously improve its estimates.

Probabilistic algorithms come at the price of higher computational complexity when compared to their more traditional counterparts because instead of computing a single guess, they try to find a complete probability density. However, recent improvements made in computer hardware allow for tremendous performance at a very low cost making probabilistic algorithms more relevant than ever.

3.1.2 Probability Theory

This section presents some basic probabilistic concepts and their respective mathematical notation. Many variables used in this thesis are modeled as **random variables** that can take on multiple values depending on probabilistic laws. A random variable X could take on a specific value x. The probability of this outcome can be denoted as:

$$p(X = x) \tag{3.1}$$

The sum of the probability of all possible values of the random variable must sum up to 1. For discrete probability functions, this is expressed by:

$$\sum_{x} p(X = x) = 1 \tag{3.2}$$

[1] A robot is per definition a machine that is capable of handling complicated series of tasks automatically [44]. This definition applies to the race car that is referred to throughout this thesis, so "robot" and "vehicle" or "car" will be used synonymously.

Analogously, it holds for continuous functions that:

$$\int p(X = x)dx = 1 \tag{3.3}$$

To simplify the notation, the random variable is usually omitted, so $p(X = x)$ is abbreviated by $p(x)$.

Most random variables in this thesis will be described by continuous **probability density functions**. A very common density function is the one-dimensional **normal distribution** or **Gaussian distribution** with a mean μ and a variance σ^2:

$$p(x) = (2\pi\sigma^2)^{-\frac{1}{2}} \, exp\left(-\frac{1}{2}\frac{(x-\mu)^2}{\sigma^2}\right) \tag{3.4}$$

The normal distribution is abbreviated using $\mathcal{N}(x; \mu, \sigma^2)$.

If x is a multi-dimensional vector, the **multivariate** normal distribution is used. It is defined like this:

$$p(x) = det(2\pi\Sigma)^{-\frac{1}{2}} \, exp\left(-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)\right) \tag{3.5}$$

μ is the mean vector, while Σ represents the **covariance matrix** which is a symmetric and positive semidefinite matrix.

Alternatively, a multivariate Gaussian distribution can also be represented in **canonical parametrization** using the **information matrix** Ω and the **information vector** ξ. These parameters are defined as

$$\Omega = \Sigma^{-1} \tag{3.6}$$

and

$$\xi = \Sigma^{-1}\mu \tag{3.7}$$

The canonical parametrization of a Gaussian can be obtained by multiplying out the exponent in Equation (3.5):

$$p(x) = det(2\pi\Sigma)^{-\frac{1}{2}} \, exp\left(-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)\right) \tag{3.8}$$

$$= det(2\pi\Sigma)^{-\frac{1}{2}} \, exp\left(-\frac{1}{2}x^T\Sigma^{-1}x + x^T\Sigma^{-1}\mu - \frac{1}{2}\mu^T\Sigma^{-1}\mu\right) \tag{3.9}$$

$$= det(2\pi\Sigma)^{-\frac{1}{2}} \, exp\left(-\frac{1}{2}\mu^T\Sigma^{-1}\mu\right) \, exp\left(-\frac{1}{2}x^T\Sigma^{-1}x + x^T\Sigma^{-1}\mu\right) \tag{3.10}$$

When using η as a normalizer to combine all constants in the equation above, the Gaussian can be represented by:

$$p(x) = \eta \, exp\left(-\frac{1}{2}x^T\Sigma^{-1}x + x^T\Sigma^{-1}\mu\right) \tag{3.11}$$

$$= \eta \, exp\left(-\frac{1}{2}x^T\Omega x + x^T\xi\right) \tag{3.12}$$

The **joint distribution** of two random variables X and Y is given by:

$$p(x, y) = p(X = x \text{ and } Y = y) \tag{3.13}$$

X and Y are **independent** random variables if:

$$p(x, y) = p(x)p(y) \tag{3.14}$$

A probability

$$p(x \mid y) = p(X = x \mid Y = y) \tag{3.15}$$

is called **conditional probability** and is defined as:

$$p(x \mid y) = \frac{p(x, y)}{p(y)} \quad \forall \quad p(y) > 0 \tag{3.16}$$

If X and Y are independent, the following relation can be concluded from Equations (3.14) and (3.16):

$$p(x \mid y) = \frac{p(x)p(y)}{p(y)} = p(x) \quad \forall \quad p(y) > 0 \tag{3.17}$$

This means that if X and Y are independent, knowing about Y carries no information if X is of interest.

A consequence of this is the so called **Theorem of total probability** for discrete values:

$$p(x) = \sum_{y} p(x \mid y)p(y) \tag{3.18}$$

Analogously, for continuous values:

$$p(x) = \int_{y} p(x \mid y)p(y)dy \tag{3.19}$$

Another important result is the **Bayes rule**:

$$p(x \mid y) = \frac{p(y \mid x)p(x)}{p(y)} \tag{3.20}$$

Since $p(y)$ does not depend on x, it is the same for all possible values x. Therefore, the notation of Equation (3.20) is often simplified by replacing $p(y)^{-1}$ by a normalizer η:

$$p(x \mid y) = \eta p(y \mid x)p(x) \tag{3.21}$$

The Bayes rule will play an important role in this thesis. If x is to be inferred from y, then $p(x)$ will be referred to as **prior probability distribution**, and y is called **data**. After incorporating y, the distribution $p(x \mid y)$ is referred to as **posterior probability distribution**. Bayes rule allows this to be flipped, i.e. to infer y assuming that x was the case. The probability $p(y \mid x)$ is often called the **generative model** because it describes how state variables X cause sensor measurements Y.

The rules presented thus far can be conditioned on arbitrary variables, for example a variable Z. Conditioning the Bayes rule on $Z = z$ results in:

$$p(x, y \mid z) = \frac{p(y \mid x, z)p(x \mid z)}{p(y \mid z)} \quad \forall \quad p(y \mid z) > 0 \tag{3.22}$$

For independent values, conditioning Equation (3.14) on another variable z results in a property called **conditional independence**:

$$p(x, y \mid z) = p(x \mid z)p(y \mid z) \tag{3.23}$$

This is equivalent to:

$$p(x \mid z) = p(x \mid z, y) \tag{3.24}$$

$$p(y \mid z) = p(x \mid z, x) \tag{3.25}$$

It is important to note that conditional independence does not generally imply absolute independence and vice versa:

$$p(x, y \mid z) = p(x \mid z)p(y \mid z) \implies p(x, y) = p(x)p(y) \tag{3.26}$$

$$p(x, y) = p(x)p(y) \implies p(x, y \mid z) = p(x \mid z)p(y \mid z) \tag{3.27}$$

3.1.3 State, Map and Environment Interaction

State

The vehicle and its surroundings can be described by the state. The state is referred to as x, the state at a specific time t is denoted as x_t. The contents of the state depend on the specific context and may include different variables. In this thesis, the vehicle pose as well as landmark locations are frequently used:

The **vehicle pose** consists of the location and the orientation of the vehicle in a given reference frame. In three dimensions there are three Cartesian coordinates and three angles, so six variables in total. However, the algorithms described in this thesis operate in two dimensions where there are only two Cartesian coordinates x_v and y_v and one angular component φ_v that can be combined in a vehicle pose vector:

$$x = \begin{pmatrix} x_v \\ y_v \\ \varphi_v \end{pmatrix} \tag{3.28}$$

Note that in this case x_v is a scalar describing a Cartesian coordinate and not a state vector.

Alternatively, the vehicle pose could be represented as a **special Euclidean group** $SE(n)$ with n being the number of dimensions. This representation has the benefit of performing a transformation of an object with a single operation. In two dimensions the same vehicle pose can be represented as:

$$x = \begin{pmatrix} R & t \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} \cos\varphi_v & -\sin\varphi_v & x_v \\ \sin\varphi_v & \cos\varphi_v & y_v \\ 0 & 0 & 1 \end{pmatrix} \tag{3.29}$$

Here, R is a rotational matrix and t is a translational vector.

The **environment** surrounding the robot can also be captured in the state. There are different ways to describe the environment (see Section 3.1.3). One such way is to use landmarks. Landmarks are stationary, distinct features, well suited to be used for orientation. For example, in two dimensions the state of the environment could be described by two Cartesian coordinates. This assumes the orientation of a landmark to be irrelevant. The resulting state could be described as:

$$x = \begin{pmatrix} x_1 \\ y_1 \\ x_2 \\ y_2 \\ \vdots \end{pmatrix} \tag{3.30}$$

In this case, x_j and y_j represent the position of the j-th landmark.

There are of course many more potential state variables, since the description of the robot and its environment highly depends on the specific use case. Possibilities range from binary states describing a sensor's status to states containing point clouds describing the environment received from range scanners. However, none of these will be needed in this thesis, so they are not further discussed.

Map

Whether or not a map is given in advance, the map needs to be represented in a way the vehicle can process. A map m is a list of objects together with their respective properties that can be indexed in different ways:

- **Feature-based** maps use an index to describe a specific feature on the map. The content of the mapped object includes its Cartesian location.
- In **location-based** maps, the index corresponds to a location in the world reference frame.

The latter are also referred to as volumetric maps, as they contain information about any location in the world. This also includes information about the absence of an object. This is in contrast to feature-based maps where the environment is only known at the mapped features. However, this makes it easier to adjust a map, for example when a new sensor measurement improves the location estimate of a feature. Additionally, this representation is more computationally efficient because it requires less data to store a map.

Choosing the correct type of map is a matter of context and mostly depends on the kind of environment the robot is in as well as the requirements made towards the efficiency of the algorithm using this map. In this thesis, the environment is built from well distinguishable features that will be chosen as landmarks. The requirements towards real-time capability, and therefore efficiency, are very high. Because of this, feature-based maps are used throughout this thesis.

Equation (3.30) is a possible representation for this kind of map. However, a map may contain additional information about the features in it to simplify the recognition of that feature. This includes, for example, the orientation, size or color.

Environment Interaction

The robot interacts with the environment in different ways resulting in different information that is available to the robot. There are two contrasting types of interactions:

Measurements using environment sensors allow the robot to obtain information about the current state of the environment it is in. These sensors include cameras or range sensors and the incoming data will be referred to as *measurement* or *observation*. The measurements will be assumed to be received at discrete time steps neglecting for example time differences in a range scan due to a rotating sensor. A measurement at the time t will be denoted z_t. Observations usually increase the knowledge of the environment and therefore decrease a robot's uncertainty. Sequences of observations are denoted as:

$$z_{t1:t2} = z_{t1}, z_{t1+1}, \ldots, z_{t2} \text{ for } t_1 \leq t_2 \tag{3.31}$$

In contrast, the robot can use its actuators to influence the world. These operations are referred to as **control actions**. For example, the robot could assert forces to move objects in the environment. However, in this thesis the control input that is used to move the robot itself is of more relevance. Control actions are performed between two time steps, so a control input in the time interval $(t-1;t]$ is referred to as u_t. Sequences of control inputs are denoted analogously to Equation 3.31. These operations change the state and since the robot's actuators are noisy and the environment might also change due to external factors, the robot's uncertainty usually increases.

A typical control input in robots would be the robot's velocity. However, since the velocity is usually a result of the controller rather than the manipulated variable itself, the velocity can alternatively be measured and then interpreted as a control input. Sensors to gather information about the velocity could be wheel encoders (sensors that count wheel revolutions) or an inertia measurement unit. These sensors are referred to as **odometry sensors** and deliver **odometry data**.

For convenience, the robot is assumed to deliver a control input at every time step. If the state does not actually change between the times $t-1$ and t, this is done by setting the control u_t to zero.

It is assumed that the robot first executes the control input u_t and then receives the measurement z_t.

Marcov Assumption

A state x_t is called **complete** if information about past states, controls or measurements do not improve the ability to predict the future state. This does not require the future to be a deterministic function of the state, but it means that variables prior to the current state do not influence the stochastic evolution of the future state.

The task of an algorithm might be to compute the current state from previous states, controls and measurements. If a state is complete, this task can be reduced like this:

$$p(x_t \mid x_{0:t-1}, z_{1:t-1}, u_{1:t}) = p(x_t \mid x_{t-1}, u_t) \tag{3.32}$$

Similarly, the process by which measurements are generated can be reduced, too:

$$p(z_t \mid x_{0:t}, z_{1:t-1}, u_{1:t}) = p(z_t \mid x_t) \tag{3.33}$$

This means that a future state is only dependent on the state before and the control input in the transition between these two time steps. A measurement at time t only depends on the state at the same time. These properties are examples of **conditional independence**. A system of such conditionally independent stochastic variables is fulfilling the **Marcov assumption**. The Marcov assumption or Marcov condition describes a directed graph where a child node is only depending on its direct parent [8].

Equation (3.32) is referred to as **state transition probability**, while Equation (3.33) is called **measurement probability**. Note that these equations are probabilistic functions rather than deterministic ones.

These two equations together describe a dynamic stochastic system of the robot and its surroundings that is referred to as **hidden Marcov model** or **dynamic Bayes network**. Figure 3.1 visualizes such a network.

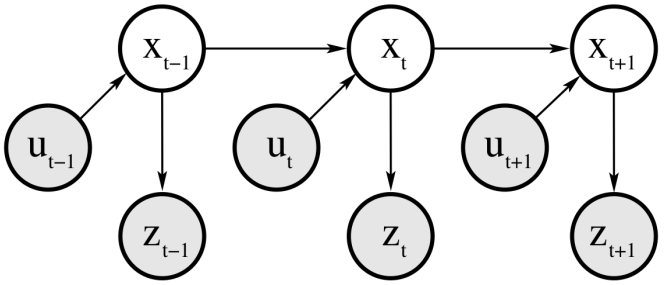


Figure 3.1: A dynamic Bayes network.

As with many theoretical assumptions, the Marcov assumption is often violated in reality. For example, there might be dynamics in the environment that are not represented in the state or the probability equations described above might be inaccurate. It would of course be possible to improve the model by, for example, adding more variables to the state in order for the description of the world to become closer to reality. However, such a model would still not fully represent reality and would increase the complexity of the system. So in practice a trade off is made in order to have systems which are real-time capable.

Prediction and Correction

A state x_t cannot be measured directly but has to be concluded from the measurements and control inputs. It is therefore distinguished between the actual state x_t and the robot's internal **belief** $bel(x_t)$. A belief can be expressed by the conditional probability:

$$bel(x_t) = p(x_t \mid z_{1:t}, u_{1:t}) \tag{3.34}$$

If the current measurement z_t is not yet incorporated into the belief, such a posterior is often referred to as **prediction** and is denoted like this:

$$\overline{bel}(x_t) = p(x_t \mid z_{1:t-1}, u_{1:t}) \tag{3.35}$$

Computing $bel(x_t)$ from $\overline{bel}(x_t)$ is called **correction** or **measurement update** because it corrects the predicted belief using the current measurement.

3.1.4 Bayes Filter

A very general approach to calculating the belief is the Bayes filter algorithm. It recursively computes the belief $bel(x_t)$, given the belief from the previous time step $bel(x_{t-1})$ as well as the measurement z_t and the control action executed by the robot u_t. In other words, the algorithm tries to recursively find the probability distribution given by Equation (3.34). For this, the state is assumed to be complete. The algorithm is derived by first applying the Bayes rule described in Equation 3.20:

$$\begin{aligned} p(x_t \mid z_{1:t}, u_{1:t}) &= \frac{p(z_t \mid x_t, z_{1:t-1}, u_{1:t}) p(x_t \mid z_{1:t-1}, u_{1:t})}{p(z_t \mid z_{1:t-1}, u_{1:t})} \\ &= \eta p(z_t \mid x_t, z_{1:t-1}, u_{1:t}) p(x_t \mid z_{1:t-1}, u_{1:t}) \end{aligned} \tag{3.36}$$

Exploiting the assumption of a complete state and the resulting conditional independence allows to simplify this equation to:

$$p(x_t \mid z_{1:t}, u_{1:t}) = \eta p(z_t \mid x_t) p(x_t \mid z_{1:t-1}, u_{1:t}) \tag{3.37}$$

Therefore, the belief is calculated as follows:

$$bel(x_t) = \eta p(z_t \mid x_t) \overline{bel}(x_t) \tag{3.38}$$

Next, the predicted belief $\overline{bel}(x_t)$ is expanded using Equation (3.19):

$$\begin{aligned} \overline{bel}(x_t) &= p(x_t \mid z_{1:t-1}, u_{1:t}) \\ &= \int p(x_t \mid x_{t-1}, z_{1:t-1}, u_{1:t}) p(x_{t-1} \mid z_{1:t-1}, u_{1:t}) dx_{t-1} \end{aligned} \tag{3.39}$$

Again, assuming a complete state results in the following simplification:

$$p(x_t \mid x_{t-1}, z_{1:t-1}, u_{1:t}) = p(x_t \mid x_{t-1}, u_t) \tag{3.40}$$

Additionally, u_t can be omitted in $p(x_{t-1} \mid z_{1:t-1}, u_{1:t})$ because it does not influence the state x_{t-1}. Therefore, Equation (3.39) can be reduced to:

$$\overline{bel}(x_t) = \int p(x_t \mid x_{t-1}, u_t) p(x_{t-1} \mid z_{1:t-1}, u_{1:t-1}) dx_{t-1} \tag{3.41}$$

Combining Equations (3.38) and (3.41) results in the Bayes filter algorithm.

Algorithm 1: Bayes filter

Input: $bel(x_{t-1}), u_t, z_t$

Output: $bel(x_t)$

1 **forall** x_t **do**
2 $\quad \overline{bel}(x_t) = \int p(x_t \mid x_{t-1}, u_t) p(x_{t-1} \mid z_{1:t-1}, u_{1:t-1}) dx_{t-1}$
3 $\quad bel(x_t) = \eta p(z_t \mid x_t) \overline{bel}(x_t)$
4 **end**
5 **return** $bel(x_t)$

The Bayes filter, described in Algorithm 1, is split into two steps. In line 2, a new control input u_t is processed to compute the **prediction**. In the second step, expressed by line 3 of the algorithm, the algorithm processes a new measurement to correct the prediction made before, hence the name **correction**.

Since the algorithm works recursively, an initial state has to be defined. This is usually done in one of two ways: Either the state is very well know, for example because it is defined that way. Then the probability is centered around that value while assigning a zero probability everywhere else. Or the state is completely unknown, in which case a uniform distribution over the domain of the initial value is used.

3.1.5 The Kalman Filter

One of the earliest and most famous implementations of the Bayes filter is the Kalman filter. The equations used in Algorithm 1 cannot be solved in a closed form for any state transition or measurement equation. However, the Kalman filter, like all algorithms in the family of the **Gaussian filters**, assumes a (multivariate) Gaussian distribution for the belief, which does make it possible to find a closed form solution.

For the belief to be Gaussian, the Kalman filter requires some assumptions on top of the Marcov assumption. First of all, the initial belief $bel(x_0)$ must be Gaussian. Using an initial mean μ_0 and an initial covariance Σ_0, according to Equation (3.5), the following formulation of the initial belief can be used:

$$bel(x_0) = p(x_0) = det(2\pi\Sigma_0)^{-\frac{1}{2}} \exp\left(-\frac{1}{2}(x_0 - \mu_0)^T {\Sigma_0}^{-1}(x_0 - \mu_0)\right) \tag{3.42}$$

Additionally, the state transition probability $p(x_t \mid u_t, x_{t-1})$ and the measurement probability $p(z_t \mid x_t)$ must be linear in its arguments, with added Gaussian noise. The former is represented by:

$$x_t = A_t x_{t-1} + B_t u_t + \epsilon_t \tag{3.43}$$

The latter is expressed with the following equation:

$$z_t = C_t x_t + \delta_t \tag{3.44}$$

Here, x_{t-1} and x_t are state vectors, u_t is the control vector, and z_t is the measurement vector at time t. All these vectors will be expressed as vertical vectors of the form:

$$x_t = \begin{pmatrix} x_{1,t} \\ x_{2,t} \\ \vdots \\ x_{n,t} \end{pmatrix}, \quad u_t = \begin{pmatrix} u_{1,t} \\ u_{2,t} \\ \vdots \\ u_{m,t} \end{pmatrix}, \quad and \quad z_t = \begin{pmatrix} z_{1,t} \\ z_{2,t} \\ \vdots \\ z_{k,t} \end{pmatrix} \tag{3.45}$$

n is the number of states, m is the dimension of the control vector and k represents the number of measurements made in this particular time step.

A_t, B_t and C_t are matrices of the sizes $n \times n$, $n \times m$, and $k \times n$, respectively. Using these matrices, the state transition function and the measurement function become linear.

ϵ_t and δ_t in Equations (3.43) and (3.44) are Gaussian random vectors modeling the uncertainties of the state transition and the measurement. They are zero mean with a covariance represented by Q_t and R_t, respectively.

Plugging Equations (3.43) and (3.44) into the definition of a multivariate Gaussian distribution as described in Equation (3.5) results in the state transition and measurement equations in the following form:

$$p(x_t \mid u_t, x_{t-1}) = det(2\pi Q_t)^{-\frac{1}{2}} \exp\left(-\frac{1}{2}(x_t - A_t x_{t-1} - B_t u_t)^T {Q_t}^{-1}(x_t - A_t x_{t-1} - B_t u_t)\right) \tag{3.46}$$

$$p(z_t \mid x_t) = det(2\pi R_t)^{-\frac{1}{2}} \exp\left(-\frac{1}{2}(z_t - C_t x_t)^T {R_t}^{-1}(z_t - C_t x_t)\right) \tag{3.47}$$

The Kalman Filter Algorithm

These assumptions used on the Bayes filter described in Algorithm 1 result in the Kalman filter algorithm[2]:

Algorithm 2: Kalman Filter

Input: $\mu_{t-1}, \Sigma_{t-1}, u_t, z_t$

Output: μ_t, Σ_t

1 $\bar{\mu}_t = A_t \mu_{t-1} + B_t u_t$
2 $\overline{\Sigma}_t = A_t \Sigma_{t-1} A_t^T + Q_t$
3 $K_t = \overline{\Sigma}_t C_t^T (C_t \overline{\Sigma}_t C_t^T + R_t)^{-1}$
4 $\mu_t = \bar{\mu}_t + K_t (z_t - C_t \bar{\mu}_t)$
5 $\Sigma_t = (I - K_t C_t) \overline{\Sigma}_t$
6 **return** μ_t, Σ_t

As described above, the Kalman filter represents the belief using a Gaussian distribution, i.e. by the mean μ_t and the covariance Σ_t. The input to the algorithm is the belief of the previous time step $t-1$ represented by μ_{t-1} and Σ_{t-1} as well as the control input u_t and the measurements z_t of the current time step.

First, the predicted belief $\overline{bel}(x_t)$ is calculated and represented by $\bar{\mu}_t$ and $\overline{\Sigma}_t$. This is done using the state transition Equation (3.43) with μ_{t-1} substituted for x_{t-1} to incorporate the control input u_t. Additionally, the covariance is adjusted to account for the increased uncertainty due to the state transition.

In the following lines the measurements are incorporated to together form the correction step. For this, the **Kalman gain** K_t is computed. The Kalman gain can be seen as a measure that describes to what extent the measurement is trusted and therefore incorporated into the predicted state. This can be seen in line 4 where K_t acts as a weight between the actual measurements z_t and the **expected measurements** $C_t \bar{\mu}_t$. Finally, in line 5 the covariance is adjusted to account for the added information given by the measurements.

The Kalman algorithm then returns the new belief $bel(x_t)$ represented by μ_t and Σ_t.

The efficiency of the algorithm is dominated by either line 3 or line 5 depending on the specific use case. Line 3 requires an inversion of a $k \times k$ matrix. With the most efficient algorithms this can be done in $O(k^{2.4})$. However, the Kalman-based SLAM algorithm presented later in this thesis consists of hundreds of states and is therefore typically rather limited by the multiplication of the $n \times n$ matrices in line 5 resulting in an $O(n^2)$ complexity.

The Extended Kalman Filter

While the Kalman filter itself works very efficiently, the linear algorithm described above has a very limited field of use, as it can only be used with very simple systems.

[2] A complete mathematical derivation of the Kalman Filter can be found in section 3.2.4 of *Probabilistic robotics* [14].

Reality is highly non-linear, so the EKF loosens the linearity assumption using a nonlinear state transition function g and a non-linear measurement function h:

$$x_t = g(u_t, x_{t-1}) + \epsilon_t \tag{3.48}$$
$$z_t = h(x_t) + \delta_t \tag{3.49}$$

g replaces A_t and B_t used in Equation (3.43) and h replaces C_t in Equation (3.44). However, using these functions directly will result in a non-Gaussian belief. To include nonlinear functions into the Kalman filter, the EKF approximates the true belief by a Gaussian belief using a first order **Taylor expansion**. For the state transition, g is linearized around the mean μ_{t-1}:

$$\begin{aligned} g(u_t, x_{t-1}) &\approx g(u_t, \mu_{t-1}) + \frac{\partial g(u_t, \mu_{t-1})}{\partial \mu_{t-1}}(x_{t-1} - \mu_{t-1}) \\ &= g(u_t, \mu_{t-1}) + G_t(x_{t-1} - \mu_{t-1}) \end{aligned} \tag{3.50}$$

Similarly, h is approximated around the predicted mean $\bar{\mu}_t$:

$$\begin{aligned} h(x_t) &\approx h(\bar{\mu}_t) + \frac{\partial h(\bar{\mu}_t)}{\partial \bar{\mu}_t}(x_t - \bar{\mu}_t) \\ &= h(\bar{\mu}_t) + H_t(x_t - \bar{\mu}_t) \end{aligned} \tag{3.51}$$

Here, G_t and H_t represent the "slope" of the functions g and h and are referred to as **Jacobians**.

Using these linear approximations, the state transition and measurement equation can be formulated as Gaussian distributions:

$$\begin{aligned} p(x_t \mid u_t, x_{t-1}) = \; & det(2\pi Q_t)^{-\frac{1}{2}} \exp\Bigg(-\frac{1}{2}[x_t - g(u_t, \mu_{t-1}) - G_t(x_{t-1} - \mu_{t-1})]^T \\ & {Q_t}^{-1}[x_t - g(u_t, \mu_{t-1}) - G_t(x_{t-1} - \mu_{t-1})]\Bigg) \end{aligned} \tag{3.52}$$

$$\begin{aligned} p(z_t \mid x_t) = \; & det(2\pi R_t)^{-\frac{1}{2}} \exp\Bigg(-\frac{1}{2}[z_t - h(\bar{\mu}_t) - H_t(x_t - \bar{\mu}_t)]^T \\ & {R_t}^{-1}[z_t - h(\bar{\mu}_t) - H_t(x_t - \bar{\mu}_t)]\Bigg) \end{aligned} \tag{3.53}$$

Using these equations and the Kalman filter algorithm described in Algorithm 2, the EKF algorithm (Algorithm 3) can be obtained.

3.2 Graph Theory

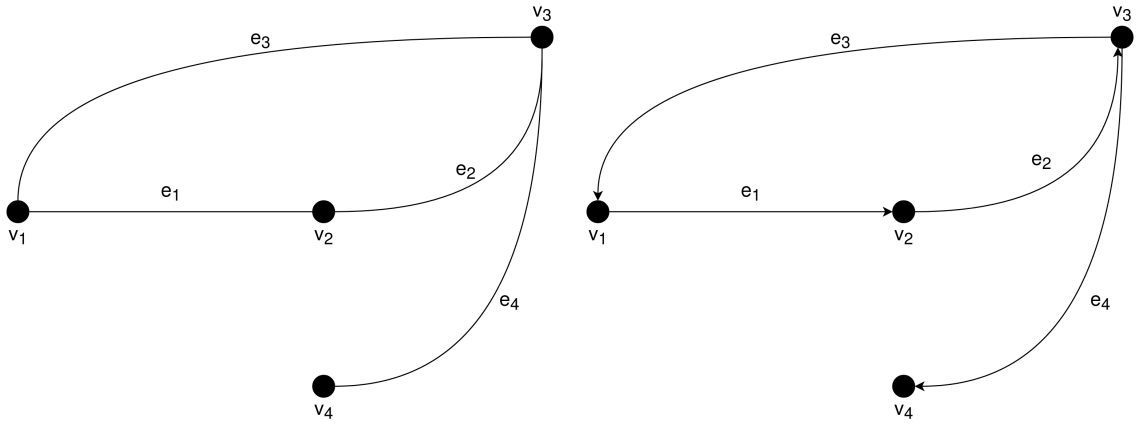
A graph in the sense of discrete mathematics is a structure of objects where some pairs of these objects may be related to one another. The GraphSLAM algorithm presented later in this thesis is based on representing the SLAM problem in such a structure, therefore it is introduced here. The foundation for this section is taken from [9].

Algorithm 3: Extended Kalman Filter

Input: $\mu_{t-1}, \Sigma_{t-1}, u_t, z_t$

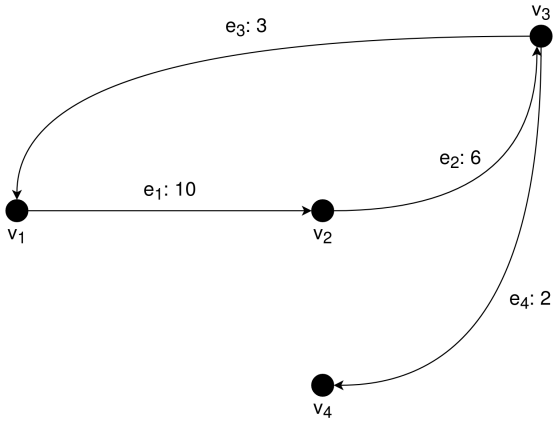
Output: μ_t, Σ_t

1 $\bar{\mu}_t = g(u_t, \mu_{t-1})$
2 $\overline{\Sigma}_t = G_t \Sigma_{t-1} G_t^T + Q_t$
3 $K_t = \overline{\Sigma}_t H_t^T (H_t \overline{\Sigma}_t H_t^T + R_t)^{-1}$
4 $\mu_t = \bar{\mu}_t + K_t(z_t - h(\bar{\mu}_t))$
5 $\Sigma_t = (I - K_t H_t)\overline{\Sigma}_t$
6 **return** μ_t, Σ_t



(a) A graph consisting of 4 vertices and 4 edges.

