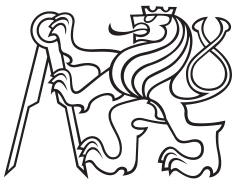
Master Thesis



Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Computer Science

Navigation System for Autonomous Student Formula

Tomáš Roun

Supervisor: Ing. Jan Čech, PhD
Field of study: Open Informatics
Subfield: Artificial Intelligence
May 2021

Acknowledgements

I would like to sincerely thank everybody who supported me in completing this thesis and in my work for eForce. I would especially like to thank Jan Čech, my thesis supervisor. His guidance and experience were invaluable in making the thesis what it is. I am incredibly grateful for his expertise and his willingness to guide me and to help me in any way, shape, or form for the past two years.

I also want to thank all my fellow past and current teammates in eForce without whom this thesis would not exist. I am grateful for everybody who contributed to the success of our team. However, I would like to name a few who had the biggest impact on our team and this thesis:

- Ondřej Šereda & Marek Szeles, for giving me the chance to be a part of eForce and singlehandlely keeping our team together during its rocky start and continuing to do so to this day.
- Dan Štorc, for filling the boots after Marek as the captain of eForce Driverless, devoting his every waking hour to eForce and for just always being available when I needed his help. Dan was instrumental in the successful completion of this thesis, giving me the freedom to focus on my thesis full-time.
- Matěj Kopecký, for his immense expertise in LaTeX and helping me with typesetting TikZ figures.

A big thank you also goes to the entire eForce Driverless team – Michal Horáček, Jan Svoboda, Pavel Král, Roman Šíp, Andrea Hauptová, Vít Ramba, Ondřej Kuban, Hynek Zamazal, Antonín Gazda and many more.

Furthermore, I want to thank the European Organization for Nuclear Research (CERN) and my former supervisor Sebastian Bukowiec for giving me the opportunity to join such an amazing place. Apart from being one of the best experiences of my life, my stay at CERN gave me the time I needed to finish my thesis.

Finally, I want to thank my family, especially my parents and my sister, for their unwavering support throughout my studies. Last but not least, I owe a big thank you to my girlfriend Anna for her advice and help on numerous occasions, but most importantly, her support and patience which were paramount in completing this thesis.

Declaration

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze, 21. května 2021

Abstract

The thesis presents the design and implementation of a navigation system for an autonomous formula participating in the Formula Student competition. In particular, the thesis proposes a real-time implementation of the Simultaneous localization and mapping problem (SLAM). In this problem, a mobile robot is placed in an unknown environment. The robot moves through the environment and uses the information from its sensors to construct a virtual map and simultaneously localize itself within the map. The thesis discusses the relevance of SLAM to the Formula Student competition. Following, a thorough theoretical overview of SLAM from the probabilistic viewpoint is provided. Then, a parallel real-time implementation of FastSLAM 1.0 using GPUs is proposed. We provide a detailed description of our implementation encompassing the whole architecture, the technologies, and data structures used. The proposed implementation is evaluated on a series of simulated and real-world datasets assessing the accuracy and performance. Compared to the implementation provided by Python Robotics, our implementation achieves higher accuracy while being orders of magnitude faster. Based on the experimental results, we conclude that our implementation is suitable for a real-time navigation system for an autonomous formula.

Keywords: Formula Student, Navigation System, SLAM, GPU, Real-time

Supervisor: Ing. Jan Čech, PhD
Karlovo náměstí 13,
Praha 2

Abstrakt

Tato práce představuje návrh a implementaci navigačního systému pro studentskou autonomní formuli účastnící se soutěže Formula Student. Konkrétně tato práce navrhuje real-time implementaci problému Simultánní lokalizace a mapování (SLAM). V tomto problému je mobilní robot umístěn do neznámého prostředí. Robot se v tomto prostředí pohybuje a za použití senzorů vytváří virtuální mapu a zároveň se v této mapě lokalizuje. Tato práce rozebírá relevantnost SLAMu pro soutěž Formula Student. Následně je uveden teoretický přehled SLAMu z pravděpodobnostního hlediska. Poté je navržena paralelní real-time implementace algoritmu FastSLAM 1.0 využívající GPU. Zároveň je poskytnut podrobný popis naší implementace zahrnující celou architekturu, použité technologie a datové struktury. Navrhovaná implementace je vyhodnocena na sérii simulovaných a reálných datasetů, kde se měří její přesnost a rychlost. V porovnání s implementací od Python Robotics naše implementace dosahuje vyšší přesnosti a je několikařádově rychlejší. Na základě těchto experimentálních výsledků vyvozujeme, že naše implementace je vhodná pro real-time navigační systém autonomní formule.

Klíčová slova: Formula Student, Navigační systém, SLAM, GPU, Real-time

Překlad názvu: Navigační systém pro autonomní studentskou formuli

Contents

Figures

Tables

List of symbols

Symbol	Meaning
k	discrete time index
$\boldsymbol{x}$	state vector
$\boldsymbol{z}$	measurement vector
$\boldsymbol{u}$	odometry vector
$\boldsymbol{x}_{1:k}$	a sequence of variables $\boldsymbol{x}_1, \boldsymbol{x}_2, .., \boldsymbol{x}_k$
$p(\boldsymbol{x})$	probability density
$p(\boldsymbol{x}, \boldsymbol{y})$	joint probability density
$p(\boldsymbol{x}\|\boldsymbol{y})$	conditional probability density
$\boldsymbol{x} \sim p(\boldsymbol{x})$	$\boldsymbol{x}$ is distributed according to $p(\boldsymbol{x})$
$\hat{\boldsymbol{x}}$	estimate of $\boldsymbol{x}$
$\mathbb{E}[\boldsymbol{x}]$	expectation of $\boldsymbol{x}$
$\mathbb{E}[\boldsymbol{x}\|\boldsymbol{y}]$	conditional expectation of $\boldsymbol{x}$ given $\boldsymbol{y}$
$Var[\boldsymbol{x}]$	variance of $\boldsymbol{x}$
$\boldsymbol{\mu}$	mean vector
σ^2	variance
Σ	covariance matrix
$\boldsymbol{\Omega}$	information matrix
$\mathcal{N}(\boldsymbol{\mu}, \Sigma)$	normal distribution
$\mathcal{N}(\boldsymbol{x}; \boldsymbol{\mu}, \Sigma)$	normal distribution evaluated at $\boldsymbol{x}$
$\mathcal{U}(a, b)$	uniform distribution
$\boldsymbol{x}^i$	particle i
$f(.)$	process model
$g(.)$	motion model
$h(.)$	measurement model
Q	motion model covariance
R	measurement model covariance
$\mathcal{O}(n)$	big O notation
$\Theta(n)$	big Theta notation

Chapter 1

Introduction

The thesis presents the design and implementation of a navigation system for an autonomous student formula participating in the Formula Student competition. Formula Student is one of the biggest and most prestigious engineering competitions in the world. Every year, university teams from all over the world try to build the best racing formula possible. Hundreds of teams compete in many racing events held during the year, mimicking the famous Formula 1 competition. The competition consists of three classes – Combustion vehicles (CV), Electric Vehicles (EV), and the newest class added in 2017 – Driverless vehicles (DV). In the Driverless class, the task is not only to design a racing formula that can be driven by a human pilot, but in addition, the car is also required to be capable of a fully autonomous operation. This means that the car has to carry extra sensors and hardware that allows it to safely operate without human intervention. For this reason, the Driverless class is the most challenging of the three classes, combining not only various fields of mechanical and electrical engineering but also many fields of computer science and robotics such as computer vision, machine learning, autonomous driving and many more.

The thesis aims specifically to provide a navigation system based on simultaneous localization and mapping for an autonomous formula. Simultaneous localization and mapping (SLAM) is a fundamental problem in robotics [13]. In this problem, a moving robot is placed in an unknown environment and its goal is to simultaneously build a map (mapping) of the environment and keep track of its own position with respect to the map (localization). The challenge is made difficult by assuming that both the robot control and the sensors perceiving the environment contain a certain level of noise. This leads to a seemingly impossible chicken-and-egg-like problem in which the robot has to map the environment using noisy information about its pose and at the same time localize itself by taking noisy measurements of the environment. The SLAM problem is relevant in many areas such search and

Figure 1.1: The autonomous formula for the 2021 season

rescue, underground and underwater exploration, self-driving cars, and many more. Despite the seeming difficulty of SLAM, several SLAM algorithms have been successfully used in many real-world applications using a wide array of distinct approaches to solving the problem.

The driverless formula is being developed by the team eForce Driverless which is part of the parent team eForce FEE Prague Formula operating at CTU Prague. eForce has been founded more than a decade ago and has become very successful in the electric class of Formula Studenet, placing in top 3 in many races and even winning several. eForce Driverless hopes to replicate its success in the Driverless class. Fig. 1.1 shows our autonomous car for the 2021 season. The formula is equipped with several sensors. On the front wing, two Intel Realsense cameras are placed on opposite sides. In addition, an Ouster OS1 LiDAR is placed in the middle of the front wing. A Stereolabs ZED camera is also placed at the top of the main hoop.

The main contribution of the thesis is the design and implementation of a simultaneous localization and mapping system for real-time autonomous navigation. The implementation is aimed mainly for use in the Formula Student competition, but is written to be general purpose so that it can be used in a variety of different systems. To achieve real-time capability, the algorithm is implemented using GPUs, which allows for a high update frequency. The proposed implementation is evaluated on several simulated and real-world datasets to assess its accuracy and performance. The experimental results confirm the suitability of our implementation for accurate real-time SLAM.

1.1 Thesis structure

The rest of the thesis is organized into the following chapters:

- **Formula Student competition** - This chapter describes the Formula Student competition and the history of eForce Driverless form its conception up to the current season (2021).
- **Recursive Bayesian filter** - In this chapter, we describe the recursive Bayesian filter and how it can model the problems concerning stochastic state estimation. As we show, most of the theory of SLAM is built on top of the Bayesian filter.
- **Kalman filter** - This chapter discusses the Kalman filter, a popular state estimation technique widely used in robotics, which is also a special case of the Bayesian filter.
- **Particle filter** - This chapter introduces particle filters and their use for state estimation. Particle filters, together with Kalman filters, form the basic structure of several SLAM algorithms.
- **Simultaneous localization and mapping** - In this chapter, the problem of SLAM is described in detail. We explain how the problem is formalized using a probabilistic framework and show three different solution approaches - EKF-SLAM [15], FastSLAM [64] and Graph SLAM [57]. The chapter also gives a brief overview of the different SLAM variants and discusses current approaches to the problem.
- **FastSLAM GPU implementation** - This chapter proposes an efficient implementation of the FastSLAM algorithm implemented on a GPU for real-time SLAM applications such as the Formula Student competition.
- **Experimental results** - In this chapter, the accuracy and performance of the proposed implementation is evaluated on a variety of simulated and real SLAM datasets.
- **Conclusion** - This chapter summarizes the results and the contributions made in the thesis. We also discuss future work and possible improvements to the implementation.

The first chapter describes the Formula Student competition and a brief history of eForce Driverless. The next three chapters are meant to be introductory and explain the theory behind recursive state estimation and various specializations such as Kalman and particle filters. For experienced readers, these chapters can be skipped and one can instead start with the chapter introducing SLAM. Throughout the thesis, we also ocassionaly refer

the reader to the appendix for more detailed information on various topics which are not included in the main text to keep it focused. Similarly, the appendix serves merely as a supplement to the main text. However, for the interested reader, the appendix contains various proofs of theorems mentioned in the thesis and extra material related to particle filters and SLAM.

Chapter 2

Formula Student competition

The Formula Student competition organizes many annual racing events. Some of the most popular races are held in Germany, Italy, Spain, Hungary, United States, United Kingdom and Czechia. Each individual race consists of several events which are divided into static and dynamic. In static events, the goal is to present the conceptual design of the car, which is judged by experts from the automotive industry. Static events include the engineering design event, business presentation, and the manufacturing & cost event. In dynamic events, the cars compete on several different tracks for the best time. The dynamic events are different for each class. In the DV class, the dynamic events consist of acceleration, skidpad, autocross, and trackdrive. For each of these events, teams earn a certain number of points based on the performance in that particular event. The overall winner is then the team with the most points.

In all dynamic events, the vehicle is required to stay within the given track delineated by traffic cones of different colors shown in Fig. 2.1. Hitting a cone or leaving the track is penalized by extra time or even a disqualification from that particular event. In the acceleration event depicted in Fig. 2.2, the car has to autonomously accelerate as much as possible in a straight 75 meters long line before safely stopping at the end. The goal of this event is to push the car to the limit in terms of maximum speed and the ability to accelerate quickly. The skidpad track shown in Fig. 2.3 consists of two pairs of concentric circles in a figure of eight pattern. The vehicle starts in the middle, entering the right circle first, then entering the left, and alternating until completing four circles on each side. The autocross and trackdrive events are very similar. Both are driven on a closed-loop circuit of up to 1km in length containing many challenging features such as chicanes, hairpin turns, and decreasing-radius turns. The difference is that in the autocross event, only a single lap is completed with teams trying to get the fastest single-lap time. In trackdrive, on the other hand, the full 10 laps are driven and the

Figure 2.1: Traffic cones are used to delineate the track.

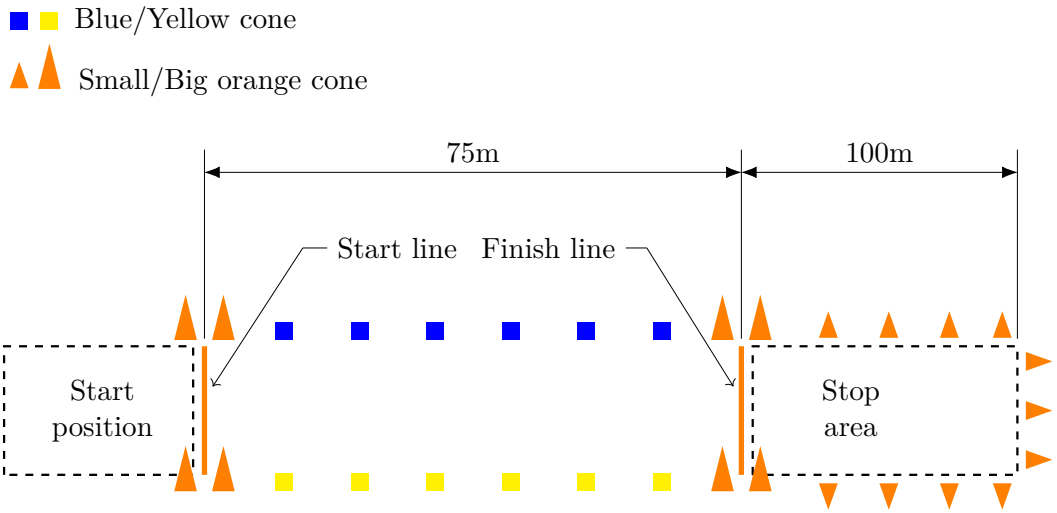


Figure 2.2: A depiction of the track for the acceleration event

teams compete for the fastest 10-lap time. A small segment of a track used for these two events is shown in Fig. 2.4.

2.1 Autonomous System

To ensure that the autonomous cars successfully complete all dynamic events, a multitude of sensors need to be used. The decision of which specific sensors to use is up to the individual teams. The most common proprioceptive sensors include wheel and ground speed sensors, GPS units, and internal measurement units. Every car is also usually equipped with either an RGB camera (mono or stereo) or a LiDAR or a combination of both. It is common to encounter a setup with a LiDAR and multiple cameras positioned at different angles to cover a wide field of view. However, using multiple sensors brings its own difficulties such synchronization and sensor fusion.

In the years since the driverless class was established, two distinct approaches to the design of the autonomous system emerged – a reactive approach and a SLAM-capable approach. With the reactive system, the behavior of the car is determined solely by the current sensor inputs. In

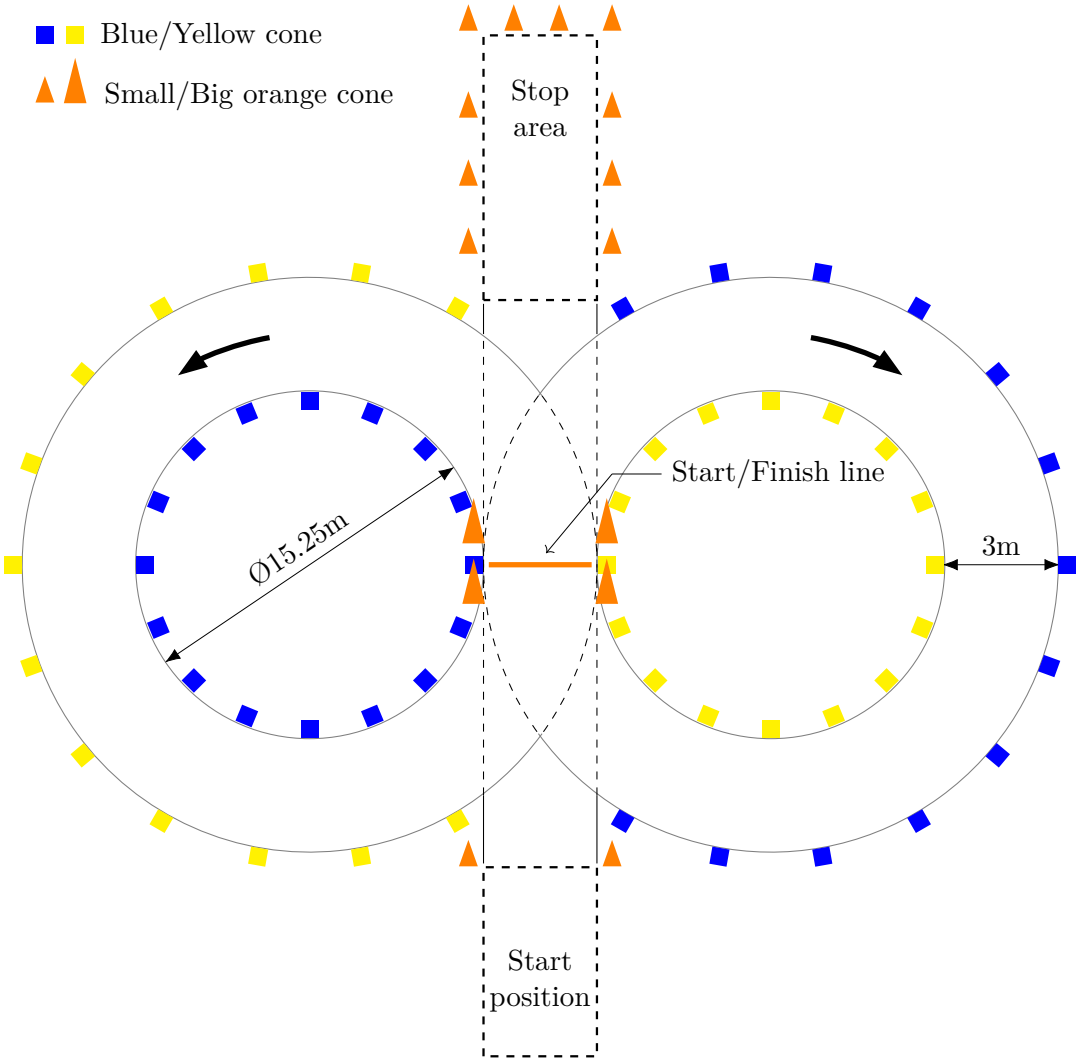


Figure 2.3: A depiction of the track for the skidpad event

other words, the reactive system does not retain any information about the environment from the past observations. The reactive autonomous system is both conceptually simple and easy to implement. An example of a reactive algorithm is following the center line of the visible part of the track. The center line is computed every time using the position of the cones extracted from the current sensor input. A major disadvantage of this approach is that it significantly limits the speed that the car can safely maintain. This is because the car only sees as far as the sensors do and thus has to drive defensively.

If a system that incorporates SLAM is used, the formula can use information from the previous lap to reach faster speeds in areas where the vehicle dynamics allow it. Knowledge of the full track also improves the path planning of the system. It can, for example, anticipate sharp turns and plan

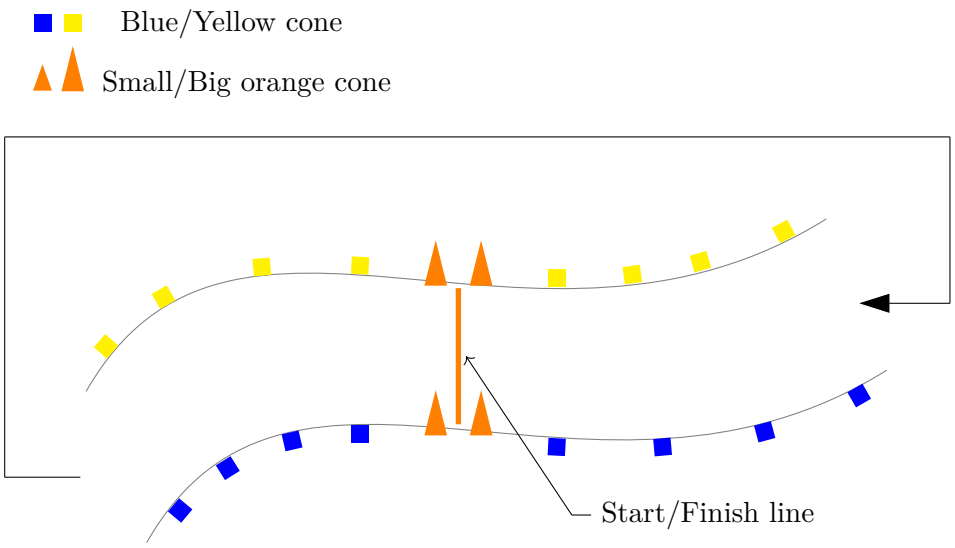


Figure 2.4: A small section of the trackdrive/autocross track

accordingly. Thus, a SLAM-based system can bring a big advantage to the team. However, compared to a reactive system, a successful implementation of SLAM is significantly more complex to achieve. In addition, to ensure safety, the autonomous system including SLAM, has to operate in real-time. Considering that an electric formula can reach velocities exceeding 100km/h, the system has to operate reliably at very high frequencies.

2.2 eForce Driverless

Team eForce Driverless was officially founded in 2019 as the first Formula Student Driverless team in the Czech Republic. eForce Driverless was founded by members of eForce FEE Prague Formula team which at that point had been operating for nearly a decade in the electric category. From an organizational standpoint, eForce Driverless remains as a semi-independent section of the main electric team sharing resources and most importantly, members. In the next few paragraphs, we summarize the history of the team, the challenges we had to face, and a recollection of our first and unfortunately, as of writing this thesis, the last race our team has attended. An in-depth description of the founding of the team, its history, and an analysis from a project management standpoint is given by [96].

eForce Driverless was founded by two long-time members of eForce at the beginning of 2019. The original team counting approximately ten members was assembled in March of that year. The first few months of this newly founded team were challenging. None of the members had any previous

experience with autonomous driving and the first few months were spent doing research on the topic. Eventually, we built up a general idea of how an autonomous formula should operate and what were the main components. We realised that the work could essentially be split into two parts – hardware & software. The hardware part included adapting one of the older formulas, outfitting it with sensors and self-driving features, and making sure it complies with the rules of Formula Student. The software part entailed writing the actual software stack together with the design of detection, navigation, and planning algorithms to allow for autonomous operation.

As we found out, the software part was comparatively easier. This was mainly because at first, the software and algorithms could be implemented on any computer without needing the physical car for testing. As such, after a few months we were already working on cone detection and path planning algorithms. The biggest hurdle slowing down our progress was the hardware part. This part required completely rebuilding one of the cars from previous seasons. The changes were numerous. Both steering and braking had to be made automatic, which required many changes. In addition, the car now had to carry more electronics, including cameras, a LiDAR, and the computing unit, which necessitated reworking the power delivery as well. Many circuit boards had to be redesigned and many new boards had to be made.

However, the problems we encountered were not just technical. The biggest problem we faced was the lack of time and manpower. Our newly assembled team did not possess the knowledge and experience that was required to implement the hardware modifications. The members of the parent team, which had the knowledge, were often busy working on the new electric car. Up to this point, the team has never attempted to build two new cars in one season. Unfortunately, this meant that we were not ready to participate in the 2019 season. However, during the summer of 2019, we started cooperating with Ing. Jan Čech, Ph.D. from the Center for Machine Perception at CTU Prague. Mr. Čech proved to be instrumental in propelling our team forward, providing much needed expertise in machine learning and computer vision. The Center for Machine Perception also provided us with resources to purchase several sensors, computing units, and various tools for which we are extremely grateful. Thanks in no small part to Jan Čech and the Center for Machine Perception, we were on schedule to compete in the 2020 season, which would have been our first season ever.

After passing the entrance tests, we qualified to multiple races held that season. The races were FS Czech, FS Spain, and FS Germany, which is famously difficult to be accepted to. However, due to the Covid-19 pandemic, which was by that time in full swing in Europe, all planned races for the 2020 season were cancelled. Instead, a fully online competition called FS Online was organized to at least partially replace the cancelled races. As this was our only chance to compete that season, our team decided to participate in

Figure 2.5: The FS Online simulator used to facilitate virtual races

this virtual event. Other teams that attended included teams from MIT, Delft, Munich, Hamburg and others. Since it was not possible to organize an in-person event, FS Online was to be held completely online including the races. To make this possible, we were provided with a simulation environment and a virtual formula which would be controlled by our autonomous system. The simulation environment is shown in Fig. 2.5.

The formula was equipped with virtual cameras, LiDARs, an inertial measurement unit, ground speed sensors, and GPS. FS Online held two dynamic events – autocross and trackdrive. These events were held for 3 consecutive days and each day the difficulty of the track increased. Because the competition was announced at a relatively short notice, many components of our autonomous system were not ready in time. Hence, we decided to design a simple yet robust reactive system using input from the virtual LiDAR. Using a combination of filtering and clustering, the LiDAR pointclouds were turned into individual traffic cones delineating the track. With these cones, waypoints marking the center line were constructed. The controller then steered the car so that it stayed as close to these waypoints as possible. This system proved as a fairly reliable and robust option bringing us a lot of success. In a few events, our formula was the only one able to complete the full track without a problem. Overall, the FS Online competition gave us a lot of experience and confidence in our capabilities and our algorithms and pushed us even more to more sophisticated solutions, most notably, SLAM.

Chapter 3

Recursive Bayesian filter

Recursive Bayesian filtering [16] is a technique to estimate the state of a system using the knowledge of the system state in the past combined with noisy measurements. The system is assumed to be stochastic and evolving over time. Recursive filtering allows one to repeatedly estimate the system state as new measurements become available. More specifically, the Recursive Bayesian filter works by recursively estimating the posterior probability density of the system state given all previous measurements. Recursive Bayesian filtering is used in a wide range of areas such as signal processing, navigation, tracking, econometrics, and healthcare [52, 44, 27, 75, 10, 76, 87, 85]. In robotics, many problems involving tracking, localization, and mapping are modeled and solved using special cases of the Recursive Bayesian filter such as Kalman and Particle filters [9, 97, 5, 88, 72]. Furthermore, many SLAM formulations, e.g., EKF-SLAM [15] and FastSLAM [64] are derived directly from a Recursive Bayesian filter.

3.1 Derivation

Consider a dynamical system or a stochastic process that evolves in discrete time steps denoted as $1, 2, ..., k$. The process model, which describes how the system evolves in time, is stochastic and given as follows

$$\boldsymbol{x}_k = f_k(\boldsymbol{x}_{1:k-1}) + \boldsymbol{q_k}, \tag{3.1}$$

where $\boldsymbol{x}_k$ is the system state at time k, f_k is the process model at time k, $\boldsymbol{x}_{1:k-1}$ is a sequence of system states from 1 to $k-1$ and $\boldsymbol{q_k}$ is a random variable functioning as a source of additive noise. Notice that the evolution

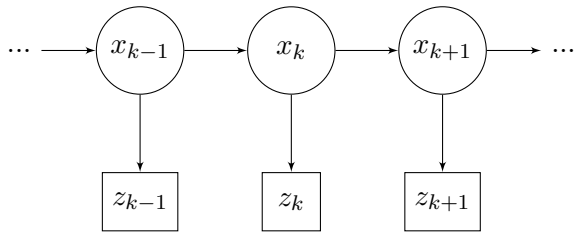


Figure 3.1: The recursive Bayes filter can be applied to any stochastic system that can be modeled as a hidden Markov model (HMM). That is, the model is described by a stochastic state transition function together with a stochastic observation.

of the system consists of a deterministic component f_k and a stochastic component $\boldsymbol{v}_k$. In an alternative notation, the new state can be expressed as:

$$\boldsymbol{x}_k \sim p(\boldsymbol{x}|\boldsymbol{x}_{1:k-1}) \tag{3.2}$$

We assume that the process model has the Markov property, i.e., it holds that

$$p(\boldsymbol{x}_k|\boldsymbol{x}_{1:k-1}) = p(\boldsymbol{x}_k|\boldsymbol{x}_{k-1}). \tag{3.3}$$

In order to make estimates of the system state, we also need a way to measure the state. Hence, we assume the ability to take noisy measurements, also called observations, of the true system state which is not known. The measurement of $\boldsymbol{x}_k$ is given as

$$\boldsymbol{z}_k = h_k(\boldsymbol{x}_{1:k}) + \boldsymbol{r}_k \tag{3.4}$$

Here, $\boldsymbol{z}_k$ is the measurement at time k, h_k is the measurement model and $\boldsymbol{r}_k$ is again a source of additive noise. We may write this as

$$\boldsymbol{z}_k \sim p(\boldsymbol{z}|\boldsymbol{x}_{1:k}). \tag{3.5}$$

Similarly to the process model, the measurement model is also assumed to depend only on the current state x_k. Mathematically:

$$p(\boldsymbol{z}_k|\boldsymbol{x}_{1:k}) = p(\boldsymbol{z}_k|\boldsymbol{x}_k). \tag{3.6}$$

The above description essentially constitutes a Hidden Markov Model (HMM) with a state transition function f_k and a measurement function h_k as shown in Fig. 3.1. In Bayesian filtering, the posterior density $p(\boldsymbol{x}_k|\boldsymbol{z}_{1:k})$ is computed by combining the previous belief $p(\boldsymbol{x}_{k-1}|\boldsymbol{z}_{1:k-1})$ and a new measurement $\boldsymbol{z}_k$. All information about the system is contained in the posterior. The computation of the posterior is split into two steps – the prediction step and the correction step. In the prediction step, the filter first computes the prior (also called proposal) distribution $p(\boldsymbol{x}_k|\boldsymbol{z}_{1:k-1})$ using only the process model. This step can be thought of as propagating the previous belief given by $p(\boldsymbol{x}_{k-1}|\boldsymbol{z}_{1:k-1})$ using the system dynamics. To give a concrete example, in the robotics setting, the prediction step can be imagined as predicting the new position of a moving robot based on the data provided by an inertial measurement unit (IMU) or a GPS. The prior distribution is derived as follows:

$$p(\boldsymbol{x}_k|\boldsymbol{z}_{1:k-1}) = \frac{p(\boldsymbol{x}_k, \boldsymbol{z}_{1:k-1})}{p(\boldsymbol{z}_{1:k-1})} = \frac{\int p(\boldsymbol{x}_k, \boldsymbol{x}_{k-1}, \boldsymbol{z}_{1:k-1}) d\boldsymbol{x}_{k-1}}{p(\boldsymbol{z}_{1:k-1})} \tag{3.7}$$

$$= \frac{\int p(\boldsymbol{x}_k|\boldsymbol{x}_{k-1}, \boldsymbol{z}_{1:k-1}) p(\boldsymbol{x}_{k-1}|\boldsymbol{z}_{1:k-1}) p(\boldsymbol{z}_{1:k-1}) d\boldsymbol{x}_{k-1}}{p(\boldsymbol{z}_{1:k-1})} \tag{3.8}$$

$$= \int p(\boldsymbol{x}_k|\boldsymbol{x}_{k-1}, \boldsymbol{z}_{1:k-1}) p(\boldsymbol{x}_{k-1}|\boldsymbol{z}_{1:k-1}) d\boldsymbol{x}_{k-1} \tag{3.9}$$

$$\overset{Markov}{=} \int \underbrace{p(\boldsymbol{x}_k|\boldsymbol{x}_{k-1})}_{\text{process model}} \underbrace{p(\boldsymbol{x}_{k-1}|\boldsymbol{z}_{1:k-1})}_{\text{posterior at } k-1} d\boldsymbol{x}_{k-1}. \tag{3.10}$$

In equation (3.7), we used the Chapman-Kolmogorov identity. In the last equation, we used the Markov property of the process model. Note that the result only depends on the process model and the previous posterior. The second step of the filter is the correction step. Informally, the correction step *corrects* the prior distribution using the information from the newly acquired measurement. Formally, it computes the new posterior density $p(\boldsymbol{x}_k|\boldsymbol{z}_{1:k})$ using the prior $p(\boldsymbol{x}_k|\boldsymbol{z}_{1:k-1})$ together with the measurement $\boldsymbol{z}_k$:

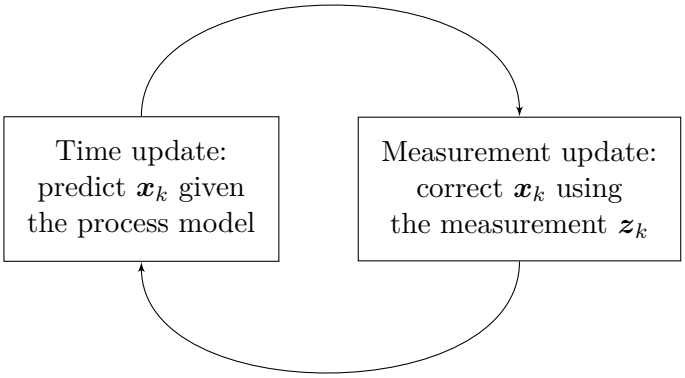


Figure 3.2: The recursive Bayesian filter computes the posterior in two steps. First, in the prediction step, the prior is computed by propagating the state using the process model. In the correction step, the prior is corrected using the newly acquired measurement.

$$
\begin{aligned}
p(\boldsymbol{x}_k|\boldsymbol{z}_{1:k}) &= p(\boldsymbol{x}_k|\boldsymbol{z}_k, \boldsymbol{z}_{1:k-1}) && (3.11)\\
&= \frac{p(\boldsymbol{x}_k, \boldsymbol{z}_k|\boldsymbol{z}_{1:k-1})}{p(\boldsymbol{z}_k|\boldsymbol{z}_{1:k-1})} && (3.12)\\
&= \frac{p(\boldsymbol{z}_k|\boldsymbol{x}_k, \boldsymbol{z}_{1:k-1})p(\boldsymbol{x}_k|\boldsymbol{z}_{1:k-1})}{p(\boldsymbol{z}_k|\boldsymbol{z}_{1:k-1})} && (3.13)\\
&\overset{Bayes}{=} \frac{p(\boldsymbol{z}_k|\boldsymbol{x}_k)p(\boldsymbol{x}_k|\boldsymbol{z}_{1:k-1})}{p(\boldsymbol{z}_k|\boldsymbol{z}_{1:k-1})} && (3.14)\\
&= \frac{p(\boldsymbol{z}_k|\boldsymbol{x}_k)p(\boldsymbol{x}_k|\boldsymbol{z}_{1:k-1})}{\int p(\boldsymbol{z}_k|\boldsymbol{x}_k)p(\boldsymbol{x}_k|\boldsymbol{z}_{1:k-1})d\boldsymbol{x}_k} && (3.15)\\
&\propto \underbrace{p(\boldsymbol{z}_k|\boldsymbol{x}_k)}_{\substack{\text{measurement}\\ \text{model}}} \underbrace{p(\boldsymbol{x}_k|\boldsymbol{z}_{1:k-1})}_{\text{prior}}. && (3.16)
\end{aligned}
$$

In equation (3.15), we used the measurement independence assumption (3.6). In essence, the correction step is an application of the Bayes formula. The measurement model $p(\boldsymbol{z}_k|\boldsymbol{x}_k)$ is the likelihood, $p(\boldsymbol{x}_k|\boldsymbol{z}_{1:k-1})$ is the prior and the denominator $p(\boldsymbol{z}_k|\boldsymbol{z}_{1:k-1})$ is the evidence. Note that the evidence does not depend on the state. It can thus be thought of as a normalizing constant and in many applications it need not be computed at all. Fig. 3.2 illustrates the prediction-correction loop of the filter. Of note is also the fact that both the prediction and correction steps only need to remember the previous posterior to compute the next one. It is therefore not necessary to remember the full history of the filter.

Since a Bayesian filter provides the full posterior, we have the freedom of

computing various estimators of this probability density. There are many different possibilities, however, here we list some of the most common estimators. A thorough review is provided in [16].

1. Maximum a posteriori (MAP) – The MAP estimate finds the largest mode of the posterior which is given by

$$\hat{\boldsymbol{x}}_k^{MAP} = \arg\max_{\boldsymbol{x}_k} \; p(\boldsymbol{x}_k|\boldsymbol{z}_{1:k}).$$

2. Maximum likelihood (ML) – The ML estimate is a special case of MAP in which the prior given by $p(\boldsymbol{x}_k|\boldsymbol{z}_{1:k-1})$ is disregarded. The estimate is then

$$\hat{\boldsymbol{x}}_k^{ML} = \arg\max_{\boldsymbol{x}_k} \; p(\boldsymbol{z}_k|\boldsymbol{x}_k).$$

3. Minimum mean square error (MMSE) – MMSE selects $\hat{\boldsymbol{x}}_k^{MMSE}$ which minimizes

$$\mathbb{E}\left[||\boldsymbol{x}_k - \hat{\boldsymbol{x}}_k||^2|\boldsymbol{z}_{1:k}\right] = \int ||\boldsymbol{x}_k - \hat{\boldsymbol{x}}_k||^2 \; p(\boldsymbol{x}_k|\boldsymbol{z}_{1:k}) d\boldsymbol{x}_k.$$

   As we show in Appendix A.1.1, this formulation leads to

$$\hat{\boldsymbol{x}}_k^{MMSE} = \mathbb{E}\left[\boldsymbol{x}_k|\boldsymbol{z}_{1:k}\right].$$

   In other words, the estimator is a conditional expectation of the posterior. A comparison between the MAP and MMSE estimators is shown in Fig. 3.3.

It is important to realise that computing the full posterior of the Bayesian filter is not tractable in the vast majority of cases. To obtain the posterior, we need to evaluate several integrals which typically do not have a closed-form solution, except for a few restrictive cases. One such case is when the process and measurement models are linear and the noise is Gaussian. These restrictions lead to the well-known Kalman filter [43] which we describe in the following chapter. In other cases when the posterior cannot be evaluated exactly, a numerical integration can be used to approximate the posterior. A particle filter [4] is an example of a Monte Carlo technique that approximates the posterior using a finite set of samples called particles. The theory behind particle filters, which play an integral part in FastSLAM, is described in more detail in chapter 5.

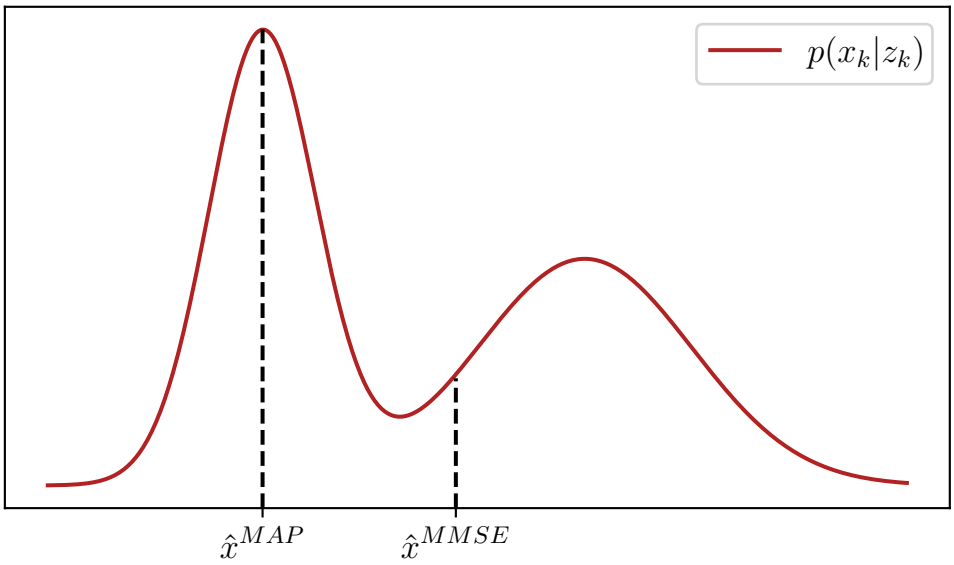


Figure 3.3: The difference between the maximum a posteriori (MAP) and the minimum mean square error (MMSE) estimator on a multimodal posterior. For $\hat{\boldsymbol{x}}^{MMSE}$, the estimator is equal to the conditional expectation while for $\hat{\boldsymbol{x}}^{MAP}$, it is the highest mode of the posterior. In the Gaussian case, the two estimators coincide.

Chapter 4

Kalman filter

Kalman filter is one of the few special cases of the Recusive Bayesian filter for which there is an exact analytical solution [43]. Despite being introduced more than 50 years ago, Kalman filter still plays an important role in navigation and tracking of autonomous vehicles and aircraft, signal processing, econometrics and many more [39, 61, 65, 40, 5, 88, 72]. In robotics, Kalman filter is mainly used in tasks involving tracking and Simulatenous Localization and Mapping [7, 64].

Kalman filter is of particular importance due to the fact that under certain assumptions, the filter is unbiased and optimal. Optimal in this case means that the Kalman filter minimizes the minimum mean square error (MMSE) described in (3). Unfortunately, the required assumptions are rather restrictive. Kalman filter requires a linear process and measurement model in combination with a Gaussian noise. As very few systems can be modeled using linear models in practise, the application of the basic Kalman filter is rather limited. However, the linearity assumption of the filter may be relaxed. This relaxation sacrifices the optimality of the filter. Nevertheless, the so-called Extended Kalman filter (EKF) [59] is a widely used modification that performs well provided the nonlinearity of the system is not too severe.

4.1 Model description

The Kalman filter requires that the process model be linear. We can write this as

$$\boldsymbol{x}_k = f_k(\boldsymbol{x}_{k-1}, \boldsymbol{u}_k) + \boldsymbol{q}_k \tag{4.1}$$
$$= F_k \boldsymbol{x}_{k-1} + B_k \boldsymbol{u}_k + \boldsymbol{q}_k. \tag{4.2}$$

In the above equation, F_k is the state-transition matrix and B_k is the control-input model applied to the control vector $\boldsymbol{u}_k$. The process model is again stochastic containing additive noise given by $\boldsymbol{q}_k \sim \mathcal{N}(0, Q_k)$ which is a zero-mean Gaussian with covariance Q_k. Similarly, the measurement model is also linear:

$$\boldsymbol{z}_k = h_k(\boldsymbol{x}_k) + \boldsymbol{r}_k \tag{4.3}$$
$$= H_k \boldsymbol{x}_k + \boldsymbol{r}_k. \tag{4.4}$$

The matrix H_k is the linear measurement model. Analogously, $\boldsymbol{v}_k \sim \mathcal{N}(0, R_k)$ is the zero-mean Gaussian measurement noise with covariance R_k.

4.2 Derivation

The Kalman filter can be derived in many ways. Originally, it was derived using the orthogonal projection method, however, it can also be posed as a least-squares minimization. Since Kalman filter is a linear instance of the Recursive Bayesian filter, it is also possible to derive the prediction and correction step directly from the Bayesian filter. Since both the state transition and measurement functions are linear, both $p(\boldsymbol{x}_k|\boldsymbol{x}_{k-1}, \boldsymbol{u_k})$ and $p(\boldsymbol{z}_k|\boldsymbol{x}_k)$ are normally distributed:

$$p(\boldsymbol{x}_k|\boldsymbol{x}_{k-1}, \boldsymbol{u}_k) = \mathcal{N}(\boldsymbol{x}_k; F_k \boldsymbol{x}_{k-1} + B_k \boldsymbol{u}_k, Q_k) \tag{4.5}$$
$$p(\boldsymbol{z}_k|\boldsymbol{x}_k) = \mathcal{N}(\boldsymbol{x}_k; H_k \boldsymbol{x}_k, R_k). \tag{4.6}$$

Suppose that the posterior at $k-1$ is normally distributed according to

$$p(\boldsymbol{x}_{k-1}|\boldsymbol{z}_{1:k-1}, \boldsymbol{u}_{1:k-1}) = \mathcal{N}(\boldsymbol{x}_{k-1}; \hat{\boldsymbol{x}}_{k-1|k-1}, P_{k-1|k-1}). \tag{4.7}$$

As we shall see later, the posterior distribution of the Kalman filter is indeed Gaussian. Now, substituting these into the prediction step of the Recursive Bayesian filter (3.7), we obtain

$$p(\boldsymbol{x}_k|\boldsymbol{z}_{1:k-1}, \boldsymbol{u}_{1:k}) = \int p(\boldsymbol{x}_k|\boldsymbol{x}_{k-1}, \boldsymbol{u}_k)p(\boldsymbol{x}_{k-1}|\boldsymbol{z}_{1:k-1}, \boldsymbol{u}_{1:k-1})d\boldsymbol{x}_{k-1} \tag{4.8}$$

$$= \int \mathcal{N}(\boldsymbol{x}_k; F_k\boldsymbol{x}_{k-1} + B_k\boldsymbol{u}_k, Q_k)\mathcal{N}(\boldsymbol{x}_{k-1}; \hat{\boldsymbol{x}}_{k-1|k-1}, P_{k-1|k-1})d\boldsymbol{x}_{k-1}. \tag{4.9}$$

The above integral is a product of two Gaussians which has a closed-form solution. The integral evaluates to a Gaussian with mean $\hat{\boldsymbol{x}}_{k|k-1}$ and covariance $P_{k|k-1}$. As a Gaussian distribution is fully described by its first two moments, this gives us an explicit prediction step of the Kalman filter:

$$\hat{\boldsymbol{x}}_{k|k-1} = F_k\hat{\boldsymbol{x}}_{k-1|k-1} + B_k\boldsymbol{u}_k \qquad \text{predicted estimate} \tag{4.10}$$

$$P_{k|k-1} = F_kP_{k-1|k-1}F_k^T + Q_k \qquad \text{estimate covariance} \tag{4.11}$$

The update step is derived analogously. We again show only the first steps for brevity, however, the complete derivation can be found in [58].

$$p(\boldsymbol{x}_k|\boldsymbol{z}_{1:k}\boldsymbol{u}_{1:k}) = \frac{p(\boldsymbol{z}_k|\boldsymbol{x}_k, \boldsymbol{u}_{1:k})p(\boldsymbol{x}_k|\boldsymbol{z}_{1:k-1}, \boldsymbol{u}_{1:k-1})}{p(\boldsymbol{z}_k|\boldsymbol{z}_{1:k-1})} \tag{4.12}$$

$$= \frac{p(\boldsymbol{z}_k|\boldsymbol{x}_k)p(\boldsymbol{x}_k|\boldsymbol{z}_{1:k-1})}{\int p(\boldsymbol{z}_k|\boldsymbol{x}_k)p(\boldsymbol{x}_k|\boldsymbol{z}_{1:k-1})d\boldsymbol{x}_k} \tag{4.13}$$

$$= \frac{\mathcal{N}(\boldsymbol{z}_k; H_k\boldsymbol{x}_k, R_k)\ \mathcal{N}(\boldsymbol{x}_k; \hat{\boldsymbol{x}}_{k|k-1}, P_{k|k-1})}{\int \mathcal{N}(\boldsymbol{z}_k; H_k\boldsymbol{x}_k), R_k)\ \mathcal{N}(\boldsymbol{x}_k; \hat{\boldsymbol{x}}_{k|k-1}, P_{k|k-1})\ d\boldsymbol{x}_k} \tag{4.14}$$

The posterior can again be shown to be Gaussian. This means that at every step of the filter, the distribution at hand is Gaussian. As such, to represent the filter, one only needs to store the mean and covariance. Specifically, the correction step is given by the following equations:

$$\boldsymbol{y}_k = \boldsymbol{z}_k - H_k\hat{\boldsymbol{x}}_{k|k-1} \qquad \text{residual} \tag{4.15}$$

$$S_k = H_kP_{k|k-1}H_k^T + R_k \qquad \text{residual covariance} \tag{4.16}$$

$$K_k = P_{k|k-1}H_k^TS_k^{-1} \qquad \text{Kalman gain} \tag{4.17}$$

$$\hat{\boldsymbol{x}}_{k|k} = \hat{\boldsymbol{x}}_{k|k-1} + R_k\boldsymbol{y}_k \qquad \text{corrected estimate} \tag{4.18}$$

$$P_{k|k} = (I - K_kH_k)\,P_{k|k-1} \qquad \text{corected covariance} \tag{4.19}$$

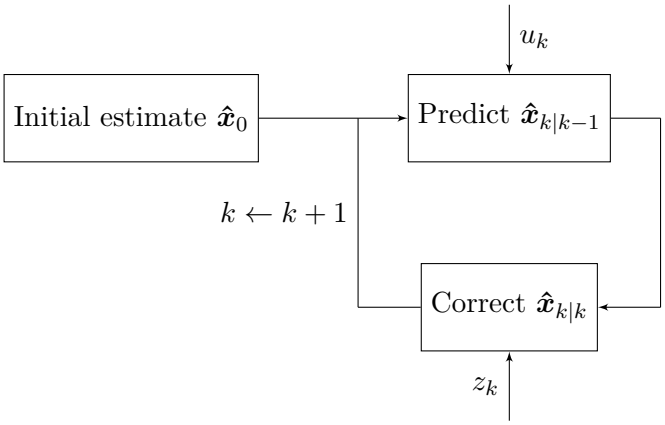


Figure 4.1: A block diagram of the recursive operation of the Kalman filter. The filter starts with an initial estimate $\hat{\boldsymbol{x}}_0$ and P_0. Then, in a loop, the filter first uses the control input to predict the next state of the system, which is subsequently refined using the measurement $\boldsymbol{z}_k$.

Here, $\boldsymbol{y}_k$ is called the innovation or measurement residual. S_k is the residual covariance and K_k is the Kalman gain, a measure of how much the filter trusts the measurement $\boldsymbol{z}_k$ over the predicted state. When the covariance of the estimate is small compared to the measurement covariance, the correction step tends to put more weight on the predicted estimate which manifests by the Kalman gain going to zero. On the other hand, when the estimate covariance is relatively large, the filter will correct closer to the measurement. A rigorous demonstration of this is provided in the Appendix A.2.1. Finally, the filter computes a new estimate covariance, taking into account the uncertainty of the new measurement. In a typical case, the resulting estimate uncertainty is lower than either the measurement or the previous estimate uncertainty. A block diagram of the Kalman filter operation is shown in Fig. 4.1. A single correction step of the Kalman filter is shown graphically in Fig. 4.2.

4.3 Intuition for a 1D case

The Kalman filter equations may seem impenetrable at first glance. To build some intuition for them, it may be helpful to derive the Kalman filter for the one-dimensional case where matrix multiplication becomes a simple scalar multiplication. It is then possible to draw some parallels between this simple case and the general multidimensional case. First, suppose we wish to combine two Gaussian measurements (x_1, σ_1^2) and (x_2, σ_2^2) into a single estimate. If $\sigma_1^2 = \sigma_2^2$, we can simply take the average to get $\hat{x} = \frac{1}{2}(x_1 + x_2)$. If the uncertainties of the measurements differ, we can use the maximum likelihood principle to derive the most likely estimate. Formally, given two i.i.d. measurements x_1 and x_2 with corresponding uncertainties σ_1^2 and σ_1^2, we wish to maximize the likelihood

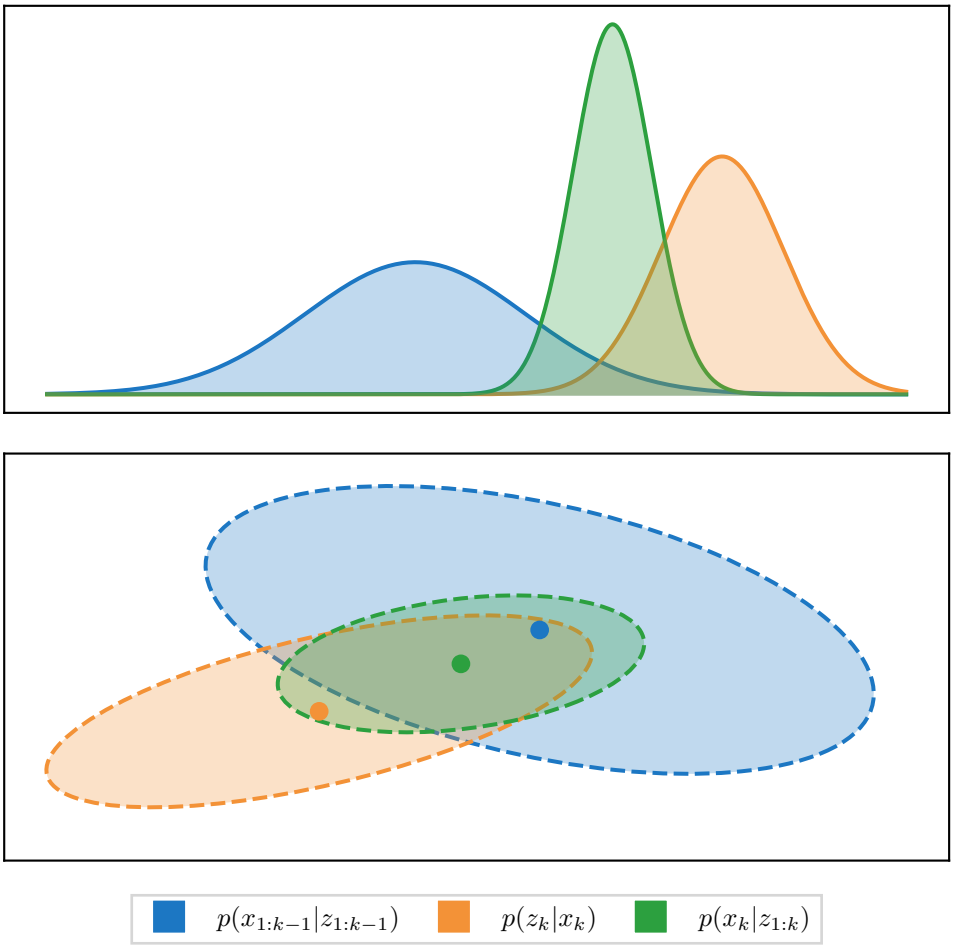


Figure 4.2: An example of a 1D (**top**) and a 2D (**bottom**) Kalman update. The previous belief (**blue**) and a measurement (**orange**) combine into the corrected belief (**green**). Note that the new variance is lower than either of the previous estimate or the measurement.

$$p(x_1, x_2|\hat{x}) = p(x_1|\hat{x})p(x_2|\hat{x}) = \prod_{i=1}^{2} \exp\left(-\frac{(x_i - \hat{x})^2}{\sigma_i^2}\right). \tag{4.20}$$

More generally, for n measurements we maximize:

$$p(x_1, x_2, ..., x_n|\hat{x}) = \prod_{i=1}^{n} p(x_i|\hat{x}) = \prod_{i=1}^{n} \exp\left(-\frac{(x_i - \hat{x})^2}{\sigma_i^2}\right). \tag{4.21}$$

Taking the natural logarithm and computing the derivative to find the maximum, we obtain:

$$\mathcal{L} = \ln\left(\prod_{i=1}^{n} \exp\left(-\frac{(x_i - \hat{x})^2}{\sigma_i^2}\right)\right) = -\sum_{i=1}^{n} \frac{(x_i - \hat{x})^2}{\sigma_i^2} \tag{4.22}$$

$$\frac{\partial \mathcal{L}}{\partial \hat{x}} = \sum_{i=1}^{n} \left(\frac{x_i - \hat{x}}{\sigma_i^2}\right) = 0. \tag{4.23}$$

Solving for two measurements ($n = 2$), we arrive at the following formula

$$\hat{x} = \frac{\sigma_2^2}{\sigma_1^2 + \sigma_2^2} x_1 + \frac{\sigma_1^2}{\sigma_1^2 + \sigma_2^2} x_2. \tag{4.24}$$

Since this is a linear combination of two Gaussians, the variance of $\hat{x}$ is given as

$$\hat{\sigma}^2 = Var[\hat{x}] = Var\left[\frac{\sigma_2^2}{\sigma_1^2 + \sigma_2^2} x_1\right] + Var\left[\frac{\sigma_1^2}{\sigma_1^2 + \sigma_2^2} x_2\right] \tag{4.25}$$

$$= \frac{\sigma_2^4 \sigma_1^2}{(\sigma_1^2 + \sigma_2^2)^2} + \frac{\sigma_1^4 \sigma_2^2}{(\sigma_1^2 + \sigma_2^2)^2} \tag{4.26}$$

$$= \frac{\sigma_1^2 \sigma_2^2}{\sigma_1^2 + \sigma_2^2}. \tag{4.27}$$

Let us now consider a very simple one-dimensional dynamic system given by the following process and measurement models:

$$x_k = x_{k-1} + c + w_k \tag{4.28}$$

$$z_k = x_k + v_k. \tag{4.29}$$

Here, c is a constant and w_k and v_k are the process and measurement noise, respectively. If we rearrange the formulas for $\hat{x}$ and $\hat{\sigma}^2$ and apply them to this model, we get a result that is starting to resemble the Kalman filter equations.

$$\hat{x} = \left(\frac{\sigma_2^2}{\sigma_1^2 + \sigma_2^2}\right) x_1 + \left(\frac{\sigma_1^2}{\sigma_1^2 + \sigma_2^2}\right) x_2 = x_1 + \frac{\sigma_1^2 (x_2 - x_1)}{\sigma_1^2 + \sigma_2^2} \tag{4.30}$$

$$\hat{\sigma} = \frac{\sigma_1^2 \sigma_2^2}{\sigma_1^2 + \sigma_2^2} = \sigma_1^2 - \frac{\sigma_1^4}{\sigma_1^2 + \sigma_2^2} \tag{4.31}$$

If we think of the values (x_1, σ_1^2) and (x_2, σ_2^2) respectively as the prior belief and the current measurement, we can notice similarities with the general Kalman filter equations. Setting $y = x_2 - x_1$ and $k = \sigma_1^2(\sigma_1^2 + \sigma_2^2)^{-1}$, we can see that the one-dimensional filter equations are almost identical to the general case:

General KF	1D KF	
$\hat{\boldsymbol{x}}_{k\|k-1} = F_k\hat{\boldsymbol{x}}_{k-1\|k-1} + B_k\boldsymbol{u}_k$	$\hat{x}_{k\|k-1} = x_{k-1\|k-1} + c$	(4.33)
$P_{k\|k-1} = F_k P_{k-1\|k-1} F_k^T + Q_k$	$\hat{\sigma}_{k\|k-1} = \hat{\sigma}_{k-1\|k-1} + q_k^2$	(4.34)
$\boldsymbol{y}_k = \boldsymbol{z}_k - H_k\hat{\boldsymbol{x}}_{k\|k-1}$	$y_k = z_k - \hat{x}_{k\|k-1}$	(4.35)
$S_k = H_k P_{k\|k-1} H_k^T + R_k$	$s_k^2 = \hat{\sigma}_{k\|k-1} + r_k^2$	(4.36)
$K_k = P_{k\|k-1} H_k^T S_k^{-1}$	$k_k = \hat{\sigma}_{k\|k-1}(s_k^2)^{-1}$	(4.37)
$\hat{\boldsymbol{x}}_{k\|k} = \hat{\boldsymbol{x}}_{k\|k-1} + K_k\boldsymbol{y}_k$	$\hat{x}_{k\|k} = \hat{x}_{k\|k-1} + K_k y_k$	(4.38)
$P_{k\|k} = (I - K_k H_k)\, P_{k\|k-1}$	$\hat{\sigma}_{k\|k} = (1 - k_k)\hat{\sigma}_{k\|k-1}$	(4.39)

4.4 Asymptotic behavior and filter divergence

The Kalman filter provides an ubiased estimate of the system state since $\mathbb{E}[\mathbb{E}[\boldsymbol{x}_k|\boldsymbol{z}_k]] = x_k$ by the law of total expectation. The convergence of the estimate itself is conditioned on the process and measurement models. If they are both noisy, the estimate covariance will not converge to zero. Moreover, under certain conditions, the Kalman filter can diverge, i.e., the estimation error may grow unbounded. The most common reason for a filter divergence is an incorrect system model. If either the process or measurement models do not accurately reflect the system dynamics, the filter may diverge. A thorough investigation of the divergence conditions is given by [25, 94].