(b) A directed graph consisting of 4 vertices and 4 edges.



(c) A weighted, directed graph consisting of 4 vertices and 4 edges.

Figure 3.2: Graph examples.

3.2.1 Definitions

A graph G is a mathematical construct consisting of a set of **vertices** (or **nodes**) $V(G)$, a set of **edges** $E(G)$ and relations between these two sets, i.e. an edge is associated with two vertices called its endpoints.

A **directed graph** H consists of a set of vertices $V(G)$, a set of edges $E(G)$ and an **ordered pair of vertices** for each edge. The first vertex of the ordered pair is called the tail of the edge, the second vertex is the head.

A **weighted graph** J consists of a set of vertices $V(G)$, a set of edges $E(G)$ and a function assigning a **weight** to each edge. A weighted graph may also be directed.

Figure 3.2 shows an example for these a types of graphs.

3.2.2 Matrix Representation

Instead of listing all edges and vertices, a graph can be represented using an **adjacency matrix**. Let G be a graph with a set of vertices $V(G) = v_1, \ldots, v_n$ and a set of edges $E(G) = e_1, \ldots, e_m$. The adjacency matrix $A(G)$ is a $n \times n$ matrix, in which each entry $a_{i,j}$ is the number of edges of G with the endpoints v_i, v_j. The adjacency matrix is always symmetric.

The adjacency matrix for the graph depicted in Figure 3.2a is:

$$A(G) = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \tag{3.54}$$

For weighted graphs, an entry of the adjacency matrix $a_{i,j}$ contains the sum of weights associated with all edges with the endpoints v_i, v_j. If v_i and v_j are not connected, $a_{i,j}$ is zero. The adjacency matrix $A(J)$ for the weighted graph in Figure 3.2c is:

$$A(J) = \begin{pmatrix} 0 & 10 & 3 & 0 \\ 10 & 0 & 6 & 0 \\ 3 & 6 & 0 & 2 \\ 0 & 0 & 2 & 0 \end{pmatrix} \tag{3.55}$$

3.3 Least Squares Optimization

The following section quickly introduces the idea behind least squares optimization. For more details, [21] is recommended.

The GraphSLAM algorithm presented in this thesis formulates SLAM as an optimization problem. It is solved by minimizing a cost function built by a sum of constraints:

$$F(x) = \sum_{i,j} e(x_i, x_j, z_{ij})^T \Omega_{ij} e(x_i, x_j, z_{ij}) \tag{3.56}$$

$$x^* = \underset{x}{\operatorname{argmin}}\, F(x) \tag{3.57}$$

x is a state vector where x_i and x_j each represent a block of that state. They are connected by a constraint with the mean z_{ij} and the information matrix Ω_{ij}. x^* is the state where the cost function is minimal. $e(x_i, x_j, z_{ij})$ describes an error function measuring how well that constraint is satisfied by x_i and x_j. For simplicity, the error function is written as:

$$e(x_i, x_j, z_{ij}) = e_{ij}(x) \tag{3.58}$$

Various optimization techniques are available, with Gauss-Newton and Levenberg-Marquardt being some of the most popular. The idea behind both of these algorithms is to approximate the error function by its first order Taylor expansion around an initial guess $\tilde{x}$:

$$e_{ij}(\tilde{x} + \Delta x) \approx e_{ij}(\tilde{x}) + J_{ij}\Delta x \tag{3.59}$$

J_{ij} is the Jacobian of e_{ij} computed in $\tilde{x}$. This is used to approximate a constraint in Equation (3.56):

$$\begin{aligned} F_{ij}(\tilde{x} + \Delta x) &= e_{ij}(\tilde{x} + \Delta x)^T \Omega_{ij} e_{ij}(\tilde{x} + \Delta x) && (3.60)\\ &\approx (e_{ij}(\tilde{x}) + J_{ij}\Delta x)^T \Omega_{ij}(e_{ij}(\tilde{x}) + J_{ij}\Delta x) && (3.61)\\ &= \underbrace{e_{ij}^T \Omega_{ij} e_{ij}}_{c_{ij}} + 2\underbrace{e_{ij}^T \Omega_{ij} J_{ij}}_{b_{ij}}\Delta x + \Delta x^T \underbrace{J_{ij}^T \Omega_{ij} J_{ij}}_{H_{ij}}\Delta x && (3.62)\\ &= c_{ij} + 2b_{ij}\Delta x + \Delta x^T H_{ij}\Delta x && (3.63) \end{aligned}$$

The full cost function in Equation (3.56) is then approximated:

$$\begin{aligned} F(\tilde{x} + \Delta x) &= \sum_{i,j} F_{ij}(\tilde{x} + \Delta x) && (3.64)\\ &\approx \sum_{i,j} c_{ij} + 2b_{ij}\Delta x + \Delta x^T H_{ij}\Delta x && (3.65)\\ &= c + 2b^T\Delta x + \Delta x^T H \Delta x && (3.66) \end{aligned}$$

Here, $c = \sum c_{ij}$, $b = \sum b_{ij}$ and $H = \sum H_{ij}$.

This quadratic form of the cost function is minimized in Δx by solving the linear system

$$H\Delta x^* = -b \tag{3.67}$$

and adding the increment Δx^* onto the initial guess $\tilde{x}$

$$x^* = \tilde{x} + \Delta x^* \tag{3.68}$$

The Gauss-Newton approach will solve the optimization problem by iterating the linearization in Equation (3.63), the solution in Equation (3.67) and adding the increment in Equation (3.68) until a termination criterion is met (e.g. maximum number of iterations or minimal increment size).

By adding a damping factor λ to Equation (3.67), the Levenberg-Marquardt algorithm controls the convergence to prevent overshooting:

$$(H + \lambda I)\Delta x^* = -b \tag{3.69}$$

Algorithm 4: Levenberg-Marquardt Algorithm [23]

Input: $\tilde{x}$

Output: x^*

```
1  λ = λ_init
2  while !converged do
3  |   H, b = buildLinearSystem(x̃)
4  |   e = error(x̃)
5  |   x̃_old = x̃
6  |   Δx̃ = solve((H + λI)Δx̃ = −b)
7  |   x̃ += Δx̃
8  |   if e < error(x̃) then // Repeat iteration with increased damping
9  |   |   x̃ = x̃_old
10 |   |   λ *= 2
11 |   else // Reduce damping for faster convergence
12 |   |   λ /= 2
13 end
14 x* = x̃
15 return x*
```

The Levenberg-Marquardt algorithm is described in Algorithm 4. First, the damping is initialized in line 1. The system is then linearized in line 3 and the current value of the cost function is determined in line 4. In line 5 the current guess is backed up, in case the iteration needs to be repeated. After solving the linear system and adding the resulting increment onto the current guess in lines 6 and 7, the cost function is evaluated again. If the overall error was reduced by the last step, the damping is decreased to increase the step size and to therefore reduce the time needed for convergence. If the error has increased, the damping is increased and the last iteration is repeated.

By adding a (small) value to the main diagonal of the H-matrix, it is also assured that the linear system always is of **full rank**, which prevents numerical instabilities.

The g^2o-library [21] used for the GraphSLAM algorithm presented in this thesis uses some more advanced techniques to improve the optimization, for example:

- The library utilizes a generalization to solve optimization problems that are not parameterized in Euclidean space but, for example, in special Euclidean groups; see Equation (3.29).
- Additionally, because of the way the cost function is built, the resulting linearized system will have a sparse structure that can be used to solve the optimization problem more efficiently. For more details, as to how the optimization is improved, see [21].

3.4 Data Association

A crucial part of any SLAM algorithm is the so called data association. While it is not part of this thesis, it is part of the algorithms presented here, so it is briefly introduced. The theoretical foundation for this section is taken from [10].

Data association is the process of matching a measured landmark to a mapped landmark in order to perform the correction resulting from this measurement. This is a difficult problem because both the measured and the mapped landmark are not known precisely but are estimated in a probabilistic way. Because of this, the correct match is not always obvious. A wrong association may lead to a divergence of the SLAM algorithm that the data association is used for, which is why it is important to use a robust algorithm for this task.

There are various ways to solve this problem. However, only the algorithm used in this thesis will be presented. The Joint Compatibility Branch and Bound (JCBB) algorithm promises to be very robust while maintaining an efficiency that allows it to be real-time capable, which is why it is used for the given use case.

To solve the data association problem, the JCBB algorithm uses an **interpretation tree**. The tree is built of k levels where k is the number of measurements made in the particular time step. A node at level i contains an interpretation of all measurements above the i-th level. From every node, there are n branches. Here, n is the number of possible matchings, i.e. the number of mapped landmarks plus one additional branch for the possibility that no matching is possible. The task of the data association algorithm is to choose the path through the tree that contains the best interpretations of the measurements. Mathematically, this means finding the **hypothesis** $\mathcal{H}_k$ that pairs most observed landmarks to a mapped landmark.

To determine which branch is actually the one containing the best associations, the local - or individual - compatibility and the global - or joint - compatibility are used.

The **individual compatibility** is computed from the actual measurement z_i and the expected measurement $\hat{z}_i$. The expected measurement in turn is computed using the **inverse measurement equation**:

$$\hat{z}_i = h(x_v, m) \tag{3.70}$$

x_v expresses the current vehicle state, while m is the map. This equation will be further discussed in Section 5.3. From this, the **Mahalanobis distance** of the measured landmark i to the mapped landmark j is computed:

$$D_{ij}^2 = (z_i - \hat{z}_i)^T P_{ij}^{-1} (z_i - \hat{z}_i) \tag{3.71}$$

Here, P_{ij} corresponds to the covariance matrix representing the mapped landmark's uncertainty. An observed and a mapped landmark are deemed compatible depending on the Chi-squared test with confidence level α:

$$D_{ij}^2 < \chi_\alpha^2 \tag{3.72}$$

The **joint compatibility** is computed using the interpretation of all previous associations. The joint innovation $f_{\mathcal{H}_i}$ is calculated like this:

$$f_{\mathcal{H}_i} = \begin{pmatrix} f_{\mathcal{H}_{i-1}} \\ z_i - \hat{z}_i \end{pmatrix} = \begin{pmatrix} z_0 - \hat{z}_0 \\ z_1 - \hat{z}_1 \\ \vdots \\ z_i - \hat{z}_i \end{pmatrix} \tag{3.73}$$

The joint compatibility is calculated as follows:

$$D^2_{\mathcal{H}_i} = f^T_{\mathcal{H}_i} C^{-1}_{\mathcal{H}_i} f_{\mathcal{H}_i} \tag{3.74}$$

$C_{\mathcal{H}_i}$ is the covariance of the joint innovation, computed from the uncertainties of the mapped landmarks. Analogously to the individual compatibility, a joint compatibility is deemed consistent when:

$$D^2_{\mathcal{H}_i} < \chi^2_\alpha \tag{3.75}$$

To increase the algorithm's efficiency, a branch and bound approach is used. A search for the best hypothesis can be stopped if no better hypothesis than the current one is possible. Also, the most promising nodes in the interpretation tree can be explored first. The core of the JCBB algorithm is the recursive function described in Algorithm 5.

Algorithm 5: JCBB

Input: $\mathcal{H}$, i

Output: Best hypothesis

```
1  if i > k then // Current node is leaf node
2      if pairings(H) > pairings(Best) then
3          Best = H
4      end
5  else
6      foreach j in m do // Traverse down branches from current node
7          if individual_compatibility(i, j) and joint_compatibility([H j]) then
8              JCBB([H j]), i + 1) // Accept matching
9          end
10     end
11     if (pairings(H) + size(m) - i > pairings(Best)) then // Better hypothesis
        possible
12         JCBB([H 0], i + 1) // Declare node as unmatched
13     end
14 end
```

Since the algorithm works recursively, it expects a hypothesis as input together with the current level i. The first hypothesis to call the algorithm will just be an empty one.

If the current node is a leaf node, the number of matchings in the current hypothesis is compared to the number of matchings in the best hypothesis so far. If the current hypothesis is better, it is declared as the new best.

If the current node is not a leaf node, the branches from the current node are tested for individual and joint compatibility. In other words, it is checked if the currently looked at measurement i is a possible matching for the mapped landmark j. If so, the matching is accepted and the algorithm is called recursively for the next level in the interpretation tree.

If no compatible matching could be found but it is still possible to find more matchings than the currently best hypothesis, the node is declared as unmatched and the algorithm moves on to the next level. If the latter is not possible, the current hypothesis can safely be disregarded, limiting the computational complexity.

3.5 Localization

Localization is the problem of finding the robot's pose in a given environment. The pose cannot be sensed directly in the sense of a robot possessing noise-free sensors determining the pose. Instead, the pose has to be computed from data. The localization problem covered in this thesis is a local one because compared to a global problem the initial pose is roughly known and can be estimated using a Gaussian probability distribution. Additionally, the environment is assumed to be static, which is a valid approach seeing that the tracks the vehicle competes on are made up of cones that aren't moved (see Section 4.1.3) and it is ensured that neither people nor animals are on the track at any time.

3.5.1 Marcov Localization

A very straightforward approach to solve the localization problem is to apply the Bayes filter described in Algorithm 1 to the problem, resulting in the Marcov localization algorithm:

Algorithm 6: Marcov Localization

Input: $bel(x_{t-1}), u_t, z_t, m$

Output: $bel(x_t)$

1 **forall** x_t **do**
2 $\quad \overline{bel}(x_t) = \int p(x_t \mid x_{t-1}, u_t, m) p(x_{t-1} \mid z_{1:t-1}, u_{1:t-1}, m) dx_{t-1}$
3 $\quad bel(x_t) = \eta p(z_t \mid x_t, m) \overline{bel}(x_t)$
4 **end**
5 **return** $bel(x_t)$

The algorithm is very similar to the Bayes filter. The difference is that in the Marcov localization the map m is incorporated into the algorithm. It plays a role mainly in the measurement update (line 3), but it is sometimes incorporated into the prediction (line 2) as well. With a roughly known initial pose, the first belief can be expressed by:

$$bel(x_0) = det(2\pi\Sigma_0)^{-\frac{1}{2}} \exp\left(-\frac{1}{2}(x_0 - \mu_0)^T {\Sigma_0}^{-1}(x_0 - \mu_0)\right) \tag{3.76}$$

3.5.2 EKF Localization

The EKF localization algorithm described in Algorithm 7 can be derived from the Marcov localization, just as the general EKF algorithm was derived from the Bayes filter.

Algorithm 7: EKF Localization

Input: $\mu_{t-1}, \Sigma_{t-1}, u_t, z_t$

Output: μ_t, Σ_t

1 $\bar{\mu}_t = g(u_t, \mu_{t-1}, m)$
2 $\overline{\Sigma}_t = G_t \Sigma_{t-1} G_t^T + Q_t$
3 $K_t = \overline{\Sigma}_t H_t^T (H_t \overline{\Sigma}_t H_t^T + R_t)^{-1}$
4 $\mu_t = \bar{\mu}_t + K_t(z_t - h(\bar{\mu}_t, m))$
5 $\Sigma_t = (I - K_t H_t)\overline{\Sigma}_t$
6 **return** μ_t, Σ_t

Again, the map is incorporated into the measurement update. In practice, this is used to compute the expected measurement. For this, the expected location of a measurement is needed, which is the mapped location. Additionally, the map may be used during the state transition, however, this is not always the case.

In the case of localization, the state vector x_t typically contains only the robot's pose. An implementation of this algorithm is discussed below in Chapter 6 where the EKF localization algorithm used in this thesis is presented.

With the theoretical foundation discussed in this chapter, the next chapter presents the system that the SLAM algorithms need to work within and other boundary conditions constraining the implementation of the algorithms.

4 Underlying Conditions and Preliminaries

4.1 Formula Student

Formula Student is an international design competition for students. With self-developed and self-built race cars, the participating teams compete in various events all over the world.[1]

4.1.1 History

In 1981, the Formula Student SAE[2] was held for the first time [51]. Starting as a small event with four participating schools in the United States of America, the concept grew more and more popular with the first competition in Europe being held in 1998 at MIRA Proving Ground in the United Kingdom [50]. In 2006, the German event, FSG, took place for the first time at the Hockenheimring [49].

So far, only race cars powered by an internal combustion engine had been competing. In 2010, FSG introduced a competition for electric vehicles. Finally, in 2017, FSG added a competition for autonomous race cars, the FSD.

4.1.2 KA-RaceIng

KA-RaceIng is a Formula Student team consisting of students of the Karlsruhe Institute of Technology (KIT). Founded in 2006, the team currently consists of around 70 active members. Every year, the team competes with multiple vehicles. Driverless race cars have been developed since the introduction of FSD in 2017. Recent accomplishments in the driverless competition include multiple second places in Formula Student Germany and overall first places in Formula Student East (2018) and Formula Student Spain (2019) [45].

4.1.3 Disciplines

The goal of a Formula Student event is to collect as many points as possible in multiple disciplines. These disciplines also consist of so called **static disciplines**, including the **Engineering Design** competition, where the teams have to explain and justify their respective vehicle design to experts from motorsport and the automotive industry. Especially in FSD, a substantial amount of points can be scored here, so the focus of the teams is not only to build the fastest car but also to get a deep understanding of their work and to be able to justify engineering decisions made throughout the entire development phase.

[1] While the various events are independent competitions, they follow similar rules and require the same challenges to be overcome. Because of this, they are often summarized as one competition. In this thesis, the term "Formula Student" will be used for this.

[2] SAE: Society of Automotive Engineers

Overall, a team is able to score up to 1000 points [48]. Table 4.1 shows how the points are split between the different disciplines.

Table 4.1: FSD - Maximum points awarded [48].

	Points
Static Events	
Business Plan Presentation	75
Cost and Manufacturing	100
Engineering Design	300
Dynamic Events	
Skid Pad	75
Acceleration	75
Autocross	100
Efficiency	75
Trackdrive	200
Overall	1000

Aside from the above-mentioned Engineering Design, the **dynamic events** are most relevant in the context of developing a SLAM algorithm with the different disciplines having different requirements of the algorithm.

The tracks are marked with **cones** that use a color code as depicted in Figure 4.1. Yellow and blue cones mark the right and the left boundary of the track, respectively, while small orange cones mark exit and entry lanes and big orange cones will be placed before and after start, finish and timekeeping lines.

(a) Start, finish and timekeeping lines.

(b) Exit and entry lanes.

(c) Right track boundary.

(d) Left track boundary.

Figure 4.1: The cones used for marking the track [47].

Acceleration

In the Acceleration event the cones are positioned in a straight line with a length of 75 m. The rules state that the cones will be placed approximately 5 m apart with a track width of at least 3 m. Since teams are allowed to walk the track before the event and use analogue measurements devices, it is possible to measure cone

positions and generate an approximate map for the event offline, in which case the vehicle does not need to solve the SLAM problem but will only have to localize itself on that map.

Skidpad

Opposed to the acceleration runs, in the Skidpad event, the teams showcase the maximum lateral acceleration their vehicle can handle. In an "8"-shaped layout the vehicle needs to drive four circles, two in each half of the "8", with only the respective second lap being timed. Figure 4.2 shows the track layout with all previously known measurements. The circle diameters as well as the number of cones on each circle is known, so again, an approximate map could be generated offline. The exact positioning of a cone is not known beforehand, though.

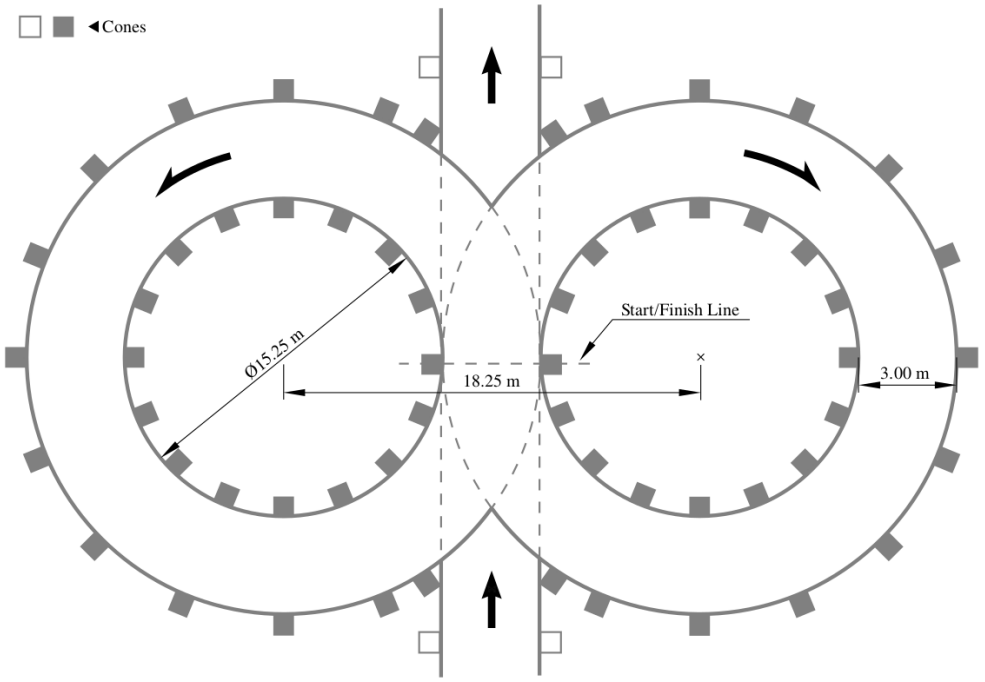


Figure 4.2: Skidpad track layout [48].

Autocross

The Autocross event is the most demanding discipline in the context of the SLAM algorithm because the track layout is not known beforehand, which means that solving the SLAM problem is necessary. However, the rules state some guidelines as to how the track can be designed [48]:

- Straights: No longer than 80 m.
- Constant turns: up to 50 m diameter.
- Hairpin turns: Minimum of 9 m outside diameter (of the turn).
- Miscellaneous: Chicanes, multiple turns, decreasing radius turns, etc.
- The minimum track width is 3 m.
- The length of one lap is approximately 200 m to 500 m.

An exemplary map is shown in Figure 4.3 that was generated using the data recorded during the 2019 FSG Autocross event and processed by the GraphSLAM algorithm presented in this thesis.

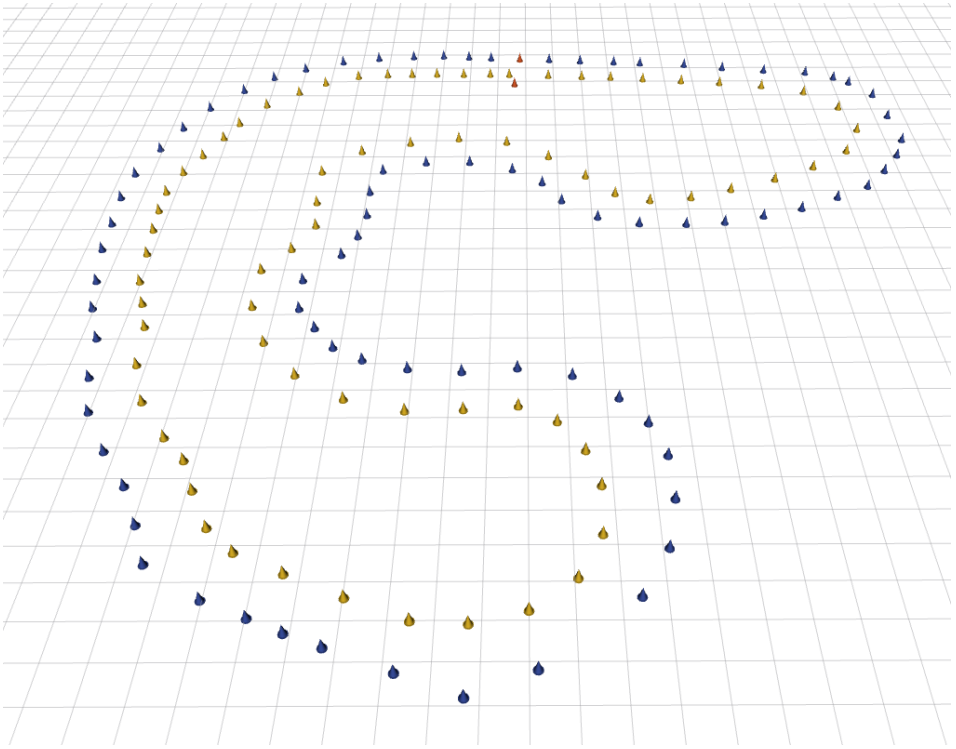


Figure 4.3: FSG 2019 Autocross map. Cone sizes increased for better visibility.

The primary goal in Autocross is finishing one lap as quickly as possible. However, the teams should also try to generate a high-quality map in this run because information collected during Autocross may be used in the Trackdrive event.

Trackdrive

On the same track used in Autocross, the teams now have to complete ten consecutive laps. The map generated in Autocross can and should be used. Therefore, the challenges for the SLAM algorithm are different from the Autocross event. The algorithm must be robust enough to not diverge in ten laps. The location given by the SLAM algorithm is also used for counting laps, as the vehicle has to stop on its own after finishing the run. Also, because the map is known beforehand, the speed of the vehicle is generally a lot higher compared to Autocross, decreasing the quality of sensor measurements.

Efficiency

The score for the Efficiency discipline is not determined in a separate event but during the Trackdrive event. The time integrated voltage and current output are multiplied to determine the energy used for the run. Regenerated energy is partially subtracted from the total energy used and a score is calculated from that value as well as the overall best efficiency of all teams and the time needed for the Trackdrive.

Scoring

Aside from the time needed for a run, there are other factors influencing the final score of a dynamic discipline. Additional points will be awarded for finishing at least one run without a *Did Not Finish (DNF)* or *Disqualified (DQ)*. Points can be deducted for the following reasons [48]:

- A cone is *Down or Out (DOO)* if the cone has been knocked over or the entire base of the cone lies outside the box marked around the cone in its undisturbed position.
- An *Off-course (OC)* occurs when the vehicle has all four wheels outside the track boundary as indicated by edge marking.
- An *Unsafe Stop (USS)* is defined as not stopping within the specified area and/or not entering the finish-state.

Table 4.2 shows the consequences these penalties have for the score awarded for a run depending on the discipline [48]. A DNF or a DQ will of course result in zero points.

Table 4.2: Penalties in FSD [48].

	Acceleration	Skidpad	Autocross	Trackdrive
DOO	2 s	0.2 s	2 s	2 s
OC	DNF	DNF	10 s	10 s
USS	DNF	DNF	DNF	-50 points

Map Availability

When it comes to the SLAM module of the software stack, the biggest difference between the various dynamic disciplines is whether or not a map is available. If a map is available, the vehicle only needs to localize itself, i.e. the SLAM problem does not need to be solved.

Acceleration and Skidpad are special cases because it is possible to generate an approximate map, based only on information taken from the rule book. This has proven to work in the past but with an improved SLAM algorithm it may not be necessary. Depending on whether or not a map is generated, the algorithms will either solve the SLAM problem or work in a localization-only mode.

For the purposes of this thesis, only data generated during Trackdrive and Autocross will be evaluated because these are the most demanding disciplines when it comes to the SLAM module. From the results of this analysis, conclusions can then be drawn with respect to Acceleration and Skidpad as well.

4.2 Base Car

The data sets used in this thesis were recorded on KA-RaceIng's 2019 driverless car, the KIT19d, depicted in Figure 4.4. The hardware platform is provided by the KIT15e that originally competed in the 2015 Formula Student Electric competition. The vehicle was then retrofitted with the sensors needed to race autonomously.

Figure 4.4: Hardware platform: The KIT19d.

The electric, all-wheel drive single seater has a top speed of approximately 120 $\frac{\text{km}}{\text{h}}$ and accelerates from standstill to 100 $\frac{\text{km}}{\text{h}}$ in roughly 2.5 s. Combined, the four motors - one per wheel - deliver a continuous power output of 92 kW and a short term output of up to 120 kW.

4.2.1 Sensors

To measure the environment, the vehicle uses two types of sensors: In the front of the car three lidars (IBEO LUX Model 2010) deliver information about the distance of the surroundings in the form of a point cloud. Mounted on the highest position of the car, the main roll hoop, are the two cameras (Basler dart daA1600-60uc) that are used to validate cone measurements and to find color information.

Additionally, each of the four motors is fitted with a hall sensor measuring its respective angular velocity. Using the fixed gearing ratio and the wheel's diameter, the wheel speeds are calculated from this measurement. An Inertia Measurement Unit (IMU) (Xsens MTi-G-710) is mounted in the center of the car. Together, these sensors are used to compute a velocity estimate of the vehicle.

The steering system was retrofitted with a steering actuator to allow the car to follow a planned trajectory. The steering system was also equipped with a steering sensor measuring the current steering angle.

4.2.2 Autonomous Computing Unit

The Autonomous Computing Unit (ACU), often referred to as "Car PC", is mounted on the back of the car. It consists of consumer-grade components that are fitted in a carbon fiber housing. The specifications are listed in Table 4.3.

Table 4.3: Components of the ACU.

Mainboard	Gigabyte Z390N
RAM	32BG G-Skill RipJaws V
CPU	Intel Core i7 9700K (8 Core) (TDP 95W)
GPU	None

Most notably, the ACU works without a Graphics Processing Unit (GPU). This is achieved with the architecture of the perception pipeline that uses a neural net but is capable of running on the CPU. See Section 4.3.3 for details. This is a great benefit because GPUs draw a lot of power, reducing the efficiency scoring. Additionally, because the ACU is powered by the low voltage system of the car, the power the ACU can draw from the battery is very limited.

4.2.3 Safety

Because racing with autonomous prototypes is dangerous, safety is the number one priority. This is why the Formula Student rules require the vehicle to be equipped with a shutdown circuit as depicted in Figure 4.5.

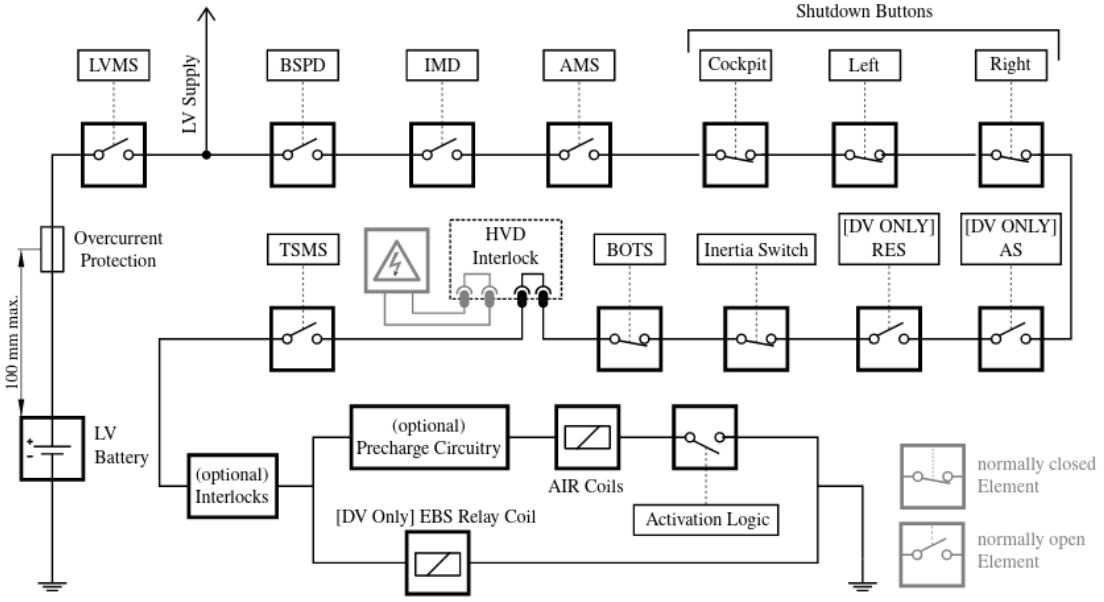


Figure 4.5: The shutdown circuit [48].

In essence, if any module of the shutdown circuit fails, the car will perform an emergency braking maneuver. This means that the high voltage system powering

the drive train is disconnected, the Emergency Brake System (EBS) is released and the emergency state is indicated with an intermittent sound and flashing blue lights. The EBS is a passive pneumatically actuated system.

An emergency brake maneuver will for example be performed in the following cases:

- Emergency stop requested by the user, using the Remote Emergency System (RES).
- The ACU requests a stop because one of the nodes has stopped responding.
- The battery management system detects a fault with the battery, for example overheating.

Especially the first two examples are relevant to the SLAM algorithm: If the algorithm diverges, the car will likely go off track and must be stopped manually by the team. Should the algorithm, for example, enter a state where it is not responsive, the supervisor node (see Section 4.3.2) has to engage the emergency brake maneuver for safety reasons. Of course, no points will be rewarded in such a case so having a robust, real-time capable SLAM algorithm is essential.

4.3 Autonomous Pipeline

The SLAM algorithms presented in this thesis have to be implemented into a complete autonomous pipeline to allow for driverless racing. The system can roughly be divided into the following modules:

- Perception: Measuring the environment and delivering the cone positions to the SLAM algorithm.
- SLAM: Taking the measurements from the perception pipeline as well as odometry measurements to estimate the map and the current position on it.
- Planning: Recognizing the track on the given map and planning the best possible trajectory on it, including the target velocity.
- Control: Minimizing the error between the estimated and the target pose and velocity and communicating needed wheel torques and steering angles with the hardware interfaces.

The utilized software stack of the KIT19d is introduced in great detail by Nekkah et al. [42] but shall be summarized in the following section.

4.3.1 Software Framework

All above mentioned modules are realized using Robot Operating System (ROS)[3] nodes. ROS is an open-source robotics software framework and the standard software tool in many robotics applications. It was started in 2007 at the Stanford Artificial Intelligence Laboratory and is being maintained and developed by the Open Source Robotics Foundation since 2014 [46]. The framework allows for the code to be written in multiple programming languages. All code in the implementation of this autonomous pipeline is written in *C++* with the exception of the image processing that is implemented using *Python*.

[3] `www.ros.org`

ROS allows the developer to implement an abstract, decentralized system architecture by using peer-to-peer communication. **Nodes** communicate directly with each other via a message system. **Publishers** will send a message to a **topic** and **subscribers** will listen to that topic. The publisher is not affected, whether there are none, one, or multiple subscribers to a topic. This allows for great flexibility. The full autonomous pipeline can be split into smaller modules that can be developed, maintained and tested as separate units. Individual units can be exchanged without affecting the overall system.

In addition to the beneficial architecture, ROS comes with a number of handy tools to improve the development workflow. To name a few:

- *RViz* allows visualizing messages.
- *rosbags* can record, play and manipulate messages. Testing a new algorithm can be accomplished by sending messages from a recording while having the new node running and processing these recorded messages.
- *Dynamic reconfigure* allows to change parameters online using a graphical user interface.
- *rqt_graph* is able to visualize the nodes and the messages they are sending.

4.3.2 Architecture

Figure 4.6 shows an overview of the complete software architecture of the autonomous system. The individual nodes are explained in more detail below.

The software pipeline starts with the *driver_ws*. In this workspace, all drivers needed to process the incoming sensor measurements are implemented. Here, all data is transformed into ROS messages and published to be made available to the rest of the system. For example, the lidar point clouds and the cameras images will be used in the *perception_ws*.

4.3.3 Perception

The perception pipeline takes advantage of the different strengths of both sensor types, the lidar and the camera. The main advantage of the lidar is the highly precise distance measurement, while the main advantage of the cameras is the semantic information that can be extracted from a camera image. Due to the high accuracy of the lidar, the camera serves only as a verification of objects, while the actual measurements are taken from the lidar directly.

The lidars and cameras are **synchronized** using a hardware trigger. After a scan of the environment, the lidar point clouds from the different lidar sensors are transformed into the same coordinate frame for further processing. To **filter** and **cluster** the point cloud, the DBSCAN algorithm is used [6]. Clusters that represent potential cone measurements are then projected into image space.

To reduce computational complexity, only areas of the image where a cone is proposed by the lidar, will be further investigated. For this, the distance of the potential cone is used to mark the region of interest in the image using a **bounding box** as shown in Figure 4.7. Using only the projected cluster as the center of the region of interest has been shown to not be robust enough, as can be seen in Figure 4.7a. To

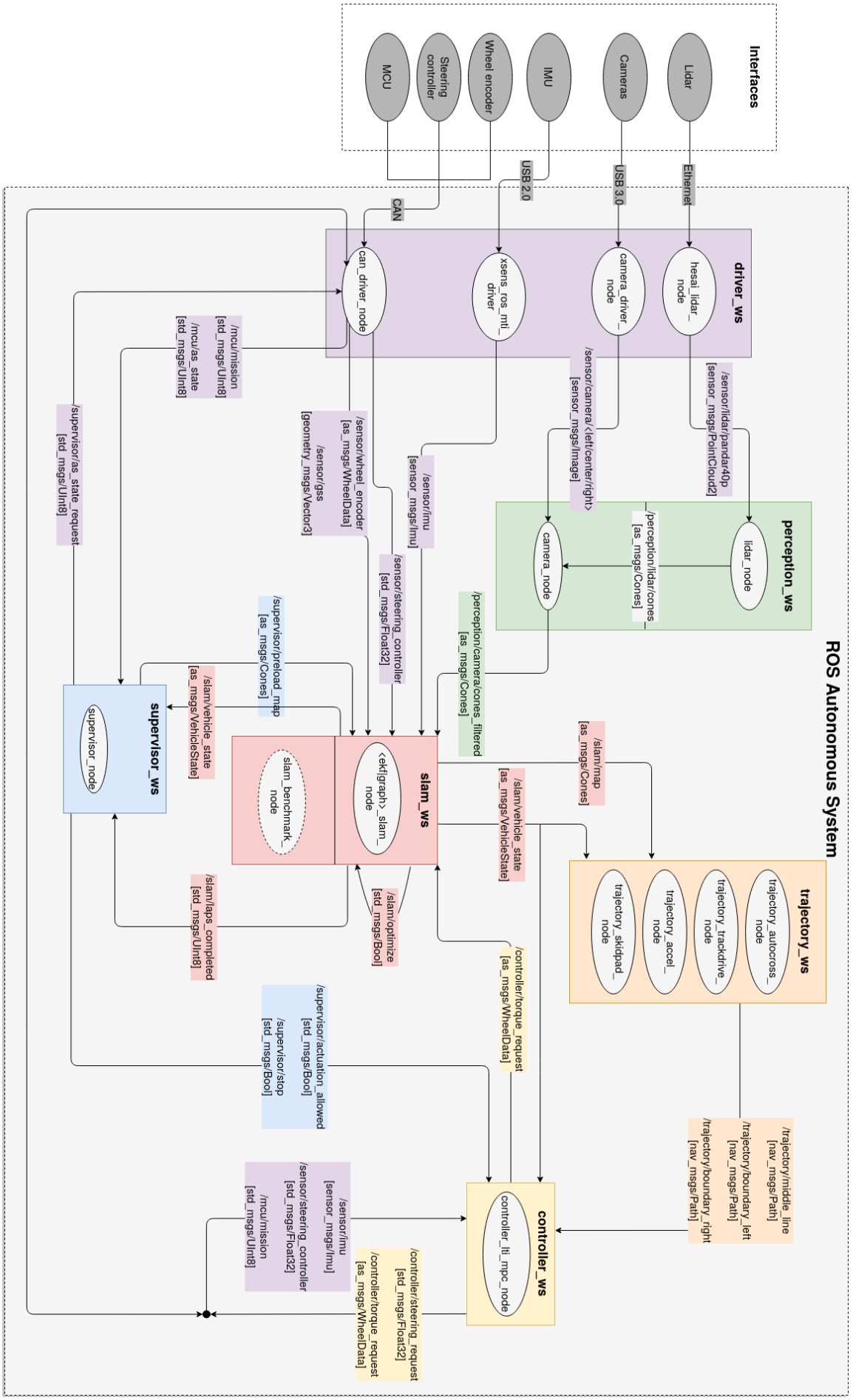
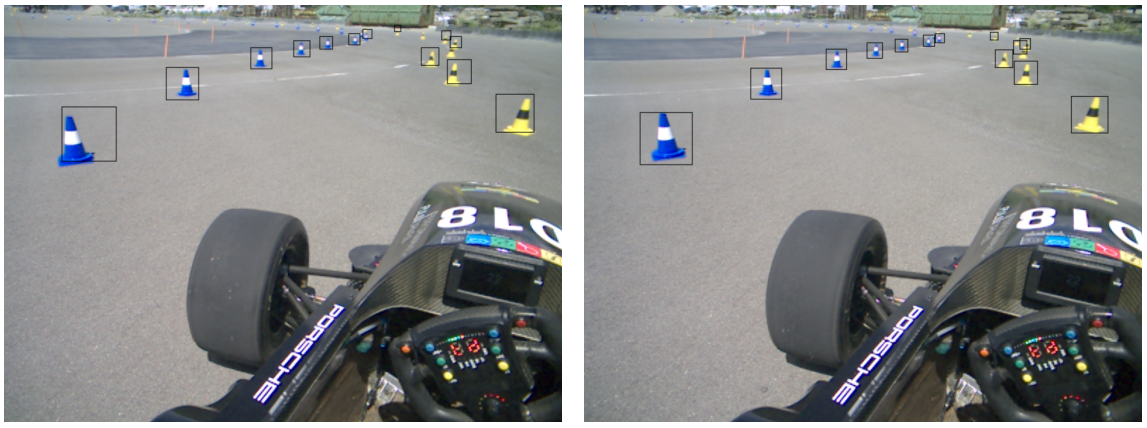


Figure 4.6: Overview over the full system architecture.

correct the bounding boxes, the unique color of the cones is taken advantage of. Using the center of the area of the respective color, the center of the bounding box is moved as depicted in Figure 4.7b.



(a) Before the bounding box correction. (b) After the bounding box correction.

Figure 4.7: Correction of the projected bounding boxes to fully contain a cone.

The bounding box and its content is then cut out of the full image and fed into a **convolutional neural network**. The network will then **classify** the content of the image to either be no cone or a cone of either blue, yellow or orange colour. Because this is a rather easy task for such a network and because only the region of interest is investigated, no GPU is needed. This is a great advantage in terms of power consumption.

All verified cone measurements will then be passed on to the SLAM algorithm.

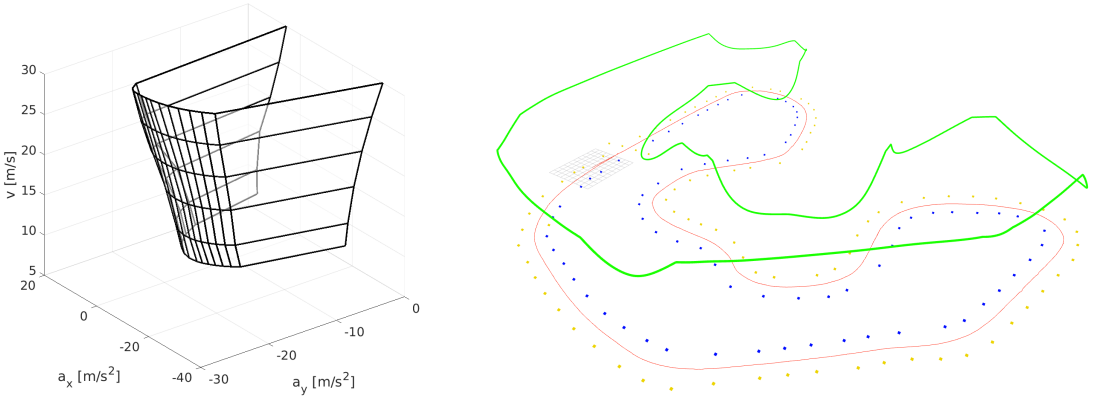
4.3.4 Planning

The trajectory planning algorithm differs depending on the driven discipline. For example, in Trackdrive the full path can be planned before the car starts driving because the map is known. In Autocross, however, the trajectory needs to be planned live and updated at every time step to account for newly mapped cones or corrections of their respective position. But in principle, the planning algorithm works the same in all of the disciplines and can only deliver good results if a sufficiently accurate map is supplied.

First of all, the **way points** need to be found. Way points are points on the middle line of the track that will be used to plan the path. They are computed by connecting all cones using Delauny triangulation and then sorting and filtering these triangles using geometric constraints like maximum opening angles or vertex distances. The way points are the middle points of the edges of the remaining triangles. The way points are connected by cubic splines to form the planned path.

The next step is to find the **target velocity** for each point on the planned path. For this purpose, a GGS-diagram is used. In this diagram, the maximum possible lateral and longitudinal acceleration are plotted depending on the vehicle speed. The latter is necessary because the aerodynamics of a Formula Student race car greatly affect the maximum possible acceleration by increasing the downforce with increasing velocity. Figure 4.8a shows the GGS-diagram of a Formula Student race

car. Using the maximum acceleration, the maximum velocity is computed for each point on the track. Figure 4.8b shows the final result of the planning module. The cones are represented as blue and yellow dots with the planned path in red. On the vertical axis, in green, the target velocity is shown.



(a) GGS diagram of a FS race car. (b) Planned path (red) and velocity (green).

Figure 4.8: Planning module: GGS diagram and final result.

The trajectory can now be optimized to, for example, resemble the racing line instead of passing through the middle points. However, since a Formula Student track is very narrow compared to the car, it is only sensible to do so if the localization is very precise. Otherwise the vehicle might hit more cones and the overall score will be worse because of the resulting penalties.

4.3.5 Controller

The task of the controller is to reduce the distance between the planned path and the estimated vehicle pose in lateral direction, and to reduce the difference between the planned and the estimated longitudinal velocity. The information available to the controller is the velocity and pose estimate delivered by the SLAM module as well as the planned trajectory delivered by the planning module. The outputs of the controller are torque requests for each of the four wheels and the desired steering angle. Implemented are two separate controllers:

In **longitudinal direction**, a simple PI-Controller minimizes the error between the current and the target velocity.

In **lateral direction**, the vehicle utilizes a Model Predictive Controller (MPC) [12] to reduce the distance to the planned path. By predicting the state for the near future (the *prediction horizon*) with an implemented vehicle model, the controller can react to errors, before they occur.

The output of the longitudinal controller is the desired acceleration, the output of the lateral controller is the desired yaw moment, resulting in an overdetermined system.

To solve this and to implement an active yaw control (torque vectoring), **Optimal Control Allocation** [15] is used. By solving an optimization problem, the torques can be split between the wheels to achieve the desired linear acceleration and yaw

rate, whilst also considering constraints like the limits of grip of tires with a lower load.

The controller can of course only decrease the pose error efficiently if the actual vehicle pose delivered by the SLAM module is an accurate estimate.

4.4 SLAM: Initial Situation

The KIT19d, of course, also solved the SLAM problem, otherwise no autonomous driving would have been possible. While the implemented algorithm was working in principle, the SLAM module turned out to be one of the performance bottlenecks of the system.

The approach used in 2019 was based on an **EKF**. In principle, it was based on the EKF localization algorithm as described in Algorithm 7. To build the map, a very simple and therefore efficient method was chosen: A landmark was mapped at the position of its very first detection and then never corrected afterwards. With this simplification, the landmark positions did not need to be included in the state.

The 2019 FSG Autocross track consisted of 155 cones. Since the EKF's computational complexity is squared in the number of states ($O(n^2)$), including the map into the state makes a enormous difference. While the 2019 squared number of states was $n^2 = 3^2 = 9$, an EKF SLAM algorithm that will update all cone positions in all time steps would have $n^2 = (3 + 155 * 2)^2 = 97969$.

However, the high efficiency is paid for with a very low accuracy. In fact, the vehicle performance had to be reduced in certain situations to limit the wheel slip, which in turn improved the odometry measurements. Having good odometry measurements reduces the overall drift in a SLAM algorithm, which means that corrections using the measurements of the surroundings are less important to reduce drift.

Since racing is about always going to the limit in terms of performance, the SLAM algorithm used by the car needed to be improved. To make an informed and objective decision, two algorithms should be implemented and compared with respect to their performance.

In the remainder of this thesis, two algorithms - EKF SLAM and GraphSLAM - are presented, their performance is compared and finally, a conclusion is drawn as to which of the algorithms should be used in the future at KA-RaceIng.

5 System Modeling

5.1 Reference Frames

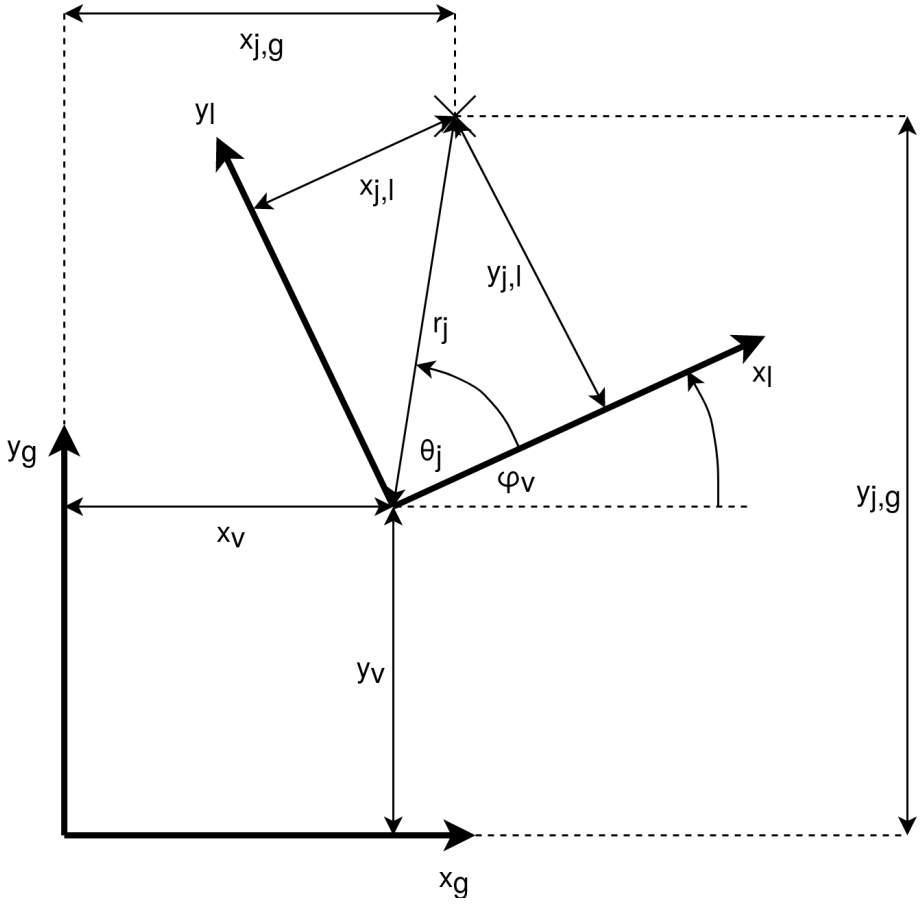


Figure 5.1: Reference frames and relevant variables used throughout this thesis. The 'X' marks the locations of the j-th landmark.

The reference frames that will be used throughout this thesis are depicted in Figure 5.1. All problems will be regarded as two-dimensional. This greatly reduces the computational complexity and due to the flat nature of a Formula Student track, this is a valid assumption.

The **global reference frame** (or world reference frame) $\{x_g, y_g\}$ is defined by the starting position of the vehicle. When initializing, the vehicle will define its starting position to be $x_g = 0, y_g = 0$, heading in the direction of the global x-axis. The vehicle coordinates are defined by the global position (x_v, y_v) and the vehicle heading φ_v:

$$x = \begin{pmatrix} x_v \\ y_v \\ \varphi_v \end{pmatrix} \tag{5.1}$$

The global coordinates of the j-th landmark are described by $x_{j,g}$ and $y_{j,g}$. Because

cones are used as landmarks, the heading is neither measurable nor helpful when solving the SLAM problem, so only their respective position is of interest.

The **local reference frame** (or vehicle reference frame) $\{x_l, y_l\}$ is defined by the vehicle coordinates: The origin is located at the current position of the car, i.e. x_v and y_v. The local x-axis is pointing in the direction of the vehicle heading, i.e. φ_v.

The position of the landmark within this local reference frame can be described either by the cartesian coordinates $x_{j,l}$ and $y_{j,l}$ or the respective cylindrical coordinates r_j and θ_j. The transformations between these coordinates are given by:

$$\begin{pmatrix} x_{j,l} \\ y_{j,l} \end{pmatrix} = \begin{pmatrix} r_j \cos(\theta_j) \\ r_j \sin(\theta_j) \end{pmatrix} \tag{5.2}$$

$$\begin{pmatrix} r_j \\ \theta_j \end{pmatrix} = \begin{pmatrix} \sqrt{x_{j,l}^2 + y_{j,l}^2} \\ atan2(y_{j,l}, x_{j,l}) \end{pmatrix} \tag{5.3}$$

$$\begin{pmatrix} r_j \\ \theta_j \end{pmatrix} = \begin{pmatrix} \sqrt{(x_{j,g} - x_v)^2 + (y_{j,g} - y_v)^2} \\ atan2(y_{j,g} - y_v, x_{j,g} - x_v) - \varphi_v \end{pmatrix} \tag{5.4}$$

5.2 Motion Model

The motion model is used to compute the state transition and describes the robot's movement in the environment, given the control input u_t. It is a common approach to actually measure a robot's motion resulting from a control input instead of using the control input itself, as it eliminates error sources caused, for example, by inaccurate actuators. The motion model used throughout this thesis is a very simple one because it must be computationally efficient. Also, it only has to be accurate in the short term, as the predicted pose will be corrected using measurement data.

The vehicle is assumed to move with constant velocity between time steps. This assumption will of course be violated in reality. However, the prediction is performed at a high frequency and results have shown this assumption to be accurate enough for the given purpose. The resulting motion model can be expressed like this:

$$\begin{pmatrix} x_{v,t} \\ y_{v,t} \\ \varphi_{v,t} \end{pmatrix} = \begin{pmatrix} x_{v,t-1} \\ y_{v,t-1} \\ \varphi_{v,t-1} \end{pmatrix} + \begin{pmatrix} \cos(\varphi_v) & -\sin(\varphi_v) & 0 \\ \sin(\varphi_v) & \cos(\varphi_v) & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \hat{v}_x \\ \hat{v}_y \\ \hat{\dot{\varphi}} \end{pmatrix} \Delta t \tag{5.5}$$

$$= \begin{pmatrix} x_{v,t-1} \\ y_{v,t-1} \\ \varphi_{v,t-1} \end{pmatrix} + \begin{pmatrix} \cos(\varphi_v)\hat{v}_x\Delta t - \sin(\varphi_v)\hat{v}_y\Delta t \\ \sin(\varphi_v)\hat{v}_x\Delta t + \cos(\varphi_v)\hat{v}_y\Delta t \\ \hat{\dot{\varphi}}\Delta t \end{pmatrix} \tag{5.6}$$

The 3×3 matrix in this equation is effectively a rotation matrix transforming the vehicle speed into the global coordinate frame. $\hat{v} = (\hat{v}_x, \hat{v}_y, \hat{\dot{\varphi}})^T$ is the true velocity, which is unknown, so the motion model is approximated using the measured velocity and Gaussian noise with a covariance of Q_t:

$$\begin{pmatrix} x_{v,t} \\ y_{v,t} \\ \varphi_{v,t} \end{pmatrix} = \begin{pmatrix} x_{v,t-1} \\ y_{v,t-1} \\ \varphi_{v,t-1} \end{pmatrix} + \begin{pmatrix} \cos(\varphi_v)v_x\Delta t - \sin(\varphi_v)v_y\Delta t \\ \sin(\varphi_v)v_x\Delta t + \cos(\varphi_v)v_y\Delta t \\ \dot{\varphi}\Delta t \end{pmatrix} + \mathcal{N}(0, Q_t) \tag{5.7}$$

$$= g(u_t, x_{t-1}) + \mathcal{N}(0, Q_t) \tag{5.8}$$

As described in Equation (3.50), the function g can be approximated using a Taylor expansion:

$$g(u_t, x_{t-1}) \approx g(u_t, \mu_{t-1}) + G_t(x_{t-1} - \mu_{t-1}) \tag{5.9}$$

$g(u_t, \mu_{t-1})$ is obtained by replacing the unknown state x_{t-1} by the mean of the current belief μ_{t-1}. The Jacobian G_t can be expressed like this:

$$G_t = \begin{pmatrix} 1 & 0 & (v_x sin(\mu_{t-1,\varphi_v}) - v_y \cos(\mu_{t-1,\varphi_v}))\Delta t \\ 0 & 1 & (v_x \cos(\mu_{t-1,\varphi_v}) - v_y \sin(\mu_{t-1,\varphi_v}))\Delta t \\ 0 & 0 & \Delta t \end{pmatrix} \tag{5.10}$$

Handling Dropped Odometry Messages

For various reasons, it may occur from time to time that an odometry message is not processed in time and the next message is therefore dropped. Generally, this is not a big issue. The pose estimate using the odometry data is a rough guess anyway, which is why the pose is corrected in the update step of EKF SLAM or the optimization of GraphSLAM.

If messages are dropped too frequently, however, the pose estimate may be too fraudulent to be corrected later on. Because of this, and to improve the short-term estimate (which is the one used by the controller), dropped messages are considered in the motion model by increasing the time step Δt depending on the number of dropped messages.

The motion model used here can be regarded as a numerical integration of the velocity estimates to form a pose estimate. Increasing the step width is therefore equivalent to reducing the resolution of the integration. While this obviously reduces the precision when compared to integrating with full resolution, the estimate is still more precise than it would be when just skipping a step of the integration.

5.2.1 Velocity Estimation

The motion model described above depends on a velocity estimate as input. While the rotational velocity $\dot{\varphi}$ is delivered directly from the IMU, the linear velocity is not measured directly. Instead, the axle speeds are measured at the electric motors ω_{mot} and a linear velocity v_x is computed using the gearing ratio i_{gear} and the tire radius r_{tire}. Experiments have shown that the best results are obtained when using the wheel speeds of the rear axle ω^{rl}_{mot} and ω^{rr}_{mot} and combining them into a single velocity estimate to account for different speeds of the left and right side when turning.

$$v_x = \frac{2\pi r_{\text{wheel}} \frac{(\omega^{rl}_{\text{mot}} + \omega^{rr}_{\text{mot}})}{2}}{i_{\text{gear}}} = \frac{\pi r_{\text{wheel}} (\omega^{rl}_{\text{mot}} + \omega^{rr}_{\text{mot}})}{i_{\text{gear}}} \tag{5.11}$$

This of course only allows for a longitudinal velocity estimate. Since there are no sensors on the car that allow for a velocity estimate in lateral direction, the lateral velocity is assumed to be zero.

$$v_y = 0 \tag{5.12}$$

While this assumption is clearly fraudulent, it can be used here because the lateral velocity is low enough for the SLAM algorithm to be able to correct the resulting positioning error.

5.3 Measurement Model

The measurements of surroundings landmarks are processed using cylindrical coordinates. This way, the two dimensions of a measurement can be regarded as independent. The position measurements are made by a lidar sensor, so this assumption is feasible and results in zeros in the off-diagonal entries of the measurement noise matrix R_t.

The measurement model is formulated using Equation (5.4). Analogously to the motion model, the measurement is not exact, therefore the measurement model is approximated using a Gaussian noise term:

$$\begin{pmatrix} r_j \\ \theta_j \end{pmatrix} = \begin{pmatrix} \sqrt{(x_{j,g} - x_v)^2 + (y_{j,g} - y_v)^2} \\ atan2(y_{j,g} - y_v, x_{j,g} - x_v) - \varphi_v \end{pmatrix} + \mathcal{N}(0, R_t) \tag{5.13}$$

$$= h(x_t, j, m) + \mathcal{N}(0, R_t) \tag{5.14}$$

The Taylor approximation is:

$$h(x_t, j, m) \approx h(\mu_t, j, m) + H_t(x_t - \mu_t) \tag{5.15}$$

When defining

$$\delta = \begin{pmatrix} \delta_x \\ \delta_y \end{pmatrix} = \begin{pmatrix} \mu_{x_{j,g}} - \mu_{x_v} \\ \mu_{y_{j,g}} - \mu_{y_v} \end{pmatrix} \tag{5.16}$$

and $q = \delta^T \delta$, the Jacobian H_t can be expressed like this:

$$H_t = \frac{\partial h(\mu_t, j, m)}{\partial \mu_v} = \begin{pmatrix} -\sqrt{q}\delta_x & -\sqrt{q}\delta_y & 0 \\ \delta_y & -\delta_x & -q \end{pmatrix} \tag{5.17}$$

Here, $\mu_j = (\mu_{x_{j,g}} \ \mu_{y_{j,g}})^T$ is the robot's belief of the j-th landmark's position in the global coordinate frame and $\mu_v = (\mu_{x_v} \ \mu_{y_v} \ \mu_{\varphi_v})^T$ is the internal belief of the vehicle's pose in the world frame.