The asymptotic behaviour of the Kalman filter can be studied using the the observability and reachability properties of dynamical systems.

Definition 4.1. A dynamical system given by

$$\boldsymbol{x}_k = F\boldsymbol{x}_{k-1} + \boldsymbol{v}_k \tag{4.40}$$
$$\boldsymbol{z}_k = H\boldsymbol{x}_k + \boldsymbol{w}_k, \tag{4.41}$$

is said to be **observable** if $\boldsymbol{x}_1$ can be unambiguously determined from $\{\boldsymbol{z}_1, \boldsymbol{z}_2, ..., \boldsymbol{z}_n, \}$.

Definition 4.2. A dynamical system given by

$$\boldsymbol{x}_k = F\boldsymbol{x}_{k-1} + B\boldsymbol{u}_k + \boldsymbol{v}_k \tag{4.42}$$
$$\boldsymbol{z}_k = H\boldsymbol{x}_k + \boldsymbol{w}_k, \tag{4.43}$$

is said to be **reachable** if given a starting state $\boldsymbol{x}_1 = \mathbf{0}$ and an end state $\boldsymbol{x}_n$, there is a sequence of control inputs $\{\boldsymbol{u}_1, \boldsymbol{u}_2, ..., \boldsymbol{u}_n, \}$ such that $\boldsymbol{x}_n$ can be reached from $\boldsymbol{x}_1$.

Observability and reachability can be tested by constructing special matrices and verifying they have full rank. More details can be found in [98]. Assuming the system in question is observable, the error covariance P is bounded. Moreover, if the system is also reachable, the error covariance is guaranteed to converge and can be computed in a closed form from the following system of equations:

$$M = FPF^T + Q \tag{4.44}$$
$$K = PH^T(HMH^T + R)^{-1} \tag{4.45}$$
$$P = (I - KH)\,M \tag{4.46}$$

The speed of convergence is typically very fast as is shown in figure (4.3).

4.5 Extended Kalman filter

Unfortunately, most systems cannot be accurately modeled by a set of linear equations. When either the process model or the measurement model is nonlinear or both, the Kalman filter cannot be used. However, the Kalman filter can be modified to work with these nonlinearities. This modification is the so-called Extended Kalman filter (EKF) [59], which works by linearizing the process and measurement models by means of the first-order Taylor approximation. The resulting filter can no longer guarantee optimality. Nevertheless, it typically performs well, provided the model nonlinearities are not too severe [71, 51, 89].

The original Kalman filter formulation requires only minor changes to adapt it to the nonlinear case. Below, we show the formulations of both KF and EKF side-by-side for easy comparison.

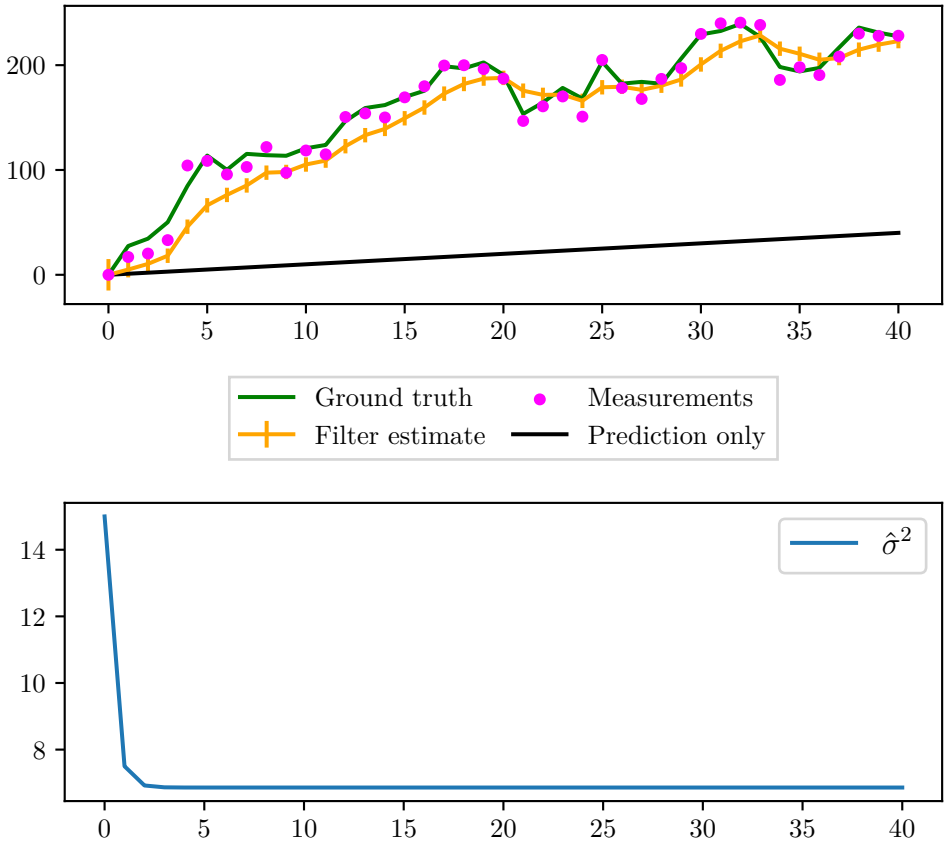


Figure 4.3: An example of a one-dimensional Kalman filter estimating a system state. The system model is given by $x_k = x_{k-1} + c + w_k$ and $z_k = x_k + v_k$. The process noise is large enough for x_k to deviate significantly from the prediction (**black**). Using the measurements (**pink**), the filter can estimate the real value of x_k reasonably well. The second plot shows the uncertainty of $\hat{x}_k$. Additionally, we can observe that the uncertainty quickly converges. Indeedn, the estimated uncertainty is within 10^{-5} of the theoretical value after 40 iterations.

Kalman filter	Extended Kalman filter	
$\hat{\boldsymbol{x}}_{k\|k-1} = F_k\hat{\boldsymbol{x}}_{k-1\|k-1} + B_k\boldsymbol{u}_k$	$\hat{\boldsymbol{x}}_{k\|k-1} = f_k\left(\hat{\boldsymbol{x}}_{k-1\|k-1}, \boldsymbol{u}_k\right)$	(4.47)
$\hat{R}_{k\|k-1} = F_k P_{k-1\|k-1} F_k^T + Q_k$		(4.48)
$\boldsymbol{y}_k = \boldsymbol{z}_k - H_k\hat{\boldsymbol{x}}_{k\|k-1}$	$\boldsymbol{y}_k = \boldsymbol{z}_k - h_k\left(\hat{\boldsymbol{x}}_{k\|k-1}\right)$	(4.49)
$R_k = H_k P_{k\|k-1} F_k^T + R_k$		(4.50)
$R_k = P_{k\|k-1} H_k^T R_k^{-1}$		(4.51)
$\hat{\boldsymbol{x}}_{k\|k} = \hat{\boldsymbol{x}}_{k\|k-1} + R_k\boldsymbol{y}_k$		(4.52)
$P_{k\|k} = (R - R_k H_k)\, P_{k\|k-1}$		(4.53)

In the case of EKF, F_k and H_k are replaced by the Jacobians of the process and measurement models.

$$F_k = \left.\frac{\partial f}{\partial \boldsymbol{x}}\right|_{\hat{\boldsymbol{x}}_{k-1|k-1}, \boldsymbol{u}_k} \tag{4.54}$$

$$H_k = \left.\frac{\partial h}{\partial \boldsymbol{x}}\right|_{\hat{\boldsymbol{x}}_{k|k-1}} \tag{4.55}$$

The Jacobians are evaluated at the previous state estimate and the predicted state, respectively. Note that the only major difference between KF and EKF is the need to compute the Jacobians of the process and measurement models. The prediction and correction equations remain unchanged.

Chapter 5

Particle filter

A limitation of the Kalman filter is the requirement for a parametric posterior given by a multivariate Gaussian. This means that the Kalman filter fundamentally cannot represent non-Gaussian, possibly multimodal distributions. Multimodal posteriors arise in situations where the motion and measurement models induce ambiguity due to the inherent noise. An example from robotics is the problem of localization in a known environment. In that case, ambiguity can manifest if multiple locations in the environment have similar features and thus are difficult to distinguish. Since multiple hypotheses are possible, this situation cannot be adequately modeled by a Gaussian.

Particle filters are a nonparametric method and thus do not place any constraints on the posterior as the Kalman filter does. As a consequence, particle filters can represent arbitrary distributions. In addition, they can handle nonlinear and non-Gaussian motion and measurement models. Together with Kalman filters, Partical filters belong to the family of Recursive Bayes filters. The fundamental idea which makes particle filters different, is the way they represent the posterior distribution. The posterior is given as a sum of a finite number of weighted samples called particles. Each particle represents one possible state of the system. The more particles are used, the more closely the filter is able to approximate the true posterior density.

Particle filters have been successfully used in practise to solve many estimation and tracking problems [97, 33, 32, 29]. One notable example is Monte Carlo localization (MCL) [20]. In MCL, a robot is placed in an unknown location in an otherwise known environment. The goal of the robot is to move through the environment and localize itself by observing features in the environment. The ability to represent multiple hypotheses together with nonlinear motion makes particle filters comparatively robust. However, the number of particles needed to accurately model the full posterior grows exponentially with the dimension of the state in the worst case. This restricts

particle filters to low-dimensional problems such as the aforementioned Monte Carlo localization.

5.1 Representing the posterior

Formally, particle filters approximate the posterior by a weighted sum of Dirac impulses:

$$p(\boldsymbol{x}_k|\boldsymbol{z}_{1:k}) \approx \sum_{i=1}^{N} w_k^i \delta(\boldsymbol{x}_k - \boldsymbol{x}_k^i), \tag{5.1}$$

where N is the number of particles, w_k^i is the weight of particle $\boldsymbol{x}^i$, also called an importance factor, $\{\boldsymbol{x}_k^1, ..., \boldsymbol{x}_k^N\}$ are the particles and $\delta(.)$ is the Dirac Delta distribution. To make the approximation an actual probability density, we require that the particle weights are normalized:

$$\sum_{i=1}^{N} w_k^i = 1. \tag{5.2}$$

Informally, the particle weights correspond to how much a given particle differs from the true state. If the weight is small, the particle is far away from the true state and vice versa. Intuitively, if we use enough particles, the approximated density should in principle get arbitrarily close to the true posterior density we are interested in. However, since the number of particles is necessarily finite, particle filters are not optimal state estimators but merely an approximation. Nevertheless, convergence to the true posterior can be shown in certain cases [18].

5.2 Importance sampling

In an ideal situation, the particles would be sampled directly from the posterior density. This is in general impossible as most of the time, the posterior does not have a closed form or is difficult or impossible to sample from. We instead use a technique called importance sampling that allows us to sample from a different, typically much simpler *proposal* distribution. The particle weights are then adjusted to correct these samples. Suppose we want to compute a statistic of the form $\mathbb{E}_p[f(x)]$ of the distribution $p(x)$. Assume that $p(x)$ is

difficult or impossible to sample from. Luckily, with importance sampling, we can sample from a different distribution by using a simple algebraic trick:

$$\mathbb{E}_p\left[f(x)\right] \overset{def}{=} \int f(x)p(x)dx \tag{5.3}$$

$$= \int \frac{f(x)p(x)}{q(x)}q(x)dx \tag{5.4}$$

$$\overset{def}{=} \mathbb{E}_q\left[\frac{f(x)p(x)}{q(x)}\right]. \tag{5.5}$$

Here, $q(x)$ is called the proposal density. This density can be arbitrary, subject only to $p(x) > 0 \Rightarrow q(x) > 0$. That is, the support of p is contained in q. The above result states that to compute an estimate of $\mathbb{E}_p[f(x)]$, we can instead sample from a different proposal distribution and compute the estimate using this weighted sample. For example, we can estimate the expectation $\mathbb{E}_p[x]$ as follows:

$$\hat{\mu} = \frac{1}{n}\sum_{i=1}^{n}\frac{x^i p(x^i)}{q(x^i)} = \frac{1}{n}\sum_{i=1}^{n} w^i x^i. \tag{5.6}$$

Returning back to filtering, the posterior $p(\boldsymbol{x}_k|\boldsymbol{z}_{1:k})$ can therefore be approximated by sampling particles from a suitable proposal distribution $q(\boldsymbol{x}_k|\boldsymbol{z}_{1:k})$ and adjusting the weights accordingly:

$$w_k^i \propto \frac{p(\boldsymbol{x}_k^i|\boldsymbol{z}_{1:k})}{q(\boldsymbol{x}_k^i|\boldsymbol{z}_{1:k})}. \tag{5.7}$$

5.3 Sequential Importance Sampling

Sequential importance sampling (SIS) is the simplest example of a particle filter. SIS takes advantage of the fact that under certain assumptions, sampling particles and calculating particle weights can be done recursively by using the particles and weights from the previous time step. To derive the SIS algorithm, we need to consider the full posterior $p(\boldsymbol{x}_{1:k}|\boldsymbol{z}_{1:k})$ as opposed to the marginal posterior $p(\boldsymbol{x}_k|\boldsymbol{z}_{1:k})$ used so far. That is, $\boldsymbol{x}_{1:k}$ contains the full history of the states from time step 1 to k. The approximate posterior density changes slightly:

$$p(\boldsymbol{x}_{1:k}|\boldsymbol{z}_{1:k}) \approx \sum_{i=1}^{N} w^i \delta(x_{1:k} - x^i_{1:k}). \tag{5.8}$$

Here, the particles contain the full history $\boldsymbol{x}^i_{1:k} = \{\boldsymbol{x}^i_1, ..., \boldsymbol{x}^i_N\}$. Each particle can thus be thought of as a separate path or trajectory through the state space. The weight is then computed analogously as

$$w^i_k \propto \frac{p(\boldsymbol{x}_{1:k}|\boldsymbol{z}_{1:k})}{q(\boldsymbol{x}_{1:k}|\boldsymbol{z}_{1:k})}. \tag{5.9}$$

We now show how the above weight update is computed recursively using weights w_{k-1}. First, we can expand the posterior into a product of the measurement and process model and the posterior at a previous time step:

$$p(\boldsymbol{x}_{1:k}|\boldsymbol{z}_{1:k}) \overset{Bayes}{=} \frac{p(\boldsymbol{z}_k|\boldsymbol{x}_{1:k}, \boldsymbol{z}_{1:k-1})p(\boldsymbol{x}_{1:k}|\boldsymbol{z}_{1:k-1})}{p(\boldsymbol{z}_k|\boldsymbol{z}_{1:k-1})} \tag{5.10}$$

$$= \frac{p(\boldsymbol{z}_k|\boldsymbol{x}_{1:k}, \boldsymbol{z}_{1:k-1})p(\boldsymbol{x}_k|\boldsymbol{x}_{1:k-1}, \boldsymbol{z}_{1:k-1})p(\boldsymbol{x}_{1:k-1}|\boldsymbol{z}_{1:k-1})}{p(\boldsymbol{z}_k|\boldsymbol{z}_{1:k-1})} \tag{5.11}$$

$$\overset{Markov}{=} \frac{p(\boldsymbol{z}_k|\boldsymbol{x}_k)p(\boldsymbol{x}_k|\boldsymbol{x}_{k-1})p(\boldsymbol{x}_{1:k-1}|\boldsymbol{z}_{1:k-1})}{p(\boldsymbol{z}_k|\boldsymbol{z}_{1:k-1})} \tag{5.12}$$

$$\propto p(\boldsymbol{z}_k|\boldsymbol{x}_k)p(\boldsymbol{x}_k|\boldsymbol{x}_{k-1})p(\boldsymbol{x}_{1:k-1}|\boldsymbol{z}_{1:k-1}) \tag{5.13}$$

Note that we can obtain the standard recursive formula of the marginal posterior $p(\boldsymbol{x}_k|\boldsymbol{z}_{1:k})$ from the above equation (5.12) by marginalizing out $\boldsymbol{x}_{1:k-1}$ (see Appendix A.3.3 for more details). For the next step, we assume that the proposal distribution factorizes in a natural way:

$$q(\boldsymbol{x}_{1:k}|\boldsymbol{z}_{1:k}) = q(\boldsymbol{x}_k|\boldsymbol{x}_{1:k-1}, \boldsymbol{z}_{1:k})q(\boldsymbol{x}_{1:k-1}|\boldsymbol{z}_{1:k-1}). \tag{5.14}$$

Substituting (5.13) and (5.14) into (5.9), we obtain

$$w_k^i \propto \frac{p(\boldsymbol{x}_{1:k}|\boldsymbol{z}_{1:k})}{q(\boldsymbol{x}_{1:k}|\boldsymbol{z}_{1:k})} \tag{5.15}$$

$$= \frac{p(\boldsymbol{z}_k|\boldsymbol{x}_k^i)p(\boldsymbol{x}_k^i|\boldsymbol{x}_{k-1}^i)}{q(\boldsymbol{x}_k^i|\boldsymbol{x}_{1:k-1}^i, \boldsymbol{z}_{1:k})} \underbrace{\frac{p(\boldsymbol{x}_{1:k-1}^i|\boldsymbol{z}_{1:k-1})}{q(\boldsymbol{x}_{1:k-1}^i|\boldsymbol{z}_{1:k-1})}}_{w_{k-1}^i} \tag{5.16}$$

$$= w_{k-1}^i \frac{p(\boldsymbol{z}_k|\boldsymbol{x}_k^i)p(\boldsymbol{x}_k^i|\boldsymbol{x}_{k-1}^i)}{q(\boldsymbol{x}_k^i|\boldsymbol{x}_{1:k-1}^i, \boldsymbol{z}_{1:k})}. \tag{5.17}$$

If we further assume that the proposal depends only on $\boldsymbol{x}_{k-1}$ and $\boldsymbol{z}_k$, that is,

$$q(\boldsymbol{x}_k|\boldsymbol{x}_{1:k-1}, \boldsymbol{z}_{1:k}) = q(\boldsymbol{x}_k|\boldsymbol{x}_{k-1}, \boldsymbol{z}_k), \tag{5.18}$$

the weight correction becomes

$$w_k^i \propto w_{k-1}^i \frac{p(\boldsymbol{z}_k|\boldsymbol{x}_k^i)p(\boldsymbol{x}_k^i|\boldsymbol{x}_{k-1}^i)}{q(\boldsymbol{x}_k^i|\boldsymbol{x}_{k-1}^i, \boldsymbol{z}_k)}. \tag{5.19}$$

Note that in the last equation, the conditioning on the full path $\boldsymbol{x}_{1:k}$ disappears. This means that the particles only need to store the latest state estimate as opposed to the whole path history. Moreover, only the last measurement $\boldsymbol{z}_k$ need to be stored. As a consequence of this fact, the memory required to store the particles is bounded and independent of the number of time steps. The SIS filter uses the above recursive formula as shown in Algorithm 1. In every iteration of the SIS filter, particles are drawn from the proposal distribution and their weights are corrected using (5.19).

Unfortunately, the SIS filter is susceptible to *sample degeneracy* [23, 4]. Sample degeneracy is a phenomenon caused by the fact that the variance of the unnormalized weights always increases regardless of the choice of the proposal distribution [23]. This means that at a certain point, all but one particle will have an almost zero weight thus having a negligible contribution to the posterior density.

5.4 Sampling importance resampling filter

The Sampling importance resampling (SIR) [45] filter attempts to solve the problem of sample degeneracy by adding a resampling step to the SIS

Algorithm 1: SIS Filter

for $k \leftarrow 1$ **to** K **do**
    Draw $\boldsymbol{x}_k^i \sim q(\boldsymbol{x}_k|\boldsymbol{x}_{k-1}^i, \boldsymbol{z}_k)$
    Compute particle weights w_k^i according to (5.19)
end

algorithm. During the resampling step, a new unweighted posterior is created from the old one in order to improve the quality of the overall particle set. The goal is to remove particles with very low weights whose contribution to the distribution is miniscule. Resampling is typically done by drawing particles with replacement from the old set with a probability of being selected proportional to the weight of the particle. In this way, particles with larger weights have a higher chance of being selected multiple times. On the other hand, particles with a very low weight are likely to die out. This step is reminiscent of genetic algorithms [100] where a similar scheme ensures the so-called *survival of the fittest*. Fig. 5.1 depicts the resampling step. The approximate posterior produced by the SIR filter has been shown to converge to the true posterior almost surely [18]. There are many popular resampling strategies which have been the subject of many studies [66, 67, 69, 28, 35, 21, 38, 91]. In the following sections, we will introduce a few popular techniques described in the literature.

Algorithm 2: SIR filter

for $k \leftarrow 1$ **to** K **do**
    Draw $\boldsymbol{x}_k^i \sim q(\boldsymbol{x}_k|\boldsymbol{x}_{k-1}^i, \boldsymbol{z}_k)$
    Compute particle weights w_k^i according to (5.19)
    Resample particles based on 5.5
end

5.5 Resampling strategies

The resampling step aims to reduce the variance of the weights and thus ensure that the majority of the particles contribute meaningfully to the posterior approximation. In practical terms, this is done by sampling with replacement from the old particle set. The sampling is carried out in proportion to the particle weight so that particles with higher weight are selected on average more often than particles with a lower weight. We can state this condition precisely, but in order to do that, we need to define a few terms:

Definition 5.1. An *offspring* vector denoted as $\{o_1, ..., o_N\}$ given particles $\{\boldsymbol{x}_k^1, ..., \boldsymbol{x}_k^N\}$ is a vector where o_i defines how many times a particle $\boldsymbol{x}_k^i$ will be replicated in the resampled particle set.

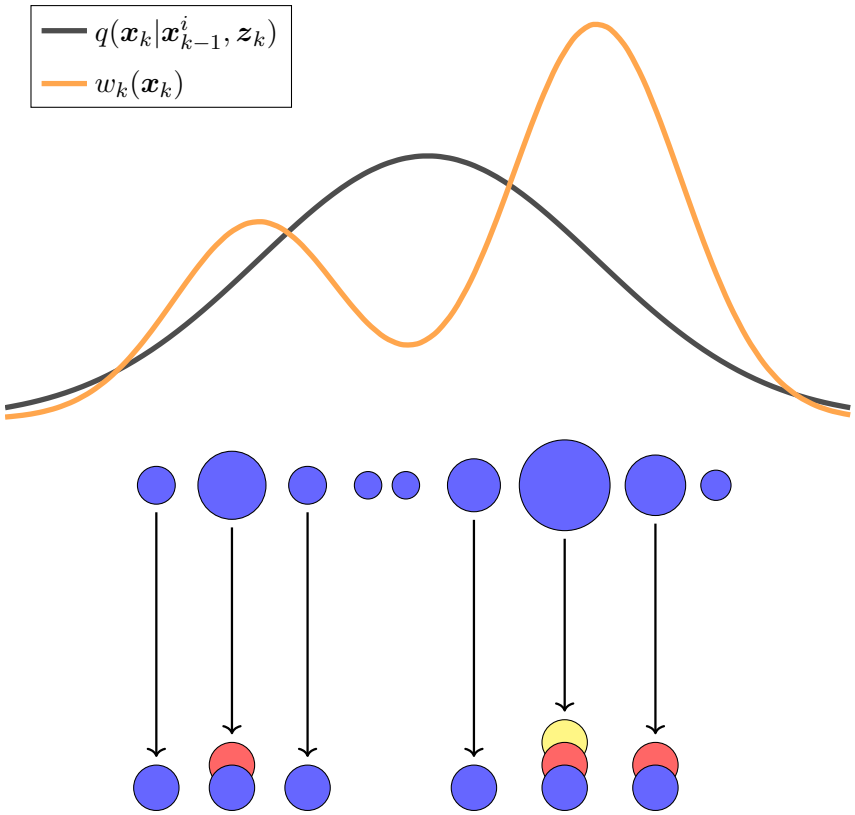


Figure 5.1: A visual representation of resampling in particle filters. The black curve represents the proposal distribution while the orange curve shows the corrected weights. The particles are depicted as circles whose size depends on the particle weight. After resampling, the weights are reset. Notice also that particles with higher weight are sampled multiple times.

Definition 5.2. An *ancestor* vector denoted as $\{a_1, ..., a_N\}$ given particles $\{\boldsymbol{x}_k^1, ..., \boldsymbol{x}_k^N\}$ is a vector where a_i defines which particle will appear at index i in the resampled particle set. In other words, x^{a_i} is the ancestor of the particle at index i.