In the case of EKF SLAM, the state vector will contain the landmark position. Therefore, the complete Jacobian contains zeros in all places except for the partial Jacobians $\frac{\partial h(\mu_t, j, m)}{\partial \mu_v}$ and $\frac{\partial h(\mu_t, j, m)}{\partial \mu_j}$. For the latter follows:

$$H_{t,j} = \frac{\partial h(\mu_t, j, m)}{\partial \mu_j} = \begin{pmatrix} \sqrt{q}\delta_x & \sqrt{q}\delta_y \\ -\delta_y & \delta_x \end{pmatrix} \tag{5.18}$$

5.3.1 Inverse Measurement Model

When a new landmark is initialized, the inverse measurement model h^{inv} is needed. This equation uses a measurement and transforms it into state space:

$$\begin{pmatrix} x_{j,g} \\ y_{j,g} \end{pmatrix} = \begin{pmatrix} x_v + r_j \cos(\varphi_v + \theta_j) \\ y_v + r_j \sin(\varphi_v + \theta_j) \end{pmatrix} + \mathcal{N}(0, R_t) \tag{5.19}$$

$$= h^{inv}(x_v, z_j) + \mathcal{N}(0, R_t) \tag{5.20}$$

The Taylor approximations with respect to the mean of the belief of the vehicle state μ_v and the measurement z_j are:

$$H_v^{inv} = \frac{\partial h^{inv}(\mu_v, z_j)}{\partial \mu_v} = \begin{pmatrix} 1 & 0 & -r_j \sin(\mu_{\varphi_v} + \theta_j) \\ 0 & 1 & r_j \cos(\mu_{\varphi_v} + \theta_j) \end{pmatrix} \tag{5.21}$$

$$H_j^{inv} = \frac{\partial h^{inv}(\mu_v, z_j)}{\partial z_j} = \begin{pmatrix} \cos(\mu_{\varphi_v} + \theta_j) & -r_j \sin(\mu_{\varphi_v} + \theta_j) \\ \sin(\mu_{\varphi_v} + \theta_j) & r_j \cos(\mu_{\varphi_v} + \theta_j) \end{pmatrix} \tag{5.22}$$

5.4 Uncertainties

The uncertainties of the measurements are represented in the covariance matrices of the motion model (Q_t) and the measurement model (R_t). Both of these uncertainties can be tied to sensor input. Finding good values for these uncertainties is crucial because they have an enormous impact on the algorithm's performance. Wrong uncertainties quickly lead to a diverging filter because the algorithm over- or underestimates the certainty of the vehicle's pose or a landmark's location. To make matters more complicated, the uncertainties are not generally constant values and may change depending on the vehicle state.

To find the best possible values, the error of the control and measure input was modeled. Assuming worst-case scenarios, the error is calculated given the current vehicle state. Interpreting the resulting error as the standard deviation allows to find the covariance by squaring the resulting error. This is of course a very rough estimate of the uncertainty. There may be additional error sources not considered in the error models. And since worst-case scenarios are used, the sensor input might actually be more accurate than expected. However, if the error would be known exactly, there wouldn't be a need to look at this from a probabilistic point of view.

5.4.1 Motion Uncertainty

Longitudinal Uncertainty

The sensor input available to the algorithms will first of all be the wheel speeds. As described above, they are used to calculate the longitudinal velocity v_x. Aside from the error induced by the hall sensor measuring the motor revolutions e_{hall}, the main error source here is the slip s resulting from the torque applied to the wheels.

The Pacejka tire model is used to determine the longitudinal and lateral forces a tire will transfer given the longitudinal slip or side slip angle, respectively. For the purposes of the following estimation, this relation will be flipped and the slip will be estimated using the linear section of the Pacejka tire model. The gradient for this section for the longitudinal tire model g_{lo} can be estimated from the plot depicted in Figure 5.2. Here, the contact force is chosen rather low. If the wheel load increases, for example for the outer wheel in a corner, the slip will actually reduce. But since the following estimation is assuming a worst-case scenario, this will not be considered.

From the definition of slip [25]

$$s = \frac{v_x - \hat{v}_x}{\hat{v}_x} \tag{5.23}$$

the error $e_{\text{slip},x}$ can be estimated by multiplying the slip with the velocity:

$$s = \frac{v_x - \hat{v}_x}{\hat{v}_x} \approx \frac{v_x - \hat{v}_x}{v_x} = \frac{e_{\text{slip},x}}{v_x} \implies e_{\text{slip},x} \approx s v_x \tag{5.24}$$

Here, v_x represents the vehicle speed estimated from wheel speeds and $\hat{v}_x$ represents the true vehicle speed.

In conclusion, for the uncertainty of v_x follows:

$$\sigma_{v_x} = e_{\text{hall}}^2 + e_{\text{slip},x}^2 = e_{\text{hall}}^2 + \left(\frac{g_{\text{lo}} i_{\text{gear}}}{r_{\text{tire}}} T_{\text{mot}} v_x\right)^2 \tag{5.25}$$

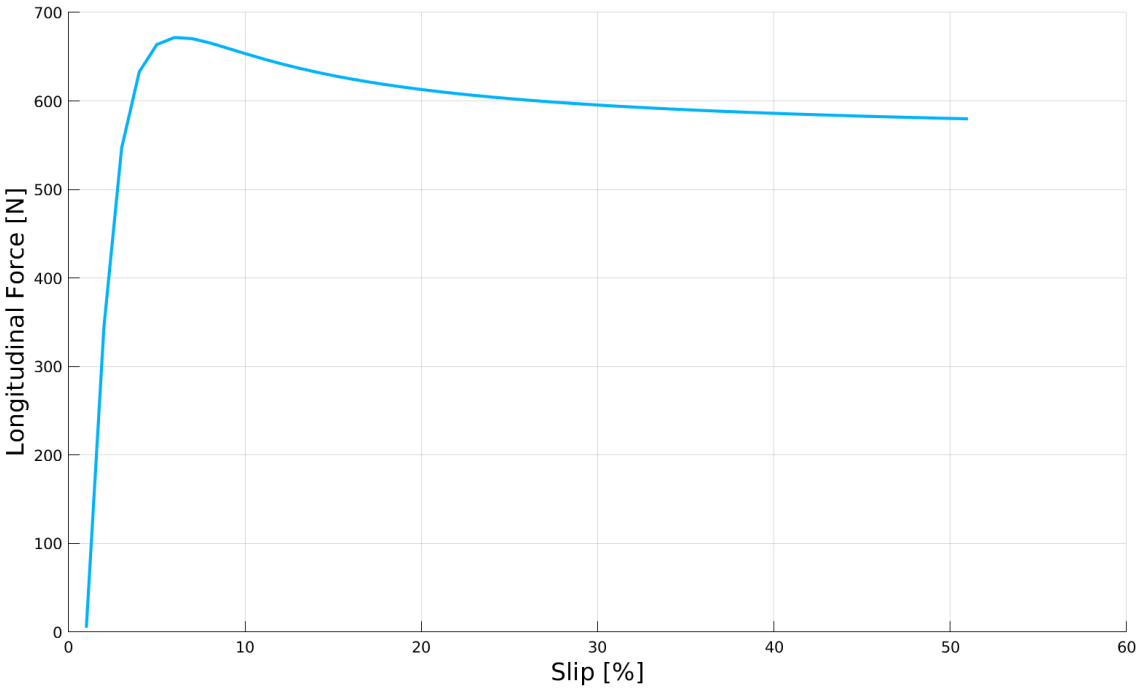


Figure 5.2: Pacejka tire model: Force vs. slip (longitudinal). Contact force: 400 N.

Since the torque is measured at the electric motors, the gear ration i_{gear} needs to be considered. r_{tire} is the tire's dynamic radius to convert the drive torque into the tire force.

Lateral Uncertainty

In lateral direction, the main error velocity v_y is slip as well. Just as in longitudinal direction, the slip is estimated using tire data. Using the side slip angle β and a small-angle approximation, the lateral velocity is

$$v_y = v \sin \beta \approx v_x \beta. \tag{5.26}$$

From the single track model depicted in Figure 5.3, β can be approximated to

$$\beta = -\frac{l_r}{R} + \alpha_r. \tag{5.27}$$

The corner radius R is approximated using a small steering angle δ and assuming no slip angles α_f and α_r:

$$R \approx \frac{l}{\tan \delta} \approx \frac{l}{\delta} \tag{5.28}$$

The rear slip angle α_r needed in Equation (5.27) is computed using the longitudinal velocity and tire data. The lateral acceleration of the car is:

$$a_y = \frac{v_x^2}{R} \tag{5.29}$$

From the acceleration and the vehicle weight m_v results the lateral force F_y

$$F_y = a_y m_v \tag{5.30}$$

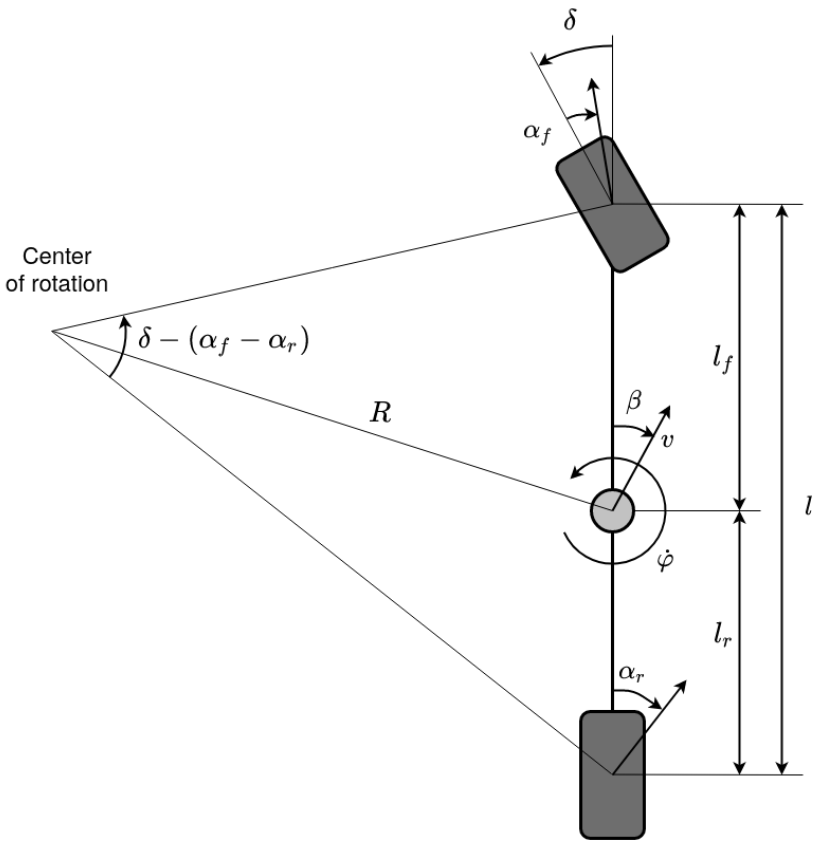


Figure 5.3: Single track model.

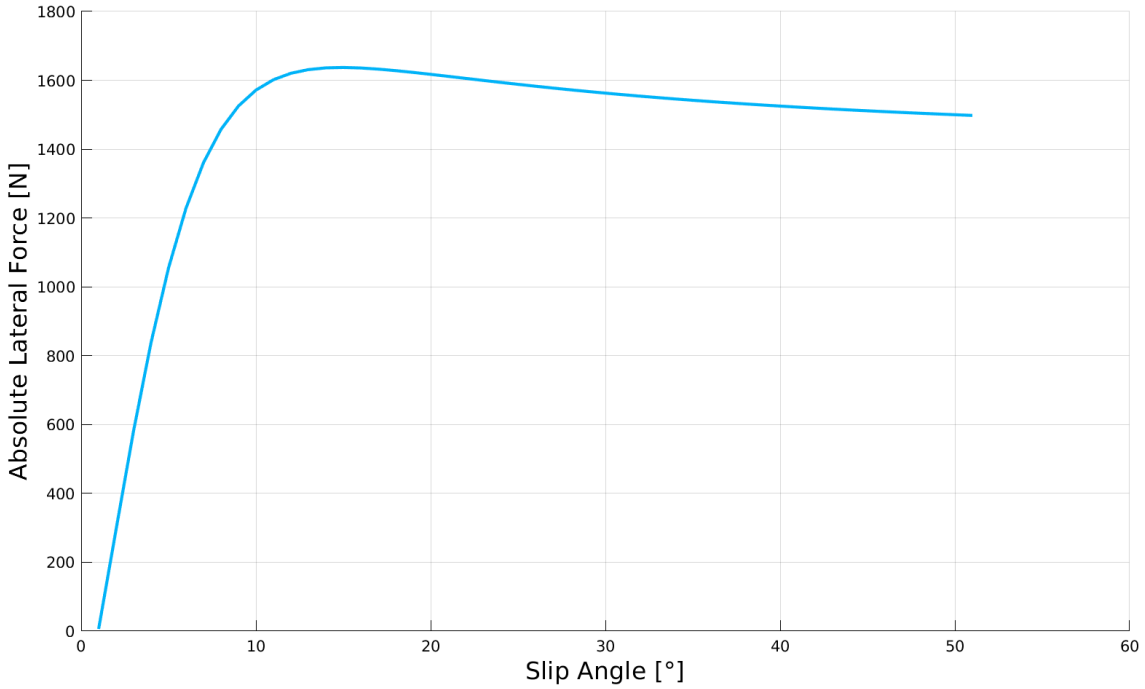


Figure 5.4: Pacejka tire model: Lateral force vs. slip angle. Contact force: 400 N.

which is used to obtain α_r

$$\alpha_r = g_{\text{lat}} F_y \tag{5.31}$$

g_{lat} is the gradient of the linear section of the lateral Pacejka tire model that can be seen in Figure 5.4.

Combining these equations results in the approximated lateral velocity error e_{v_y}:

$$\begin{aligned} e_{v_y} &= v_y && (5.32)\\ &\approx v_x \beta && (5.33)\\ &= v_x(-\frac{l_r}{R} + \alpha_r) && (5.34)\\ &\approx v_x(-\frac{l_r}{l}\delta + g_{\text{lat}} m_v v_x^2) && (5.35)\\ &= \frac{g_{\text{lat}} m_v}{l}\delta v_x^3 - \frac{l_r}{l}\delta v_x && (5.36) \end{aligned}$$

The uncertainty is again calculated by squaring the error:

$$\sigma_{v_y} = e_{v_y}^2 \tag{5.37}$$

While this is of course a very rough approximation, there now is a coupling between lateral uncertainty, speed and steering angle. High speeds and high steering angles will result in high lateral velocity uncertainty.

Rotational Uncertainty

The IMU delivers sensor data for the **yaw rate**. The error induced by this sensor itself is the main source of error for this measurement, so the uncertainty for the yaw rate is computed using:

$$\sigma_{\dot{\varphi}} = e_{\dot{\varphi}}^2 \tag{5.38}$$

Transformation into State Space

The uncertainties described here are uncertainties for the velocity in the vehicle coordinate frame, i.e. in *control space*.

$$Q_{t,l} = \begin{pmatrix} \sigma_{v_x} & 0 & 0 \\ 0 & \sigma_{v_y} & 0 \\ 0 & 0 & \sigma_{\dot{\varphi}} \end{pmatrix} \tag{5.39}$$

To apply the uncertainty to the SLAM algorithm, the noise must be transformed into *state space*. To do this, the approximated motion model $g(u_t, x_{t-1})$ described in Equation (5.8) is derived with respect to the control input u_t

$$V_t = \frac{\partial g(u_t, x_{t-1})}{\partial u_t} = \begin{pmatrix} \cos\varphi_v \Delta t & -\sin\varphi_v \Delta t & 0 \\ \sin\varphi_v \Delta t & \cos\varphi_v \Delta t & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{5.40}$$

and applied to the motion noise in control space $Q_{t,l}$ to result in the motion noise in state space Q_t:

$$Q_t = V_t \, Q_{t,l} \, V_t^T \tag{5.41}$$

5.4.2 Measurement Uncertainty

Analogously, the sensor uncertainty for the environment measurements must be determined. The landmark position will be delivered using the lidar, so first of all, the measurement noise of the lidar itself needs to be considered. Since all measurements will be processed using cylindrical coordinates, the noise of the range and the bearing measurement can be regarded as independent.

An additional error source is the clustering algorithm used. The landmark's position will be estimated by grouping all points that are considered to be corresponding to the same landmark and computing the middle point of this cluster. This does not necessarily coincide with the actual middle point of the landmark. In fact, since a landmark can just be measured from one side at a time, the clustering middle point will likely have an offset to the actual middle point. This must be considered when setting the uncertainty parameters for the SLAM algorithm. It is also worth mentioning that an offset like that breaks the Gaussian assumption. The next section explains, how the algorithms can work despite this.

5.4.3 Limiting the Minimal Uncertainty

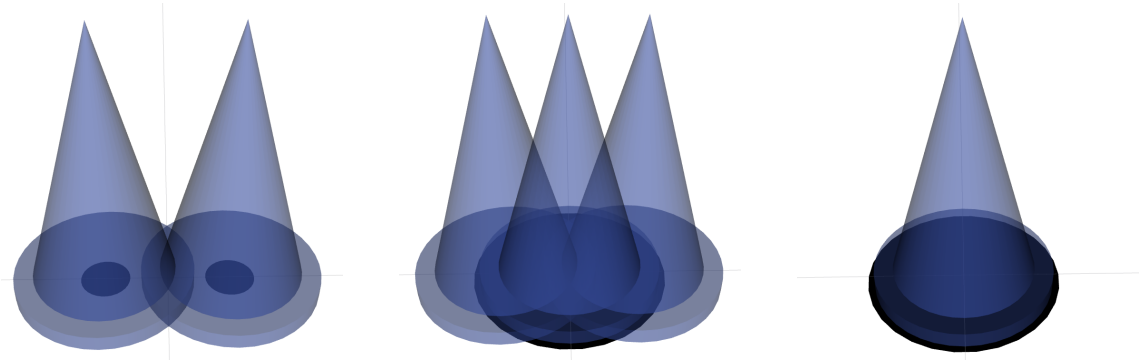
During the development of the algorithms a problem kept arising: Cones, especially near the start, were mapped twice. It seemed as if the data association algorithm failed to match cones even though they were very close together.

The reason for this lies in a systematic error made by the perception pipeline: The measurement of a cone will always resemble the outer shell of the cone rather than the center of the cone. This is because a cone can always be seen from one side only and the clustering algorithm defines the center of a measured cone to coincide with the center of the clustered points. This means that the cluster center will always be near the shell of the cone instead near its center, which results in all measurements being closer to the car than they should be. The resulting error is systematic, meaning that the measurement's probability density does not have a zero mean and is therefore not Gaussian.

This phenomenon turned out to be a big issue because the algorithms would overestimate the certainty of the cone position because the vehicle re-observes the same systematic error many times. This results in cone positions that are off by a few centimeters but have a very high certainty. Because of this, the Mahalanobis distance, that is used by the data association algorithm, grows very large. So, when re-observing the cone from another side, when for example the starting straight is visible from the opposite straight, it will be regarded as a new cone.

The phenomenon is visualized in Figure 5.5. The cones are visualized transparently so that the covariance visualization is visible. The covariance is visualized using an ellipse at the bottom of the cone with the orientation and the radii of the ellipse is determined using the Eigenvectors and the Eigenvalues, respectively. In this example, the uncertainty is equal in all directions, so the ellipse resembles a circle.

In Figure 5.5a the covariance is not limited. This results in two mapped cones, each at the position of the outer shell of the actual cone, as can be seen in Figure 5.5b. Because the covariance is not limited, with multiple measurements confirming the



(a) Missing association because of overestimated certainty.

(b) Combination of the wrong and correct associations.

(c) Measurements from both directions successfully matched.

Figure 5.5: Effect of limiting the minimal uncertainty. Covariance visualized by black ellipse.

systematic error it shrinks to very small values (i.e. the cone position is determined to be very certain).

In Figure 5.5c, the correct measurement is visualized. It is in between the two false measurements and could be associated because the minimal uncertainty was limited to account for the cone's volume.

5.5 Filtering False Positive Measurements

Depending on the quality of the sensor data and the perception pipeline, the SLAM algorithm will have to deal with a number of false positive detections. Generally, a false detection is not a big issue for the algorithm. If, for example, a fixed object like a fire extinguisher is falsely classified as an orange cone, it can still be used by the SLAM algorithm to localize the car. If a false positive measurement is the result of sensor noise or similar, it is likely that it will not occur very often in the same area, so it would not be used (often) to correct the state.

Still, false positives are an issue for two reasons. First of all, **trajectory planning** is only interested in the map of the track itself. With more landmarks other than cones on the map (such as a fire extinguisher), the trajectory planning algorithm must be increasingly robust and filter them out. Secondly, the process of **data association** becomes increasingly difficult because all false positive measurements still need to be regarded as possible matchings for new incoming measurements. The computational complexity of the data association depends on the size of the map and the number of measurements, so it will take longer if there are a large number of false positives.

A very simple way to deal with this issue is to count the number of times a particular landmark has been seen. The SLAM algorithms will only publish landmarks on the map if they are confirmed, i.e. they have been seen often enough. This already improves the situation for the trajectory planning.

To deal with the problem of data association, a **map cleaning** is performed fre-

quently. This means that with a certain rate all landmarks are checked, as to whether or not they have been seen often enough to be confirmed. If a cone is under that threshold and also outside the perception range, it is removed from the map. The latter restriction is necessary because of course every landmark is under the threshold when it is seen for the first time. As long as it could be seen for a second time, i.e. as long as it is within the perception range, it must be tracked and cannot be removed from the map.

Having discussed the basic concepts relevant to both algorithms, in the following two chapters, EKF SLAM and GraphSLAM and their respective implementation is presented.

6 EKF SLAM

6.1 EKF Localization

When the vehicle is driving in the Trackdrive discipline, the map previously created in Autocross will be available. Therefore, it is not necessary to solve the SLAM problem in this discipline, but instead the vehicle must only find its location on said map. Since this problem is easier to solve, the localization algorithm shall be discussed first. Based on the general EKF localization algorithm, described in Chapter 3 (Algorithm 7), the implementation of the EKF localization is described in Algorithm 8.

The EKF localization algorithm begins with the prediction step. The actual prediction can be found in line 6 where the new mean belief is updated according to the motion model defined in Equation (5.8). In line 7, the covariance of the belief $\overline{\Sigma}_t$ is updated using the Jacobian of the motion model G_t that was computed in line 2 and the motion uncertainty Q_t that was computed in line 3 and mapped into state space in lines 4 and 5.

The update step starts in line 8. First, the measurement noise R_t is computed. Afterwards all measurements $z_t^{1:i}$ are associated to the map. For this, the JCBB implementation of the MRPT library[1] is utilized. To improve efficiency, only landmarks within the perception range of the vehicle are considered for association. If there are landmarks for which no associations could be found, they will not be further considered in the case of localization.

The matched landmarks, however, are used to update the belief. For each of them, the corresponding expected measurement $\hat{z}_t^j$ is computed in line 13. In line 14, the Jacobian of the measurement equation is computed for the current measurement, as described in Equation (5.17). Using these prerequisites, in lines 15 through 17, the Kalman equations are calculated to compute Kalman gain K_t and the updated mean of the belief μ_t and the covariance Σ_t.

6.2 EKF SLAM

If there is no map available to the robot, the difficulty increases significantly. To create that map and to find the location on it, i.e. solving the SLAM problem, all that is available to the robot are the measurements $z_{1:t}$ and the controls $u_{1:t}$.

6.2.1 The EKF SLAM Algorithm

The EKF SLAM algorithm is very similar to the EKF localization algorithm described in Section 6.1 (Algorithm 8). However, some adjustments have to be made for the algorithm to be able to solve the SLAM problem.

[1] Mobile Robot Programming Toolkit: *www.mrpt.org*

Algorithm 8: EKF Localization (Implementation)

Input: $\mu_{t-1}, \Sigma_{t-1}, u_t, z_t^{1:i}$

Output: μ_t, Σ_t

Prediction

$\varphi_v = \mu_{t-1,\varphi_v}$

$$G_t = \begin{pmatrix} 1 & 0 & (v_x sin(\varphi_v) - v_y \cos(\varphi_v))\Delta t \\ 0 & 1 & (v_x \cos(\varphi_v) - v_y \sin(\varphi_v))\Delta t \\ 0 & 0 & \Delta t \end{pmatrix}$$

$$Q_{t,l} = \begin{pmatrix} \sigma_{v_x} & 0 & 0 \\ 0 & \sigma_{v_y} & 0 \\ 0 & 0 & \sigma_{\dot{\varphi}} \end{pmatrix}$$

$$V_t = \begin{pmatrix} \cos\varphi_v \Delta t & -\sin\varphi_v \Delta t & 0 \\ \sin\varphi_v \Delta t & \cos\varphi_v \Delta t & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$Q_t = V_t \, Q_{t,l} \, V_t^T$

$$\bar{\mu}_t = \mu_{t-1} + \begin{pmatrix} \cos(\varphi_v) v_x \Delta t - \sin(\varphi_v) v_y \Delta t \\ \sin(\varphi_v) v_x \Delta t + \cos(\varphi_v) v_y \Delta t \\ \dot{\varphi} \Delta t \end{pmatrix}$$

$\overline{\Sigma}_t = G_t \Sigma_{t-1} G_t^T + Q_t$

Update

$$R_t = \begin{pmatrix} \sigma_r & 0 \\ 0 & \sigma_\theta \end{pmatrix}$$

associateObservedLandmarks()

forall *matched landmarks* z_t^j **do**

$$\delta = \begin{pmatrix} \delta_x \\ \delta_y \end{pmatrix} = \begin{pmatrix} \mu_{x_{j,g}} - \mu_{x_v} \\ \mu_{y_{j,g}} - \mu_{y_v} \end{pmatrix}$$

$q = \delta^T \delta$

$$\hat{z}_t^j = \begin{pmatrix} \sqrt{q} \\ atan2(\delta_y, \delta_x) \end{pmatrix}$$

$$H_t = \begin{pmatrix} -\sqrt{q}\delta_x & -\sqrt{q}\delta_y & 0 \\ \delta_y & -\delta_x & -q \end{pmatrix}$$

$K_t = \overline{\Sigma}_t H_t^T (H_t \overline{\Sigma}_t H_t^T + R_t)^{-1}$

$\bar{\mu}_t = \bar{\mu}_t + K_t(z_t^j - \hat{z}_t^j)$

$\overline{\Sigma}_t = (I - K_t H_t)\overline{\Sigma}_t$

end

$\mu_t = \bar{\mu}_t$

$\Sigma_t = \overline{\Sigma}_t$

return μ_t, Σ_t

The state is expanded to additionally contain information about the landmark states because they become part of the estimation. The orientation of the cones that will be used as landmarks in the algorithm cannot be measured with the available sensors and is therefore of no use. Instead, only the respective x- and y-position of a landmark will be considered and estimated. The resulting state vector will look like this:

$$x = \begin{pmatrix} x_v & y_v & \varphi_v & x_{1,g} & y_{1,g} & x_{2,g} & y_{2,g} & \dots \end{pmatrix}^T \tag{6.1}$$

Here, $x_{1,g}$ and $y_{1,g}$ describe the coordinates of the first landmark in the global reference frame and so on. The state vector is therefore of size $(2N + 3) \times 1$ where N is the number of mapped landmarks.

Operating with higher-dimensional vectors and matrices of course increases computational complexity. Therefore, if not all entries of a vector or a matrix need to be adjusted in an equation, block operations are utilized where only parts of the vector or the matrix are changed. This is expressed using *vector.block(beginRow, numberOfRows)* and *matrix.block(beginRow, beginColumn, numberOfRows, numberOfColumns)*. Due to the length of the algorithm, it is split into two parts. The prediction is described in Algorithm 9 and the update is described in Algorithm 10.

Prediction

Algorithm 9: EKF SLAM Prediction

Input: $\mu_{t-1}, \Sigma_{t-1}, u_t$

Output: $\bar{\mu}_t, \bar{\Sigma}_t$

1 $\varphi_v = \mu_{t-1,\varphi_v}$

2 $G_t = \begin{pmatrix} 1 & 0 & (v_x sin(\varphi_v) - v_y \cos(\varphi_v))\Delta t \\ 0 & 1 & (v_x \cos(\varphi_v) - v_y \sin(\varphi_v))\Delta t \\ 0 & 0 & \Delta t \end{pmatrix}$

3 $Q_{t,l} = \begin{pmatrix} \sigma_{v_x} & 0 & 0 \\ 0 & \sigma_{v_y} & 0 \\ 0 & 0 & \sigma_{\dot{\varphi}} \end{pmatrix}$

4 $V_t = \begin{pmatrix} \cos\varphi_v \Delta t & -\sin\varphi_v \Delta t & 0 \\ \sin\varphi_v \Delta t & \cos\varphi_v \Delta t & 0 \\ 0 & 0 & 1 \end{pmatrix}$

5 $Q_t = V_t \, Q_{t,l} \, V_t^T$

6 $\bar{\mu}_{t,v} = \mu_{t-1,v} + \begin{pmatrix} \cos(\varphi_v) v_x \Delta t - \sin(\varphi_v) v_y \Delta t \\ \sin(\varphi_v) v_x \Delta t + \cos(\varphi_v) v_y \Delta t \\ \dot{\varphi} \Delta t \end{pmatrix}$

7 $\overline{\Sigma}_t = \begin{pmatrix} G_t \Sigma_{vv,t-1} G_t^T & G_t \Sigma_{vm,t-1} \\ (G_t \Sigma_{vm,t-1})^T & \Sigma_{mm,t-1} \end{pmatrix} + Q_t$

8 **return** $\bar{\mu}_t, \overline{\Sigma}_t$

Since the environment is assumed to be static, in the prediction step only the vehicle state is adjusted while the landmark states remain constant. The first 5

lines are equal to the respective lines in the EKF localization algorithm described in Algorithm 8. Line 6 looks very similar, too, but it should be noted that a block operation is performed here where only the first three entries of the state vector are adjusted.

The belief of the covariance matrix is split into four blocks to increase the efficiency of the algorithm:

$$\Sigma_t = \begin{pmatrix} \Sigma_{vv,t} & \Sigma_{vm,t} \\ \Sigma_{mv,t} & \Sigma_{mm,t} \end{pmatrix} \tag{6.2}$$

$\Sigma_{vv,t}$ is the 3×3 block in the top left corner, $\Sigma_{mm,t}$ is the $2N \times 2N$ block in the lower left corner and $\Sigma_{vm,t}$ and $\Sigma_{mv,t}$ are $3 \times 2N$ and $2N \times 3$ matrices in the top right and the botton left corner, respectively. Since the covariance matrix is symmetric, it follows that:

$$\Sigma_{vm,t}^T = \Sigma_{mv,t} \tag{6.3}$$

Since the map wasn't adjusted in the prediction, neither is $\Sigma_{mm,t}$, which in practice is by far the largest block.

Update

The update step described in Algorithm 10 begins by performing the data association. This step is identical to the EKF localization algorithm. However, landmarks that could not be matched will be added to the map. This is done using Algorithm 11, which is explained below.

The update of the EKF SLAM algorithm varies to that of the EKF localization algorithm. Because the covariance matrix may be a very large matrix in the case of EKF SLAM, it will only be adjusted once per time step by using a so called *batch update*. As before, the expected measurement $\hat{z}_t^j$ and its respective difference Δz to the actual measurement z_t^j is computed for each observation in lines 7 to 10. But instead of performing the update directly, the computed information is accumulated in a $2i \times 1$ matrix with i being the total number of measurements in the time step.

Similarly, the Jacobi Matrix H is computed for each measurement. Since the measurement equation only depends on the vehicle state and the state of the currently looked at landmark, only the respective columns of H are non-zero. To increase computational efficiency, only the non-zero elements are computed:

$$H_{t,v}^j = \frac{\partial h(\bar{\mu}_t, j, m)}{\partial \bar{\mu}_v} = \begin{pmatrix} -\sqrt{q}\delta_x & -\sqrt{q}\delta_y & 0 \\ \delta_y & -\delta_x & -q \end{pmatrix} \tag{6.4}$$

$$H_{t,j}^j = \frac{\partial h(\bar{\mu}_t, j, m)}{\partial \bar{\mu}_j} = \begin{pmatrix} \sqrt{q}\delta_x & \sqrt{q}\delta_y \\ -\delta_y & \delta_x \end{pmatrix} \tag{6.5}$$

Again, the computed data is accumulated in a matrix with two lines per measurement. The Jacobian with respect to the vehicle state $H_{t,v}^j$ is inserted into the first three columns while the Jacobian with respect to the landmark state $H_{t,j}^j$ is inserted into the columns that correspond to its position in the state vector (its unique *id* id_j). To conclude the iteration, the accumulated measurement uncertainty matrix R_t is filled. R_t is zero except for its main diagonal.

Finally, the Kalman equations are calculated in lines 18 to 22.

Algorithm 10: EKF SLAM Update

Input: $\bar{\mu}_t, \overline{\Sigma}_t, z_t^{1:i}$

1 associateObservedLandmarks()
2 initializeUnmatchedLandmark()
3 $H_t = Zeros(2\ i, N)$ // i = number of matched measurements, N = number of states
4 $\Delta z_t = Zeros(2\ i, 1)$
5 $R_t = Zeros(2\ i, 2\ i)$
6 **forall** *matched landmarks* $z_t^j = \begin{pmatrix} r_t^j & \theta_t^j \end{pmatrix}^T$ **do**
7 $\quad \delta = \begin{pmatrix} \delta_x \\ \delta_y \end{pmatrix} = \begin{pmatrix} \bar{\mu}_{x_{j,g}} - \bar{\mu}_{x_v} \\ \bar{\mu}_{y_{j,g}} - \bar{\mu}_{y_v} \end{pmatrix}$
8 $\quad q = \delta^T \delta$
9 $\quad \hat{z}_t^j = \begin{pmatrix} \sqrt{q} \\ atan2(\delta_y, \delta_x) \end{pmatrix}$
10 $\quad \Delta z_t.block(2j, 2) = z_t^j - \hat{z}_t^j$
11 $\quad H_{t,v}^j = \begin{pmatrix} -\sqrt{q}\delta_x & -\sqrt{q}\delta_y & 0 \\ \delta_y & -\delta_x & -q \end{pmatrix}$
12 $\quad H_{t,j}^j = \begin{pmatrix} \sqrt{q}\delta_x & \sqrt{q}\delta_y \\ -\delta_y & \delta_x \end{pmatrix}$
13 $\quad H_t.block(2j, 0, 2, 3) = H_{t,v}^j$
14 $\quad H_t.block(2j, 2\ id_j + 3, 2, 2) = H_{t,j}^j$
15 $\quad R_t(2j, 2j, 1, 1) = \sigma_r$
16 $\quad R_t(2j+1, 2j+1, 1, 1) = \sigma_\theta$
17 **end**
18 $K_t = \overline{\Sigma}_t H_t^T (H_t \overline{\Sigma}_t H_t^T + R_t)^{-1}$
19 $\bar{\mu}_t = \bar{\mu}_t + K_t \Delta z_t$
20 $\overline{\Sigma}_t = (I - K_t H_t)\overline{\Sigma}_t$
21 $\mu_t = \bar{\mu}_t$
22 $\Sigma_t = \overline{\Sigma}_t$
23 **return** μ_t, Σ_t

Initializing New Landmarks

Algorithm 11: Initializing New Landmarks

Input: $\bar{\mu}_t, \overline{\Sigma}_t\ z_t^{1:k}$

1 **forall** *unmatched measurements* $z_t^i = \begin{pmatrix} r_t^i & \theta_t^i \end{pmatrix}^T$ **do**

2 $\quad \bar{\mu}_{i,x} = \bar{\mu}_{x_v} + r_t^i \cos(\bar{\mu}_{\varphi_v} + \theta_t^i)$

3 $\quad \bar{\mu}_{i,y} = \bar{\mu}_{y_v} + r_t^i \sin(\bar{\mu}_{\varphi_v} + \theta_t^i)$

4 $\quad \bar{\mu}_t = \begin{pmatrix} \bar{\mu}_t & \bar{\mu}_{i,x} & \bar{\mu}_{i,y} \end{pmatrix}^T$

5 $\quad H_v^{\text{inv}} = \begin{pmatrix} 1 & 0 & -r_j \sin(\bar{\mu}_{\varphi_v} + \theta_t^i) \\ 0 & 1 & r_j \cos(\bar{\mu}_{\varphi_v} + \theta_t^i) \end{pmatrix}$

6 $\quad H^{\text{inv}} = Zeros(2, N)$

7 $\quad H^{\text{inv}}.block(0, 0, 3, 3) = H_v^{\text{inv}}$

8 $\quad H_i^{\text{inv}} = \begin{pmatrix} \cos(\bar{\mu}_{\varphi_v} + \theta_t^i) & -r_k \sin(\bar{\mu}_{\varphi_v} + \theta_t^i) \\ \sin(\bar{\mu}_{\varphi_v} + \theta_t^i) & r_k \cos(\bar{\mu}_{\varphi_v} + \theta_t^i) \end{pmatrix}$

9 $\quad R_t^i = \begin{pmatrix} \sigma_r & 0 \\ 0 & \sigma_\theta \end{pmatrix}$

10 $\quad \overline{\Sigma}_t = \begin{pmatrix} \overline{\Sigma}_t & \overline{\Sigma}_t^T H^{\text{inv}} \\ H^{\text{inv}} \overline{\Sigma}_t & H^{\text{inv}} \overline{\Sigma}_t H^{\text{inv},T} + H_i^{\text{inv}} R_t^i H_i^{\text{inv},T} \end{pmatrix}$

11 **end**

12 **return** $\bar{\mu}_t, \bar{\Sigma}_t$

To initialize newly observed landmarks, the algorithm described in Algorithm 11 is used. First, the measurement is transformed into state space using the inverse measurement model described in Section 5.3. This new state is appended to the current predicted belief $\bar{\mu}_t$ in line 4.

The covariance matrix is updated using the Jacobians of the inverse measurement model h^{inv}. The Jacobian with respect to the measurement H_i^{inv} is used to transform the measurement uncertainty into state space. The Jacobian with respect to the state is computed in lines 5 to 7. Since the Jacobian is zero at the columns corresponding to the landmark states, only the first 2×3 block is non-zero.

The new covariance matrix is defined in line 10 by appending 2 new lines and columns that are computed using the Jacobians described above.

6.2.2 EKF SLAM Implementation

While the EKF SLAM algorithm utilizes the equations described above, some specific implementation details should be explained. There are a number of factors that required a different implementation to the usual Kalman architecture.

First of all, the high speeds of the car require very high update rates. The controller works with a frequency of 50 Hz, so a new pose estimate should be delivered at least every 20 ms.

Secondly, the computation times of the prediction and the update step differ quite significantly. In the prediction step, only the vehicle pose and the respective blocks of the covariance needs to be manipulated, while in the update step, the full map is corrected and therefore the complete state and covariance matrix need to be adjusted. Additionally, in the update step, the data association needs to be performed, which may require a significant amount of the available computation time depending on the amount of landmarks seen in a particular time step.

Finally, the sensors delivering the input data for the algorithm work at different frequencies. Odometry data is available at roughly 100 Hz or every 10 ms, whereas a perception scan is only delivered at 25 Hz or every 40 ms. It would of course be possible to only use every fourth odometry measurement and to predict over a longer time step, i.e. 40 ms. This, however, causes problems with the real-time capability of the overall system, due to the update frequency the controller requires.

Linear Structure

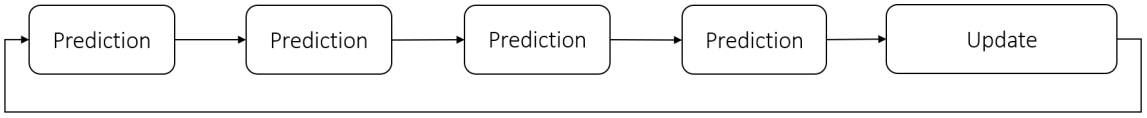


Figure 6.1: Linear structure of the EKF SLAM algorithm.

Originally, the algorithm was implemented as depicted in Figure 6.1. The Kalman filter would perform multiple predictions before an update step was computed. While this implementation did work in principal and solved the problems of the different input frequencies and the required output frequency, the algorithm turned out to not be real-time capable. The reason for that is the amount of time taken by the update step. Due to the linear architecture, the update step needed to be completed before the next prediction started, which often wasn't the case.

Parallel Structure

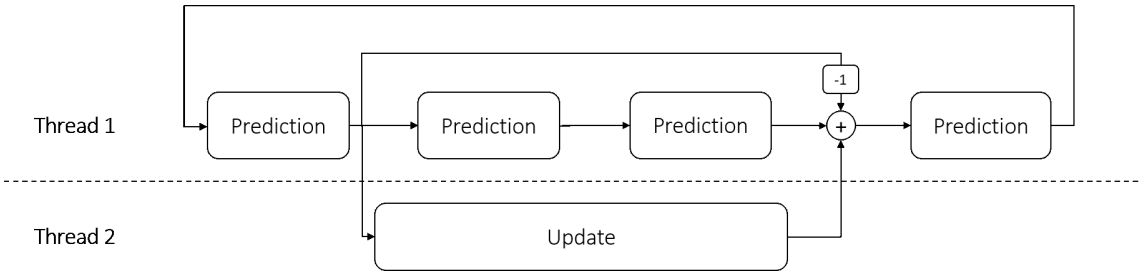


Figure 6.2: Parallel structure of the EKF SLAM algorithm.

The answer to the problem of real-time capability is parallelization. Figure 6.2 depicts the architecture that the implemented algorithm works with. It is very similar to the linear structure shown in Figure 6.1, however, the update step is moved onto a second thread. Doing so allows the algorithm to keep processing odometry data and publishing predicted pose estimates while in the background computing the update. Once the update is completed, the pose the update started

with is corrected. To account for the predictions made since the update started, the pose delta since then is added onto the updated pose.

This process solves all of the problems mentioned above: The input data is processed at different frequencies, there is always a short-term pose estimate for the controller to work with and the update step may take up all of the available time until a new measurement is available. In fact, the algorithm has proven to be robust enough to work with update steps taking even longer than that. In that case, information is lost because the algorithm has to drop incoming messages but the predictions can continue without interruption.

Example

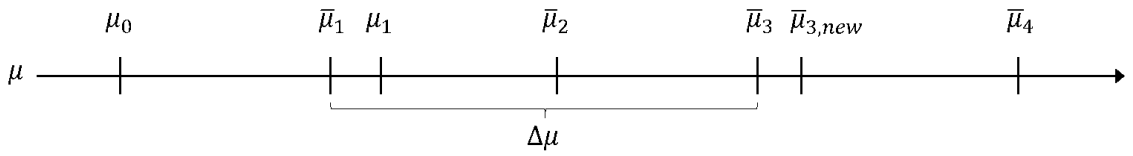


Figure 6.3: Exemplary estimates made by the parallel EKF SLAM implementation.

Figure 6.3 shows a 1-dimensional example of the estimates made by the parallel structure of EKF SLAM. The input in this example will be (in order):

- One odometry measurement
- One set of observations
- Three odometry measurements

In this example, computing a prediction will take 1 ms, whereas computing an update will take 25 ms. Just as in the hardware platform presented in Section 4.2, odometry measurements will be available every 10 ms, while observations of the environment are delivered every 40 ms.

Starting from the robot's initial belief μ_0, the **first received odometry measurement** will trigger a prediction which will result in a new predicted estimate $\bar{\mu}_1$. Next, the perception pipeline delivers a **new set of measured landmarks**. This allows the SLAM algorithm to correct the estimate made in the prediction step just before.

However, since the update step may take up a long time, it is computed on a second thread. Because in this example, computing the update step will take 25 ms, **two odometry measurements** will be received in the meantime. Without the parallelization, these measurements could not be taken into account and the information would be lost. With the parallelization, however, the odometry measurements will trigger predictions, even though the update is computed in the background. Since $\bar{\mu}_1$ is the most current estimate, the predictions will continue from that state, resulting in the predicted states $\bar{\mu}_2$ and $\bar{\mu}_3$. This allows the SLAM algorithm to continuously publish vehicle pose estimates with a constant frequency of 100 Hz, ensuring that there is always an up-to-date pose estimate available to the controller.

Once the **update step has been completed**, there is a corrected estimate μ_1 for the state that the update step started with, $\bar{\mu}_1$ in this example. Next, the algorithm needs to account for the predictions made since the update started. It does so, by

first computing the delta of the states from the estimate the algorithm started with and the most current estimate. In this example, this means:

$$\Delta\bar{\mu} = \bar{\mu}_3 - \bar{\mu}_1 \tag{6.6}$$

The computed delta is then added onto the updated state μ_1 to adjust the newest predicted pose $\bar{\mu}_3$, resulting in $\bar{\mu}_{3,new}$:

$$\bar{\mu}_{3,new} = \mu_1 + \Delta\bar{\mu} \tag{6.7}$$

Note that this is still a predicted state, even though it is only available after an update step is completed. Also, when computing $\Delta\bar{\mu}$, only the difference in the vehicle pose needs to be considered. The environment is assumed to be static so it is not changed in the prediction step.

Finally, the **last odometry measurement** is delivered and a new predicted state $\bar{\mu}_4$ is computed starting from the most current state estimate $\bar{\mu}_{3,new}$. After this, the cycle repeats.

Even though the update step took very long, no information was lost in this example. This would even be the case if computing the update step took up to 40 ms (time until new observations are delivered).

Before the EKF SLAM algorithm presented in this chapter is analysed in terms of its performance, the GraphSLAM algorithm is presented in Chapter 7.

7 GraphSLAM

The GraphSLAM algorithm differs from the EKF SLAM algorithm in many ways. Instead of filtering the input data (control inputs $u_{1:t}$ and measurements $z_{1:t}$), GraphSLAM, as the name suggests, builds a graph containing all available information in the form of mathematical constraints. This allows the algorithm to solve the full SLAM problem, i.e. to find an estimate for all landmarks and *all* vehicle poses. The task of the algorithm is to build the graph from the given control inputs and measurements, to formulate a minimization problem and to finally find the configuration for all vehicle poses and landmark positions that best fulfills all constraints by solving this minimization problem.

Extracting the best estimate by solving the graph is computationally expensive. Adding a new constraint to the graph, however, is a very cheap operation. This property will later be used in the implementation to allow the algorithm to perform in real-time.

The **computational complexity** of the algorithm is linear in the number of constraints. As a consequence, the GraphSLAM's complexity increases over time opposed to the EKF SLAM which has a complexity that grows only when the map size increases.

Keeping track of all constraints is a crucial advantage of the GraphSLAM algorithm because it allows for the motion and measurement equations to be linearized again at a later point in time. EKF SLAM is not able to do this, which is why GraphSLAM promises more accurate estimates.

The GraphSLAM algorithm presented in this chapter is implemented utilizing the g^2o library [21].

7.1 Graph Structure

The graph used to represent the SLAM problem is a weighted, directional graph consisting of two types of edges and two types of vertices[1]. The graph structure is visualized in 7.1.

Each vehicle pose is represented by a **pose vertex** and each mapped landmark is represented by a **landmark vertex**. Consecutive pose vertices are connected by **odometry edges**. Landmark vertices are connected to all vehicle poses they have been measured from using **landmark edges**.

The GraphSLAM will be represented using the information matrix Ω and the information vector ξ. New edges will result in a local addition of an edge to the information matrix.

[1] *Vertex* and *node* are synonymous in this context and will be used as such.

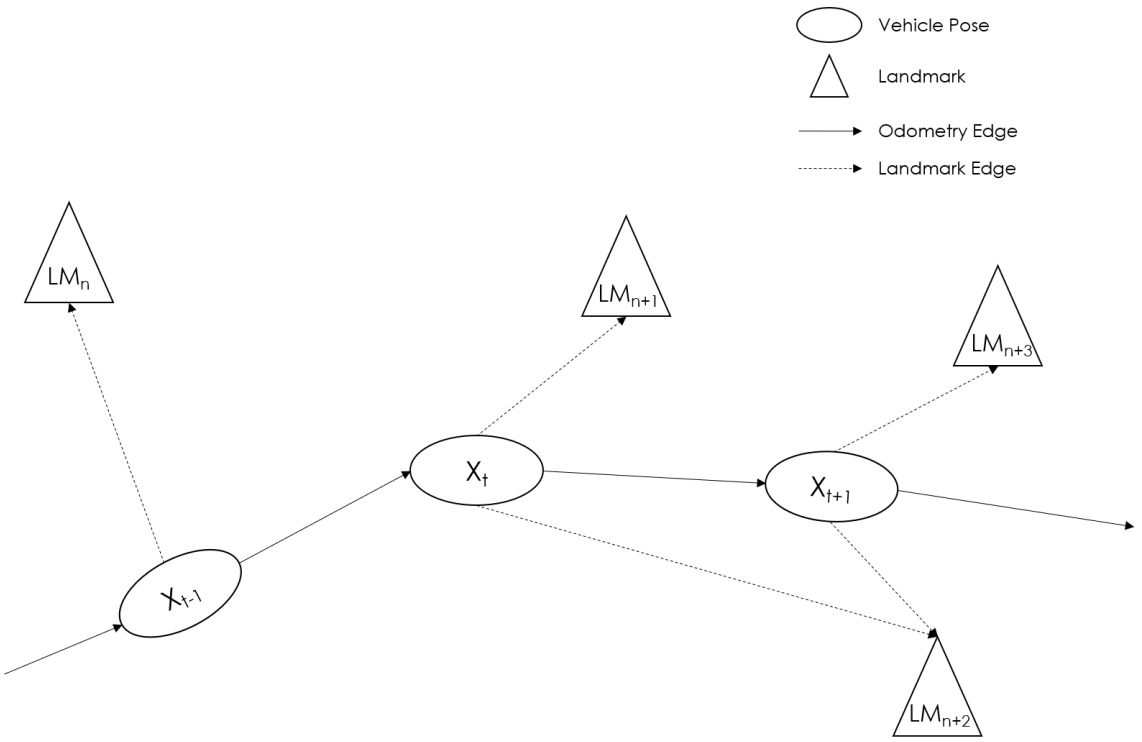


Figure 7.1: Structure of the graph representing the SLAM problem.

7.2 The GraphSLAM Algorithm

Just as the EKF SLAM algorithm, the GraphSLAM algorithm uses the control command $u_{1:t}$ and the measurements $z_{1:t}$ as an input. Each of these inputs triggers an event, wich results in the creation of new edges and possibly new vertices.

A **new odometry input** is used to generate a new pose vertex using the motion model described in Section 5.2. Additionally, the new pose vertex is connected with the pose node before it. The constraint that links these poses can be formulated as follows:

$$(x_t - g(u_t, x_{t-1}))^T Q_t^{-1} (x_t - g(u_t, x_{t-1})) \tag{7.1}$$

A **new observation** will first of all trigger the data association. Observations that were able to be matched to a mapped landmark will be linked to the vehicle pose that the measurement had been made from using a landmark edge. The resulting constraint is:

$$(z_t^i - h(x_t, m))^T R_t^{-1} (z_t^i - h(x_t, m)) \tag{7.2}$$

For each **unmatched observation**, i.e. for landmarks that have never been seen before, a new landmark vertex will be created and connected using a landmark edge just as the matched landmarks above. h in Equation (7.2) is the measurement model discussed in Section 5.3.

The resulting information matrix will be zero everywhere except for the entries on the main diagonal and those corresponding to an edge. For example, the structure of the information matrix corresponding to Figure 7.1 is visualized in Table 7.1.

Table 7.1: Structure of the information matrix corresponding to Figure 7.1.

	x_{t-1}	x_t	x_{t+1}	LM_n	LM_{n+1}	LM_{n+2}	LM_{n+3}
x_{t-1}	■	■		■			
x_t	■	■	■		■	■	
x_{t+1}		■	■			■	■
LM_n	■			■			
LM_{n+1}		■			■		
LM_{n+2}		■	■			■	
LM_{n+3}			■				■

The sum of all constraints has the form:

$$\begin{aligned} J_{\text{GraphSLAM}} =& x_0^T \Omega_0 x_0 + \sum_t (x_t - g(u_t, x_{t-1}))^T Q_t^{-1} (x_t - g(u_t, x_{t-1})) \\ &+ \sum_t \sum_i (z_t^i - h(x_t, m))^T R_t^{-1} (z_t^i - h(x_t, m)) \end{aligned} \tag{7.3}$$

The first constraint $x_0^T \Omega_0 x_0$ is called **anchoring constraint** and fixes the problem to a global reference frame by initializing the first vehicle pose to $x_0 = (0, 0, 0)^T$. This is needed because all constraints discussed above are relative constraints and hold no information about the global reference frame. Without the anchoring constraint, the cost function is invariant to a rigid body transformation, which results in the system of equations being under-determined.

The cost function $J_{\text{GraphSLAM}}$ is a sum of squared errors which has the structure of Equation (3.56). The task of the algorithm is to minimize all errors, i.e. to minimize the cost function. This is done using the Levenberg-Marquardt algorithm as explained in Section 3.3.

7.2.1 Initializing New Landmarks

To provide the best initial guess to the optimization algorithm, in Autocross newly observed landmarks are mapped using the inverse measurement equation discuessed in Section 5.3.1:

$$\begin{pmatrix} x_{j,g} \\ y_{j,g} \end{pmatrix} = \begin{pmatrix} x_v + r_j \cos(\varphi_v + \theta_j) \\ y_v + r_j \sin(\varphi_v + \theta_j) \end{pmatrix} \tag{7.4}$$

7.2.2 Fixing Nodes

Depending on the discipline that the vehicle is in, specific nodes may need to be fixed. In Autocross, the very first vehicle pose node is fixed. By doing so, the connecting odometry edge acts as an anchoring constraint, as explained above.

In Trackdrive, the map is available from the start and will not be adjusted anymore. Although it would be possible to further correct the map, in Trackdrive the vehicle tends to drive faster than it would in Autocross. This results in less accurate sensor measurements which is why the map quality would likely not improve during Trackdrive.

In practice, that means that all nodes representing landmarks will be fixed when the map is loaded initially. They will then not be further corrected during the optimization. For the application of the algorithm this is a great advantage because it essentially works the same in both Autocross and Trackdrive.

The j-th node can be fixed by manipulating the H-matrix in Equation (3.67) in one of two ways:

- The identity matrix is added to the block corresponding to the j-th node.
- The rows and columns corresponding to the j-th node are suppressed and not considered during the optimization.

7.2.3 Outlier Handling

Since neither the sensors nor the perception pipeline deliver perfect results, there is always the chance of outlier measurements. Outlier measurement may have a serious impact on the estimation quality because the optimization problem is formulated using squared errors. A squared error cost function is a good way to quickly correct bigger deviations of the correct state. However, in the case of outlier measurements, they imply a big deviation that is in fact not correct and results in a loss of accuracy if it is not dealt with.

To prevent outliers from having a fatal impact, the cost function is formulated in a slightly adjusted way using a so called **Huber kernel**. The idea is to penalize errors up to a certain threshold in a quadratic manner but to not increase that penalty as drastically if the error is above the threshold. This is done using the Huber loss function:

$$J_{\text{Huber}}(e) = \begin{cases} \frac{e^2}{2} & \text{if } |e| < \delta \\ \delta(|e| - \frac{c}{2}) & \text{otherwise} \end{cases} \tag{7.5}$$

Here, δ is the parameter defining the error threshold.

The Huber loss function is visualized and compared to the squared error loss function in Figure 7.2.

7.3 GraphSLAM Implementation

Just as EKF SLAM, GraphSLAM must process incoming odometry data and observations. However, in GraphSLAM the correction of the map and the vehicle pose is moved into a separate step, the optimization. The computational complexity of the optimization is linear in the number of edges, so depending on the number of edges, the optimization may take a long time. Because of this, the implementation of the GraphSLAM algorithm is parallelized. The key idea is to move the optimization onto a second thread so that it can run in the background, while the algorithm is still collecting all incoming data. The implementation of the two threads is discussed in the next section.

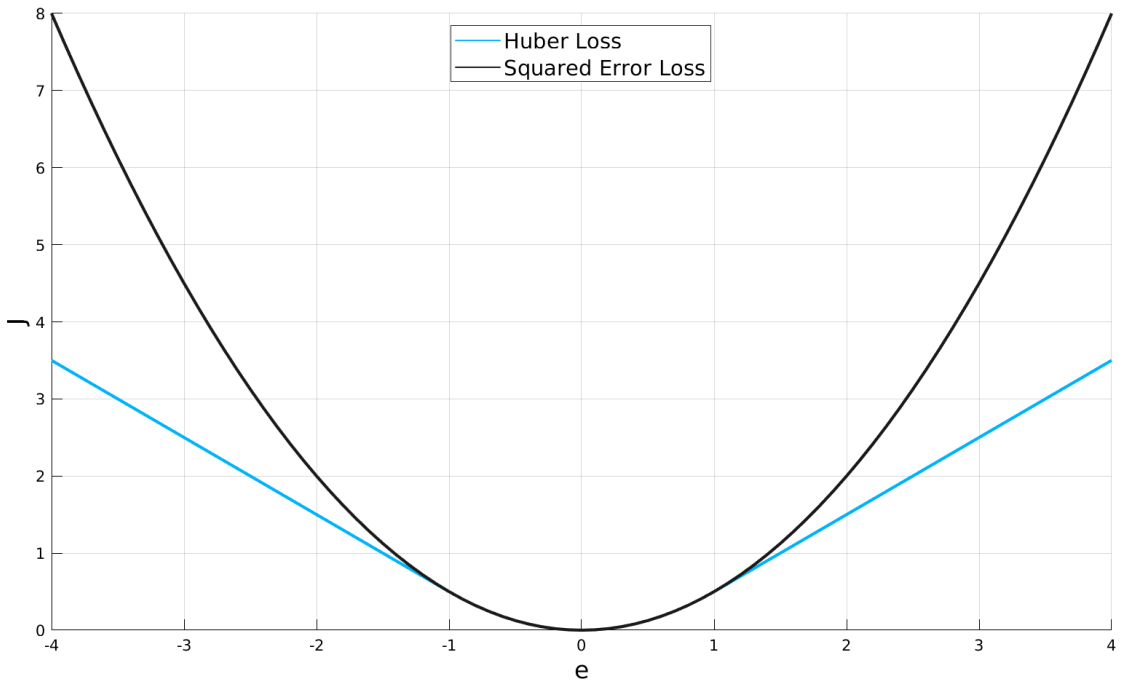


Figure 7.2: Comparison of the Huber loss function (blue, $\delta = 1$) and the squared error loss function (black).