Both the offspring and ancestor vectors encode the output of a resampling procedure. These two representations are equivalent and one can easily convert between them as shown in Algorithm 4 and 3. Sometimes it is easier to work with one representation over the other. With these definitions, we can formalize what we mean by sampling in proportion to the weights. The most common way to define the proportionality is as follows [67, 21]:

$$\mathbb{E}(o_i|\{w^1, ..., w^N\}) = \frac{Nw^i}{\sum_{j=1}^N w^j}, \tag{5.20}$$

which, assuming normalized weights, simplifies to

$$\mathbb{E}(o_i|\{w^1, ..., w^N\}) = Nw^i. \tag{5.21}$$

Here, N is again the number of particles, w^i is the particle weight and and o_i is an element of the offspring vector. In simple terms, this formula states that in expectation, the replication count for a given particle should be proportional to its weight. This is sometimes called the unbiassedness condition [67]. There exists a large number of both sequential and parallel resampling algorithms which satsify this condition. The most common sequential ones include multinomial, stratified, and systematic sampling. Common parallel algorithms include Metropolis-Hastings and Acceptatnce-Rejection sampling [67]. To keep this chapter focused on the theory of particle filters, a detailed description of these algorithms is given in the Appendix A.3.1.

Algorithm 3: Offspring-To-Ancestor

input : An offspring vector $\{o_1, ..., o_N\}$
output : An ancestor vector $\{a_1, ..., a_N\}$

```
1
2 k = 1
3 for i ← 1 to N do
4     for j ← 1 to o_i do
5         a_k = i
6         k = k + 1
7     end
8 end
```

Algorithm 4: Ancestor-To-Offspring

input : An ancestor vector $\{a_1, ..., a_N\}$
output : An offspring vector $\{o_1, ..., o_N\}$

```
1
2 for i ← 1 to N do
3     j = a_i
4     o_j = o_j + 1
5 end
```

5.6 Effective sample size

Care must be taken when deciding how often the resampling step of a particle filter should be carried out. One might assume that resampling after every iteration would be the most optimal, however, that is usually not the case. First, there is no need to resample when no new measurement is taken as no new information is gained and the particle weights remain

unchanged. Second, depending on the implementation, resampling can be a costly operation relative to the rest of the filter. Hence, less frequent resampling is often preferred for performance reasons. In addition, resampling too often can lead to *sample impoverishment* when the process noise is low [55]. An impoverished particle set contains very little diversity in its state estimates. In other words, the posterior distribution is very peaked. Such an impoverished sample fails to properly capture the shape of the posterior density. On the other hand, not resampling often enough may introduce *Sample degeneracy* instead [55]. It is therefore important to carefully balance the effects of sample impoverishment and sample degeneracy by resampling only when needed. A common way to decide whether to resample is to measure the the effective sample size metric[47, 56, 23] N_{eff} defined as

$$N_{eff} = \frac{1}{\sum_{i=1}^{N} (w^i)^2}. \tag{5.22}$$

We again assume normalized weights. When N_{eff} of the particles drops below a chosen threshold, the particle set is resampled. The choice of the threshold is typically a fraction of N, the size of the particle set. A Common value of the threshold is $0.5N$, meaning resampling is only carried out when $N_{eff} < 0.5N$. Nevertheless, the exact value of the threshold should be tuned and validated for the model at hand as the model characteristics will have an impact on the behavior of the particles. We can incorporate N_{eff} into the SIR filter to obtain the most common variant of the particle filter. The pseudocode is shown in Algorithm 5.

Algorithm 5: General particle filter

```
1
2 for k ← 1 to K do
3     Draw x_k^i ~ q(x_k | x_{k-1}^i, z_k)
4     Compute particle weights w_k^i according to (5.19)
5     Compute N_eff
6     Resample particles if N_eff below a threshold
7 end
```

5.7 Choice of proposal distribution

As we have discussed previously, particle filters use a proposal density to sample particles. This because the true density is often difficult or even impossible to sample from. In order for the filter to work recursively, we imposed some conditions on this proposal distribution. We required that it factorizes nicely and that it has the Markov property. Any distribution

satsifyinn these conditions can in theory be used. However, some distributions are more suitable than others. A less informed distribution will produce most samples in low-density regions leading to very low particle weights. This means that a lot of computational effort is devoted to particles that are likely to be resampled away. On the other hand, a more informed distribution reduces the effect of sample degeneracy and can thus achieve the same level of accuracy with significantly fewer particles.

A popular choice of the proposal distribution is the process model given by $p(\boldsymbol{x}_k|\boldsymbol{x}_{k-1})$. The process model is a good choice for two reasons. First, it is usually pretty well understood and easy to sample from. Second, in the absence of measurements, the process model is the best guess for how the system may evolve. Substituting the process model into (5.19) yields:

$$w_k^i \propto w_{k-1}^i \frac{p(\boldsymbol{z}_k|\boldsymbol{x}_k^i)p(\boldsymbol{x}_k^i|\boldsymbol{x}_{k-1}^i)}{p(\boldsymbol{x}_k^i|\boldsymbol{x}_{k-1}^i)} \tag{5.23}$$

$$= w_{k-1}^i \underbrace{p(\boldsymbol{z}_k|\boldsymbol{x}_k^i)}_{\substack{\text{measurement} \\ \text{model}}} \tag{5.24}$$

A different choice for the proposal is $p(\boldsymbol{x}_k|\boldsymbol{x}_{k-1}^i, \boldsymbol{z}_k)$. As is shown in [23], this density minimizes the variance of the resulting weights conditioned on $\boldsymbol{x}_{1:k-1}$ and $\boldsymbol{z}_{1:k}$. For a detailed proof of this statement, see Appendix A.3.4. The distribution can be expanded as follows:

$$p(\boldsymbol{x}_k|\boldsymbol{x}_{k-1}^i, \boldsymbol{z}_k) = \frac{p(\boldsymbol{x}_k, \boldsymbol{x}_{k-1}^i, \boldsymbol{z}_k)}{p(\boldsymbol{x}_{k-1}^i, \boldsymbol{z}_k)} \tag{5.25}$$

$$= \frac{p(\boldsymbol{z}_k|\boldsymbol{x}_k, \boldsymbol{x}_{k-1}^i)p(\boldsymbol{x}_k|\boldsymbol{x}_{k-1}^i)p(\boldsymbol{x}_{k-1}^i)}{p(\boldsymbol{z}_k|\boldsymbol{x}_{k-1}^i)p(\boldsymbol{x}_{k-1}^i)} \tag{5.26}$$

$$= \frac{p(\boldsymbol{z}_k|\boldsymbol{x}_k, \boldsymbol{x}_{k-1}^i)p(\boldsymbol{x}_k|\boldsymbol{x}_{k-1}^i)}{p(\boldsymbol{z}_k|\boldsymbol{x}_{k-1}^i)} \tag{5.27}$$

$$\overset{Markov}{=} \frac{p(\boldsymbol{z}_k|\boldsymbol{x}_k)p(\boldsymbol{x}_k^i|\boldsymbol{x}_{k-1})}{p(\boldsymbol{z}_k|\boldsymbol{x}_{k-1}^i)} \tag{5.28}$$

Substituting the last expression into (5.19), we obtain:

$$w_k^i \propto w_{k-1}^i \frac{p(\boldsymbol{z}_k|\boldsymbol{x}_k^i)p(\boldsymbol{x}_k^i|\boldsymbol{x}_{k-1}^i)}{\frac{p(\boldsymbol{z}_k|\boldsymbol{x}_k^i)p(\boldsymbol{x}_k^i|\boldsymbol{x}_{k-1}^i)}{p(\boldsymbol{z}_k|\boldsymbol{x}_{k-1}^i)}} \tag{5.29}$$

$$= w_{k-1}^i p(\boldsymbol{z}_k|\boldsymbol{x}_{k-1}^i) \tag{5.30}$$

$$= w_{k-1}^i \int p(\boldsymbol{z}_k|\boldsymbol{x}_k)p(\boldsymbol{x}_k|\boldsymbol{x}_{k-1}^i)\ d\boldsymbol{x}_k \tag{5.31}$$

This distribution, albeit optimal in theory, is difficult to use in practise. The reason is twofold. First, we need to be able to sample from the proposal $p(\boldsymbol{x}_k|\boldsymbol{x}_{k-1}^i, \boldsymbol{z}_k)$ and second, we need to compute the integral in (5.31), neither of which is possible in general. However, as noted by [4], there are a few special cases in which this can be done. One such case is when the the noise is Gaussian and the measurement model is linear. The proposal is then also a Gaussian which is easily sampled. In cases when the proposal cannot be evaluated in a closed form, linearization techniques may be used to obtain a reasonable approximation.

5.8 Comparison with EKF

To demonstrate the ability of a particle filter to represent arbitrary multimodal distributions, we compare it with an Extended Kalman filter on a variant of a standard one-dimensional benchmarking model [4, 28, 45, 14, 30]. The state-space equations are

$$x_k = x_{k-1} + \cos(k) + q_k \tag{5.32}$$

$$z_k = x_k^2 + r_k, \tag{5.33}$$

where q_k and r_k are the process and observation noise, respectively. The noise is modeled as a zero-mean Gaussian with $q_k \sim \mathcal{N}(0, 10)$ and $r_k \sim \mathcal{N}(0, 1)$. This model is of interest as it demonstrates the limitations of the EKF. The combination of a large process noise and an ambiguous observation model induced by x_k^2 produces a multimodal posterior which the EKF fundamentally cannot represent. In such situations, the EKF either tracks the mean or chooses one of the possibly incorrect modes as shown in Fig. 5.2.

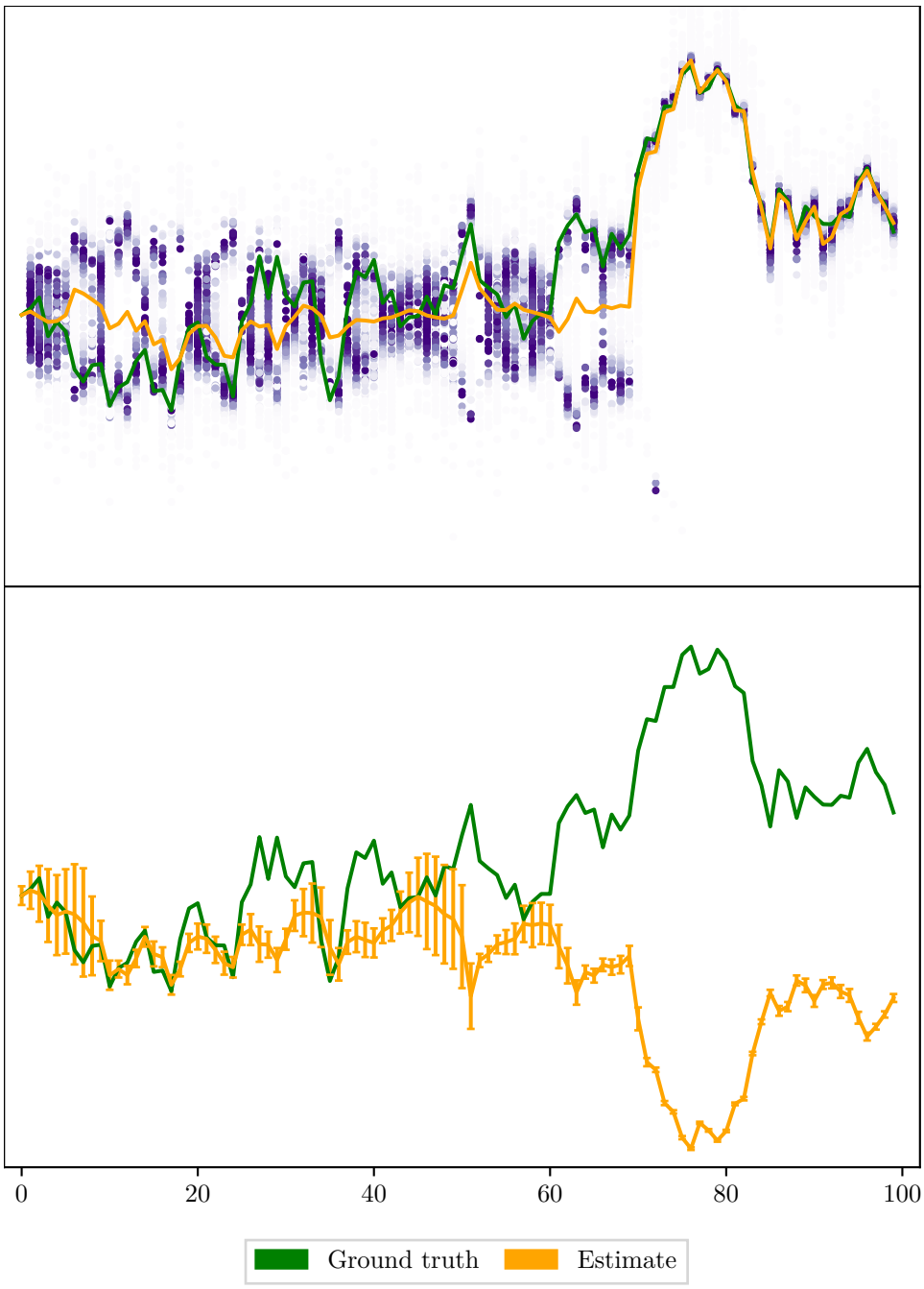


Figure 5.2: This figure shows a comparison of a particle filter **(top)** and an Extended Kalman filter **(bottom)** on a simple non-linear stochastic model given by (5.32). For each timestep, the particles making up the approximate posterior are displayed as dots in the top graph. The color of a particle corresponds to its weight – the lighter the color, the higher the weight of the particle. We can see that the particles form two peaks around the two likely positions of the true state. The EKF on the other hand can only represent a single mode and in this case chooses the wrong one.

Chapter 6

Simultaneous localization and mapping

The Simulatenous localization and mapping problem (SLAM) is one of the most prominent and difficult problems in robotics [24, 6]. The SLAM problem can be described as follows. A mobile robot is placed in an unknown environment of which it has no prior knowledge. The robot can move around and perceive the environment using onboard sensors such as a LiDAR or a camera. The goal of SLAM is for the robot to construct a virtual map of the environment and simultaneously localize itself with respect to the map. During the more than thirty-year history, SLAM went from a mere theoretical formulation to many real applications in the industry. SLAM has been successfully applied in a wide range of environments such as indoor and outdoor, underwater, and airborne. In many settings, The SLAM problem can now be claimed to be solved. However, due to the immense variety of environments and robot-sensor combinations, many such combinations remain challenging, e.g., fast robot dynamics, highly dynamic environments, or strict performance requirements [13].

The SLAM problem combines in itself the problem of localization in a known environment and the problem of mapping with a known path. However, in SLAM neither the map nor the path is known apriori. The robot may measure its odometry to calculate an approximate dead-reckoning pose. Due to the inherent noise present in the motion, the odometry estimate will inevitably drift over time. The difficulty of SLAM lies mostly in the dependence of an accurate pose on the map quality and vice versa. To build an accurate map, the robot needs to know its pose with a high degree of certainty. This is because the robot measurements are relative to its pose. On the other hand, to accurately estimate its pose, the robot needs to have an accurate map.

6.1 SLAM taxonomy

The SLAM problem can be divided into many different types based on the characteristics given by the environment, robot, and sensor configuration [93]. Here, we provide a non-exhaustive list of popular SLAM variants found in the literature:

- Dense vs feature-based SLAM: Dense SLAM uses the raw output of the sensors, e.g., LiDAR pointclouds, to construct a map, thus providing a dense reconstruction of the environment. Feature-based SLAM, also called landmark SLAM, first preproceses the raw sensor data and extracts specific features, also called landmarks, which describe the environment. These landmarks are typically larger and more complex objects in the scene which can be detected, e.g., by means of a neural network. Taking an example from Formula Student, the environment features are the traffic cones delineating the track.

  While computationally more demanding, incorporating the raw sensor input tends to bring increased accuracy as all available data are considered. In Formula Student, the advantage of dense SLAM is less pronounced due to the environment. Formula Student tracks are typically built on airport runways or large industrial car parks. Such environments are very feature-sparse. Typically, the only visible features are the cones themselves.

- Known vs unknown correspondences: Some SLAM formulations assume known correspondences between features observed at different time steps. If the feature identity cannot be discerned from the features themselves, the measurements need to be associated with the most likely correspondences. This problem is called data association. Due to sensor noise and spurious measurements, data association is a difficult problem on its own. Moreover, incorrect data association can significantly degrade the accuracy of SLAM.

- Online vs offline: Online SLAM computes only the most recent pose estimate using the knowledge of the previous estimate. Online SLAM is typically based on a recursive filter without the ability to refine estimates made in the past. Offline SLAM on the other hand, computes the whole robot path. This is more computationally demanding but produces better estimates since the algorithm can consider all available information.

- GPS vs no GPS: In outdoor environments, GPS is typically available to help guide the robot pose. In other environments, such as indoor or underwater, where GPS is not available, the robot can only rely on the sensor readings and the pose estimated by its motion model.

- Static vs dynamic environment: Most SLAM formulations consider the environment to be static, that is, the features are assumed not to move. Formulations that reason about the environment dynamics are more computationally demanding but tend to be more robust.
- Active vs passive: in passive SLAM, the SLAM algorithm cannot control the robot movement. This is useful in situations when a robot needs to carry out other tasks unrelated to SLAM. On the other hand, in active SLAM, the SLAM algorithm has control over the robot motion and can actively choose to search areas that improve its map estimate.
- Single-robot vs multi-robot: SLAM is typically formulated as a single-robot problem, however, a lot of attention has recently been devoted to multirobot cooperative SLAM [73, 81, 99, 54]. In this setting, multiple robots move independently within the same environment. The robots can communicate with each other and exchange information about the environment. In some formulations, communication constraints such as maximum distance may be imposed.

6.2 Loop closure

One important characteristic of SLAM is that the robot pose uncertainty always increases unless the robot revisits a previously seen area. The act of revisiting a known area is called *loop closure*. Loop closing is an important aspect of SLAM which allows the robot to decrease its own pose uncertainty. However, correctly detecting a loop closure is difficult when the robot is returning back to a known area after a large loop as depicted in Fig 6.1. Due to the accumulated error during the traversing of the loop, the robot may not be able to close the loop correctly or close it at all. Loop closure, regardless of its correctness, always decreases the robot uncertainty. As such, once a loop closure is made, it is difficult to revise it later. In addition, failure to close a loop introduces global inconsistencies in the map. Local consistency in SLAM is comparatively simpler as the error in a small time interval grows much slower. We note that loop closure is a concern only for the case of unknown data association. If the data association is known, the robot uses the landmark identities themselves to close the loops instead.

6.3 SLAM formulation

Based on the SLAM classification introduced previously, the rest of this chapter focuses on the case of feature-based passive SLAM with known and unknown correspondence in a static environment. This case is the most

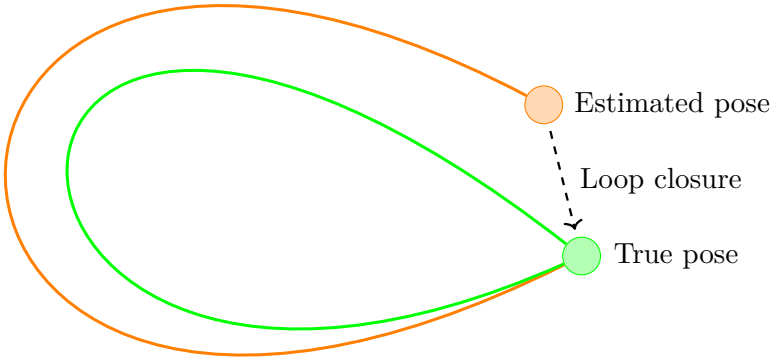


Figure 6.1: This figure shows the problem of loop closure in SLAM. The true robot path is shown in green, while the estimated path is shown in orange. When a robot returns back to a known area after a long period of exploring, its map may be distorted by the cummulative error it acrued along the way. If the error is large enough, the robot may fail to close the loop leading to a global inconsistency in the map.

relevant to the Formula Student competition. In this formulation, we consider a discrete time given by time steps denoted as $0, 1, ..., k$. The robot path is then described as a sequence of individual poses at these time steps. Formally, we have

$$\boldsymbol{x}_{0:k} = \{\boldsymbol{x}_0, ..., \boldsymbol{x}_k\}, \tag{6.1}$$

where $\boldsymbol{x}_{0:k}$ is the robot path from 0 to k and $\boldsymbol{x}_k$ is an individual pose. To further simplify the formulation, we assume without loss of generality that the robot moves in a two-dimensional world. The robot pose is then characterized by a 3-dimensional vector (x, y, θ) giving the position and orientation in the plane. Furthermore, the robot has access to its odometry data. Odometry provides information about the relative change between consecutive poses. For example, in the case of a simple two-wheeled robot, the odometry can be characterized as the forward and angular velocity (v, ω). For the purposes of this formulation, we assume that the robot can measure its odometry either by recording the control commands given to the motors or by using some other proprioceptive sensors such as wheel encoders. The robot odometry is denoted as

$$\boldsymbol{u}_{1:k} = \{\boldsymbol{u}_1, ..., \boldsymbol{u}_k\}, \tag{6.2}$$

where $\boldsymbol{u}_k$ provides information about the change between poses $\boldsymbol{x}_{k-1}$ and $\boldsymbol{x}_k$. Finally, the robot uses its sensors to observe the environment. In landmark SLAM, the environment consists of individual landmarks that the robot can detect:

$$\boldsymbol{m} = \{\boldsymbol{m}_1, ..., \boldsymbol{m}_M\}, \tag{6.3}$$

where $\boldsymbol{m}_j$ is a single landmark. We assume that at every time step, the robot makes a single measurement. Thus we have a sequence of measurements

$$\boldsymbol{z}_{1:k} = \{\boldsymbol{z}_1, ..., \boldsymbol{z}_k\}, \tag{6.4}$$

where $\boldsymbol{z}_k$ is a measurement at time k. This assumption serves only to simplify the formulation and we show later how to work with multiple simultaneous measurements. To use a specific example of a landmark and a measurement, we again use Formula Student as an example. The race track is delineated by a set of traffic cones with every cone constituting a single landmark. Considering a flat plane, a traffic cone is given by its position in the plane, ignoring the color. A sensor detecting this cone reports the range and bearing of the cone relative to the sensor.

With the above definitions of the robot path, odometry, map and measurements, SLAM can be posed in a probabilistic setting as the problem of estimating the posterior

$$p(\boldsymbol{x}_{0:k}, \boldsymbol{m} | \boldsymbol{z}_{1:k}, \boldsymbol{u}_{1:k}). \tag{6.5}$$

In other words, SLAM can be posed as a problem of estimating the joint posterior over the robot path and the map given the measurements and the odometry. For a visual representation of SLAM, see Fig. 6.2 and 6.3. To work with this formulation further, we need the robot motion and measurement models. These are analogous to the process and measurement models of the Recursive Bayesian filter. The robot motion model is given by its exact kinematics combined with an additive noise:

$$\boldsymbol{x}_k = g(\boldsymbol{x}_{k-1}, \boldsymbol{u}_k) + q_k, \tag{6.6}$$

where $q_k \sim \mathcal{N}(0, Q_k)$ is a zero-mean Gaussian. This may again be rewritten as

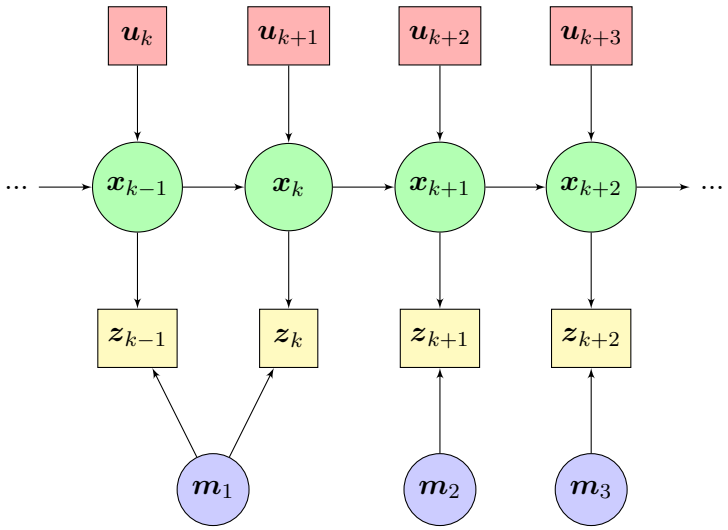


Figure 6.2: A graphical representation of the SLAM problem as a dynamic Bayes network (DBN). DBN is a generalization of HMMs which relates variables from consecutive time steps. The network clearly shows the relationship between the robot path ($\boldsymbol{x}_k$), odometry ($\boldsymbol{u}_k$), measurements ($\boldsymbol{z}_k$) and the landmarks ($\boldsymbol{m}_j$).

$$\boldsymbol{x}_k \sim p(\boldsymbol{x}|\boldsymbol{x}_{k-1}, \boldsymbol{u}_k). \tag{6.7}$$

Analogously, we have the measurement model:

$$\boldsymbol{z}_k = h(\boldsymbol{x}_k) + r_k, \tag{6.8}$$
$$\boldsymbol{z}_k \sim p(\boldsymbol{z}|\boldsymbol{x}_k), \tag{6.9}$$

where $r_k \sim \mathcal{N}(0, R_k)$. In the literature, one can find three main SLAM algorithms using the formulation given above. In the following sections, we introduce them in chronological order, starting from the oldest and finishing with the most recent algorithm. Finally, the last section of this chapter describes the problem of data association in SLAM and several different solution approaches to this problem.

6.4 EKF-SLAM

EKF-SLAM is the oldest SLAM algorithm that has been successfully used in real applications such as in indoor and underwater environments, underground exploration, and space exploration [3, 102, 37, 2, 42]. As the name suggests,

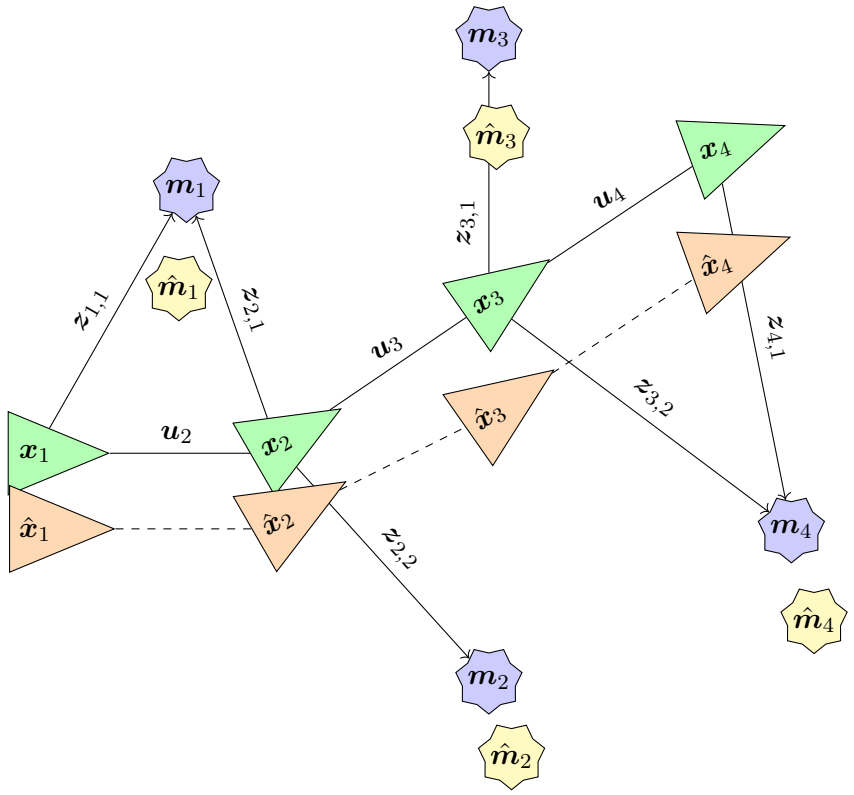


Figure 6.3: A visualization of SLAM in two dimensions. The true path (**green**) is shown alongside the path estimated by SLAM (**orange**). The robot observes landmarks in the environment (**blue**) to construct its own map estimate (**yellow**).

EKF-SLAM is an extension of the previously described Extended Kalman filter. In fact, EKF-SLAM can be viewed as an instance of an EKF, where the estimated state is the robot pose combined with the map. Despite the initial success of EKF-SLAM, it has been surpassed by more efficient and robust algorithms such as FastSLAM [64, 63]. As we will see, the computational complexity of EKF-SLAM makes it difficult to use for large-scale environments. However, various techniques to reduce the complexity have been proposed [74, 41]. The second limitation of EKF-SLAM relates to the robustness of data association. Different data associations commonly lead to vastly different maps. However, EKF-SLAM maintains only the most likely data association hypothesis, which is insufficient when there are multiple plausible associations. As such, incorrect data association can have a great negative effect on the accuracy of EKF-SLAM.

6.4.1 Formulation

The basic operation of EKF-SLAM is largely unchanged compared to a standard EKF. The main difference between the two is that the state of EKF-SLAM contains not only the robot pose, but all map landmarks as well. The pose and landmarks form a single joint state and the algorithm estimates the posterior over both. Before proceeding further, we make a few simplifying assumptions. First, we assume that the size of the map, M, is known apriori. This is not required, but simplifies the algorithm by allowing us to initialize the state mean and covariance to the correct size at the beginning. In addition, we assume that the data association is known. Here, the data association is encoded as a mapping variable $\boldsymbol{c}$ which maps measurements to their corresponding landmarks. That is, a measurement $\boldsymbol{z}_k$ maps to a landmark $j = \boldsymbol{c}_k$. Data association is a general problem in SLAM and is explained separately in section 6.8.

Being an instance of an EKF, EKF-SLAM represents the posterior as a Gaussian parametrized by its mean and covariance. The mean at time k is a $3 + 2M$-dimensional vector

$$\boldsymbol{\mu}_k = (\underbrace{x, y, \theta}_{pose}, \underbrace{m_{1,x}, m_{1,y}, ..., m_{M,x}, m_{M,y}}_{map})^T \tag{6.10}$$

$$= (\boldsymbol{x}_k, \boldsymbol{m}_{k,1}, ..., \boldsymbol{m}_{k,M})^T \tag{6.11}$$

$$= (\boldsymbol{x}_k, \boldsymbol{m}_k)^T. \tag{6.12}$$

In addition to the robot pose, the full state vector $\boldsymbol{\mu}_k$ contains the landmark positions. The covariance is then given in a straightforward fashion:

$$\Sigma_k = \begin{pmatrix} \Sigma_{\boldsymbol{x}_k\boldsymbol{x}_k} & \Sigma_{\boldsymbol{x}_k\boldsymbol{m}_1} & \dots & \Sigma_{\boldsymbol{x}_k\boldsymbol{m}_M} \\ \Sigma_{\boldsymbol{m}_1\boldsymbol{x}_k} & \Sigma_{\boldsymbol{m}_1\boldsymbol{m}_1} & \dots & \Sigma_{\boldsymbol{m}_1\boldsymbol{m}_M} \\ \vdots & \vdots & \ddots & \vdots \\ \Sigma_{\boldsymbol{m}_M\boldsymbol{x}_k} & \Sigma_{\boldsymbol{m}_M\boldsymbol{m}_1} & \dots & \Sigma_{\boldsymbol{m}_M\boldsymbol{m}_M} \end{pmatrix} \tag{6.13}$$

$$= \begin{pmatrix} \Sigma_{\boldsymbol{x}_k\boldsymbol{x}_k} & \Sigma_{\boldsymbol{x}_k\boldsymbol{m}} \\ \Sigma_{\boldsymbol{m}\boldsymbol{x}_k} & \Sigma_{\boldsymbol{m}\boldsymbol{m}} \end{pmatrix}. \tag{6.14}$$

Note that the covariance matrix Σ_k is composed of four blocks. $\Sigma_{\boldsymbol{x}_k\boldsymbol{x}_k}$ is the robot pose covariance. $\Sigma_{\boldsymbol{x}_k\boldsymbol{m}}$ and $\Sigma_{\boldsymbol{m}\boldsymbol{x}_k}$ is the covariance between the pose and landmark positions. Finally, $\Sigma_{\boldsymbol{m}\boldsymbol{m}}$ is the map covariance.

6.4.2 Initialization

In a standard EKF, we typically have an initial estimate to seed the filter with. This is not required in SLAM, however. Since we are interested in the robot path, the map, and how they relate to each other, we are free to define the initial robot pose as the origin of the coordinate frame:

$$\boldsymbol{\mu}_0 = (x, y, \theta, m_{1,x}, m_{1,y}, ..., m_{M,x}, m_{M,y})^T \tag{6.15}$$
$$= (0, 0, 0, 0, 0, ..., 0, 0)^T. \tag{6.16}$$

The initial estimate has the following covariance:

$$\Sigma_0 = \begin{pmatrix} 0 & 0 & 0 & 0 & \dots & 0 \\ 0 & 0 & 0 & 0 & \dots & 0 \\ 0 & 0 & 0 & 0 & \dots & 0 \\ 0 & 0 & 0 & \infty & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ \underbrace{0}_{3} & \underbrace{0}_{3} & \underbrace{0}_{3} & \underbrace{0}_{2M} & \underbrace{\dots}_{2M} & \underbrace{\infty}_{2M} \end{pmatrix}. \tag{6.17}$$

The initial pose was deliberately set, thus its uncertainty $\Sigma_{\boldsymbol{x}_0\boldsymbol{x}_0}$ is zero. To distinguish landmarks which were not yet observed, we set the landmark uncertainty to ∞. As this is the initial state, no landmarks were observed yet and thus all of them start with an infinite uncertainty. Note that the infinite uncertainty is just a mathematical convenience. In a real implementation, a sufficiently large number would be used instead to avoid possible problems with representing infities in programming languages.

6.4.3 Prediction step

The prediction step is analogous to EKF. First, we compute the new state prediction:

$$\hat{\boldsymbol{\mu}}_k = g(\boldsymbol{\mu}_{k-1}, \boldsymbol{u}_k), \tag{6.18}$$

where $g(.)$ is the motion model and would normally return a new robot pose (x, y, θ). However, the state also contains the map and so to use the

same EKF equations, the motion model needs to be modified to agree in dimensions with the state vector:

$$\hat{\boldsymbol{\mu}}_k = g(\boldsymbol{\mu}_{k-1}, \boldsymbol{u}_k) \tag{6.19}$$

$$= \begin{pmatrix} g_{\boldsymbol{x}}(\boldsymbol{\mu}_{k-1}, \boldsymbol{u}_k) \\ \boldsymbol{m}_{k-1} \end{pmatrix}. \tag{6.20}$$

Here, $g_{\boldsymbol{x}}(\boldsymbol{\mu}_{k-1}, \boldsymbol{u}_k) \in \mathbb{R}^3$ is the standard motion model and $\boldsymbol{m}_{k-1}$ is the previous map belief. In other words, the g function is simply extended from $\mathbb{R}^3$ to $\mathbb{R}^{3+2M}$ such that it has the same dimension as the state vector. Note that the motion model simply copies the old map without modifying it. To compute the covariance, we first need to compute the Jacobian of the motion model evaluated at the previous estimate. This procedure is the same as in EKF. The Jacobian of g is

$$G_k = \frac{\partial g(\boldsymbol{\mu}_{k-1}, \boldsymbol{u}_k)}{\partial \boldsymbol{\mu}_{k-1}} = \begin{pmatrix} G_{\boldsymbol{x}} & \mathbf{0} \\ \mathbf{0} & I \end{pmatrix}, \tag{6.21}$$

where $G_{\boldsymbol{x}} \in \mathbb{R}^{3\times 3}$ is a Jacobian of $g_{\boldsymbol{x}}$ i.e. the Jacobian of the standard motion model, and $I \in \mathbb{R}^{2M\times 2M}$ is an identity matrix. The covariance update can then be written as

$$\hat{\Sigma}_k = G_k \Sigma_{k-1} G_k^T + Q_k \tag{6.22}$$

$$= \begin{pmatrix} G_{\boldsymbol{x}} & \mathbf{0} \\ \mathbf{0} & I \end{pmatrix} \begin{pmatrix} \Sigma_{\boldsymbol{x}_k \boldsymbol{x}_k} & \Sigma_{\boldsymbol{x}_k \boldsymbol{m}} \\ \Sigma_{\boldsymbol{m}\boldsymbol{x}_k} & \Sigma_{\boldsymbol{m}\boldsymbol{m}} \end{pmatrix} \begin{pmatrix} G_{\boldsymbol{x}}^T & \mathbf{0} \\ \mathbf{0} & I \end{pmatrix} + Q_k \tag{6.23}$$

$$= \begin{pmatrix} G_{\boldsymbol{x}} \Sigma_{\boldsymbol{x}_k \boldsymbol{x}_k} G_{\boldsymbol{x}}^T & G_{\boldsymbol{x}} \Sigma_{\boldsymbol{x}_k \boldsymbol{m}} \\ (G_{\boldsymbol{x}} \Sigma_{\boldsymbol{x}_k \boldsymbol{m}})^T & \Sigma_{\boldsymbol{m}\boldsymbol{m}} \end{pmatrix} + Q_k. \tag{6.24}$$

The process noise Q_k needs to be extended so that it matches the dimensions of the full covariance:

$$Q_k = \begin{pmatrix} Q_x & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{pmatrix}. \tag{6.25}$$

In the above, Q_x is the original motion model noise. The rest of the entries of the matrix are zero since the process noise only affects the robot pose.

Notice that the covariance update leaves the landmark uncertainties intact. Only the robot pose and the pose-landmark covariance is updated. This is because the landmark estimates depend on the the pose estimate and thus any uncertainty in the pose will propagate into the landmarks. Because of this, the asymptotic complexity of the prediction step is $\mathcal{O}(M)$ where M is the number of landmarks. This completes the prediction step of the EKF-SLAM algorithm. The only major difference compared to the EKF is that the state vector is augmented with the map estimate and the motion model g and noise Q_k are extended to match the dimensions of the new state.

6.4.4 Correction step

Assuming the data association between measurements and landmarks is known, the update step is again analogous to the standard EKF update rule. If the association was unknown, an extra data association step would have to be performed before the correction. If we recall the EKF update equations

$$\boldsymbol{y}_k = \boldsymbol{z}_k - h_k\left(\hat{\boldsymbol{\mu}}_k\right), \tag{6.26}$$
$$S_k = H_k \hat{\Sigma}_k H_k^T + R_k, \tag{6.27}$$
$$K_k = \hat{\Sigma}_k H_k^T S_k^{-1}, \tag{6.28}$$
$$\boldsymbol{\mu}_k = \hat{\boldsymbol{\mu}}_k + K_k \boldsymbol{y}_k, \tag{6.29}$$
$$\Sigma_k = \left(I - K_k H_k\right) \hat{\Sigma}_k, \tag{6.30}$$

it is evident that the only unknown quantity we need to compute is the measurement Jacobian H_k. All other quantities are either already computed from the prediction step $(\hat{\boldsymbol{\mu}}_k, \hat{\Sigma}_k)$ or are supplied by the user $(\boldsymbol{z}_k, R_k)$. Here we assume that the robot observes one measurement at a time. Later we show how to adapt EKF-SLAM to incorporate simultaneous measurements. The measurement Jacobian is defined as

$$H_k = \frac{\partial h(\boldsymbol{\mu}_k)}{\partial \boldsymbol{\mu}_k}. \tag{6.31}$$

Assuming the measurement function h provides a two-dimensional measurement, e.g., a range and bearing measurement (r, ϕ), the resulting Jacobian has dimensions $H_k \in \mathbb{R}^{2\times 3+2M}$. In addition, the resulting matrix is sparse as the only nonzero elements will be $\frac{\partial h(\boldsymbol{\mu}_k)}{\partial \boldsymbol{x}_k}$ and $\frac{\partial h(\boldsymbol{\mu}_k)}{\partial \boldsymbol{m}_{k,j}}$, where $\boldsymbol{m}_{k,j}$ is the landmark corresponding to $\boldsymbol{z}_k = h(\boldsymbol{\mu}_k)$:

$$H_k = \frac{\partial h(\boldsymbol{\mu}_k)}{\partial \boldsymbol{\mu}_k} = \begin{pmatrix} \frac{\partial h}{\partial \boldsymbol{x}_k} & \mathbf{0} & \dots & \frac{\partial h}{\partial \boldsymbol{m}_{k,j}} & \dots & \mathbf{0} \end{pmatrix}. \tag{6.32}$$

Now that we have the Jacobian H_k, the next step is to simply follow the EKF update equations as one would in a normal EKF. The correction step needs to update the whole estimate covariance leading to a time complexity of $\mathcal{O}(M^2)$. Finally, a distinction has to be made as to whether the given measurement corresponds to a landmark that has been seen before. If the landmark has been seen before, the standard EKF update described above applies. If it is a new landmark, its position needs to be initialized before the correction step is carried out as follows:

$$\boldsymbol{m}_{k,j} = h^{-1}(\hat{\mu}_k, \boldsymbol{z}_k). \tag{6.33}$$

The function $h^{-1}(.)$ returns the position of the landmark in the global coordinate frame given the robot pose and the measurement. With this, we have covered the whole EKF-SLAM algorithm. The full pseudocode is provided in Algorithm 6.

6.4.5 Multiple measurements

The above formulation of EKF-SLAM needs to be extended to the case when the robot perceives multiple simultaneous measurements. A naive way to do this is to execute the correction step for every measurement. However, due to the quadratic time complexity, this may be costly. There is an alternative way to incorporate all measurements in a single step. We can think of the individual measurements $\boldsymbol{z}_{k,1}, ..., \boldsymbol{z}_{k,n}$ as a single measurement

$$\boldsymbol{z}_k^* = \begin{pmatrix} \boldsymbol{z}_{k,1} \\ \vdots \\ \boldsymbol{z}_{k,n} \end{pmatrix}. \tag{6.34}$$

This combined measurement has the corresponding uncertainty:

$$R^* = \begin{pmatrix} R & & \\ & \ddots & \\ & & R \end{pmatrix}. \tag{6.35}$$

Finally, the Jacobian is just the stacked Jacobians of the individual measurements:

Algorithm 6: EKF-SLAM

input : Previous estimate $\boldsymbol{\mu}_{k-1}, \Sigma_{k-1}$
Measurement $\boldsymbol{z}_k$
Control input $\boldsymbol{u}_k$
Data association $\boldsymbol{c}_k$
Process and measurement noise Q_k, R_k

output : A new estimate $\boldsymbol{\mu}_k, \Sigma_k$

1
2 // prediction
3 $\hat{\boldsymbol{\mu}}_k = g(\boldsymbol{\mu}_{k-1}, \boldsymbol{u}_k)$
4 $\hat{\Sigma}_k = G_k \Sigma_{k-1} G_k^T + Q_k$
5
6 // correction
7 $j = \boldsymbol{c}_k$
8 **if** *landmark never seen before* **then**
9 // initialize landmark
10 $\boldsymbol{m}_{k,j} = h^{-1}(\boldsymbol{\mu}_{k-1}, \boldsymbol{z}_k)$
11 **end**
12
13 $\boldsymbol{y}_k = \boldsymbol{z}_k - h(\boldsymbol{z}_k)$
14 $S_k = H_k \hat{\Sigma}_k H_k^T + R_k$
15 $K_k = \hat{\Sigma}_k H_k^T S_k^{-1}$
16 $\boldsymbol{\mu}_k = \hat{\boldsymbol{\mu}}_k + K_k \boldsymbol{y}_k$
17 $\Sigma_k = (I - K_k H_k) \hat{\Sigma}_k$

$$H_k^* = \begin{pmatrix} \frac{\partial h_1(\boldsymbol{\mu}_k)}{\partial \boldsymbol{\mu}_k} \\ \vdots \\ \frac{\partial h_n(\boldsymbol{\mu}_k)}{\partial \boldsymbol{\mu}_k} \end{pmatrix} = \begin{pmatrix} H_{k,1} \\ \vdots \\ H_{k,n} \end{pmatrix}. \tag{6.36}$$

With these modifications, it is sufficient to carry out only a single EKF update as opposed to updating the state for every measurement separately.

6.4.6 Final remarks

Albeit conceptually simple, EKF-SLAM performs well in real applications. Unfortunately, its simplicity is outweighed by several underisable properties. The most prohibitive is its high computational complexity. The algorithm requires $\mathcal{O}(M^2)$ memory and $\mathcal{O}(M^2)$ time per step since the algorithm stores the full covariance matrix and the whole matrix needs to be updated in every step. This prevents EKF-SLAM from being efficiently used in large-scale environments as the computational complexity grows quadratically

with the map size. Furthermore, an issue shared with EKF is the Gaussian assumption. Due to this, EKF-SLAM struggles with highly nonlinear motion and measurement models where the linearization is less precise. Finally, EKF-SLAM is very sensitive to errors in data association. Since it can only represent a single data association hypothesis, any error in the correspondence can lead to filter divergence in the future as there is no mechanism to undo or refine previous associations.

6.5 FastSLAM

The FastSLAM algorithm [64, 63] fixes many of the aforementioned issues present in EKF-SLAM. Most notably, the time complexity of FastSLAM is significantly reduced compared to EKF-SLAM. Moreover, FastSLAM can more easily handle nonlinear process models and is more resilient to data association errors. FastSLAM achieves this by factorizing the SLAM posterior (6.5) which allows it to use a combination of particle filters and EKFs. Thanks to this factorization, the computational complexity is reduced to $\mathcal{O}(M)$ in a simple implementation and $\mathcal{O}(\log M)$ by using optimal tree data structures. Moreover, since FastSLAM uses a particle filter to track the robot pose, nonlinear motion models are handled more easily. Recall that EKF-SLAM first needs to linearize the motion model using the Taylor expansion. Finally, since data association is carried out on a per-particle basis, FastSLAM is much more robust to incorrect data association.

In the literature, one can find two variants of the FastSLAM algorithm – FastSLAM 1.0 [64] and FastSLAM 2.0 [63]. FastSLAM 2.0 is a newer and much improved version of the original FastSLAM 1.0. Since the two algorithms are conceptually very similar, we first explain the original algorithm in full and then explain the differences introduced in FastSLAM 2.0. As a final note, for the sake of brevity, when we refer to FastSLAM, we mean FastSLAM 1.0 unless otherwise specified.

6.5.1 Factoring the posterior

A key property of the SLAM posterior exploited by FastSLAM is that the landmark locations are conditionally independent given the robot pose. If we view the SLAM problem as a dynamic Bayes network as shown in Fig. 6.2, the structure of network makes the landmarks *d-separated* by the robot path. D-separation is a property of a Bayesian network which is defined on a pair of nodes. It can be shown that if a pair of nodes is d-separated, the nodes are conditionally independent. This property allows us to factor the SLAM posterior in the following way:

$$p(\boldsymbol{x}_{0:k}, \boldsymbol{m}|\boldsymbol{z}_{1:k}, \boldsymbol{u}_{1:k}) = p(\boldsymbol{x}_{0:k}|\boldsymbol{z}_{1:k}, \boldsymbol{u}_{1:k})p(\boldsymbol{m}|\boldsymbol{x}_{0:k}, \boldsymbol{z}_{1:k}, \boldsymbol{u}_{1:k}) \tag{6.37}$$

$$= p(\boldsymbol{x}_{0:k}|\boldsymbol{z}_{1:k}, \boldsymbol{u}_{1:k})p(\boldsymbol{m}|\boldsymbol{x}_{0:k}, \boldsymbol{z}_{1:k}) \tag{6.38}$$

$$= \underbrace{p(\boldsymbol{x}_{0:k}|\boldsymbol{z}_{1:k}, \boldsymbol{u}_{1:k})}_{\text{pose posterior}} \prod_{j=1}^{M} \underbrace{p(\boldsymbol{m}_j|\boldsymbol{x}_{0:k}, \boldsymbol{z}_{1:k})}_{\text{landmark posterior}}. \tag{6.39}$$

In the last equation, we used the conditional independence to rewrite the map posterior as a product over landmark posteriors. Indeed, the key insight behind fastSLAM is the fact that the SLAM posterior can be factored into a pose posterior and separate landmark posteriors. This factorization tells us that we can estimate the robot pose and landmark positions independently. Specifically, FastSLAM uses a particle filter to estimate the robot pose. Each particle in turn contains independent Extended Kalman filters to track individual landmark locations. Thanks to this factorization, both the particle filter and the EKFs are low-dimensional. Assuming a two-dimensional world, the particle filter estimates the robot pose (x, y, θ) while the EKFs track the two-dimensional landmark positions. From this, it is immediately evident that unlike in EKF-SLAM, incorporating a new measurement in FastSLAM is $\Theta(1)$. This is because the landmark filter has a constant size (2×2) with respect to the map.

The basic FastSLAM algorithm is almost identical to that of a particle filter described in Algorithm 5. First, robot poses are sampled from a proposal distribution. Then, the measurements are incorporated into the map. This is done by updating the corresponding EKFs for each landmark using the measurement model. Third, the importance weights are computed and finally, the particles are resampled. The only fundamental difference between FastSLAM and a standard particle filter is the need to update the map with the measurements. However, the landmark update itself is a standard EKF update. In summary, FastSLAM combines two standard filtering algorithms in a clever way, which increases the robustness of the algorithm and decreases its computational complexity. For a graphical representation of FastSLAM particles, see Fig. 6.4. For notational convenience, we will drop the particle index $\boldsymbol{x}^i$ used for variables belonging to particular particles in situations where the context permits it.

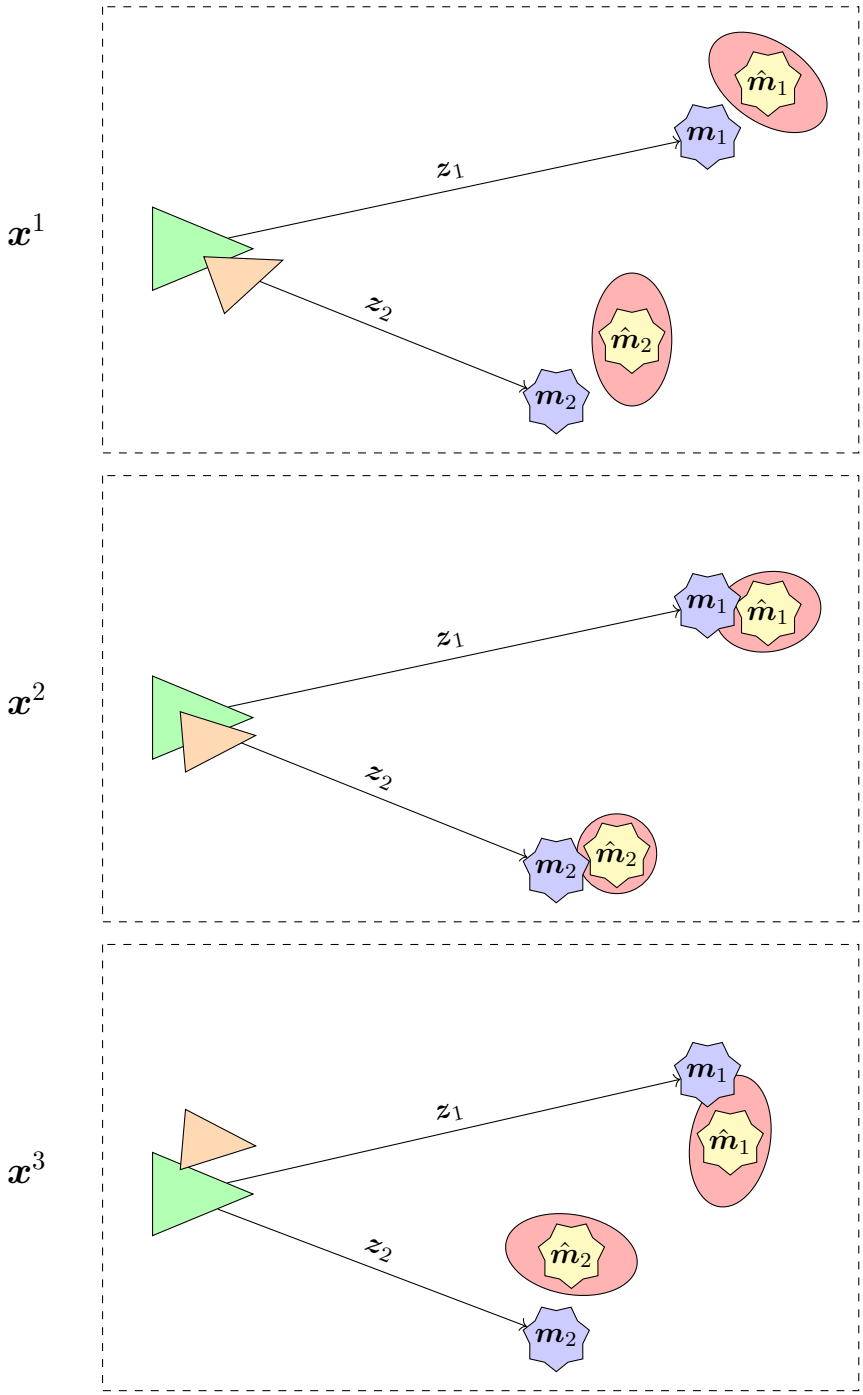


Figure 6.4: The FastSLAM algorithm represents the posterior as separate particles each containing its own map estimate. Because of this, data association is performed on a per-particle basis leading to a more robust behavior. Here is an example with three particles showing the robot pose (**green**) and its estimate (**orange**) together with the landmarks (**blue**) and their estimates (**yellow**) with uncertainties shown as red ellipses.

6.5.2 Posterior representation

FastSLAM represents the posterior as a weighted sum of particles with the following structure:

$$\Big(\underbrace{\boldsymbol{x}_k}_{\text{robot pose}}, \boldsymbol{\mu}_{k,1}, \Sigma_{k,1}, ..., \underbrace{\boldsymbol{\mu}_{k,j}, \Sigma_{k,j}}_{\substack{\text{landmark mean} \\ \text{and covariance}}}, ..., \boldsymbol{\mu}_{k,M}, \Sigma_{k,M} \Big). \tag{6.40}$$

The particle contains the robot pose $\boldsymbol{x}_k = (x, y, \theta)$ and the mean and covariance of every observed landmark. FastSLAM is initialized similarly to EKF-SLAM. If there are no constraints on the coordinate system, the initial set of particles is created with the robot pose set at the origin. The particles start with an empty map and new landmarks are added to the particles when they are first observed by the robot. These steps are explained in more detail in the following sections.

6.5.3 Prediction step

In the prediction step, new robot poses are sampled from the proposal distribution. As we have discussed in chapter 5, there are many choices for a suitable proposal distribution. In particular, FastSLAM 1.0 samples new poses from the motion model. This is a reasonable choice as the motion model is typically well-behaved and easy to sample from. As we show later, a more sophisticated choice of the proposal is possible, which is exploited by FastSLAM 2.0. Mathematically, new poses are sampled in the following manner:

$$\boldsymbol{x}_k^i \sim p(\boldsymbol{x}_k | \boldsymbol{x}_{k-1}^i, \boldsymbol{u}_k). \tag{6.41}$$

The actual mechanism of sampling new robot poses from the motion model is fairly straightforward. Assuming the motion model is given by (6.6), where $q_k \sim \mathcal{N}(0, Q_k)$, we first sample q_k from the normal distribution. A new pose is then computed as the sum of the deterministic motion model $g(\boldsymbol{x}_{k-1}^i, \boldsymbol{u}_k)$ and the sampled noise q_k.

6.5.4 Correction step

The correction step of the FastSLAM algorithm consists of two parts – the map update and the weight update. For simplicity, we assume that the robot

receives only a single measurement at a given time. In addition, we assume that the data association is known apriori. Later we will show how to remove both of these assumptions just as we did with EKF-SLAM.

Map update

FastSLAM represents the map using M independent Extended Kalman filters. Each EKF tracks a single landmark. Every particle maintains its own map relative to its estimated robot path. The implementation of the EKF is straightforward. Since the landmarks are assumed to be stationary, there is no prediction step. The EKF correction step is in principle similar to the landmark update in EKF-SLAM. To update a landmark $\boldsymbol{m}_j$, we need to distinguish three cases:

1. Landmark $\boldsymbol{m}_j$ was not observed in the current time step k. In this case, the landmark estimate remains unchanged. The update rule is then simple:

$$\boldsymbol{\mu}_{k,j} = \boldsymbol{\mu}_{k-1,j} \tag{6.42}$$
$$\boldsymbol{\Sigma}_{k,j} = \Sigma_{k-1,j}. \tag{6.43}$$

2. Landmark $\boldsymbol{m}_j$ was observed for the first time. In this case, the new landmark is initialized using

$$\boldsymbol{\mu}_{k,j} = h^{-1}(\boldsymbol{x}_k, \boldsymbol{z}_k) \tag{6.44}$$
$$\boldsymbol{\Sigma}_{k,j} = (H_k R_k^{-1} H_k^T)^{-1}, \tag{6.45}$$

   where $\boldsymbol{z}_k$ is the measurement, $h^{-1}(.)$ is the inverse of the measurement model and H_k is the Jacobian of the measurement model analogous to EKF-SLAM.