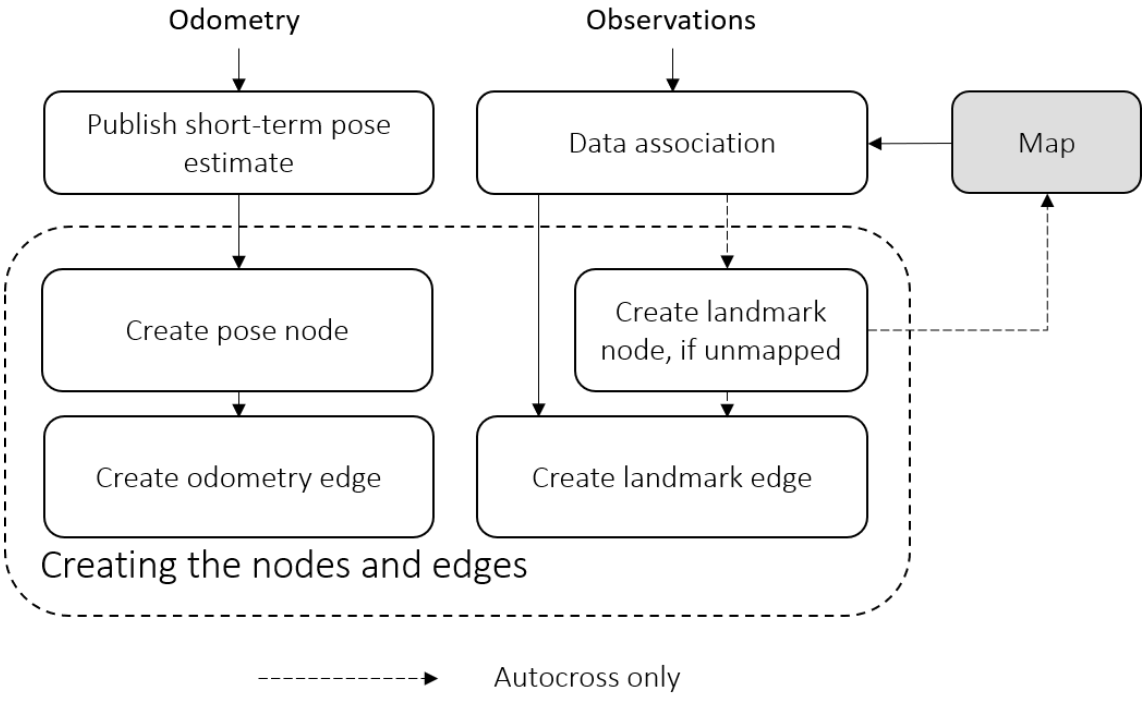


Figure 7.3: Architecture of thread 1 of the GraphSLAM implementation.

7.3.1 Thread 1

Figure 7.3 shows a flow chart of the first thread. As described above, the task of the first thread is to **collect all incoming data**. Additionally, it publishes short-term vehicle pose estimates, so that there is always a continuous, high-frequency estimate available to the controller. Whenever new data is available, a function is triggered within the code.

Incoming **odometry data** is used to first of all publish a new pose estimate using the motion model described in Section 5.2. Next, the corresponding graph objects, a new pose node and odometry edge, are created based on the new, estimated pose. However, these graph objects are not added to the graph yet. This is to allow thread 2 of the implementation to work on an optimization in the background. Once an optimization has been started, it is not possible to change the optimization problem by adding new nodes and edges to the graph, therefore, the new graph objects will instead be stored in a buffer.

Incoming **observations** of the environment must first of all be associated to the map, using the algorithm described in Section 3.4. Depending on the current discipline, i.e. depending on whether or not the algorithm works on a given map, it will add unknown landmarks to the map and create a landmark node or it will discard them. For each measurement, a landmark edge is created. Again, the nodes and edges created in this step will not be added to the graph yet.

7.3.2 Thread 2

A flow chart of the second thread is depicted in Figure 7.4. The task of this thread is to use all the information collected in the first thread to **build and solve the graph**, increasing the overall accuracy of the estimated path and map.

Because a graph optimization is a computationally expensive process, thread 2 is usually slower than thread 1. This means that when thread 2 starts over, it is likely that thread 1 created some nodes and edges in the meantime. Thread 2 will then empty the buffer created in thread 1 to extend the graph with these new graph objects. In the case of Trackdrive, where a map will be loaded at the start of the run, the landmark nodes are fixed because the map will not be further optimized in this discipline.

Now the graph optimization can be started. Iteratively, the motion and measurement equations are linearized around the current estimates of the respective nodes, the cost function is built and minimized. In Autocross, once the optimization is finished, the map will be updated with the new estimates. After this, thread 2 can start over by adding new nodes and edges created in thread 1.

7.3.3 Adjusting New Poses

Similar to the parallel EKF SLAM implementation presented in Section 6.2.2, after an optimization is done, the short-term pose estimates made since the optimization started must be adjusted. This is done by applying the motion model to all poses computed since the latest optimized pose using all velocity estimates made in the time the graph was being optimized. Because this operation is performed on the

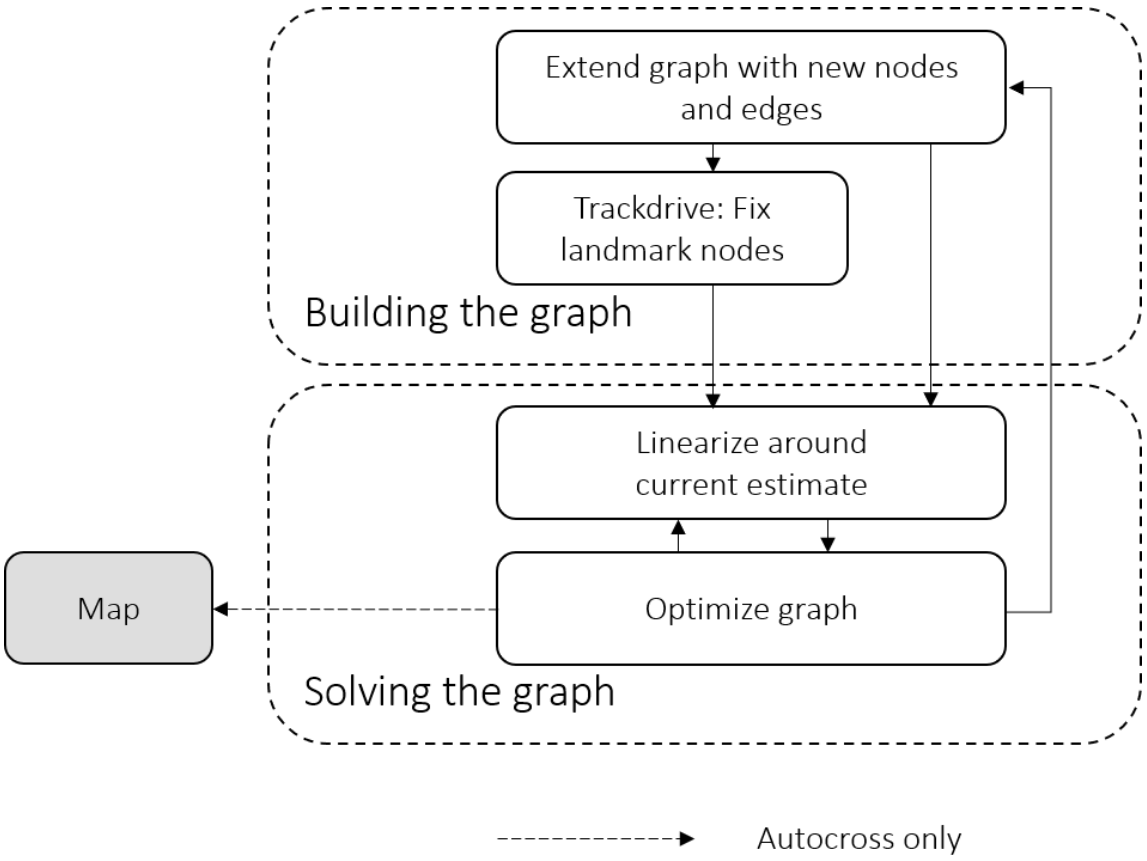


Figure 7.4: Architecture of thread 2 of the GraphSLAM implementation.

same data that the odometry processing works on, the latter has to wait for this adjustment to finish.

7.3.4 Sliding Window

The GraphSLAM algorithm presented in this thesis has a sliding window implemented that may be activated if necessary. The idea behind this approach is to reduce the computational burden of the graph optimization step by reducing the number of edges and therefore the number of constraints. The sliding window can be adjusted by a parameter (online or offline) that limits the number of edges that the graph will use in the optimization. Of course, this approach is a trade-off: Reducing the number of edges improves the algorithm's efficiency but reduces its accuracy.

In Chapters 6 and 7, both algorithms and their respective implementation were presented. The next chapter discusses the evaluation of both algorithms with respect to their performance to make an objective decision about which algorithm will be used in the future.

8 Experimental Evaluation

Figure 8.1 shows a visualization of the map and path estimated by both of the algorithms. From looking at these results, it can be concluded that the algorithms work as intended and that the SLAM problem can be solved in principle. However, drawing a conclusion on which of the algorithms worked better (and how much better) than the other is not possible when only considering the visualization of the resulting maps and paths.

To determine whether a SLAM algorithm works well or not, two things should be considered. The

- accuracy and the
- efficiency

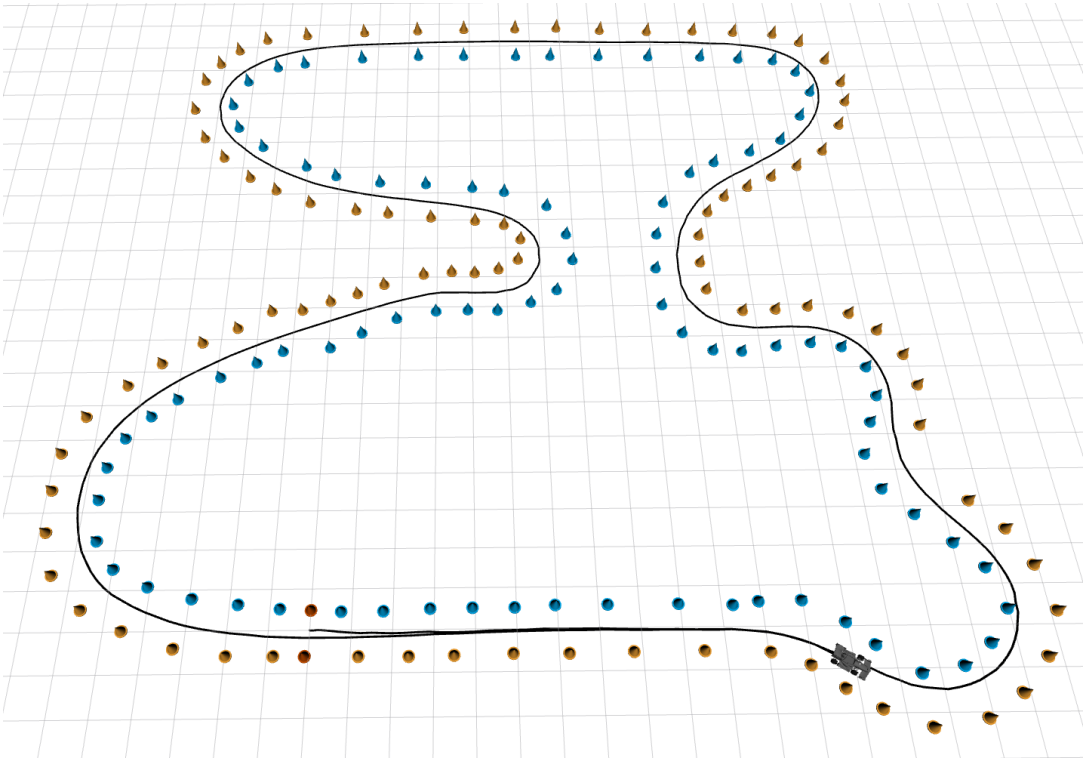
of the algorithm.

The latter can be measured rather easily. To measure the **efficiency** of an algorithm, two measures were used: The time taken to process incoming data (and in the case of GraphSLAM, the time taken for the optimization itself) and the CPU usage over the course of a lap, or in the case of Trackdrive: Ten laps.

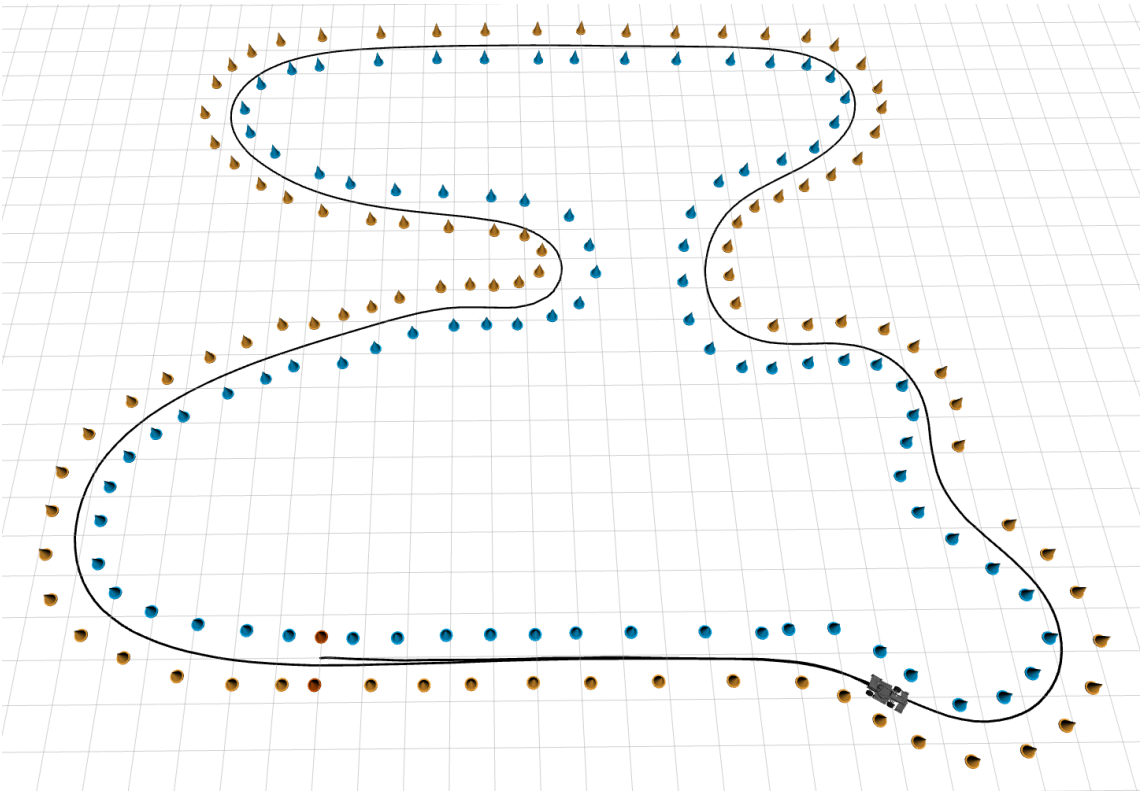
The **run times** are important because if one of the callbacks (e.g. processing odometry data) takes too long, the next message cannot be processed in time and information will be lost. While both algorithms are robust enough to handle dropped messages occasionally, if this occurs too often, it is likely for the SLAM algorithm to diverge. The **CPU usage** is another important factor. For the vehicle to be able to drive autonomously, a number of algorithms and services need to run at the same time. Namely the perception pipeline, the trajectory planning, the controller, the sensor drivers and a supervisor managing all algorithms. All these algorithms tend to become more complex as their development continues. Additionally, the computational resources are limited by external factors, such as the budget, cooling, packaging and power consumption, so keeping the CPU usage of the individual algorithms as low as possible is essential.

Measuring the **accuracy** is a difficult problem in itself. When comparing the resulting map and path of the algorithms, one might be able to spot differences in landmark positions. However, it is impossible to say which of the algorithms is closer to the correct placement of landmarks without having a reference. Luckily, it was possible to acquire such a reference in the form of a **ground truth map** of a track built on the KIT's campus. The positions of all cones have been marked on the ground and were measured using a high-precision Differential Global Positiong System (DGPS) device. The track layout is depicted in Figure 8.2. The DGPS measurements have an error below 1 cm. Considering the extent of the marking itself, it is safe to assume that the error of the measured cone positions is below 5 cm.

The performance of the algorithms is evaluated not only for the algorithms presented in this thesis but last year's algorithm is used as a base-line. To differentiate the algorithms, last year's algorithm will be referred to as "EKF SLAM 2019", whereas the EKF SLAM algorithm presented here will be referred to as "EKF SLAM 2020".



(a) EKF SLAM 2020



(b) GraphSLAM

Figure 8.1: The map and path estimated by the SLAM algorithms running on the same data set of one lap of Autocross. Cone size increased for better visibility.

Figure 8.2: Track layout that was measured using DGPS.

8.1 Data Sets

The performance analysis was conducted on two data sets, one for Autocross and one for Trackdrive. The Autocross data set was recorded on the campus of the KIT on above mentioned track. To better understand the results of the performance measurements, the data set was analyzed visually and checked for **false positive** and **false negative** measurements. The result is a total of:

- False positive measurements: 5
- False negative measurements: 3

As discussed in Section 5.5, the algorithms are set such that a cone will only be added to the map after it has been seen at least 3 times. This reduces the number of false positive landmarks on the map to 2. Increasing the parameter further than that is not sensible. Because of the delay in mapping introduced by it, the trajectory planning algorithm may not know the track far enough ahead and the car will have to reduce its speed.

The false negative measurements may be the result of something, e.g. another cone, disturbing the view so that it is never detected. This analysis is necessary because false positive and false negative measurements are errors occurring outside the scope of the SLAM algorithm. Therefore, they should not influence the performance measurements of the SLAM algorithms.

The second data set is a Trackdrive recording. It is from the FSG Trackdrive event of 2019. Since in Trackdrive the map is preloaded, an evaluation of the map will not help with the comparison of the algorithms. Therefore, the Trackdrive analysis does not necessarily have to be conducted on the measured track and will only be done with respect to the efficiency of the algorithms.

8.2 Architecture

When developing the performance measurements, the goal was to implement a tool that is **easy to use**, **expandable** and has a **minimal impact** on the measurements themselves. To achieve this, the accuracy measurements were implemented in a **separate ROS node** that runs **in parallel** to either one of the SLAM nodes.

The **efficiency** must be measured within the code of the SLAM algorithm, but measuring the efficiency has very little impact on the performance. Additionally, the efficiency measurements can be deactivated using a pre-compile directive, to completely remove the impact the measurements have on the overall system.

Measuring the **accuracy** is a process that takes more resources, but moving the tool into a separate node means that no code of the SLAM algorithm needs to be modified and that the measurements can be turned on and off as is preferred. Additionally, since both SLAM algorithms use the same interfaces, the performance measurements need to be implemented only once and the comparison could for example be expanded to other SLAM approaches. The accuracy and efficiency measurements will then be published in a unique namespace and can either be visualized live using *rqt_plot* or recorded and the visualized in a post-processing script developed in *Matlab*. Naturally, processing data in retrospective allows for some additional analysis, so that is the method used in all plots presented in this chapter.

8.3 Accuracy

8.3.1 Method

Data Association

As described in the beginning of this chapter, the data available to measure the SLAM algorithm's accuracy is the SLAM-generated map and the ground truth map. Most of the accuracy measures introduced below rely on a comparison of these two maps. To be able to compare a cone's estimated position to the ground truth, it must first be determined which cone on the SLAM generated map corresponds to which cone on the ground truth map.

The reader may recognize this problem as being essentially the same problem that the SLAM algorithms must solve to find the cone on the map corresponding to a current measurement. This process is called data association and was introduced in Section 3.4. To solve this problem, the same algorithm to solve the problem of data association can be re-used for the accuracy measurements.

Reference Frame Transformation

The next challenge that arises is that the two maps that shall be compared are not in the same frame of reference. The ground truth map was generated using Universal Transverse Mercator (UTM) coordinates. To make the following easier, the origin of the reference frame was then moved onto the center of the line where the vehicle will be staged when starting a run. According to the Formula Student rules [48], that is 6 m in front of the finish line. Finally, the reference frame is rotated such

that the x-axis points in the direction of the starting straight. The SLAM-generated map will have its origin at the point where the vehicle was started with the x-axis being aligned with the vehicle heading.

Moving the ground truth reference frame in the way described makes comparing the two maps easier, but small offsets and rotations between the two coordinate frames may have big impacts on cones far away from the origin. That will first of all make it harder to find a match on the reference map and more importantly will distort the accuracy measurements.

To solve this, the two reference frames must first be matched such that the overall error of all cone positions is as small as possible. Only then does it make sense to compare the positioning of corresponding cones on both maps. To achieve this, the **Iterative Closest Point (ICP)** implementation of the *pcl* library [22] is used. ICP is an algorithm that tries to find the transformation between two point clouds that reduces the sum of squared distances between the closest neighbor points. For this purpose, the maps are interpreted as point clouds. Because of the transformation described above, the maps will already match relatively well which makes it easy for ICP to find a close match.

Once the transformation has been found for a data set, the actual accuracy measurements can be performed.

8.3.2 Cone Count

Before comparing the two maps, looking at the number of cones on the SLAM-generated map can give a first idea of how well the algorithms performed. Since the total number of cones on the track is known, looking at the total number of mapped cones can give an insight to the following:

Too many mapped cones usually occur when the data association algorithm does not match cones that do in fact correspond, which is, for example, the case when overestimating a cone's certainty (see Section 5.4.3), when there are high errors in the cone positions or if the algorithm diverged.

Too few cones could for example point to a too restrictive filtering of false positive cones (see Section 5.5).

Figure 8.3 shows the results for one lap of Autocross. The number of cones are very similar for EKF SLAM 2020 and GraphSLAM but EKF SLAM 2019 tends to map too many cones, which is likely due to the lower accuracy of the algorithm. At the end of the run, EKF SLAM 2020 and GraphSLAM are mising one cone, which can be accounted to the false positive and false negative measurements discussed above in Section 8.1.

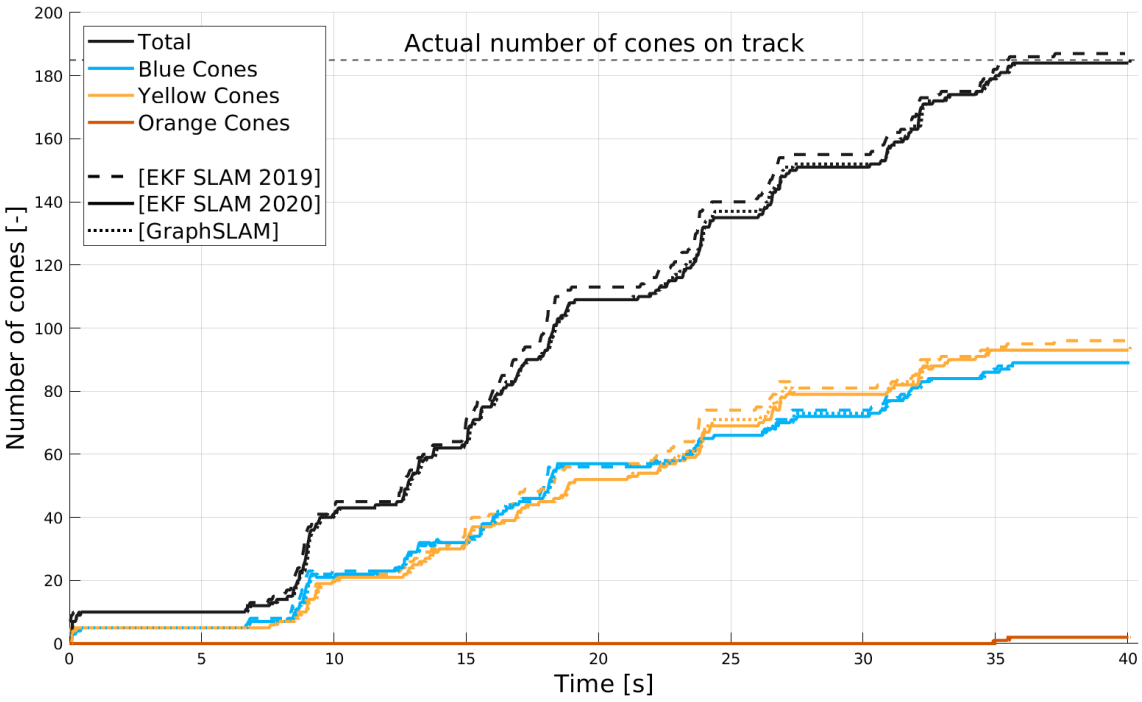


Figure 8.3: Autocross: Number of cones mapped over the course of the lap.

8.3.3 Matching Ratio and Error Threshold

The very first result, when comparing the SLAM-generated map to the ground truth map, is received when performing the data association. For how many cones of the SLAM-generated map was it possible to find a correspondence on the ground truth map? If a match could not be found, there are two possible reasons:

- The perception pipeline delivered a false positive measurement or
- the error of the estimated cone position is too big for the data association algorithm to find a match.

The **matching ratio** is calculated by dividing the number of matched cones by the number of cones on the SLAM generated map.

As described in Section 8.1, the number of false positive landmarks in the data set, after filtering, is 2. With a total of 185 cones on the track, the matching ratio at the end of the run, with an ideal SLAM on this data set, should be close to 99 %, which is the case for both EKF SLAM 2020 and GraphSLAM as can be seen in Figure 8.4. Both algorithms drop to values between 93 % and 97 % in the middle section but do recover from this. EKF SLAM 2019 drops to about 86 % in the middle section und finishes the lap with less than 90 % of matchings made.

Before looking at the concrete values of the errors of the estimated cone positions, another measure is of interest: The percentage of mapped cones with a **positioning error above a certain threshold**. In this case, the threshold is chosen to be 30 cm, which is roughly the diameter of a cone. Positioning errors below this threshold are less of an issue because the car will have to plan its trajectory with a safety margin anyway. Considering that Formula Student tracks are very narrow in comparison to the car, a larger positioning error will force the car to plan its trajectory on the middle line of the track in order to not hit any cones. While that was the case in the 2019 season, current developments of the trajectory planning and the controller

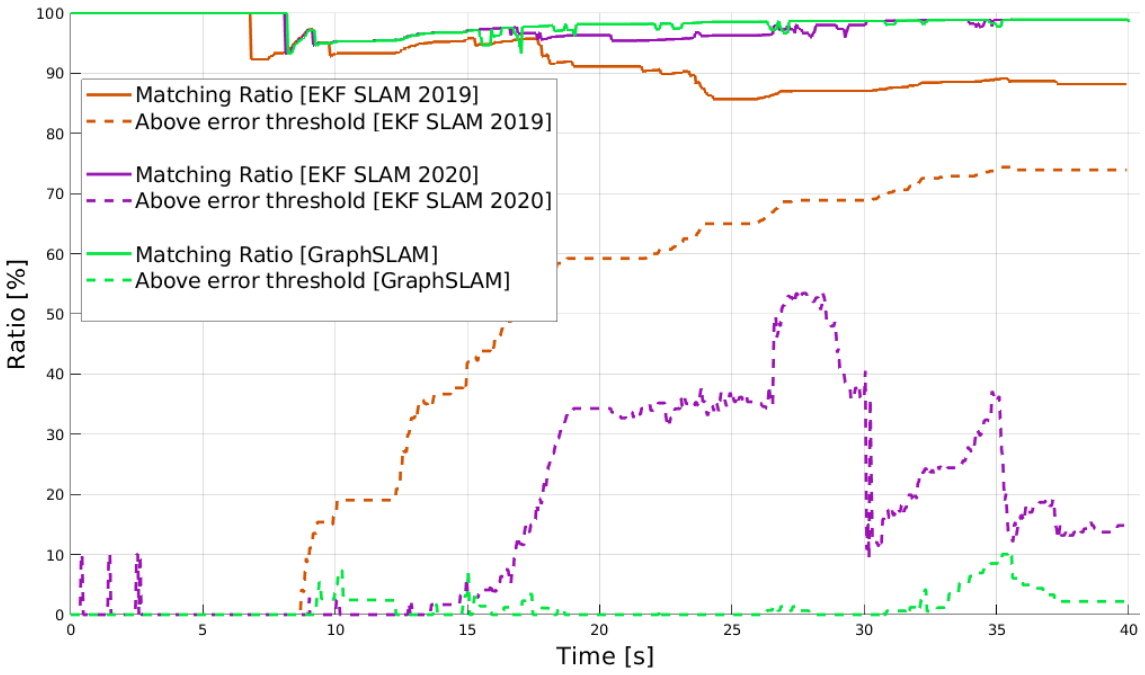


Figure 8.4: Autocross: Matching ratio and ratio of matched cones with a positioning error above a threshold of 30 cm.

of the car include using the racing line to further reduce lap times.

As shown with the dashed line in Figure 8.4, **EKF SLAM 2019** has most of the map, or about 74 %, above the threshold. In reality, it is actually worse than that. The positioning error can of course only be computed for cones where a corresponding pair could be found on both maps. As discussed, more than 10 % of the map could not be matched, which is likely because of a positioning error that is significantly higher than the threshold of 30 cm.

The graph of **EKF SLAM 2020** also rises to values of up to 53 %, however, at around 30 s and 35 s into the recording, big drops of the graph can be seen. This can be explained by **loop closures**. When leaving the opposite straight at around three quarters of the track, there are multiple spots where the car re-observes cones that were mapped very close to the beginning. Because the drift was the lowest at the start of the track, the certainty of the cone positions is very high and the positioning error of the cones is very low. This allows the algorithm to use the new measurements of the old cones to reduce the drift significantly and therefore to increase the overall accuracy.

This process can not be observed in the graph of EKF SLAM 2019 because this algorithm never corrects the cone positions and therefore cannot reduce the error.

GraphSLAM performs a lot better, with a maximum of 10.5 % of cones above the positioning error threshold and roughly 2 % at the end of the run.