3. Landmark $\boldsymbol{m}_j$ was reobserved. In the third and final case, we proceed with a standard EKF update, linearizing the measurement model if necessary:

$$S_k = H_k \Sigma_{k-1,j} H_k^T + R_k, \tag{6.46}$$
$$K_k = \Sigma_{k-1,j} H_k^T S_k^{-1}, \tag{6.47}$$
$$\boldsymbol{\mu}_{k,j} = \boldsymbol{\mu}_{k-1,j} + K_k(\boldsymbol{z}_k - \hat{\boldsymbol{z}}_k), \tag{6.48}$$
$$\Sigma_{k,j} = (I - K_k H_k)\Sigma_{k-1,j}. \tag{6.49}$$

Unlike in EKF-SLAM, incorporating a measurement is a constant time operation as it requires updating a single 2×2 EKF whereas in EKF-SLAM, the whole state covariance needs to be updated.

Weight update

The second part of the correction step is to compute the importance weights of the particles. Recall the weight update from chapter 5:

$$w_k^i \propto \frac{p(\boldsymbol{x}_{0:k}^i|\boldsymbol{z}_{1:k})}{q(\boldsymbol{x}_{0:k}^i|\boldsymbol{z}_{1:k})} = \frac{\text{target}}{\text{proposal}}. \tag{6.50}$$

To derive the weight update, we will again consider the full path posterior $p(\boldsymbol{x}_{0:k}^i|\boldsymbol{z}_{1:k}, \boldsymbol{u}_{1:k})$. This path posterior is the target distribution. Since the new poses are sampled from $p(\boldsymbol{x}_k|\boldsymbol{x}_{k-1}^i, \boldsymbol{u}_k)$, the robot paths after the prediction step are distributed according to

$$p(\boldsymbol{x}_{0:k}^i|\boldsymbol{z}_{1:k-1}, \boldsymbol{u}_{1:k}) = \underbrace{p(\boldsymbol{x}_k^i|\boldsymbol{x}_{k-1}^i, \boldsymbol{u}_k)}_{\text{motion model}} \underbrace{p(\boldsymbol{x}_{0:k-1}^i|\boldsymbol{z}_{1:k-1}, \boldsymbol{u}_{1:k-1})}_{\text{previous path posterior}} \tag{6.51}$$

Substituting this into (6.50), we obtain

$$\begin{aligned}
w_k^i &\propto \frac{p(\boldsymbol{x}_{0:k}^i|\boldsymbol{z}_{1:k}, \boldsymbol{u}_{1:k})}{p(\boldsymbol{x}_{0:k}^i|\boldsymbol{z}_{1:k-1}, \boldsymbol{u}_{1:k})} && (6.52)\\
&\overset{Bayes}{\propto} \frac{p(\boldsymbol{z}_k|\boldsymbol{x}_{0:k}^i, \boldsymbol{z}_{1:k-1}, \boldsymbol{u}_{1:k})p(\boldsymbol{x}_{0:k}^i|\boldsymbol{z}_{1:k-1}, \boldsymbol{u}_k)}{p(\boldsymbol{x}_{0:k}^i|\boldsymbol{z}_{1:k-1}, \boldsymbol{u}_{1:k})} && (6.53)\\
&= \frac{p(\boldsymbol{z}_k|\boldsymbol{x}_{0:k}^i, \boldsymbol{z}_{1:k-1})p(\boldsymbol{x}_{0:k}^i|\boldsymbol{z}_{1:k-1}, \boldsymbol{u}_k)}{p(\boldsymbol{x}_k^i|\boldsymbol{z}_{1:k-1}, \boldsymbol{u}_k)} && (6.54)\\
&= p(\boldsymbol{z}_k|\boldsymbol{x}_{0:k}^i, \boldsymbol{z}_{1:k-1}) && (6.55)\\
&= \int p(\boldsymbol{z}_k|\boldsymbol{m}_j, \boldsymbol{x}_{0:k}^i, \boldsymbol{z}_{1:k-1})p(\boldsymbol{m}_j|\boldsymbol{x}_{0:k}^i, \boldsymbol{z}_{1:k-1})d\boldsymbol{m}_j && (6.56)\\
&= \int \underbrace{p(\boldsymbol{z}_k|\boldsymbol{m}_j, \boldsymbol{x}_k^i)}_{\text{measurement model}} \underbrace{p(\boldsymbol{m}_j|\boldsymbol{x}_{0:k-1}^i, \boldsymbol{z}_{1:k-1})}_{\substack{\text{landmark estimate}\\ \text{at } k-1}} d\boldsymbol{m}_j. && (6.57)
\end{aligned}$$

The last integral is reminiscent of the EKF update rule. Indeed, it is the probability of observing $\boldsymbol{z}_k$ given the landmark estimate from the previous time step. Specifically, the weight is computed as

$$\begin{aligned}
w_k^i &\propto \mathcal{N}(\boldsymbol{z}_k; \hat{\boldsymbol{z}}_k, S_k^i) && (6.58)\\
&= \frac{1}{\sqrt{|2\pi S_k^i|}} \exp\{-\frac{1}{2}(\boldsymbol{z}_k - \hat{\boldsymbol{z}}_k)^T (S_k^i)^{-1}(\boldsymbol{z}_k - \hat{\boldsymbol{z}}_k)\}, && (6.59)
\end{aligned}$$

where $(\boldsymbol{z}_k - \hat{\boldsymbol{z}}_k)$ is the measurement residual and $S_k = H_k \Sigma_{k-1,j} H_k^T$ is the standard EKF residual covariance. The new weight is thus proportional to the likelihood that the robot observed $\boldsymbol{z}_k$ given our previous belief of the landmark position estimated by the EKF.

6.5.5 Resampling

FastSLAM estimates the robot pose using a particle filter and as such, resampling is needed to prevent sample degeneracy. Resampling in FastSLAM is analogous to resampling in a standard particle filter. The difference is that since the whole particle is resampled, not only the path hypotheses but also map hypotheses are deleted in the resampling step. This leads to the depletion of historical data of the map which effects the overall map statistics [8].

6.5.6 Multiple measurements

The FastSLAM algorithm is easily adapted to take into account multiple simultaneous measurements. Consider that at time k, the robot observes measurements $\{\boldsymbol{z}_{k,1}, ..., \boldsymbol{z}_{k,n}\}$. Provided the data association is known, the corresponding landmark EKFs are updated sequentially. The importance weight is then a product of the partial weights computed from each observation:

$$w_k^i = \prod_{j=1}^{n} w_{k,j}^i \tag{6.60}$$

$$= \prod_{j=1}^{n} \mathcal{N}(\boldsymbol{z}_{k,j}, \hat{\boldsymbol{z}}_{k,j}, S_k^i). \tag{6.61}$$

6.5.7 Landmark elimination

Due to imperfect sensors, spurious measurements may be introduced into the map. Depending on the frequency of false positives, this may be problematic as other than resampling away unlikely particles, FastSLAM has no mechanism to prune incorrect landmarks from the map. In [64], a method to track the probability of existence of every landmark is proposed. Observing a landmark provides evidence for existence, while not observing a landmark despite it being in the sensor range, provides evidence for nonexistence. When the landmark probability drops below a certain threshold, the landmark is deleted.

Let $e_j \in \{0, 1\}$ be a binary variable indicating the existence of landmark $\boldsymbol{m}_j$. The probability of existence

$$p(e_j|\boldsymbol{x}_{0:k}, \boldsymbol{z}_{1:k}) \tag{6.62}$$

is tracked using a static binary Bayesian filter. The binary Bayesian filter is another special case of the recursive Bayesian filter. The filter equation is commonly represented in its log-odds form [90]. Skipping over the derivation which is obtained from the general recursive filter, we obtain:

$$\ell(e_j|\boldsymbol{x}_{0:1}, \boldsymbol{z}_{1:k}) = \ell(e_j|\boldsymbol{x}_k, \boldsymbol{z}_k) + \ell(e_j|\boldsymbol{x}_{0:k-1}, \boldsymbol{z}_{1:k-1}) + \ell(e_j), \tag{6.63}$$

where $\ell(x)$ is the log odds ratio:

$$\ell(x) = \ln \frac{p(x)}{1 - p(x)}. \tag{6.64}$$

Assuming a uniform prior, the update amounts to adding

$$\ell(e_j|\boldsymbol{x}_k, \boldsymbol{z}_k) = \ln \frac{p(e_j|\boldsymbol{x}_k, \boldsymbol{z}_k)}{1 - (e_j|\boldsymbol{x}_k, \boldsymbol{z}_k)}, \tag{6.65}$$

whenever the landmark is in the perceptual range of the sensor. The landmark is deleted when the log-odds value drops below a chosen threshold. Note that this method relies on the knowledge of the reliability and the perceptual range of the sensor and the overall geometry of the environment to accurately compute $p(e_j|\boldsymbol{x}_k, \boldsymbol{z}_k)$. In an environment with a lot of spurious observations, one may want to set the probability lower to eliminate the false positives faster. On the other hand, in an environment with a large number of occlusions, one needs to be more conservative as not observing a landmark does not carry as much evidence for nonexistence. To give a real example, in the Formula Student competition, LiDAR is a very popular sensor among the teams. This type of sensor works best when mounted just above ground since the traffic cones are fairly short. In this setting, it is common that cones close to the car occlude cones farther away. Thus, not observing a given cone does not necessarily provide evidence for its nonexistence.

6.5.8 Final notes

This description constitutes the entirety of the FastSLAM 1.0 algorithm. We would like to remark that despite the fact that the full path posterior was used in the derivation of the importance weights, the result only depends on the current robot pose. This means that it is not necessary to remember the full robot path and hence the memory requirements are bounded. The full pseudocode of FastSLAM 1.0 is given in Algorithm 7. FastSLAM alleviates many problems present in the EKF-SLAM algorithm. Most importantly, the time complexity is reduced from $\mathcal{O}(M^2)$ to $\mathcal{O}(M)$ in a naive implementation and up to $\mathcal{O}(\log M)$ by using optimal data structures [64]. Moreover, FastSLAM can more easily handle nonlinear motion models since the path posterior is approximated using a particle filter instead of an EKF.

6.6 FastSLAM 2.0

A common problem that can noticably reduce the accuracy of FastSLAM 1.0 is when the motion model contains a lot of noise relative to the sensors. If that is the case, the prediction step will propagate most particles into areas where they will be subsequently assigned low weights in the correction step. Thus, most particles will be eliminated in the resampling step, lowering the diversity and wasting computational effort on particles that are likely to be discarded. The reason for this is that in FastSLAM 1.0, particles are sampled from the motion model which disregards information about the current measurement. If the motion model disagrees with the observations, more particles are needed to cover areas were the motion model likelihood is low, but the measurement likelihood is high. This model mismatch leads to excessive resampling. We have shown in chapter 5 that resampling may introduce sample impoverishment. This means that after a certain point in the past, all particles will share a common history. For FastSLAM, this is problematic as particles also contain map estimates. Due to this, FastSLAM cannot revise its map estimate beyond the point at which all particle histories are shared.

To remedy this, FastSLAM 2.0 [63] instead samples new poses from a distribution conditioned on the previous pose together with the current measurement. Since the sampled poses are in agreement with the measurements, most particles contribute meaningfully to the estimate and sample degeneracy is reduced. Thus, FastSLAM 2.0 typically requires significantly fewer particles to achieve the same level of accuracy compared to FastSLAM 1.0. Moreover, unlike FatSLAM 1.0, FastSLAM 2.0 has been shown to converge with just a single particle in the linear Gaussian case. This is remarkable as FastSLAM 1.0 does not converge regardless of the number of particles due to the fact that

Algorithm 7: FastSLAM 1.0

input : Particles $\boldsymbol{x}_{k-1}^1, \ldots, \boldsymbol{x}_{k-1}^N$
Measurement $\boldsymbol{z}_k$
Control input $\boldsymbol{u}_k$
Data association $\boldsymbol{c}_k$
Process and measurement noise R_k, Q_k
output : New particles $\boldsymbol{x}_k^1, \ldots, \boldsymbol{x}_k^N$

```
1
2  // prediction
3  for i ← 1 to N do
4      x_k^i ~ p(x_k | x_{k-1}^i, u_k)
5  end
6
7  j = c_k
8  for i ← 1 to N do
9      if landmark never seen before then
10         // initialize landmark
11         μ_{k,j} = h^{-1}(x_k^i, z_k)
12         Σ_{k,j} = (H_k R_k^{-1} H_k^T)^{-1}
13     else
14         // update landmark
15         S_k = H_k Σ_{k-1,j} H_k^T + R_k
16         K_k = Σ_{k-1,j} H_k^T S_k^{-1}
17         μ_{k,j} = μ_{k-1,j} + K_k(z_k − ẑ_k)
18         Σ_{k,j} = (I − K_k H_k) Σ_{k-1,j}
19     end
20     w_k^i = N(z_k, ẑ_k, S_k)
21 end
22
23 if N_eff below threshold then
24     Resample x_k^i
25 end
```

in FastSLAM 1.0, there is no way to change the trajectory of the particles after they are sampled from the motion model. In FastSLAM 2.0, by sampling from the modified proposal, the particle trajectory is corrected to agree with the observations.

6.6.1 Proposal distribution

The only fundamental difference between the FastSLAM 1.0 and 2.0 is the different proposal distribution. In FastSLAM 1.0, the poses are sampled from the motion model:

$$\boldsymbol{x}_k^i \sim p(\boldsymbol{x}_k|\boldsymbol{x}_{k-1}^i, \boldsymbol{u}_k). \tag{6.66}$$

In contrast, FastSLAM 2.0 samples poses from

$$\boldsymbol{x}_k^i \sim p(\boldsymbol{x}_k|\boldsymbol{x}_{0:k-1}^i, \boldsymbol{u}_{1:k}, \boldsymbol{z}_{1:k}). \tag{6.67}$$

In the above, we condition the new pose on the most recent measurement $\boldsymbol{z}_k$ in addition to the control $\boldsymbol{u}_k$. The above probability can be expanded using Bayes rule together with the Markov property:

$$\begin{aligned} p(\boldsymbol{x}_k|\boldsymbol{x}_{0:k-1}^i, \boldsymbol{u}_{1:k}, \boldsymbol{z}_{1:k}) &\overset{Bayes}{=} \eta p(\boldsymbol{z}_k|\boldsymbol{x}_k, \boldsymbol{x}_{0:k-1}^i, \boldsymbol{u}_{1:k}, \boldsymbol{z}_{1:k-1}) \\ &\qquad\qquad p(\boldsymbol{x}_k|\boldsymbol{x}_{0:k-1}^i, \boldsymbol{u}_{1:k}, \boldsymbol{z}_{1:k-1}) \qquad (6.68) \\ &= \eta \underbrace{p(\boldsymbol{z}_k|\boldsymbol{x}_k, \boldsymbol{x}_{0:k-1}^i, \boldsymbol{u}_{1:k}, \boldsymbol{z}_{1:k-1})}_{\text{likelihood}} \underbrace{p(\boldsymbol{x}_k|\boldsymbol{x}_{k-1}^i, \boldsymbol{u}_k)}_{\text{prior}}. \qquad (6.69) \end{aligned}$$

Here, η is a normalizing constant. In (6.69), the robot pose in the second term is conditionally independent of $\boldsymbol{z}_{k-1}$ and $\boldsymbol{x}_{0:k-2}^i$ given $\boldsymbol{x}_{k-1}^i$. Thus the second term simplifies to the motion model. Using the law of total probability, we can insert the corresponding landmark $\boldsymbol{m}_j$ into the expression:

$$= \eta \int p(\boldsymbol{z}_k|\boldsymbol{m}_j, \boldsymbol{x}_k, \boldsymbol{x}^i_{0:k-1}, \boldsymbol{u}_{1:k}, \boldsymbol{z}_{1:k-1}) p(\boldsymbol{m}_j|\boldsymbol{x}_k, \boldsymbol{x}^i_{0:k-1}, \boldsymbol{u}_{1:k}, \boldsymbol{z}_{1:k-1}) d\boldsymbol{m}_j$$
$$p(\boldsymbol{x}_k|\boldsymbol{x}^i_{k-1}, \boldsymbol{u}_k) \tag{6.70}$$
$$= \eta \int \underbrace{p(\boldsymbol{z}_k|\boldsymbol{m}_j, \boldsymbol{x}_k)}_{\text{Measurement model}} \underbrace{p(\boldsymbol{m}_j|\boldsymbol{x}^i_{0:k-1}, \boldsymbol{z}_{1:k-1}, \boldsymbol{u}_{1:k-1})}_{\text{Landmark prior}} d\boldsymbol{m}_j$$
$$p(\boldsymbol{x}_k|\boldsymbol{x}^i_{k-1}, \boldsymbol{u}_k). \tag{6.71}$$

This expression cannot be evaluated exactly due to the nonlinearity of the measurement model. However, a closed form result can be obtained using a linear approximation. Here, we skip the derivation and only show the final result. We invite the reader to consult the full derivation in [62]. Using a linear approximation, the above expression simplifies to a Gaussian with the following parameters:

$$\Sigma_x = \left(H^T_{\boldsymbol{x}} S^{-1} H_{\boldsymbol{x}} + Q_k^{-1}\right)^{-1} \tag{6.72}$$
$$\boldsymbol{\mu}_x = \Sigma_x H^T_{\boldsymbol{x}} S^{-1}(\boldsymbol{z}_k - \hat{\boldsymbol{z}}_k) + \hat{\boldsymbol{x}}_k. \tag{6.73}$$

Here, the matrix S is again the innovation covariance defined as $S = H_{\boldsymbol{m}} \Sigma_{k-1,m} H^T_{\boldsymbol{m}} R_k$. $H_{\boldsymbol{x}}$ and $H_{\boldsymbol{m}}$ are the Jacobians of the measurement model with respect to the pose and landmark position, respectively. One can think of this new distribution as a single EKF update of the predicted robot pose. The particle is first projected using the motion model and then subsequently corrected using the estimated position of the corresponding landmark.

Since the proposal distribution has changed, the importance weight needs to also be rederived. We again skip the derivation and provide only the result. The details can again be found in [62]. The new importance weight is computed using the following Gaussian:

$$w^i_k = \mathcal{N}(\boldsymbol{z}_k; \hat{\boldsymbol{z}}_k, H_{\boldsymbol{x}} Q_k H^T_{\boldsymbol{x}} + H_{\boldsymbol{m}} \Sigma_{k-1,j} H^T_{\boldsymbol{m}} + R_k). \tag{6.74}$$

In summary, the only difference between FastSLAM 1.0 and 2.0 is the different proposal distribution which takes into account not only the motion but also the measurements. This modified proposal requires to also derive a new formula for the importance weights. The pseudocode of FastSLAM 2.0 is provided in Algorithm 8.

Algorithm 8: FastSLAM 2.0

input : Particles $\boldsymbol{x}_{k-1}^1, \ldots, \boldsymbol{x}_{k-1}^N$
Measurement $\boldsymbol{z}_k$
Control input $\boldsymbol{u}_k$
Data association $\boldsymbol{c}_k$
Process and measurement noise R_k, Q_k

output : New particles $\boldsymbol{x}_k^1, \ldots, \boldsymbol{x}_k^N$

for $i \leftarrow 1$ **to** N **do**
  $\boldsymbol{x}_k^i = g(\boldsymbol{x}_{k-1}^i, \boldsymbol{u}_k)$
end

$j = \boldsymbol{c}_k$
for $i \leftarrow 1$ **to** N **do**
  if *landmark never seen before* **then**
    // initialize landmark
    $\boldsymbol{\mu}_{k,j} = h^{-1}(\boldsymbol{x}_k^i, \boldsymbol{z}_k)$
    $\Sigma_{k,j} = (H_k R_k^{-1} H_k^T)^{-1}$
    // sample pose
    $\boldsymbol{x}_k^i \sim p(\boldsymbol{x}_k | \boldsymbol{x}_{k-1}^i, \boldsymbol{u}_k)$
  else
    $S_k = H_k \Sigma_{k-1,j} H_k^T + R_k$
    // update position
    $\Sigma_x = \left(H_{\boldsymbol{x}}^T S^{-1} H_{\boldsymbol{x}} + Q_k^{-1} \right)^{-1}$
    $\boldsymbol{\mu}_x = \Sigma_x H_{\boldsymbol{x}}^T S_k^{-1} (\boldsymbol{z}_k - \hat{\boldsymbol{z}}_k) + \hat{\boldsymbol{x}}_k$
    // update landmark
    $K_k = \Sigma_{k-1,j} H_k^T S_k^{-1}$
    $\boldsymbol{\mu}_{k,j} = \boldsymbol{\mu}_{k-1,j} + K_k (\boldsymbol{z}_k - \hat{\boldsymbol{z}}_k)$
    $\Sigma_{k,j} = (I - K_k H_k) \Sigma_{k-1,j}$
    // sample pose
    $\boldsymbol{x}_k^i \sim \mathcal{N}(\boldsymbol{\mu}_x, \Sigma_x)$
  end
  $w_k^i = \mathcal{N}(\boldsymbol{z}_k; \hat{\boldsymbol{z}}_k, H_{\boldsymbol{x}} Q_k H_{\boldsymbol{x}}^T + H_{\boldsymbol{m}} \Sigma_{k-1,j} H_{\boldsymbol{m}}^T + R_k)$
end

if N_{eff} *below threshold* **then**
  Resample $\boldsymbol{x}_k^i$
end

6.6.2 Multiple measurements

Handling multiple measurements in FastSLAM 2.0 is more complicated compared to FastSLAM 1.0. This is due to the different proposal distribution which incorporates the measurements as well. We stated before that one can think of the new proposal as a single EKF update to the pose proposed by the motion model. With multiple measurements, the procedure is the same, however, it is carried out iteratively. Every measurement is used one by one to refine the final proposal distribution. First, the initial pose is sampled from the motion model. Then, for every measurement, the corrected proposal is computed using the equations below:

$$\Sigma_{\boldsymbol{x}_k,0} = Q_k \tag{6.75}$$

$$\mu_{\boldsymbol{x}_k,0} = g(\boldsymbol{x}_{k-1}, \boldsymbol{u}_k) \tag{6.76}$$

$$\Sigma_{\boldsymbol{x}_k,n} = \left(H_{\boldsymbol{x},n}^T S^{-1} H_{\boldsymbol{x},n} + \Sigma_{\boldsymbol{x}_k,n-1}^{-1}\right)^{-1} \tag{6.77}$$

$$\mu_{\boldsymbol{x}_k,n} = \mu_{\boldsymbol{x}_k,n-1} + \Sigma_{\boldsymbol{x}_k,n} H_{\boldsymbol{x},n}^T S^{-1} (\boldsymbol{z}_k - \hat{\boldsymbol{z}}_{k,n}). \tag{6.78}$$

.

These equations are essentially the EKF update equations. We sample a new pose from this new proposal, update the corresponding landmark filter using this pose, and compute the importance weight. The procedure is then repeated for the next measurement using the same equations given above. Once all measurements are processed, we use the proposal distribution to sample the final pose. This procedure indeed mirrors a standard EKF incorporating several measurements. As we introduce more measurements, the variance of the proposal will decrease and the proposal distribution will concentrate in areas which are in agreement with the measurements.

6.6.3 Extracting the map

In many applications, including the Formula Student competition, SLAM is only one part of a larger navigation system. Often times, other parts of the system, such as path planning, depend on the output of SLAM. It is therefore necessary to regularly extract the estimated position and the map from the algorithm. In EKF-SLAM, this step is rather straightforward. The algorithm maintains a single map estimate which is simple to obtain by taking the corresponding part of the state vector. In FastSLAM, however, every particle carries its own map estimate. One may then ask how to extract a single map from FastSLAM that most likely represents the environment. Taking a simple weighted average of the landmark positions across all landmarks is in general

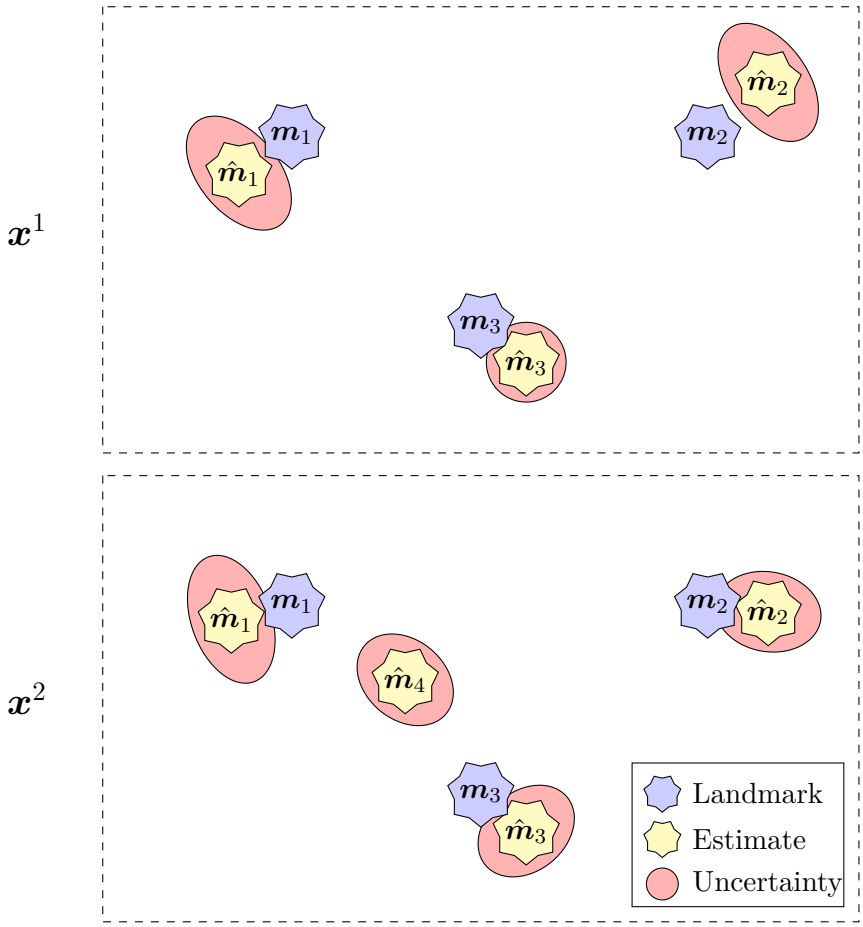


Figure 6.5: A graphical visualization of the problem of computing the final map in FastSLAM. Commonly, due to the noise in both the motion and sensors, particles can containt vastly different maps. Here we see one such example, where one particle believes the map contains 3 landmarks and the other 4.

not possible for several reasons. First, every particle can carry a different number of landmarks. Furthermore, the landmark correspondences between particles are not known. Consider, for example, the case with two particles $\boldsymbol{x}^1$ and $\boldsymbol{x}^2$. Suppose that particle $\boldsymbol{x}^1$ carries 3 landmarks in its map and particle $\boldsymbol{x}^2$ carries 4. We need to decide whether the map estimate should contain 3 or 4 particles and compute the correspondences between the two maps. For a visual representation of this situation, see Fig. 6.5. Since this is intractable for a large number of particles, heuristic algorithms are used. Common heuristic approaches are based on maximum likelihood, K-means, and Gaussian mixture models. As this problem is not directly related to SLAM, a description of these methods may be found in Appendix A.4.1.

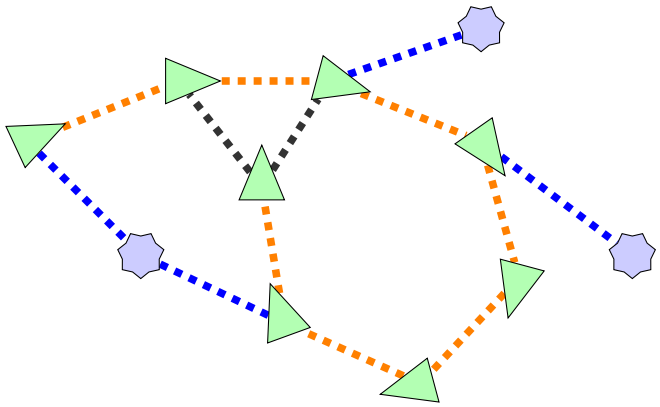


Figure 6.6: As the robot explores the environment, different kinds of constraints may be added to the graph. Using odometry, the relative transformation between successive poses may be constrained (**orange**). By recognizing previously seen areas, the robot may constrain poses which are separated by a long time interval (**black**). Finally, using sensor measuremts the robot constrains the relative position between the current pose and a ladmark (**blue**).