8.3.4 Mean Squared Error

Figure 8.5 shows the mean squared error of the positions of all mapped cones where a corresponding cone was found on the reference map, per time step. This value is interesting because it gives a very absolute and precise measure of the accuracy over

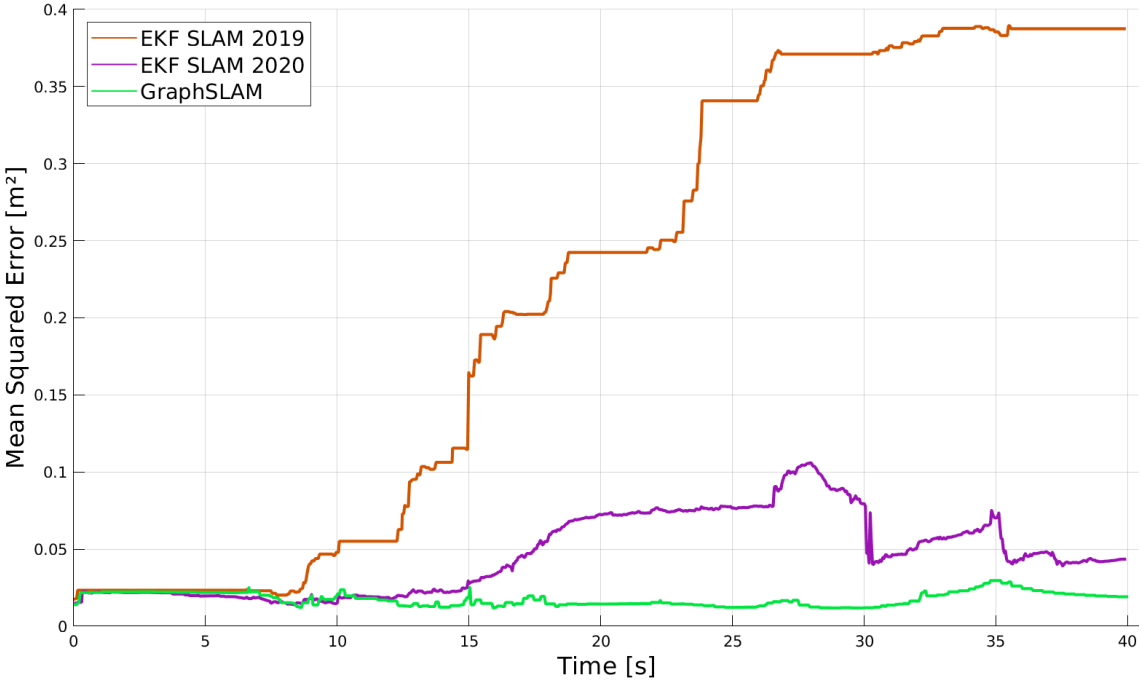


Figure 8.5: Autocross: Mean squared error of all mapped cone positions.

the course of the lap. The **squared error** is chosen to penalize far away cones more than closer estimates.

Many observations similar to the ones described in Section 8.3.3 can be made. GraphSLAM is the most accurate algorithm followed by EKF SLAM 2020 with a big improvement over EKF SLAM 2019. In Figure 8.5, the graph of the latter increases almost continuously to a value of $0.388\,\mathrm{m}^2$. Again, this is due to the lack of corrections made. The increasing error reflects the drift over the course of the lap.

EKF SLAM 2020 and GraphSLAM finish the lap with much better values of $0.044\,\mathrm{m}^2$ and $0.019\,\mathrm{m}^2$, respectively. GraphSLAM's superior accuracy is also shown by the fact that it is at constantly low values, whereas EKF SLAM 2020 increases to significantly higher values over the course of the lap that are only corrected after loop closure.

GraphSLAM: Sliding Window Accuracy

In the case of GraphSLAM, the accuracy and the efficiency of the algorithm can be weighted against each other using a sliding window as introduced in Section 7.3.4. To determine the impact a sliding window has on the accuracy of the algorithm, different window sizes are compared to a run where the sliding window is turned off. The accuracy can then be compared using the mean squared error measure presented above. The window sizes are chosen such that they include roughly 5 s, 10 s, 20 s, and 30 s worth of constraints. The results are depicted in Figure 8.6.

The results confirm the expected: The larger the sliding window, the smaller the mean squared error. However, it can also be seen that doubling the size of the sliding window does not cut the mean squared error in half. When the window is very small, the resulting error is very high. Doubling the window size from 5 s to 10 s improves the results significantly, whereas increasing the window size from 20 s

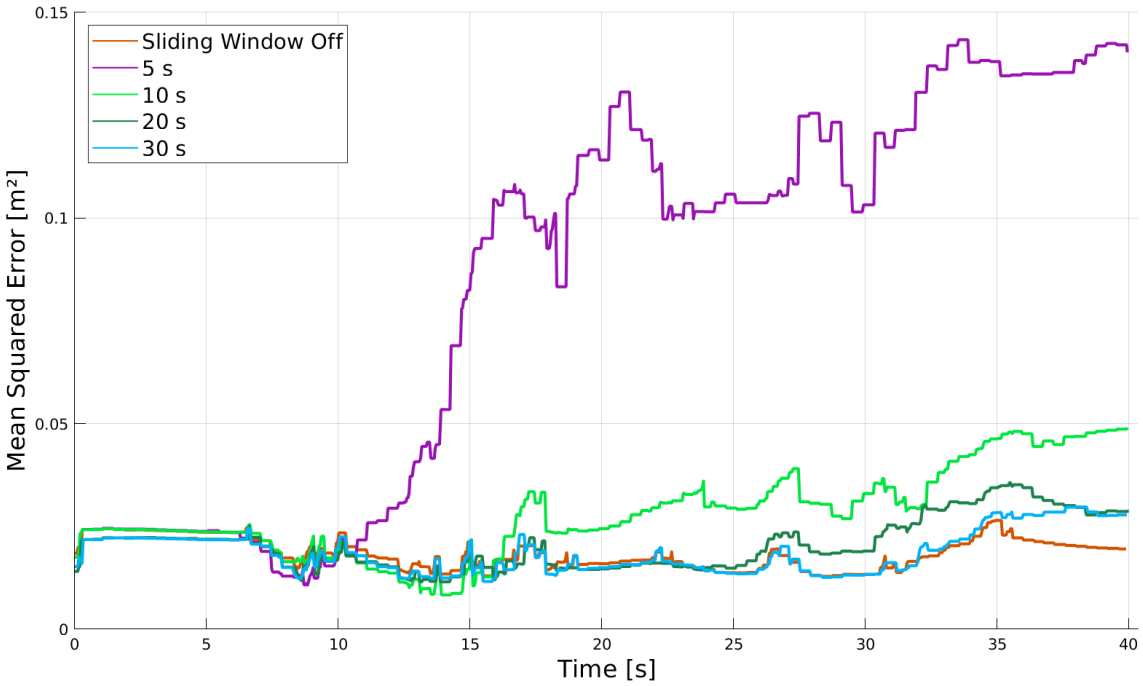


Figure 8.6: GraphSLAM: Impact of sliding windows of various sizes in regards to the algorithm's accuracy.

to 30 s barely makes a difference. The best results are, of course, achieved without a sliding window.

8.4 Efficiency

With any system that must work in real-time, efficiency is crucial. Additionally, as described in Section 4.3, there are a number of other algorithms that will have to run at the same time, meaning that computational resources, especially the CPU, must be shared between these algorithms.

The following section first discusses the run times of the algorithms which is important for real-time capability. After that, the CPU usage of all algorithms is compared. Because opposed to the accuracy measurements the efficiency measurements do not depend on a ground truth, the evaluation is performed for both Autocross and Trackdrive.

8.4.1 Run Times

Autocross

Incoming odometry data and observations trigger a callback in both algorithms. The time taken to compute the callback is measured and plotted in Figure 8.7. To smooth the result, the runtimes are filtered using a moving average filter. The maximum time an odometry processing may take before information is lost is marked in the plot.

Odometry measurements are processed so quickly that they are barely visible

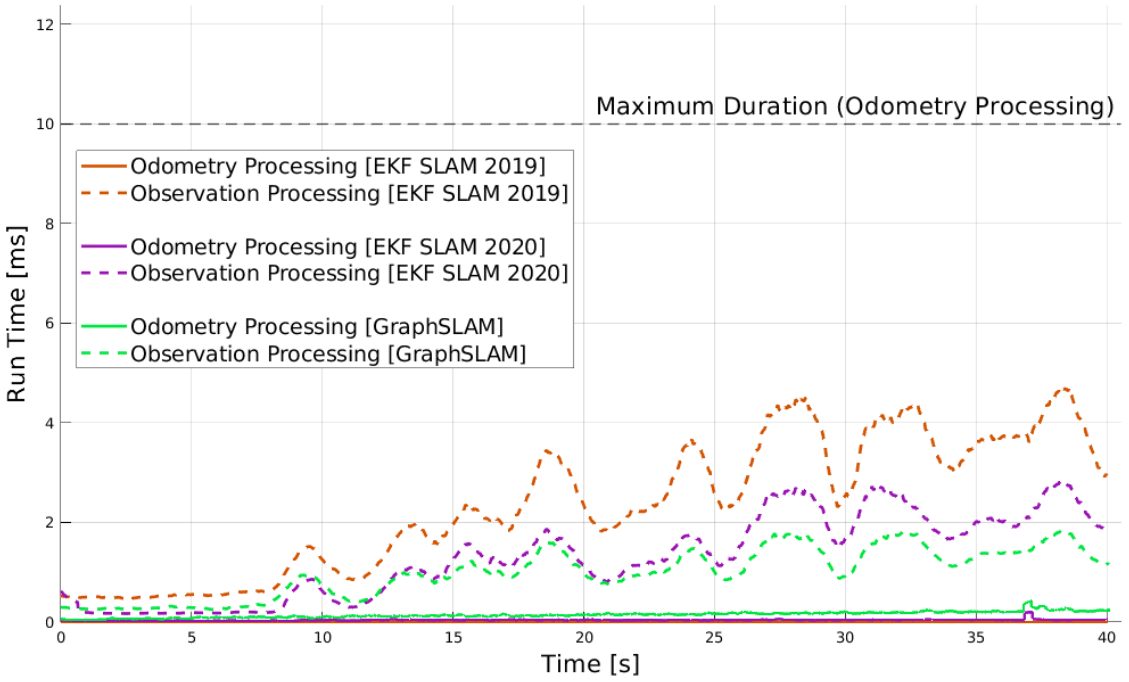


Figure 8.7: Autocross: Time taken to process new odometry data and observations (filtered).

in the plot. GraphSLAM takes the longest for this, but even the highest peak at around 37 s is below half a millisecond. The same peak can be seen in EKF SLAM 2020 and has a simple explanation. After each odometry processing, the algorithms check whether a lap has been completed. In the case of Autocross, if it has, the algorithms will write the map to the disk automatically so that it can be loaded during Trackdrive. EKF SLAM 2019 did not have that feature implemented (the map had to be extracted manually from the recorded data), hence there is no peak in the plot.

Another interesting observation in this data is that the time taken by **GraphSLAM** to process odometry data increases over the course of the lap. This can be explained by the adjustment of the new poses as presented in Section 7.3.3. As discussed further below, the duration of an optimization will increase during the lap, meaning that more short-term pose estimates are being made based on the velocity estimate. Since they need to be corrected after the optimization and because the odometry processing has to wait for that to happen, the time for the odometry processing will increase over the course of the lap. There may be room for improvement here, but as can be seen in Figure 8.7, the odometry data is easily processed quickly enough.

The dashed lines represent the times taken by the algorithms to **process observations**. First of all, it should be noted that processing an observation is allowed to take up to 40 ms. If it takes longer, new observations cannot be processed in time and information is lost. All algorithms perform their computations well within that time limit.

The first thing that stands out when looking at the three graphs are the **local peaks**. They can be explained by the number of landmarks observed in a particular time step. All algorithms ran on the same data set, so the number of observed

landmarks is the same for each algorithm at each time step, which is why the peaks are occurring at the same time for all three algorithms. The number itself mostly depends on the orientation of the vehicle with respect to the track. If the car faces the middle point of the track, it may be able to see parts of the opposing end of the track and is hence able to observe a large number of landmarks. Whereas if the car's heading is oriented towards the outside of the track, it may only be able to observe a few cones of the track right in front of it. Because every observation must be associated to the map and the SLAM algorithms themselves iterate over all observed landmarks in a time step, the run time of the callback increases.

The next observation is an unexpected one: **EKF SLAM 2019** actually takes longer than EKF SLAM 2020. This should not be the case because EKF SLAM 2019 only corrected the vehicle pose, whereas EKF SLAM 2020 corrects both the vehicle pose and the map. There are two likely explanations for this: As could be seen in Section 8.3.4, EKF SLAM 2019 is by far the least accurate of the algorithms. A less accurate map makes it more difficult for the data association algorithm to find the correct matchings. Cones mapped far away from their actual position may have multiple possible matchings, so the algorithm must check more possibilities in order to find the correct solutions, which may take a very long time.

However, the mean squared error is almost the same for all the algorithms until the car starts driving as could be seen Figure 8.5. Still, the run time of the observation processing is higher even at the beginning. This small offset cannot be explained with the data association. EKF SLAM 2020 is of course built on EKF SLAM 2019, but most of the code has been rewritten in the development process. During this phase, the code quality increased and a lot of overhead was removed by more efficient code which may explain the increased run times of EKF SLAM 2019.

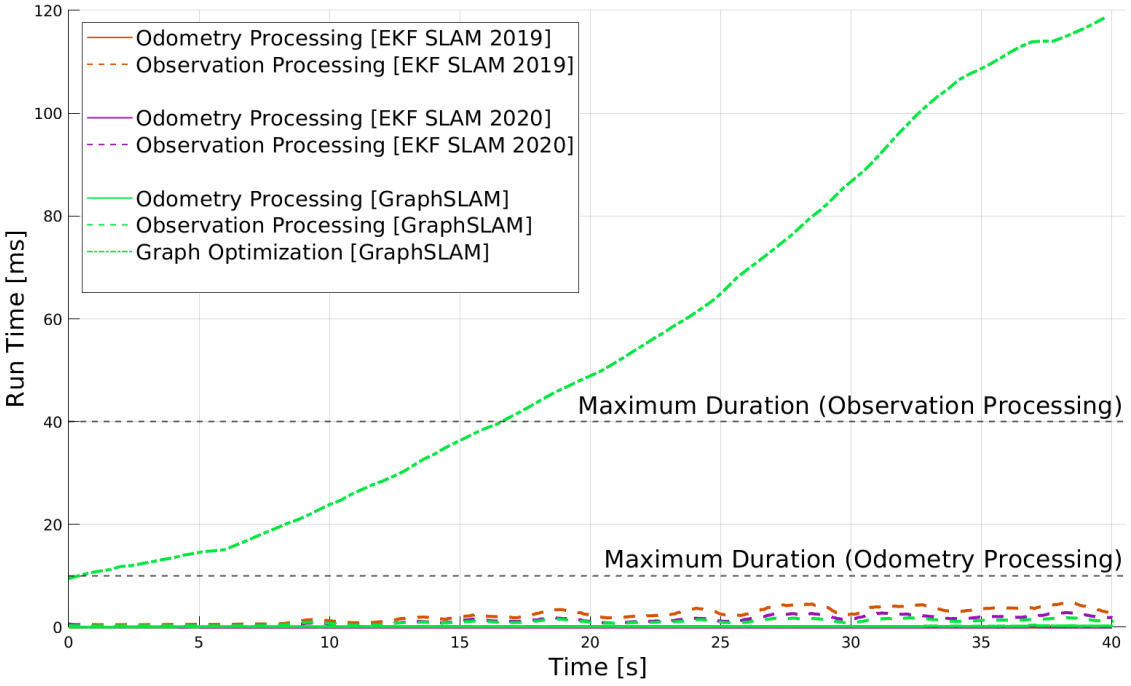


Figure 8.8: Autocross: Time taken to process new odometry data and observations and to optimize the graph (filtered).

When comparing **EKF SLAM 2020 and GraphSLAM**, it can be seen that they take a similar amount of time in the beginning with an increasing offset because EKF SLAM 2020 takes longer towards the end of the run. An EKF's computational complexity is squared in the number of states. Because the map is part of the state and built over the course of the lap, the number of states increases continuously. Therefore, the computation time of EKF SLAM 2020 increases until the map is complete.

However, the comparison between the observation processing of EKF SLAM 2020 and GraphSLAM is not a fair one. EKF SLAM 2020 will correct the map and the vehicle pose in this step, whereas GraphSLAM merely prepares the optimization, which is not shown in Figure 8.7 but is included in Figure 8.8.

As mentioned earlier, the GraphSLAM's computational complexity is linear in the number of edges and the number of edges is linear in time. This means that the time taken for the **optimization** increases of the course of a lap. As can be seen in the plot, this time dwarfs the run times of the odometry and observation processing. It should be noted that unlike processing the input data, optimizing the graph does not have a fixed time limit. Optimizing with a higher frequency will surely increase the short-term accuracy but if an optimization takes longer, no information is lost because collecting data and optimizing the graph is done on separate threads.

The accuracy presented in Section 8.3 was achieved with one optimization for every ten sets of observations, i.e. every 400 ms. This means that the optimization could even be performed three times as often without reaching that limit in this data set.

GraphSLAM: Sliding Window Run Times

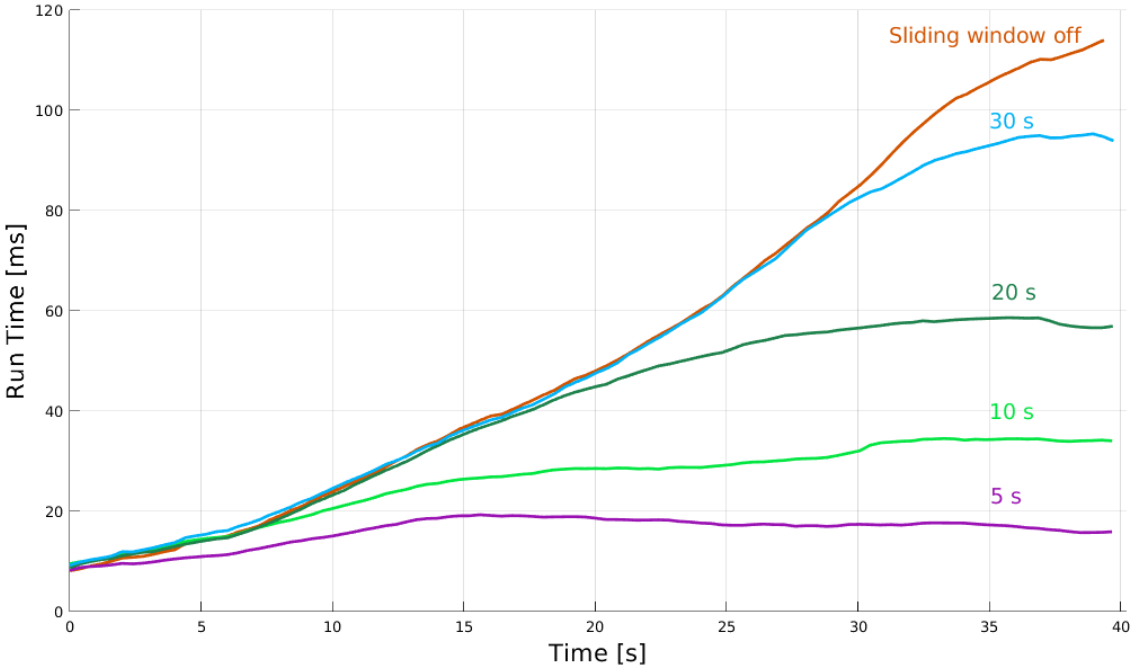


Figure 8.9: GraphSLAM: Impact of sliding windows of various sizes in regards of the algorithm's efficiency in Autocross.

As discussed in Section 8.3.4, using a sliding window reduces the accuracy. The reason to use a sliding window anyway is the improved efficiency. To understand

the gain in efficiency from using a sliding window, the same window sizes earlier compared with respect to their accuracy are now compared in terms of efficiency. The results are depicted in Figure 8.9.

As expected, the runtimes of the optimization scales linearly in the number of edges, i.e. the size of the window. Choosing the right windows size or whether a sliding window should be used at all should therefore be a compromise between the required accuracy and the available resources.

Trackdrive

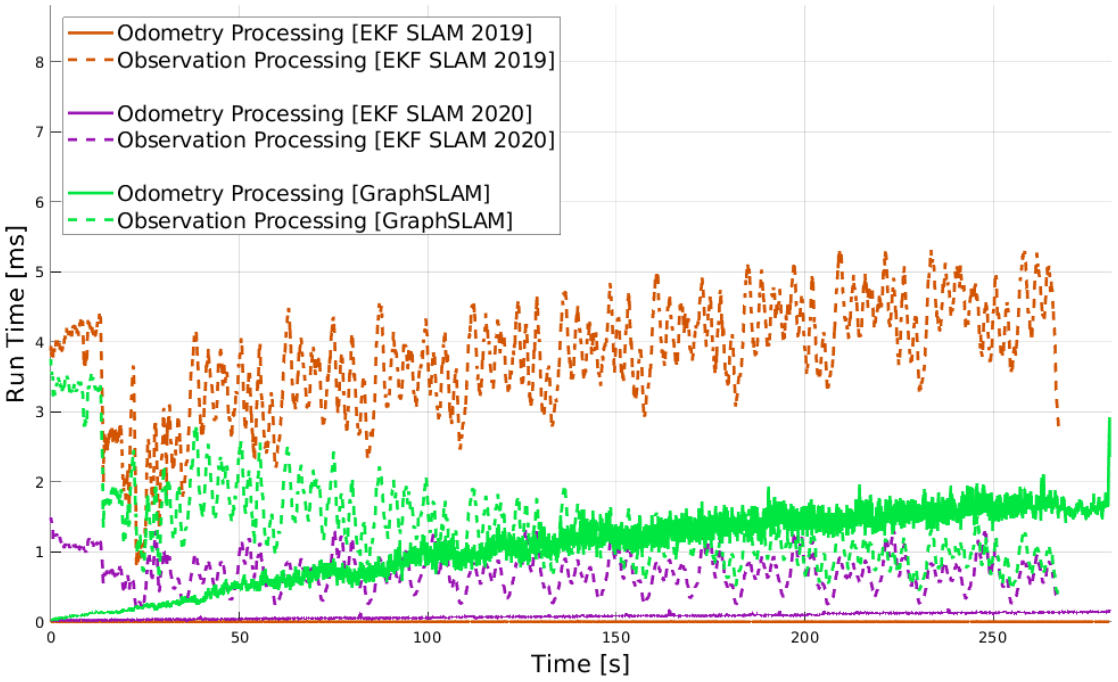


Figure 8.10: Trackdrive: Time taken to process new odometry data and observations (filtered).

Figure 8.10 shows the same analysis discussed above but in a Trackdrive scenario. This means two things: The time to complete the discipline is a lot longer because a Trackdrive consists of ten laps, and the algorithms switch to a localization-only mode since the map is known from the start. This means that both EKF SLAM algorithms now essentially work the same way. The filter state consists only of the vehicle state. Measured observations will only be used to correct the vehicle pose.

The difference that can be seen between the observation processing of EKF SLAM 2019 and EKF SLAM 2020 can only be explained by increased code quality and decreased overhead. It can also be seen that in Trackdrive, the run time of the **observation processing** of EKF SLAM is roughly constant and generally a lot lower than it was in Autocross.

Adjusting the poses after an optimization is done takes longer, the longer the optimization was. This can be observed in the graph of the **odometry processing** of GraphSLAM. The curve increases and shows that eventually the odometry processing takes longer than the observation processing. However, it is still within the time limit of 10 ms.

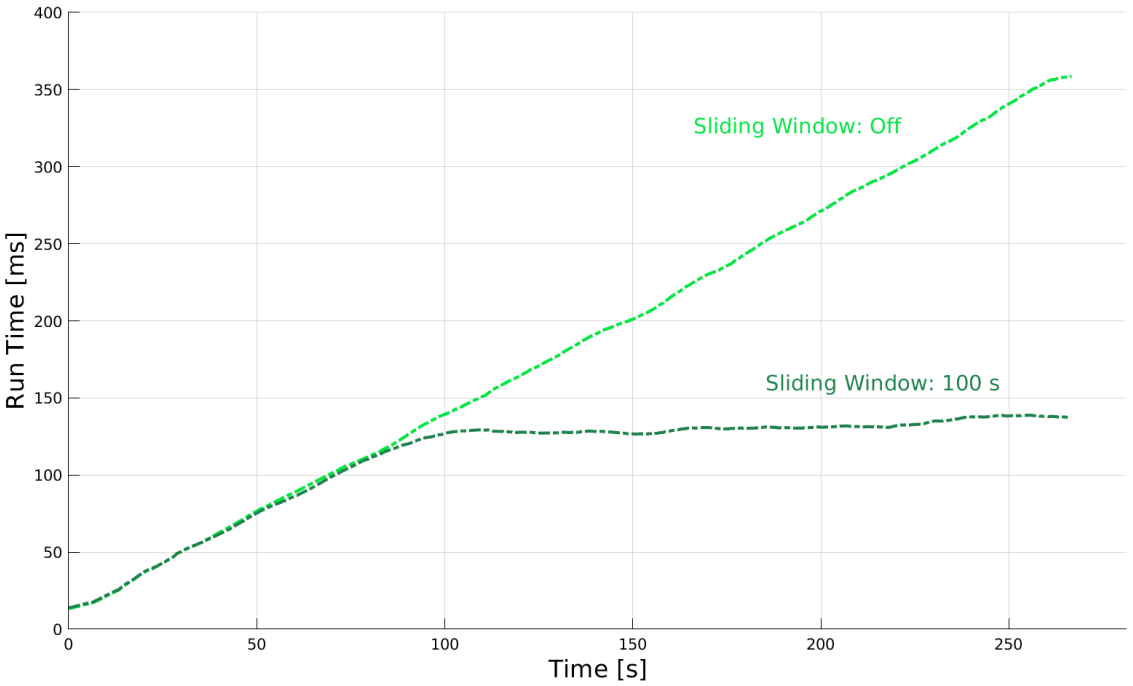


Figure 8.11: Trackdrive: Time taken to optimize the graph (filtered).

Just as in Autocross, the **optimization** that can be seen in Figure 8.11 takes significantly longer than all other processes. For better visibility, odometry and observation processing are not included in this plot. Because a Trackdrive run takes a lot longer compared to an Autocross run, the long run time of the optimization is a bigger issue as a single optimization takes over 350 ms. However, the algorithm was set to start an optimization every 400 ms, so even in this Trackdrive dataset, no optimization was missed.

Still, in Trackdrive it is sensible to use a **sliding window**. As was shown in Section 8.3.4, the accuracy of a 30 s window is very close to the accuracy when the sliding window is turned off. In the Trackdrive depicted in Figure 8.11, the sliding window was set to around 100 s. This results in a roughly constant computation time from 100 s onwards and the accuracy should be sufficient. However, without a ground truth path, the accuracy in Trackdrive cannot be measured.

8.4.2 CPU Usage

Finally, the usage of computational resources of the algorithms is analyzed. The CPU usage is measured with a frequency of 10 Hz. At every point of measurement, the percentage of time the CPU was not idling since the last measurement is calculated. For example, if none of the CPU cores was idling during the last time step, the CPU usage was at 100 %. Just as the run time measurements, the CPU measurements are smoothed with a moving average filter during post-processing.

Before discussing the actual results, Table 8.1 shows a **comparison of the CPU** used for the analysis presented in this thesis versus the CPU that is built into the ACU. The *CPU Mark* is an average of eight different tests performed on the CPU[2].

[1]Source: `www.cpubenchmark.net`
[2]Explanation of all tests: `www.cpubenchmark.net/cpu_test_info.html`

Table 8.1: Comparison: CPU used for analysis (left) vs. CPU used by ACU (right)[1].

	Intel Core i5-4690K	**Intel Core i7-9700K**
Clockspeed	3.5 GHz	3.6 GHz
Turbo Speed	Up to 3.9 GHz	Up to 4.9 GHz
Physical Cores	4	8
Single Thread Rating	2158	2906
CPU Mark	5580	14740

Higher values correspond to better performance. The CPU mark of the ACU's processor is roughly 2.6 times higher than that of the CPU that the analysis has been conducted on. This means that in absolute values, the CPU loads plotted in this section will be higher than they would be, had the algorithms run on the car. However, the relative differences can still be compared.

Autocross

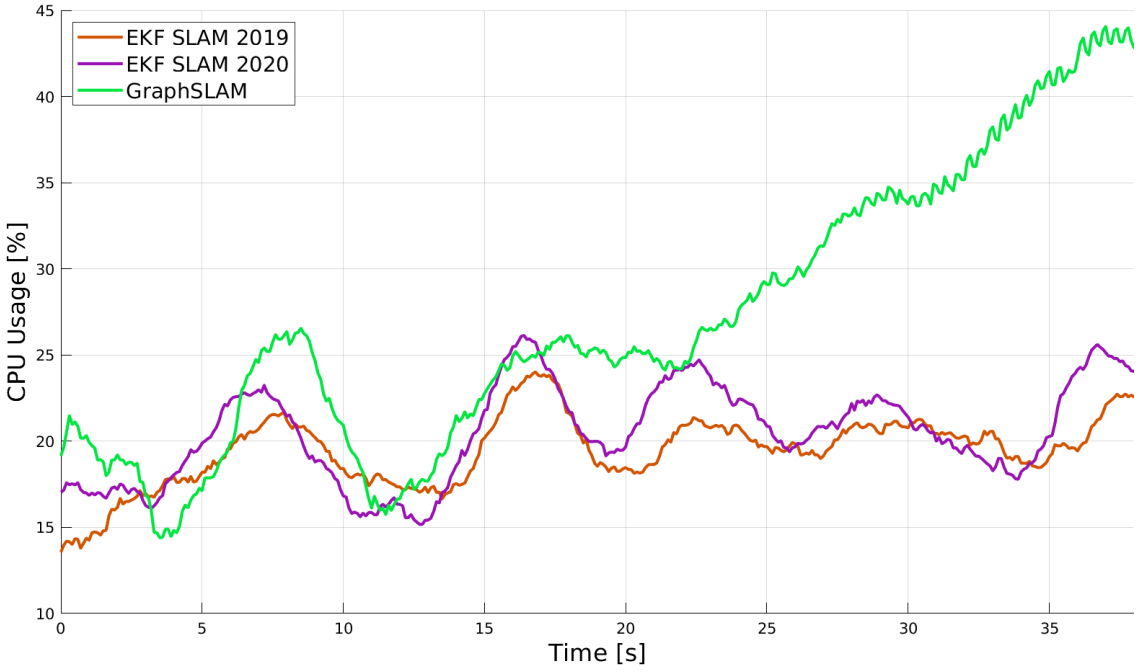


Figure 8.12: Autocross: CPU usage (filtered).

The CPU load during one lap of Autocross can be seen in Figure 8.12. When comparing the CPU loads of **EKF SLAM 2019 and EKF SLAM 2020**, a similar observation as in Section 8.4.1 can be made: The EKF SLAM 2019 should be more efficient due to it not correcting the map, but the CPU usage of both EKF SLAM algorithms is nearly the same. The reasons are likely the same too. The data association algorithm is the same but due to the less accurate map, matching the correct cones is harder. Additionally, increased code quality and decreased overhead are beneficial for an efficient algorithm. The increasing complexity of EKF SLAM 2020 due to the increasing size of the state vector can not be seen in this plot.

The CPU usage of **GraphSLAM** turns out to be a larger issue than the run times. With an increasing number of constraints, the optimization becomes the dominant

process in terms of CPU load after around 20 s. At the end of the run, GraphSLAM takes up close to 45 % of the CPU, almost twice the value of both EKF SLAM algorithms.

GraphSLAM: Sliding Window CPU Usage

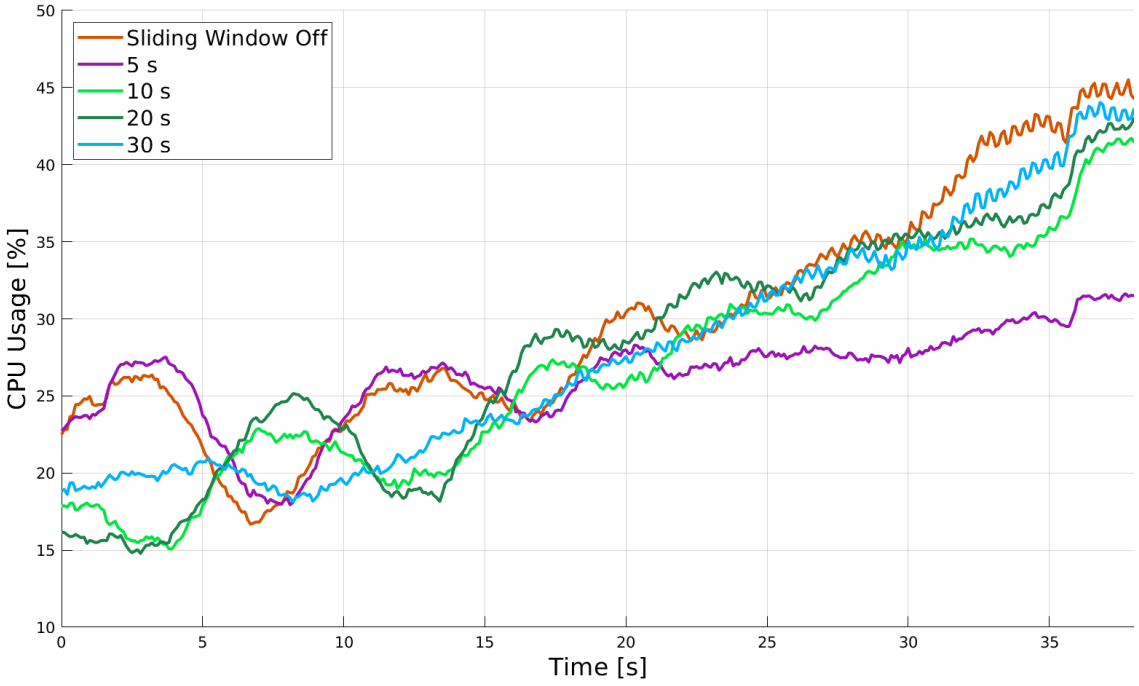


Figure 8.13: GraphSLAM: Impact of sliding windows of various sizes in regards of the algorithm's CPU usage in Autocross.

Adding a sliding window made a significant difference in the run time of the optimization run times. Figure 8.13 shows the difference in CPU usage using the same window sizes. During the first half of the run, no clear pattern can be seen, but after roughly 20 s, the CPU usage increases with the size of the sliding window. The sliding window affects only the optimization. At the start of a run, other processes, such as the data association, are more dominant which may be why no clear structure can be seen at first.

The only window size where the CPU usage is decreased significantly is the smallest one (5 s). All other windows reach a maximum CPU load of 40 % to 45 %. In conclusion: In terms of CPU usage, a sliding window does not make a big difference in Autocross.

Trackdrive

In Trackdrive, all algorithms work in a localization-only mode. That means that both EKF SLAM algorithms only have the three vehicle states in their respective state vector and that landmark nodes are fixed in the graph. This results in a CPU usage that is generally lower than in Autocross. Due to the amount of time a Trackdrive takes, the sliding window does have a significant effect on the CPU usage. In this case, the sliding window was set to 100 s, which is about two and a half laps worth of constraints.

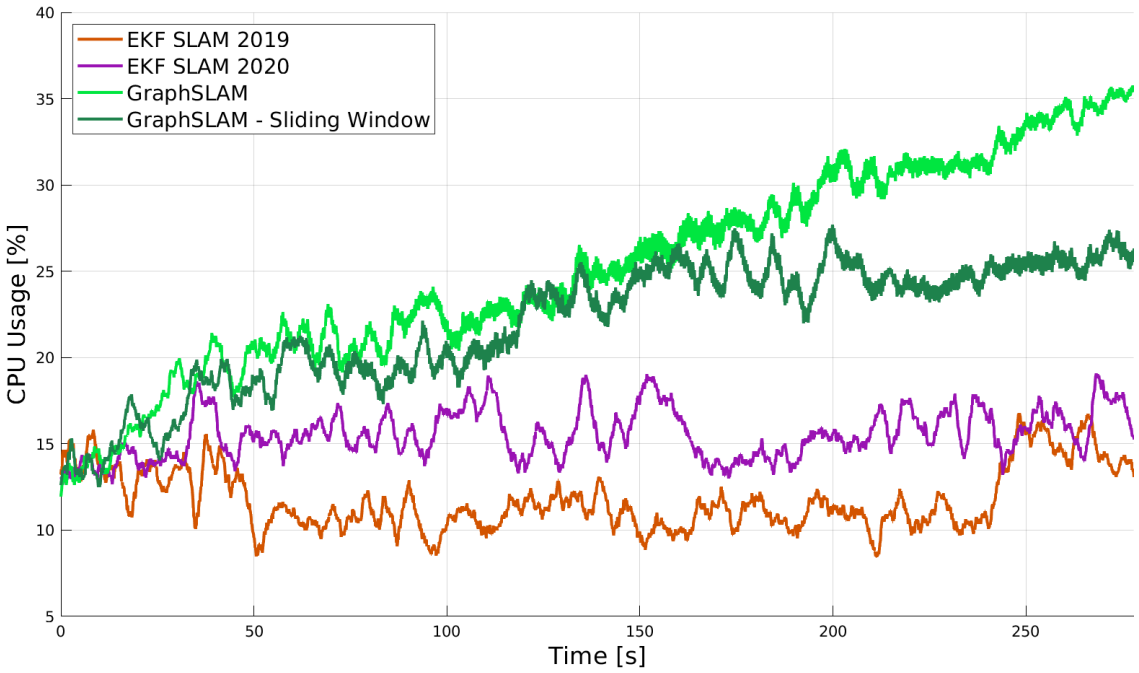


Figure 8.14: Trackdrive: CPU usage (filtered).

Just as in Autocross, the maximum GraphSLAM CPU load is roughly twice that of EKF SLAM 2020 which is between 15 % and 20 % most of the time.

The fact that the CPU usage of the EKF SLAM 2019 is actually lower than that of EKF SLAM 2020 is unexpected. Since both algorithms work essentially the same way, the CPU usage was expected to be similar as well. One reason for this may be the parallelized structure of the new algorithm. It allows the algorithm to work on multiple callbacks at the same time on different CPU cores. With the way the CPU usage is measured, the maximum CPU usage of EKF SLAM 2019 is 25 % on the machine used for the analysis because it only uses one of the four available CPU cores. EKF SLAM 2020 may use up to 100 %. Additional features implemented in the new algorithm, like online parameter changes or better visualization, utilize this increased potential at the cost of higher CPU usage.

In this chapter, the performance of both algorithms was presented in great detail. In Chapter 9, a conclusion is drawn from the results and possible future improvements of the algorithms and the method of the performance analysis are discussed.

9 Conclusion and Future Work

9.1 Summary

In this thesis, two SLAM algorithms - EKF SLAM and GraphSLAM - were presented and analyzed with respect to their performance.

The detailed performance analysis is summarized in Table 9.1. The rows marked as *Final* are the values measured at the end of the run, whereas *Min* and *Max* are the extreme values measured at any time during the run.

All **accuracy** measures lead to the same conclusion: GraphSLAM is the superior algorithm in terms of precision followed by EKF SLAM 2020 which is still a huge improvement over EKF SLAM 2019. However, while EKF SLAM 2020 delivers reasonable results at the end of the run, the measured accuracy drops quite significantly over the course of the lap and is only corrected later. This means that EKF SLAM 2020 will deliver relatively accurate maps but may struggle with localization during the run, which might require the velocity to be limited in order to not hit any cones or offend the track limits. That is different with the GraphSLAM algorithm which continuously delivers highly accurate results.

The accuracy comes at a price of course and that is visible when looking at the **efficiency** measurements, specifically the CPU load. Regardless of the discipline, GraphSLAM is more demanding in terms of the computational resources of the car. In Trackdrive, the CPU usage can be reduced by activating the sliding window. In Autocross, however, because of the shorter time span of the discipline, the benefit of reducing the amount of constraints barely makes a difference. In Trackdrive, the car does not need to map the environment as a map is available from the start, which is why the overall CPU load is lower compared to Autocross. This means that even though Autocross is shorter, it is the critical discipline with respect to both accuracy and efficiency.

The **time taken** to process incoming odometry data and observations differs between the algorithms but is not an issue due to the parallelized architecture implemented in EKF SLAM 2020 and GraphSLAM. In the case of GraphSLAM, the optimization will take an increasing amount of time the more constraints are added to the graph which may turn out to be an issue during Trackdrive. However, this issue can easily be addressed with a sliding window. Even without a sliding window, no optimization was missed in the data sets presented because GraphSLAM will deliver accurate estimates even when optimizing at a rather low rate.

Table 9.1: Summary of the results of the performance analysis discussed in Chapter 8.

			EKF SLAM 2019		**EKF SLAM 2020**		**GraphSLAM**	
			Autocross	Trackdrive	Autocross	Trackdrive	Autocross	Trackdrive
Accuracy	Number of Cones [-] *(Should be 185)*	Final	187	-	184	-	184	-
	Matching Ratio [%]	Min	85.71	-	93.33	-	93.26	-
		Final	88.24	-	98.91	-	98.91	-
	Cones Above Threshold [%]	Max	74.39	-	53.74	-	10.50	-
		Final	73.94	-	14.84	-	2.20	-
	Mean Squared Error [m^2]	Max	0.3894	-	0.1060	-	0.0296	-
		Final	0.3875	-	0.0436	-	0.0189	-
Efficiency	Maximum Run Times [ms]	Odometry Processing *(Maximum* 10 ms*)*	0.01	0.01	0.20	0.18	0.43	2.11
		Observation Processing *(Maximum* 40 ms*)*	4.67	5.30	2.83	1.50	1.84	3.56
		Optimization	-	-	-	-	118.60	358.70
	CPU Usage [%]	Max	24.03	16.79	26.14	19.08	43.93	36.09

9.2 Conclusion

In the beginning of this thesis (Section 1.2), the following research question was posed:

Which approach to the SLAM problem results in the best performance in the context of an autonomous Formula Student race car?

Using the results summarized above, this question can now be answered:

The first conclusion to be drawn is that both of these algorithms are capable of solving the SLAM problem and are therefore valid approaches. When measuring the performance of a SLAM algorithm purely using the accuracy of its estimates, the answer is straight forward: **GraphSLAM**. The algorithm has proven its superior accuracy when compared to EKF SLAM 2020 with all measures presented in Section 8.3.

However, the algorithm must run on the ACU which has limited **computational resources** due to external restrictions such as weight, power consumption and budget. In addition to the SLAM algorithm, there are a number of other algorithms running at the same time. Comparing only the accuracy of the algorithms is not sufficient for a real-world application.

While first tests on the car point towards sufficiently available processing power, this limitation of the GraphSLAM algorithm should be kept in mind. Should computational resources be an issue, using EKF SLAM 2020 instead is a valid option. The lower accuracy will the be made up for by the decreased CPU usage, which may eventually lead to a better overall performance.

9.3 Application of GraphSLAM in 2020

The GraphSLAM algorithm presented in this thesis has been successfully tested in real-world applications. The first Trackdrive using this algorithm was completed in October 2020. Thanks to extensive testing on the available data sets and the performance evaluations presented in this thesis, the algorithm was functional right away.

Because of the COVID-19 pandemic, the 2020 Formula Student events had to be cancelled. However, the **Formula Student Online (FSO)**[1] competition was introduced at short notice. FSO is a simulation event that replaced the physical events in Hungary and the Netherlands. KA-RaceIng participated along with the Formula Student teams from Delft (*MITeamDelft*), Hamburg (*e-gnition Hamburg*), Munich (*TUfast Racing*), and Prague (*eForce FEE Prague Formula*). The dynamic events were reduced to only include Autocross and Trackdrive which are the most demanding disciplines for the SLAM algorithm. The competition was split into three days. Each day would feature a unique track that was unknown beforehand where an Autocross needed to be completed followed by a Trackdrive using the generated map. A significant difference to a physical Formula Student event was the fact that access to the software pipeline was not permitted on the day of the event, which meant

[1] `formulastudentonline.com`

that all software nodes had to be started and stopped automatically and the map had to be stored and loaded without user interaction.

The software stack of the autonomous system was running on a server where it was connected to a separate computer running the simulation. The simulation would deliver sensor data, whereas the autonomous system sent torque and steering requests to the simulation. Given that the software stack on the actual race car fundamentally has the same interfaces, the autonomous pipeline was untouched for the most part. Aside from parameter changes and the inclusion of different sensors, the perception pipeline was using only the lidar due to the limited ability of the simulation to deliver robust camera data.

The results of all disciplines relevant to the SLAM module are listed in Table 9.2. Note that in contrast to most Formula Student competitions, a total score of 1250 points was possible.

Table 9.2: FSO results [41].

	Autocross		Trackdrive		Eng. Design		Overall	
	Score	Place	Score	Place	Score	Place	Score	Place
Karlsruhe	150	1	125	1	500	1	1127.4	1
Hamburg	20	4	0	4	443	2	831.5	2
Munich	0	5	0	4	422	3	667.2	3
Prague	30	3	10	3	259	4	571.1	4
Delft	56.47	2	27.89	2	0	5	84.4	5

The GraphSLAM algorithm presented in this thesis proved to be robust and accurate and helped the team win the overall event.

9.4 Future Work

9.4.1 Hardware Setup 2021

For the 2021 season, the algorithms need to be adjusted to a new hardware setup. Specifically, the perception sensors will change. The **new lidar**, a Hesai Pandar40p, has a significantly increased number of points per scan and may therefore increase the perception range and the number of cones detected within that range. While that is of course a great improvement, the increased number of detected cones must be processed by the SLAM algorithms, which may be challenging. The process of data association will take longer and both algorithms must iterate over all landmarks seen in a specific time step. It must therefore be ensured that the time taken by the algorithms to process the increased amount of data is still within the time limit dictated by the lidar's frequency. The latter will decrease to either 10 Hz or 20 Hz, depending on the lidar settings, allowing the SLAM algorithms more time to process the observations.

In 2021, the car may have an additional sensor, a Kistler SFII **ground speed sensor**. This sensor is capable of delivering high precision velocity measurements in

both longitudinal and lateral direction. Such a velocity sensor is of course a great benefit when evaluating the motion model. The velocity estimate is more precise and available in lateral direction and at a very high rate of up to 250 Hz. When having a good motion estimate, the correction made in the update step (EKF SLAM) or in the optimization using landmark edges (GraphSLAM) is smaller and the algorithms in turn are more robust.

Opposed to the 2019 season, the KIT21d will have additional hardware to improve the **aerodynamics** of the car, namely a front and rear wing. In 2019, the car occasionally reached the physical limits of the car which made this step necessary to further improve the overall performance. For the SLAM algorithms, this poses another challenge because higher velocities imply fewer landmark measurements and increased slip (although the latter would be less of an issue with the aforementioned ground speed sensor).

9.4.2 Improving the SLAM Algorithms

The SLAM algorithms themselves are of course subject to improvement as well. Depending on the physical tests with the new hardware, a decision between the algorithms must be made. As discussed in Section 9.2, this decision mainly depends on the available computational resources and the efficiency of the overall software stack.

The tool to measure the performance developed for and presented in this thesis can then play an important role in improving the chosen SLAM algorithm. The first step would be to test combinations of different **parameter settings** on test data acquired with the new hardware setup. Both algorithms are highly dependent on their respective parameters, so testing different setups and measuring their respective performance is crucial.

When acquiring the ground truth map, multiple **track variations** were measured allowing the course to either have very narrow or very fast corners. As soon as data is recorded on these track variations with the new hardware, the SLAM algorithms should be evaluated with respect to all these variants. This allows for an even better analysis of the algorithms.

The improved accuracy of the 2020 algorithms may allow for lower requirements to the **data association** algorithm that takes a significant portion of the time when processing new observation. Using a simpler algorithm, for example a nearest neighbor approach, may actually improve the overall performance, especially with the increased perception range in mind.

Another possible improvement concerns the **motion model**. The model presented in Section 5.2 is a very simple model which was chosen to ensure low odometry processing times. However, as could be seen in Section 8.4.1, the runtimes of processing odometry data are more than tenfold below the maximum allowed duration of 10 ms, so using a more sophisticated motion model is surely an option. In the case of GraphSLAM it is also conceivable to use different motion models for the short-term estimate delivered to the controller and the odometry constraint used to optimize the estimate.

The **measurement model** could be improved with respect to the systematic error discussed in Section 5.4.3. Limiting the uncertainty as it was presented in this

thesis did fix the problem, however, adjusting the measurement model to reduce the systematic error may lead to better results. This could for example be achieved by adding the mean radius of a cone to each range measurement or to use the three-dimensional measurement to account for the offset depending on the z-value of the measurement.

The parallelization of the GraphSLAM algorithm could be optimized with respect to the adjustment of new poses (see Section 7.3.3), which turned out to increase the computation time of processing new odometry data (see Section 8.4.1).

Considering the vehicle's **pitch and roll angles** when projecting the cone measurement into two-dimensional space should drastically improve the accuracy of the measurements. Strictly speaking, this is not part of the SLAM module but should be considered in the perception pipeline. However, improving the quality of measurements will of course improve the quality of the path and map estimates and simplify the process of data association.

Finally, there are **other approaches** to the SLAM problem that have not been discussed in this thesis but could be implemented and evaluated using the benchmarking tool presented in Chapter 8. However, the benefit of implementing another approach should be considered carefully.

List of Figures

List of Tables

List of Algorithms

Bibliography

[1] Randall Smith and Peter Cheeseman. "On the Representation and Estimation of Spatial Uncertainty". In: *The International Journal of Robotics Research* 5 (Feb. 1987). DOI: 10.1177/027836498600500404.

[2] H. F. Durrant-Whyte. "Uncertain geometry in robotics". In: *IEEE Journal on Robotics and Automation* 4.1 (1988), pp. 23–31. DOI: 10.1109/56.768.

[3] Randall Smith, Matthew Self, and Peter Cheeseman. "Estimating Uncertain Spatial Relationships in Robotics". In: *Autonomous Robot Vehicles*. Ed. by Ingemar J. Cox and Gordon T. Wilfong. New York, NY: Springer New York, 1990, pp. 167–193. ISBN: 978-1-4613-8997-2. DOI: 10.1007/978-1-4613-8997-2_14. URL: https://doi.org/10.1007/978-1-4613-8997-2_14.

[4] J. J. Leonard and H. F. Durrant-Whyte. "Simultaneous map building and localization for an autonomous mobile robot". In: *Proceedings IROS '91: IEEE/RSJ International Workshop on Intelligent Robots and Systems '91*. 1991, 1442–1447 vol.3. DOI: 10.1109/IROS.1991.174711.

[5] Hugh Durrant-Whyte, David Rye, and Eduardo Nebot. "Localization of Autonomous Guided Vehicles". In: *Robotics Research*. Ed. by Georges Giralt and Gerhard Hirzinger. London: Springer London, 1996, pp. 613–625. ISBN: 978-1-4471-0765-1.

[6] Martin Ester et al. "A density-based algorithm for discovering clusters in large spatial databases with noise". In: AAAI Press, 1996, pp. 226–231.

[7] F. Lu and E. Milios. "Globally Consistent Range Scan Alignment for Environment Mapping". In: *Auton. Robots* 4.4 (Oct. 1997), pp. 333–349. ISSN: 0929-5593. DOI: 10.1023/A:1008854305733. URL: https://doi.org/10.1023/A:1008854305733.

[8] Peter Spirtes, Clark N. Glymour, and Richard Scheines, eds. *Causation, prediction, and search*. 2nd ed. Adaptive computation and machine learning. Cambridge, Mass: MIT Press, 2000. ISBN: 9780262284158; 0262284154.

[9] Douglas B. West. *Introduction to Graph Theory*. 2nd ed. Prentice Hall, Sept. 2000. ISBN: 0130144002.

[10] J. Neira and J. D. Tardos. "Data association in stochastic mapping using the joint compatibility test". In: *IEEE Transactions on Robotics and Automation* 17.6 (2001), pp. 890–897.

[11] Stefan Williams. "Efficient Solutions to Autonomous Mapping and Navigation Problems". PhD thesis. Jan. 2001. Chap. 2.4.1.

[12] E.F. Camacho, C. Bordons, and C.B. Alba. *Model Predictive Control*. Advanced Textbooks in Control and Signal Processing. Springer London, 2004. ISBN: 978-1-8523-3694-3.

[13] Sebastian Thrun et al. "Simultaneous Localization and Mapping with Sparse Extended Information Filters". In: *I. J. Robotic Res.* 23 (July 2004), pp. 693–716. DOI: 10.1177/0278364904045479.

[14] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic robotics*. Cambridge, Mass.: MIT Press, 2005. ISBN: 0262201623 9780262201629.

[15] P. Tondel and T. A. Johansen. "Control allocation for yaw stabilization in automotive vehicles using multiparametric nonlinear programming". In: *Proceedings of the 2005, American Control Conference, 2005.* 2005, 453–458 vol. 1. DOI: 10.1109/ACC.2005.1469977.

[16] T. Bailey and H. Durrant-Whyte. "Simultaneous Localisation and Mapping (SLAM) Part 2 : State of the Art". In: 2006.

[17] Hugh Durrant-Whyte and Tim Bailey. "Simultaneous Localisation and Mapping (SLAM): Part I The Essential Algorithms". In: *Robotics & Automation Magazine* 13 (Jan. 2006).

[18] Sebastian Thrun and Michael Montemerlo. "The Graph SLAM Algorithm with Applications to Large-Scale Mapping of Urban Structures". In: *I. J. Robotic Res.* 25 (May 2006), pp. 403–429. DOI: 10.1177/0278364906065387.

[19] Lars B. Cremean et al. "Alice: An Information-Rich Autonomous Vehicle for High-Speed Desert Navigation". In: *The 2005 DARPA Grand Challenge: The Great Robot Race*. Ed. by Martin Buehler, Karl Iagnemma, and Sanjiv Singh. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 437–482. ISBN: 978-3-540-73429-1. DOI: 10.1007/978-3-540-73429-1_14. URL: https://doi.org/10.1007/978-3-540-73429-1_14.

[20] G. Grisetti et al. "A Tutorial on Graph-Based SLAM". In: *IEEE Intelligent Transportation Systems Magazine* 2.4 (2010), pp. 31–43. DOI: 10.1109/MITS.2010.939925.

[21] R. Kümmerle et al. "G2o: A general framework for graph optimization". In: *2011 IEEE International Conference on Robotics and Automation*. 2011, pp. 3607–3613.

[22] Radu Bogdan Rusu and Steve Cousins. "3D is here: Point Cloud Library (PCL)". In: *IEEE International Conference on Robotics and Automation (ICRA)*. Shanghai, China, May 2011.

[23] Cyrill Stachniss. *lecture notes: Robot Mapping*. Feb. 3, 2014.

[24] Evan Ackermann and Erico Guizzo. *iRobot Brings Visual Mapping and Navigation to the Roomba 980*. 2015. URL: https://spectrum.ieee.org/automaton/robotics/home-robots/irobot-brings-visual-mapping-and-navigation-to-the-roomba-980 (visited on 05/11/2020).

[25] Frank Gauterin and Hans-Joachim Unrau. *Skript zu "Grundlagen der Fahrzeugtechnik II"*. 2017.

[26] L. von Stumberg et al. "From monocular SLAM to autonomous drone exploration". In: *2017 European Conference on Mobile Robots (ECMR)*. 2017.

[27] Marcel Zeilinger et al. "Design of an Autonomous Race Car for the Formula Student Driverless (FSD)". In: May 2017.

[28] Marcus Anderson and Martin Baerveldt. "Simultaneous localization and mapping for vehicles using ORB-SLAM2". MA thesis. Gothenburg, Sweden: Chalmers University of Technology, May 2018.

[29] F. Demim et al. “SLAM Problem for Autonomous Underwater Vehicle using SVSF Filter”. In: *2018 25th International Conference on Systems, Signals and Image Processing (IWSSIP)*. 2018.

[30] Miguel Valls et al. “Design of an Autonomous Racecar: Perception, State Estimation and System Integration”. In: (Apr. 2018).

[31] Axel Brunnbauer and Markus Bader. “Traffic cone based self-localization on a 1 : 10 race car”. In: 2019.

[32] Statistisches Bundesamt. *Causes of accidents involving personal injury - External causes*. 2019. URL: https://www.destatis.de/EN/Themes/Society-Environment/Traffic-Accidents/Tables/causes-accidents-personal-injury2.html (visited on 05/01/2020).

[33] Statistisches Bundesamt. *Causes of accidents involving personal injury - Improper behaviour of pedestrians*. 2019. URL: https://www.destatis.de/EN/Themes/Society-Environment/Traffic-Accidents/Tables/causes-accidents-personal-injury3.html (visited on 05/01/2020).

[34] Statistisches Bundesamt. *Causes of accidents involving personal injury - Technical faults*. 2019. URL: https://www.destatis.de/EN/Themes/Society-Environment/Traffic-Accidents/Tables/causes-accidents-personal-injury1.html (visited on 05/01/2020).

[35] Statistisches Bundesamt. *Driver-related causes of accidents involving personal injury*. 2019. URL: https://www.destatis.de/EN/Themes/Society-Environment/Traffic-Accidents/Tables/driver-mistakes.html (visited on 05/01/2020).

[36] Juraj Kabzan et al. “AMZ Driverless: The Full Autonomous Racing System”. In: (May 2019).

[37] Felix Nobis et al. “Autonomous Racing: A Comparison of SLAM Algorithms for Large Scale Outdoor Environments”. In: Feb. 2019, pp. 82–89. DOI: 10.1145/3332305.3332319.

[38] statista. *Verteilung der Investitionen durch Unternehmen aus der Automobilindustrie nach Zieltechnologien im Jahr 2017*. 2019. URL: https://de.statista.com/statistik/daten/studie/1040890/umfrage/verteilung-der-investitionen-durch-unternehmen-der-automobilindustrie-nach-bereichen/ (visited on 05/01/2020).

[39] Kristian Wahlqvist. “A Comparison of Motion Priors for EKF-SLAM in Autonomous Race Cars”. MA thesis. Stockholm, Sweden: KTH, School of Electrical Engineering and Computer Science (EECS), July 2019.

[40] Statistisches Bundesamt. *Trend in the number of persons killed in road traffic accidents*. 2020. URL: https://www.destatis.de/EN/Themes/Society-Environment/Traffic-Accidents/_Graphic/_Interactive/traffic-accidents-persons-killed-year.html (visited on 05/01/2020).

[41] *Formula Student Online Results*. 2020. URL: https://formulastudentonline.com/?page_id=712 (visited on 05/22/2020).

[42] Sherif Nekkah et al. “The Autonomous Racing Software Stack of the KIT19d”. In: *arXiv e-prints*, arXiv:2010.02828 (Oct. 2020), arXiv:2010.02828. arXiv: 2010.02828 [cs.RO].

[43] statista. *Anzahl der gemeldeten Pkw in Deutschland in den Jahren 1960 bis 2020*. 2020. URL: https://de.statista.com/statistik/daten/studie/12131/umfrage/pkw-bestand-in-deutschland/ (visited on 05/01/2020).

[44] Oxford Advanced American Dictionary. *Definition of "robot"*. URL: https://www.oxfordlearnersdictionaries.com/definition/american_english/robot (visited on 07/08/2020).

[45] KA-Raceing e.v. *Erfolge*. URL: https://www.ka-raceing.de/erfolge (visited on 05/22/2020).

[46] Alexander Entinger. *Einführung in das Robot Operating System*. URL: https://m.heise.de/developer/artikel/Einfuehrung-in-das-Robot-Operating-System-3273655.html?seite=all (visited on 05/25/2020).

[47] Formula Student Germany. *Formula Student Handbook 2020*. URL: https://www.formulastudent.de/fileadmin/user_upload/all/2020/rules/FSG20_Competition_Handbook_v1.0.pdf (visited on 05/25/2020).

[48] Formula Student Germany. *Formula Student Rules 2020*. URL: https://www.formulastudent.de/fileadmin/user_upload/all/2020/rules/FS-Rules_2020_V1.0.pdf (visited on 05/22/2020).

[49] Formula Student Germany. *History of Formula Student Germany*. URL: https://www.formulastudent.de/about/chronicle/ (visited on 05/22/2020).

[50] Institution of Mechanical Engineers. *History of Formula Student*. URL: https://www.imeche.org/events/formula-student/about-formula-student/history-of-formula-student (visited on 05/22/2020).

[51] Formula SAE. *History of Formula SAE*. URL: https://www.fsaeonline.com/page.aspx?pageid=c4c5195a-60c0-46aa-acbf-2958ef545b72 (visited on 05/22/2020).