6.7 Graph SLAM

So far, we have discussed two formulations using which the SLAM problem may be solved – EKF-SLAM and FastSLAM. Both of these approaches are an example of online SLAM. That is, they both derive from recursive filtering and estimate only the current posterior without the ability to refine previous estimates. Graph SLAM on the other hand, is an example of offline SLAM. The basic idea behind Graph SLAM is to represent the problem as a graph induced by the robot motion and sensor measurements. This idea is rather natural. Indeed, we have seen that the SLAM problem can be modeled as a Dynamic Bayes Network introducing dependencies between the robot path, map, odometry, and sensor measurements. Graph SLAM was introduced in [57]. However, due to the lack of efficient algorithms, Graph SLAM only gained widespread popularity in the last decade.

Using the odometry information and sensor measurements, Graph SLAM constructs a constraint graph where every node represents an unknown and every edge represents a constraint between a pair of unknowns. In Feature-based SLAM, the unknowns are the robot poses and the landmarks. There are two kinds of constraints in the graph. The first kind constraints the relative transformation between a pair of robot poses. These can be added either through the knowledge of the robot odometry or through recognizing previously seen areas which adds loop closure constraints. The second kind of constraint is that between a pose and a landmark. This edge is generated by a sensor measurement which constraints the relative position of the robot and the observed landmark. These situations are depicted in Fig. 6.6.

We now wish to find a configuration of the poses and landmarks that satisfy these constraints. However, due to the noise present in the sensors and the motion, some constraints may be conflicting. This means that the problem cannot be solved exactly. Instead, we consider the constraints as soft. Let us define $\boldsymbol{z}_{ij}$ as the measurement between two nodes in the constraint graph. If the corresponding constraint is between a pose and a landmark, $\boldsymbol{z}_{ij}$ is the landmark measurement provided by the sensor. If the constraint is between two poses, $\boldsymbol{z}_{ij}$ is then the relative transformation between the two poses computed from the odometry. This is also called a virtual meaurement. We assume that these measurements are normally distributed with uncertainty given by the information matrix $\boldsymbol{\Omega}_{ij}$. Let also $\hat{\boldsymbol{z}}_{ij}$ be the predicted measurement. The log-likelihood of a constraint is then:

$$\mathcal{L}_{ij} \propto (\boldsymbol{z}_{ij} - \hat{\boldsymbol{z}}_{ij})^T \boldsymbol{\Omega}_{ij} (\boldsymbol{z}_{ij} - \hat{\boldsymbol{z}}_{ij}) \tag{6.79}$$

$$= \boldsymbol{e}_{ij}^T \boldsymbol{\Omega}_{ij} \boldsymbol{e}_{ij}, \tag{6.80}$$

where $\boldsymbol{e}_{ij} = (\boldsymbol{z}_{ij} - \hat{\boldsymbol{z}}_{ij})$. Graph SLAM uses the maximum likelihood principle to find the most likely configuration that minimizes the negative log-likelihood of the whole graph:

$$F(\boldsymbol{x}) = \sum_{i,j} \boldsymbol{e}_{ij}^T \boldsymbol{\Omega}_{ij} \boldsymbol{e}_{ij} \tag{6.81}$$

$$= \sum_{i,j} F_{ij}(\boldsymbol{x}). \tag{6.82}$$

Here, $\boldsymbol{x}$ is the state vector. The most likely state is computed as

$$\boldsymbol{x}^* = \arg\min_{\boldsymbol{x}} \; F(\boldsymbol{x}). \tag{6.83}$$

This problem can be solved using nonlinear least-squares minimization by iteratively linearizing the error function. Using a first-order Taylor expansion around $\boldsymbol{x}$, the error is approximated as

$$\boldsymbol{e}_{ij}(\boldsymbol{x} + \Delta x) \simeq \boldsymbol{e}_{ij} + \mathbf{J}_{ij} \Delta x, \tag{6.84}$$

where $\mathbf{J}_{ij}$ is the Jacobian of $\boldsymbol{e}_{ij}$ evaluated at $\boldsymbol{x}$. Substituting 6.81 into 6.84, we get

$$
\begin{aligned}
F_{ij}(\boldsymbol{x}+\Delta x) &= \boldsymbol{e}_{ij}(\boldsymbol{x}+\Delta x)^T \boldsymbol{\Omega}_{ij} \boldsymbol{e}_{ij}(\boldsymbol{x}+\Delta x) && (6.85)\\
&\simeq (\boldsymbol{e}_{ij}+\mathbf{J}_{ij}\Delta x)^T \boldsymbol{\Omega}_{ij}(\boldsymbol{e}_{ij}+\mathbf{J}_{ij}\Delta x) && (6.86)\\
&= \underbrace{\boldsymbol{e}_{ij}^T \boldsymbol{\Omega} \boldsymbol{e}_{ij}}_{c_{ij}} + 2\underbrace{\boldsymbol{e}_{ij}^T \boldsymbol{\Omega}_{ij}\mathbf{J}_{ij}}_{\boldsymbol{b}_{ij}}\Delta\boldsymbol{x} + \Delta\boldsymbol{x}^T \underbrace{\mathbf{J}_{ij}^T \boldsymbol{\Omega}_{ij}\mathbf{J}_{ij}}_{H_{ij}}\Delta x && (6.87)\\
&= c_{ij} + 2\boldsymbol{b}_{ij}\Delta\boldsymbol{x} + \Delta\boldsymbol{x}^T H_{ij}\Delta\boldsymbol{x}. && (6.88)
\end{aligned}
$$

Note that $F_{ij}(\boldsymbol{x}+\Delta x)$ is a quadratic form in Δx. This means that $F(\boldsymbol{x}+\Delta x)$ is also a quadratic form:

$$
\begin{aligned}
F(\boldsymbol{x}+\Delta x) &= \sum_{i,j} F_{ij}(\boldsymbol{x}) && (6.89)\\
&= \sum_{i,j} c_{ij} + 2\boldsymbol{b}_{ij}\Delta\boldsymbol{x} + \Delta\boldsymbol{x}^T H_{ij}\Delta\boldsymbol{x} && (6.90)\\
&= c + 2\boldsymbol{b}^T\Delta x + \Delta x^T H \Delta x. && (6.91)
\end{aligned}
$$

In the last equation, $c = \sum_{i,j} c_{ij}$, $\boldsymbol{b} = \sum_{i,j} \boldsymbol{b}_{ij}$ and $H = \sum_{i,j} H_{ij}$. The minimum of the last expression is found by setting its derivative to zero yielding a system of linear equations:

$$
H\Delta x^* = -\boldsymbol{b}. \tag{6.92}
$$

In order to minimize the negative log-likelihood of the graph, this linearization is solved iteratively until a convergence ciretorion is satisfied. To solve this system efficiently, we may take advantage of the fact that the matrix H is sparse. This is due to the structure of the constraints. Because each constraint only affects two nodes in the graph, the Jacobian of the corresponding error function is zero everywhere except for the two affected nodes. This allows us to compute $\boldsymbol{x}^*$ using sparse solvers using e.g., the sparse Cholesky decomposition of H. The pseudocode of Graph SLAM is provided in Algorithm 9. Note that in this state, the linear system in 6.92 is under determined. This is because the graph contains only relative constraints which connects nodes in the graph. Thus, the error $F(\boldsymbol{x})$ is invariant under rigid transformations applied to $\boldsymbol{x}$. This is solved by fixing the coordinates of one of the nodes in the graph.

In the above description of Graph SLAM, we discussed mainly how to optimize the constraint graph without going into detail how to construct it in

the first place. In the context of graph SLAM, graph construction is usually called the frontend, while graph optimization is called the backend. The backend works with abstract representations of the underlying sensors, which is formalized as a straightforward optimization problem. The frontend part is responsible for processing the sensor data and constructing the constraint graph, which is passed to the backend and optimized. However, the frontend part depends on the specific sensor suite available to the robot and thus cannot easily be described in general. More information about the frontend and Graph SLAM in general can be found in [31] and [13].

Algorithm 9: Graph SLAM

input : Initial guess $\boldsymbol{x}$
Constraint graph $\mathcal{C} = \{(\boldsymbol{e}_{ij}, \boldsymbol{\Omega}_{ij})\}$
output : Optimal configuration $\boldsymbol{x}^*$

```
while not converged do
    H, b ← buildLinearSystem(x, C)
    Δx ← solveSparse(HΔx = −b)
    x = x + Δx
end
x* = x
```

Final notes

In recent years, Graph SLAM has become a very popular algorithm for offline SLAM. Compared to EKF-SLAM and FastSLAM, its power lies mainly in its ability to optimize the whole graph at the same time. This allows it to refine previous estimates and thus reduce global inconsistencies, which is generally not possible with online SLAM. Furthermore, advances in sparse linear solvers make it possible to optimize graphs with hundreds of thousands of nodes and edges using libraries such as GTSAM [19] or G2o [49].

With this, we have covered the three main SLAM algorithms that have seen widespread use - EKF-SLAM, FastSLAM and Graph SLAM. The last section of this chapter is devoted to the problem of data association. Data association is general in SLAM and as such many of its algorithms are applicable to any of the SLAM formulations we have described so far.

6.8 Data association

In the derivations of EKF-SLAM, FastSLAM, and Graph SLAM, we simplified the problem by assuming that the association between the landmarks and measurements was known apriori. In other words, every measurement carries extra information about which landmark generated that measurement. However, the case of known data association is rare in real-world applications. For example, in the Formula Student competition where landmarks are represented by traffic cones delineating the track, it is impossible to accurately distinguish two cones based on the visual characteristics alone. The only difference between the cones is in the color, which helps with rejection but does not provide any information beyond that. Therefore, before a new measurement is incorporated, it is necessary to carry out the data association step. This step has two possible outcomes. Either a measurement is associated to an already existing landmark which is updated, or the measurement belongs to a landmark which has not yet been observed. In such a case, a new landmark is added to the map. The difficulty of data association is exacerbated by the the motion and measurement uncertainty. If the noise is high, the measurements become ambiguous. High motion error especially can result in several different pose hypotheses being likely. These situations are depicted in Fig.6.7 and A.3. A well-performing data association algorithm is necessary to achieve high accuracy since incorrect associations lead to inconsistent maps or even filter divergence. Visual tracking techniques may help with tracking landmark identity in subsequent time steps but are not sufficient for associating landmarks separated by a long time interval, which is crucial for correctly detecting loop closures. In the following sections, we describe some of the common data association algorithms.

6.8.1 Nearest neighbor

The nearest neighbor search is arguably the simplest method for data association [68]. In simple terms, a new measurement is always associated with the nearest landmark. Nearest is understood with respect to a type of distance function. A common choice is either the Euclidean or Mahalanobis distance. Specifically, given a measurement $\boldsymbol{z}_k$ and a set of landmarks $\{\boldsymbol{m}_{k,1}, ...\boldsymbol{m}_{k,M}\}$, the corresponding landmark is selected as follows:

$$\boldsymbol{m}^* = \underset{\boldsymbol{m}\in\{\boldsymbol{m}_{k,1},...\boldsymbol{m}_{k,M}\}}{\arg\min} d(\boldsymbol{z}_k, \hat{\boldsymbol{z}}_j). \tag{6.93}$$

Here, $\hat{\boldsymbol{z}}_j = h(\boldsymbol{m}_j)$, that is, $\hat{\boldsymbol{z}}_j$ is the predicted measurement. Following,

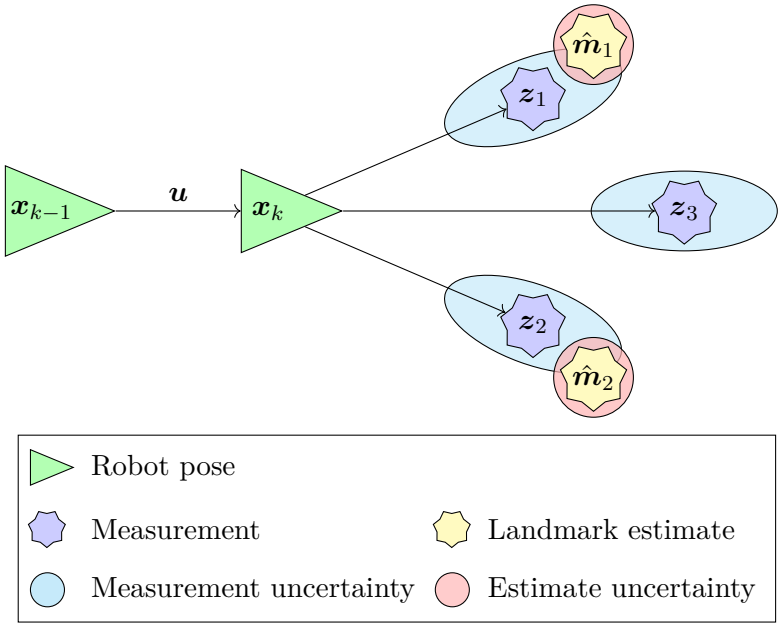


Figure 6.7: An easy instance of the data association problem. Because the sensor uncertainty is realtively low, even a simple greedy algorithm can correctly match measurements to landmarks and decide that the middle measurement constitutes a new landmark.

$d(\boldsymbol{z}_k, \hat{\boldsymbol{z}}_j)$ is the distance function. Substituting the Mahalanobis distance, we get

$$\boldsymbol{m}^* = \underset{\boldsymbol{m}\in\{\boldsymbol{m}_{k,1},\dots\boldsymbol{m}_{k,M}\}}{\arg\min} \quad d(\boldsymbol{z}_k, \hat{\boldsymbol{z}}_j) \tag{6.94}$$

$$= \underset{\boldsymbol{m}\in\{\boldsymbol{m}_{k,1},\dots\boldsymbol{m}_{k,M}\}}{\arg\min} \quad \sqrt{(\boldsymbol{z}_k - \hat{\boldsymbol{z}}_j)^T S_j^{-1} (\boldsymbol{z}_k - \hat{\boldsymbol{z}}_j)} \tag{6.95}$$

$$= \underset{\boldsymbol{m}\in\{\boldsymbol{m}_{k,1},\dots\boldsymbol{m}_{k,M}\}}{\arg\min} \quad (\boldsymbol{z}_k - \hat{\boldsymbol{z}}_j)^T S_j^{-1} (\boldsymbol{z}_k - \hat{\boldsymbol{z}}_j) \tag{6.96}$$

In the above, S_j corresponds to the innovation covariance of the measurement:

$$S_j = H\Sigma_j H^T + R, \tag{6.97}$$

where Σ_j is the landmark covariance, H is measurement Jacobian and R is the measurement noise. The nearest neighbor algorithm searches for a landmark that minimizes the Mahalanobis distance to the given observation $\boldsymbol{z}_k$. In general, the Mahalanobis distance is used to measure the distance from an observation to a set of observations with mean $\boldsymbol{m}_j$ and covariance S_j.

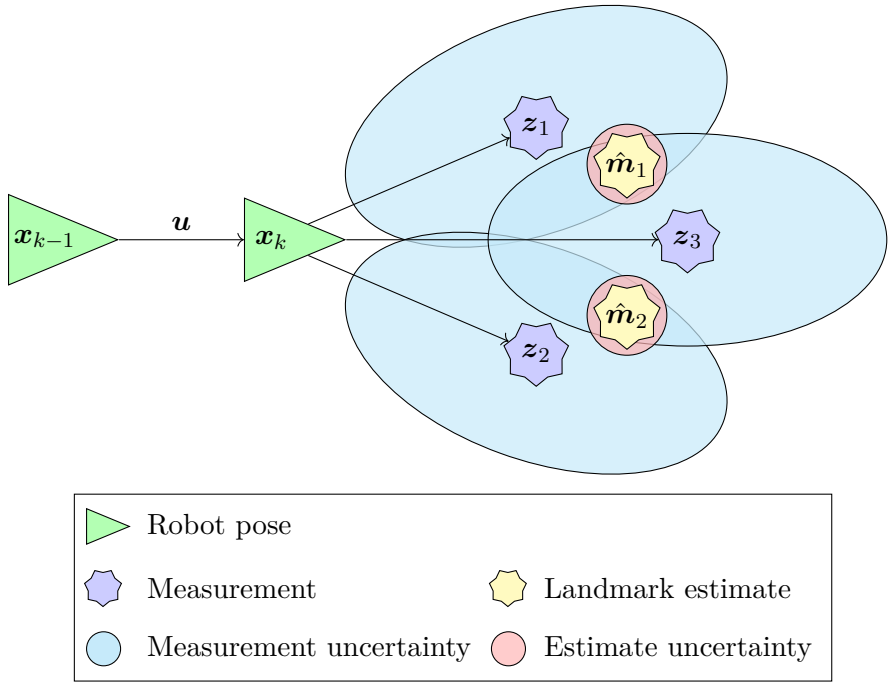


Figure 6.8: A difficult instance of the data association problem. When the measurement error is large, the association becomes ambiguous and simple data association algorithms may fail.

Alternatively, it can also be thought of as a similarity measure between two observations sampled from the same distribution with covariance S_j. The Mahalanobis distance reduces to the Euclidean distance if the covariance matrix S_j is the identity matrix.

The nearest neighbor algorithm is often favoured for its conceptual simplicity and low computational complexity of $\Theta(nM)$, where n is the number of simultaneous measurements and M is the number of landmarks. Space partitioning techniques such as kd-trees bring down the complexity to $\mathcal{O}(nlogM)$ on average. However, care must be taken to keep the tree balanced, otherwise the complexity may degrade to a linear search.

In the nearest neighbor algorithm, a measurement is considered a new landmark when the Mahalanobis distance between the measurement and the landmark is too large. This distance threshold can be defined rigourously by taking advantage of the properties of the Mahalonobis distance. If $d(x, y)$ is the Mahalanobis distance from x to y, it holds that that $d(x, y)^2 \sim \chi^2_n$, i.e., the squared Mahalanobis distance follows the Chi-squared distribution with n degrees of freedom where k is the dimension of the vectors x and y. This fact can be used to assess the individual compatibility by choosing a confidence level α and subsequently creating a new landmark only when $d(x, y)^2 > \chi^2_{n,\alpha}$. Algorithm 10 provides a pseudocode of the nearest neighbor algorithm.

Algorithm 10: Greedy association

input : Measurements $\{\boldsymbol{z}_{k,1}, ..., \boldsymbol{z}_{k,n}\}$
Landmark means $\{\boldsymbol{\mu}_{k,1}, ..., \boldsymbol{\mu}_{k,M}\}$
Landmark covariance matrices $\{\Sigma_{k,1}, ..., \Sigma_{k,M}\}$
Measurement covariance R

output : Data association c_k

```
for i ← 1 to n do
    bestDist = ∞
    bestLandmark = −1
    for j ← 1 to M do
        S_j = H_k Σ_{k,j} H_k^T + R
        d = sqrt((z_{k,i} − ẑ_{k,j})^T S_j^{-1} (z_{k,i} − ẑ_{k,j}))
        if d < bestDist and d^2 < χ^2_α then
            bestDist = d
            bestLandmark = j
    end
    c_i = bestLandmark
end
```

6.8.2 Mutual exclusion

The nearest neighbor algorithm in its simplest form ignores the fact that a sensor typically provides more than one measurement at a time. As distinct measurements must logically originate from different landmarks, two measurements cannot be associated with one landmark. We say that the two assignments mutually exclude each other. The nearest neighbor algorithm can be modified to enforce mutual exclusion as shown in Algorithm 11. Note that this change does not effect the the overall computational complexity of the algorithm. However, the benefit is that mutual exclusion makes it easier to decide when a new landmark should be created. This is because a measurement will not be associated with its closest landmark if the landmark is already taken. This forces association with a landmark which is further away, making it more likely that the measurement becomes a new landmark. On the other hand, this modification makes the algorithm very sensitive to the order in which the measurements are processed, which can lead to vastly different assignments. Moreover, a single incorrect assignment can result in a cascade of even more incorrect assignments due to the mutual exclusion constraint.

To reduce the sensitivity of the algorithm, a better approach is to first compute the distance matrix between the landmarks and the measurements. The elements of the matrix are sorted in ascending order and assigned one by one while enforcing mutual exclusion. This technique is more robust since

Algorithm 11: Improved Greedy association

input : Measurements $\{\boldsymbol{z}_{k,1}, ..., \boldsymbol{z}_{k,n}\}$
Landmark means $\{\boldsymbol{\mu}_{k,1}, ..., \boldsymbol{\mu}_{k,M}\}$
Landmark covariance matrices $\{\Sigma_{k,1}, ..., \Sigma_{k,M}\}$
Measurement covariance R

output : Data assocition c_k

```
for i ← 1 to n do
    c_i = −1
end

D ← n × M matrix      // Compute distance matrix
for i ← 1 to n do
    for j ← 1 to M do
        S_j = H_k Σ_{k,j} H_k^T + R
        D_{i,j} = sqrt((z_{k,i} − ẑ_{k,j})^T S_j^{-1} (z_{k,i} − ẑ_{k,j}))
    end
end

I ← Sort indices (i, j) of D in ascending order of D_{i,j}

for (i, j) in I do
    if c_i = −1 and D_{i,j}^2 < χ_α^2 then
        c_i = j
end
```

it is order-independent. It is however still a greedy algorithm and in many cases provides inferior solutions as shown in Fig. 6.9.

6.8.3 Hungarian algorithm

The data association problem can be posed as an optimization problem minimizing the total cost of the assignment. Consider n simulatenous measurements $\{\boldsymbol{z}_{k,1}, \ldots, \boldsymbol{z}_{k,n}\}$. The cost of the assignment is then:

$$c^* = \arg\min_{c} \sum_{i=1}^{n} d(\boldsymbol{z}_{k,i}, \hat{\boldsymbol{z}}_{k,c_i}). \tag{6.98}$$

For simplicity, we assume that $n \leq M$, that is, there are fewer measurements than landmarks. This formulation makes the data association problem an instance of the possibly unbalanced assignment problem [48], which can

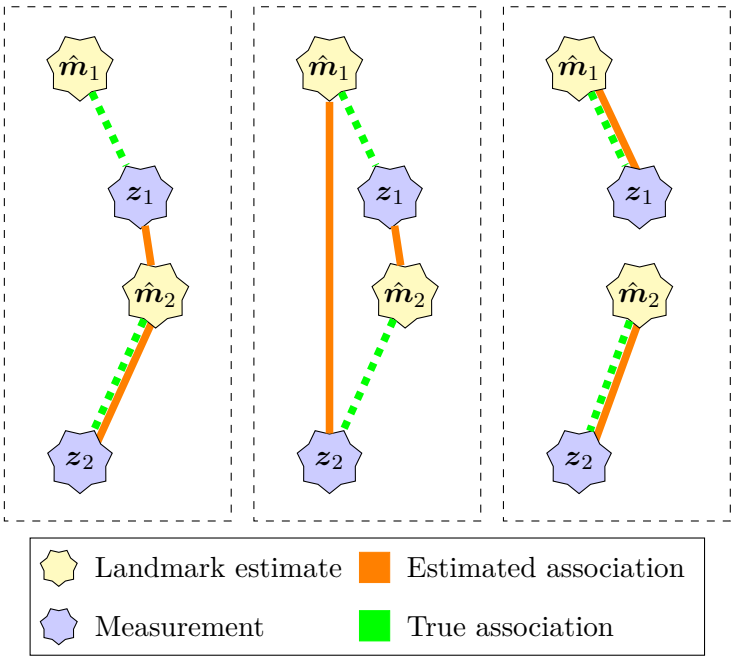


Figure 6.9: A comparison of nearest neighbor 10 (**left**), improved nearest neighbor 11 (**middle**), and the Hungarian algorithm 6.8.3 (**right**). We can see that both variants of nearest neighbor produce incorrect association. Only the Hungarian algorithm is able to match both measurements correctly.

be solved optimally using the Hungarian Algorithm in $\mathcal{O}(\max\{M, n\}^3)$. In a practical implementation, this algorithm is first used to match all measurements to landmarks. Subsequently, a new landmark is created for every unmatched measurement and every measurement where the distance is below the given threshold. See Fig. 6.9 for a visual comparison between the different methods. Other more sophisticated methods, such as the Joint compatibility branch and bound algorithm, are discussed in [68].

6.8.4 Extension to Formula Student

In this section, we provide a simple extension of the data association to the Formula Student competition, in which one can take advantage of the colors of the cones for a more robust association. The rules of Formula Student stipulate that the race track must contain two types of cones. Blue cones are placed on the left side of the track in the driving direction, and yellow cones are placed on the right side. To be precise, there are also small and big orange cones used for marking the start and end positions, but for simplicity, let us consider just the blue and yellow cones since these are by far the most common. If the cone detection system of the autonomous formula provides an information about the color of the measured cone, it is possible to design a more robust data association algorithm that takes the color of the cone

into account. The information about the color of the cones should prevent the algorithm from matching together cones of different colors, which would be otherwise possible if only the position of the cones was considered. We model this problem probabilistically, considering that the color information may not always be accurate. For example, in challenging lighting conditions, a LiDAR-based cone detector can easily mistake the color of the cone. First, we augment the measurement by adding the color information:

$$\boldsymbol{z}_k = (r_k, \phi_k, c_k). \tag{6.99}$$

Here, r and ϕ is the standard range and bearing measurement reported by the detector. In addition, the measurement contains the binary color information $c \in \{$■, ■$\}$. The probability of observing a measurement $\boldsymbol{z}_k$ given that it was generated by a landmark $\boldsymbol{m}_j$ is then

$$p(\boldsymbol{z}_k|\boldsymbol{x}_k, \boldsymbol{m}_j) = p(\boldsymbol{z}_k|\hat{\boldsymbol{z}}_k) = p(r_k, \phi_k, c_k|\hat{r}_k, \hat{\phi}_k, \hat{c}_k). \tag{6.100}$$

This can be further simplified:

$$\begin{aligned} p(r_k, \phi_k, c_k|\hat{r}_k, \hat{\phi}_k, \hat{c}_k) &\overset{def}{=} p(r_k, \phi_k|c_k, \hat{r}_k, \hat{\phi}_k, \hat{c}_k)p(c_k|\hat{r}_k, \hat{\phi}_k, \hat{c}_k) && (6.101)\\ &\overset{Markov}{=} p(r_k, \phi_k|\hat{r}_k, \hat{\phi}_k)p(c_k|\hat{r}_k, \hat{\phi}_k, \hat{c}_k) && (6.102)\\ &\overset{Markov}{=} p(r_k, \phi_k|\hat{r}_k, \hat{\phi}_k)p(c_k|\hat{c}_k) && (6.103) \end{aligned}$$

Here we made and assumption that the measured position only depends on the true position and not the color. Similarly, we also assumed that the measured color only depends on the actual color and not the position of the landmark. The second assumption may not be true in general. The distance from a landmark to the sensor can indeed influence the detected color. The final expression is a product of the standard range and bearing measurement model and the color detection model. Since the color is a binary variable, we only need to estimate $2 \times 2 = 4$ values. We can do this by evaluating the cone detection algorithm on a training dataset and computing the individual probabilities.

So far, we have assumed that the color of the landmark estimate is correct and only the measurement is susceptible to errors. This reasoning is flawed due to the fact that new landmarks are initialized from measurements. If such measurement is incorrect, there is no mechanism to correct it. This issue can be overcome by tracking the probability of the cone color over time.

This is analogous to tracking the landmark probability of existence which is employed in FastSLAM and can again be achieved using a static binary Bayes filter:

$$p(c|\boldsymbol{x}_{0:k}, \boldsymbol{z}_{1:k}). \tag{6.104}$$

The update rule is again expressed in terms of log-odds:

$$\ell(c|\boldsymbol{x}_{0:k}, \boldsymbol{z}_{1:k}) = \ell(c|\boldsymbol{x}_k, \boldsymbol{z}_k) + \ell(c|\boldsymbol{x}_{0:k-1}, \boldsymbol{u}_{1:k-1}) - \ell(c) \tag{6.105}$$

$$\overset{Markov}{=} \underbrace{\ell(c|c_k)}_{\substack{\text{inverse} \\ \text{measurement model}}} + \underbrace{\ell(c|\boldsymbol{x}_{0:k-1}, \boldsymbol{u}_{1:k-1})}_{\text{previous belief}} - \underbrace{\ell(c)}_{\text{prior}} \tag{6.106}$$

The probability $p(c|\boldsymbol{x}_{0:k}, \boldsymbol{z}_{1:k})$ is computed from the log odds using

$$p(c|\boldsymbol{x}_{0:k}, \boldsymbol{z}_{1:k}) = 1 - \frac{1}{1 + \exp(\ell(c|\boldsymbol{x}_{0:k}, \boldsymbol{z}_{1:k}))}. \tag{6.107}$$

For the purposes of data association, we can consider a given landmark blue if $p(c = \blacksquare|\boldsymbol{x}_{0:k}, \boldsymbol{z}_{1:k}) \geq 0.5$ and yellow otherwise. Alternatively, instead of minimizing the Mahalanobis distance, one can instead maximize the measurement likelihood:

$$\boldsymbol{m}^* = \arg\max_{\boldsymbol{m}_j} \; p(r_k, \phi_k|\hat{r}_k, \hat{\phi}_k) p(c_k|\hat{c}_k). \tag{6.108}$$

6.9 Final notes

In this chapter, we discussed the problem of simultaneous localization and mapping. We showed the theoretical formulation of SLAM based on maximizing the posterior probability. Moreover, we described three popular SLAM algorithms with many practical applications – EKF-SLAM, FastSLAM and Graph SLAM. In the last section, we introduced the problem of data association and its relevance to SLAM. In the following chapter, we describe our proposed implementation of FastSLAM on GPUs.

Chapter 7

FastSLAM GPU implementation

In this chapter, we describe our proposed GPU implementation of the FastSLAM 1.0 algorithm, which is the main contribution of the thesis. Despite being intended primarily for use in the Formula Student competition, the proposed implementation is general and is thus suited for any feature-based SLAM problem. The main goal of the implementation is to provide a real-time algorithm which can be deployed on a variety of different GPUs. The overall architecture is designed with speed and efficiency in mind. The reason for this is twofold. First, the implementation is intended to be deployed in an autonomous racing formula which can reach speeds of up to 100 km/h. This necessitates that the algorithm operates at a very high frequency. Second, the physical dimensions of the car, its cooling capability, and the amount of power it can provide limit the choice of the GPU. Due to these constraints, it is not possible to use the best available graphics card. Thus, the algorithm needs to be as efficient as possible in order to attain satisfactory performance even on low-end GPUs.

In the following sections, we discuss the benefits of the GPU programming paradigm and how FastSLAM can benefit from it. In addition, we describe the overall architecture and data structures used in the proposed implementation and explain the specific modifications of the FastSLAM algorithm and the reasoning behind them.

7.1 GPU programming

Graphics processing units offer a viable alternative to CPUs for programs that can be efficiently parallelized. Compared to CPUs which can commonly execute only a handful of threads in parallel, modern GPUs have the ability to execute thousands of threads. Thanks to this, GPUs provide a much higher

instruction throughput and vastly outperform CPUs in parallel workloads, despite the individual GPU cores being typically much slower. Leveraging the power of GPUs is especially enticing with respect to FastSLAM. The basic FastSLAM algorithm is *embarrassingly parallelizable* [36], that is, the algorithm is very easy to parallelize due to a lack of dependencies between the individual particles. Indeed, bar a few collective operations such as weight renormalization, FastSLAM particles are completely independent of each other. Thus, at least in theory, a GPU implementation of FastSLAM should provide an orders of magnitude speedup compared to a CPU implementation. Currently, there exist several computing platforms for GPU programming, most notably, CUDA [70], OpenCL [95] and OpenACC [101]. For our implementation, we chose to use CUDA for its wide adoption and popularity in both the industry and academia.

7.1.1 CUDA

CUDA is a GPU computing platform designed by Nvidia and first released in 2007. At the core, CUDA extends the C++ programming language by adding new syntax and macros, which allows it to target Nvidia GPUs. This way, one can easily mix host (CPU) and device (GPU) code in one program, which is then compiled using the Nvidia compiler. In addition to new language features, CUDA also includes several libraries optimized for parallel programming, such as cuBLAS for linear algebra, cuFFT for fast Fourier transform, and cuDNN for neural network support. Moreover, CUDA is not limited to programming in C/C++. Indeed, CUDA can be used with other programming languages such as Fortran or Python using the PyCUDA library [46].

Thread hierarchy

The basic building block of a CUDA program is the kernel. Kernel is a function intended to be executed on the device, distinguished from regular functions by marking the function signature with a special macro. The kernel function can then be invoked several times in parallel on the device. Before a kernel may be executed, the *execution configuration* must be specified. In simple terms, the execution configuration specifies how many threads should be available to the kernel. The threads are then organized into a thread block. The dimensions of this block determine the total number of threads. For example, a kernel may request to run in a 32×32 block configuration. Within this block, each thread has access to its position within the block. The thread can use its position to determine what part of the problem to work on or what location in memory to access. CUDA supports 1D, 2D, and 3D block configurations which naturally map to arrays, matrices, and

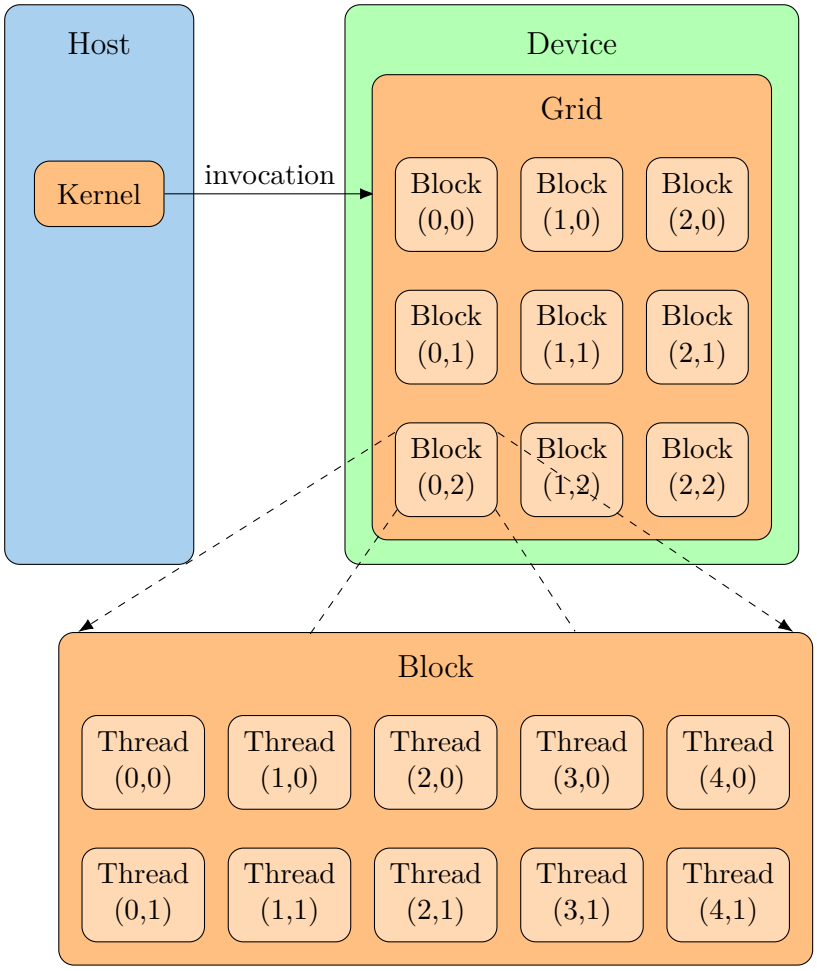


Figure 7.1: A depiction of a kernel invocation. The execution configuration of a kernel defines its grid and block dimensions which together give the total number of threads available to the kernel.

volumes. A CUDA kernel may be invoked with multiple thread blocks. These blocks are grouped into a grid whose dimensions determine the number of blocks. All blocks in a grid must have the same size. Fig. 7.1 depicts the relationship between kernels, grids, blocks, and threads. When a kernel is launched, thread blocks are assigned to the GPU streaming multiprocessors (SM) for execution in a FIFO manner. One SM may execute several blocks at any given time. When all threads in a block are finished, the block is removed from the SM and another block is moved in. Since thread blocks may be executed in any order, it is important that the code is order-independent at the block level. When a thread block is moved to an SM, it is first split into warps which are groups of 32 threads. The warps are then executed separately by warp schedulers. Typically, multiple warps of a single block are executed in parallel. However, the exact number is highly dependent on the specific architecture and on the resources available to the SM at the given time.

Memory model

CUDA defines three main types of memory available to the threads. Every thread has access to its own local memory which holds local variables and cannot be accessed by other threads. Furthermore, all threads in a block also have access to a common shared memory, which can be used to interchange data between threads within a single block. Shared memory is limited by the SM, which affects how many thread blocks may be resident at any given time. Finally, every thread has access to the global memory of the device. The global memory, albeit much larger, is considerably slower than shared memory. If a certain piece of data is to be accessed repeatedly by a thread, a common design pattern is to first copy it from the global memory to the shared memory before manipulating it. It is normally not possible to address the device memory from the host and vice versa. In order to share data between one another, explicit memory transfers are needed. However, newer versions of CUDA implement unified memory, which allows the host and the device to share the same memory address space, eliminating the need for memory transfers.

Synchronization

Parallel algorithms commonly require some level of synchronization between the threads. In CUDA, synchronization is typically needed when multiple threads write to a shared or global memory and need to wait for all write transactions to be visible to all other threads before continuing. In CUDA, this is achieved by several synchronization primitives. At the lowest level, there is the *syncwarp* directive which synchronizes threads within a warp. A level above is *syncthreads* which synchronizes threads within an entire thread block. Up until recently, there was no primitive to synchronize all threads in the whole grid. However, a grid level synchronization can be achieved by splitting the code into several kernels which are launched sequentially. In newer CUDA versions, cooperative groups can be used to achieve the same effect.

Thread synchronization helps prevent race conditions, however, a careless use of synchronization may cause a deadlock in the program. When using *syncwarp* or *syncthreads*, it is important to ensure that all concerned threads will eventually reach the synchronization barrier. If at least one thread takes a different code path due to a branch or an early return, the synchronization will never finish. This is because no thread may continue past the barrier unless all threads have reached it.

Warp divergence

CUDA thread blocks are organized into groups of 32 threads called warps, which are executed separately by warp schedulers. The defining property of a warp is that the threads are executed in lock-step. That is, at any given time, all threads are executing the same instruction. This is optimal as long as the code is branchless. If the code contains a data-dependent branch that is taken by some threads and not others, the warp scheduler needs to execute these two thread groups sequentially one after another. This is called warp divergence and depending on the number of branches present, warp divergence may significantly limit the performance. In the worst case, if every thread in a warp follows a different code path, the code becomes essentially sequential, which also incurs the context switching overhead. To maximize instruction throughput, branching within a single warp should thus be minimized, which reduces the resulting warp divergence.

7.2 Implementation architecture

In this section, we discuss the overall architecture of the implementation, the technologies and libraries used, and some main ideas behind the algorithm and how it is adapted for GPUs. In the following sections, we then explain the specific modifications which differ from the basic FastSLAM 1.0 algrotihm. We only highlight the differences of this implementation rather than explaining the whole algorithm as a thorough explanation was provided in chapter 6. To be specific, the prediction and correction steps remain largely unchanged. However, we describe the used data structures and modifications to data association and resampling.

At the highest level, the implementation is split into two main parts. The frontend, which is written in Python and the backend, which is written in CUDA. The Python frontend handles initialization, data acquisition, GPU instrumentation, and visualization. The CUDA backend, running on a CUDA-capable GPU, is responsible for executing the FastSLAM algorithm itself. See Fig. 7.2 and 7.3 for an illustration.

Using the CUDA terminology, the Python frontend acts as the host. It is responsible for setting up the GPU which involves allocating memory for the data structures and auxiliary buffers needed for the algorithm to operate. In addition, it is responsible for initializing the particles and copying all data to the GPU. The frontend also serves to acquire the robot odometry and measurements. These are preprocessed and copied to the GPU. Finally, the Python frontend extracts the map and pose estimates from the GPU and optionally produces visualizations of the operation of the algorithm. The

Python frontend is designed to either be used as a standalone application, or as a part of a larger autonomous system based on, e.g., the Robot Operating System (ROS) [77]. Indeed, the frontend can be easily adapted to serve as a ROS node. Easy interoperability is the main reason Python was chosen for the frontend as this allows us to seamlessly integrate it into our ROS-based autonomous system. A secondary reason for choosing Python is the abstractions it provides for working with CUDA GPUs. With the help of the PyCUDA library [46], GPU instrumentation is rather simple and straightforward, which reduces errors and increases maintainability.

The CUDA backend implements the FastSLAM algorithm itself. To aid with maintainability and easier debugging, the GPU code is organized into several separate kernels. These kernels are responsible for distinct parts of the algorithm, e.g., prediction, correction, resampling, and several smaller tasks such as weight renormalization. These kernels are invoked sequentially by the frontend. Thanks to the decoupled architecture, the CUDA backend can be used with a different frontend entirely, provided the same calling convention is used. To avoid dynamic memory allocation on the device, all necessary memory is preallocated at the beginning of the algorithm. This includes the memory required to store the particles, auxiliary buffers for resampling, and scratchpad memory used for data association. The FastSLAM algorithm is parallelized per particle, that is, every CUDA thread is responsible for a single particle. This is the most natural and straightforward way to parallelize the algorithm.

Throughout this implementation, memory transfers between the host and the device and vice versa are minimized to maximize performance. Aside from the initial transfer of particles, which can be removed altogether, the only data being copied is the odometry, the measurements, and the pose and map estimates. The FastSLAM algorithm is a stochastic algorithm which makes debugging very challenging. To help with reproducibility, every part of the algorithm which relies on randomness is parametrized by a seed which is used to initialize the random number generators. This means that if the algorithm is rerun with the same seed, it is guaranteed to produce the same results. This technique is essential to ensure the reproducibility of the algorithm.

7.3 Data structures

To take advantage of the massive parallelism provided by modern GPUs, we need an efficient data structure to represent the particles. The data structure has to be designed in a way to support parallel modifications to particles. The data structure should also allow for an efficient manipulation of the robot pose, the weight, and the landmarks. Regarding the map itself, retrieval, insertion, update, and deletion of landmarks should be as efficient as possible

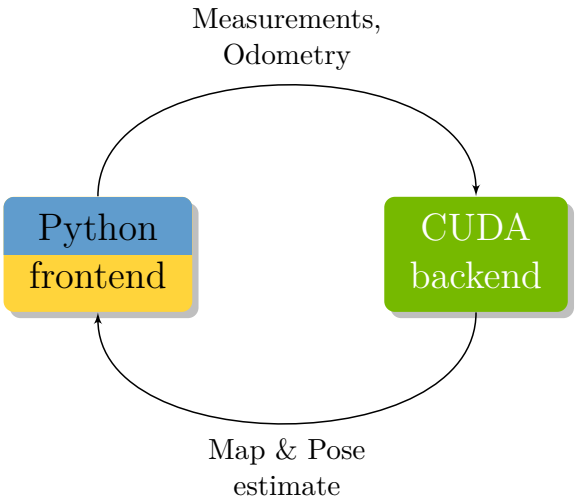


Figure 7.2: The architecture is split into a Python frontend and CUDA backend.

since these operations are very frequent.

Here, we propose a data structure that satisfies all of these requirements. We describe the structure from top to bottom. At the highest level, the data structure is stored in a linear block of memory. This block is then split into individual particles. Each particle stores the robot pose, weight and the map. The map is composed of separate landmarks represented by the EKF means and covariance matrices. The data structure is parametrized by the maximum possible map size. This is the maximum number of landmarks the particles can store. Because the maximum map size is fixed, so is the maximum size of every particle. The maximum size of a particle is a sum of the space required to store the pose, weight and the map. Since every particle has the same fixed size, it is possible to retrieve any particle from the data structure in a constant time by computing its offset from the beginning of the structure. Table 7.1 depicts the whole data structure. Each column in the table represents a single particle. For easier visualization, the structure is shown in a matrix form, although in reality, the memory layout is linear. Each particle contains the robot pose, weight, the current size of the map, and the landmarks themselves. We store the means of the Gaussians first, followed by the the covariance matrices. The means and covariance matrices are again stored at fixed offsets from the beginning of the particle. The means start at offset 4 and the matrices start at $4 + 2 * M$ where M is the maximum map size. This pattern can be extended for any extra information we wish to store in the particle, such as the probability of existence or the landmark color. These constant offsets guarantee that the retrieval, insertion, update, and deletion of any landmark are constant as well. Specifically:

1. Retrieval: To retrieve a landmark or its covariance, simply calculate the corresponding offset into the structure using the landmark ID.

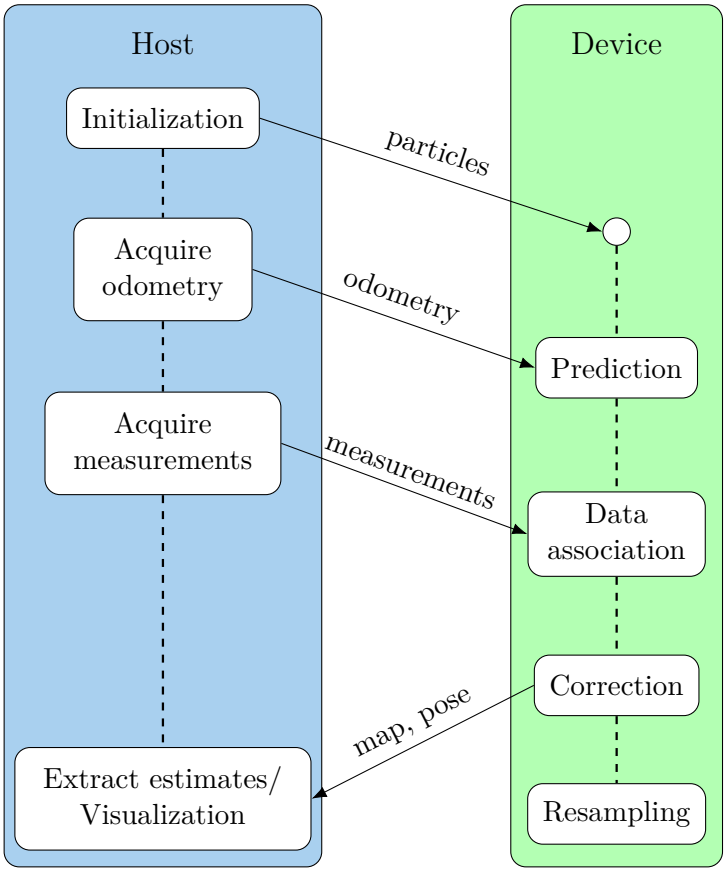


Figure 7.3: A detailed view of the overall architecture. Arrows between the CPU and GPU blocks represent memory transfers.

2. Insertion: To insert a new landmark into the map, write the relevant data to the first available slot after the last landmark and increment the map size M^i.

3. Update: First, retrieve the landmark using the computed offset, update the landmark and store it back in the same place.

4. Deletion: First, compute the landmark offset. Then, swap it with the last landmark in the map and decrement the map size M^i.

It is easily verified that all four of these operations run in $\Theta(1)$. This makes the structure very efficient and flexible. In later sections, we discuss some drawbacks of the linearity of the structure relating to data association. Note that if the real map size grows beyond the maximum limit allowed by the structure, the structure can be resized to accommodate larger maps. By doubling the size of the structure every time it is filled, the amortized time complexity of all operations is still constant. See Table 7.2 for a summary of the operations and their respective asymptotic time complexities.

(a)	x^1	x^2	$\dots$	x^N
	y^1	y^2		y^N
	θ^1	θ^2		θ^N
(b)	w^1	w^2		w^N
(c)	M^1	M^2		M^N
	$\boldsymbol{\mu}_1^1$	$\boldsymbol{\mu}_1^2$	$\dots$	$\boldsymbol{\mu}_1^N$
	$\boldsymbol{\mu}_2^1$	$\boldsymbol{\mu}_2^2$		$\boldsymbol{\mu}_2^N$
	$\vdots$	$\vdots$		$\vdots$
(d)	$\boldsymbol{\mu}_{M^1}^1$	$\boldsymbol{\mu}_{M^2}^2$		$\boldsymbol{\mu}_{M^N}^3$
	$\mathbf{0}$	$\mathbf{0}$		$\mathbf{0}$
	$\vdots$	$\vdots$		$\vdots$
	$\mathbf{0}$	$\mathbf{0}$		$\mathbf{0}$
	Σ_1^1	Σ_1^2	$\dots$	Σ_1^N
	Σ_2^1	Σ_2^2		Σ_2^N
	$\vdots$	$\vdots$		$\vdots$
(e)	$\Sigma_{M^1}^1$	$\Sigma_{M^2}^2$		$\Sigma_{M^N}^N$
	$\mathbf{0}$	$\mathbf{0}$		$\mathbf{0}$
	$\vdots$	$\vdots$		$\vdots$
	$\mathbf{0}$	$\mathbf{0}$		$\mathbf{0}$

Table 7.1: The memory layout of the particles data structure. Each column represents a single particle. Each particle contains the state (a), weight (b), the current map size (c), landmark means (d) and covariance matrices (e).

7.4 Data association

Our implementation uses the nearest neighbor algorithm (see Alg. 10) described previously to associate measurements with landmarks. This choice stems on the one hand from the fact that data association is computationally intensive and thus to achieve real-time performance, a simpler but more effi-

		Fixed	Dynamic (Amortized)
Particle	Retrieval	$\Theta(1)$	$\Theta(1)$
	Update	$\Theta(1)$	$\Theta(1)$
Landmark	Retrieval	$\Theta(1)$	$\Theta(1)$
	Update	$\Theta(1)$	$\Theta(1)$
	Insertion	$\Theta(1)$	$\Theta(1)$
	Deletion	$\Theta(1)$	$\Theta(1)$

Table 7.2: The time complexity of operations on the proposed data structure. Note that for the case in which the data structure is allowed to grow, the listed time complexity is amortized over n consecutive operations.

cient algorithm is preferred. On the other hand, since FastSLAM computes the data association on a per-particle basis, it is sufficient to use a simple algorithm as unlikely associations will be resampled away.

Assuming a single measurement, the time complexity of a naive nearest neighbor implementation is linear in the map size since the distance to every landmark has to be computed. A logarithmic time complexity may be achieved by using a tree data structure to store the map. This is difficult to implement efficiently in practise for several reasons. The tree needs to support a nearest neighbor search using Mahalanobis distance. For the Euclidean distance, k-d trees may be used. To be more efficient than the naive implementation, the tree needs to support a logarithmic time retrieval, insertion, and deletion in the worst case or at least the average case. Furthermore, the tree needs to be self-balancing or be able to rebalanced in at most logarithmic time as well. If the tree becomes unbalanced, which may be caused by the robot exploring a new area, the time complexity of all operations degrades to linear. Moreover, if mutual exclusion is enforced, the tree may need to provide multiple nearest neighbors in a single query.

Variants of a K-D-B tree [82] or a Balanced k-d tree [12] may potentially be used, however, due to the complexity involved in implementing such a structure efficiently on a GPU, we opted to use simpler heuristic optimizations instead. To avoid computing the distance to every landmark, we first filter landmarks that are within the perceptual range of the sensor. Only these filtered landmarks are then considered as a potential correspondence. This filtering step still has a linear complexity, however, if multiple simultaneous measurements need to be processed, the filtering step is executed only once, which spreads the cost over the measurements. Using the Big O notation, the complexity of the original naive impementation is $\mathcal{O}(ZM)$, where Z is the number of simultaneous measurements and M is the map size. Employing the

sensor range heuristic, the complexity becomes $\mathcal{O}(M + ZR)$ where R is the number of landmarks within the perceptual range of the sensor. Assuming the sensor range is constant, the complexity becomes $\mathcal{O}(M + Z)$ which is linear in both the map size and the number of measurements. Further heuristic improvements are possible. If the perceptual range is enlarged sufficiently compared to the speed of the robot, the filtering step need not be computed in every time step. This is because if the range is large enough, the original filtered set is still valid even after several time steps. This again helps spread the filtering cost over multiple time steps. Unfortunately, these techniques only reduce the constants in the overall complexity but are fundamentally still linear as opposed to the logarithmic cost of the tree implementation.

7.5 Resampling

The implementation uses parallel systematic resampling to resample particles (see Alg. 20). This resampling algorithm combines the low variance of the ancestor vector together with the speed of parallel resampling. Furthermore, unlike stratified resampling which has similar properties, systematic resampling requires to sample only a single random number. In chapter 5, resampling is described as the problem of constructing the ancestor vector which satisfies certain properties such as the unbiasedness condition (5.21). However, in a real implementation, resampling entails a second step. In the second step, the data structure holding the particles has to be physically copied in accordance with the ancestor vector. The most straightforward way to implement this is to maintain two buffers. A source buffer is used to hold the old particles which are copied into the destination buffer. Once the particles are copied, the buffers are swapped with the source becoming the destination buffer and vice versa. This implementation essentially doubles the memory requirements of the algorithm as it requires an auxiliary buffer.

Authors of [64] describe a more efficient way in which the particles are copied only when needed. This approach helps decrease the required memory significantly. However, due to the complexity of the algorithm, combined with the requirement of thread safety given by the GPU, we decided to forgo this implementation in favor of a simpler option which copies all particles but needs only a single buffer. This technique is called in-place resampling [66].

7.5.1 In-place resampling

In-place resampling requires only a single buffer, which reduces the memory requirements by half, compared to the source-destination buffer method. A minor complication is that given an arbitrary ancestor vector, it is generally

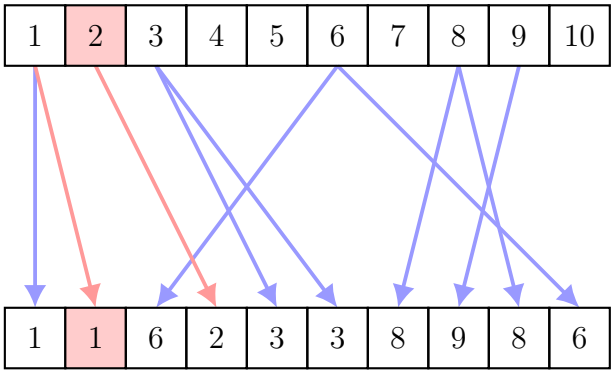


Figure 7.4: For some permutations of the ancestor vector **(bottom)**, particles are both written to and read from **(top)**. The red arrows demonstrate a conflict with the particle $\boldsymbol{x}^2$. Based on the ancestor vector, the particle is to be copied to position 4. At the same time, particle $\boldsymbol{x}^1$ writes to the position of particle $\boldsymbol{x}^2$ leading to a possible race condition.

not possible to copy the particles in-place in parallel. Consider the case shown in Fig. 7.4. The arrows are given by the ancestor vector and represent into which place the given particle should be copied. If there are multiple arrows originating from a particle, the particle is copied several times in the new generation. Looking at particle x^2, which is highlighted in red, we see that it should be copied to index 4, that is, $a_4 = 2$. At the same time, particle x^1 should be copied to index 2. If we copy all particles in parallel, the memory location of particle x^2 will be both written to and read from simultaneously. This is a read-write conflict and is an undefined behavior in CUDA. This race condition stems from the fact that multiple threads may access the same memory location at the same time, which would not be the case if the particles were copied sequentially. This problem can be remedied, however. As shown by [66], it is always possible to permute any ancestor vector such that all read-write conflicts are eliminated. Such an ancestor vector is called *conflict free*. Note that permuting the ancestor vector has no effect on the resampling itself. The permuted ancestor vector is then used to copy the particles. An ancestor vector is said to be conflict-free, provided the following sufficient condition holds:

$$o_i > 0 \implies a_i = i. \tag{7.1}$$

Here, $\boldsymbol{o}$ is the offspring vector and $\boldsymbol{a}$ is the ancestor vector defined in 5.1 and 5.2, respectively. In other words, if a particle has at least one offspring, one of the offsprings has to be copied to the same position as the original particle. This ensures a stable location from which the particles can be copied without being overwritten which eliminates race conditions. An ancestor vector can always be permuted such that (7.1) holds. This is shown in Algorithm 12. The permute algorithm works by sequentially reading a_i and

verifying that it is in the correct location, that is, $a_i = i$. If not, it checks if the correct location given by a_{a_i} already contains the same particle. If that is also false, the algorithm swaps the two positions, fixing one particle. The algorithms continues until it scans the whole vector at which point, the vector is conflict-free. The permute algorithm is sequential and runs in $\mathcal{O}(n)$. Authors of [66] propose a modification of this algorithm which is readily parallelized. An explanation of this algorithm is provided in Appendix A.5.1. In our implementation, we use the parallel algorithm.

Algorithm 12: Permute

input : An ancestor vector $\{a_1, \ldots, a_N\}$

```
1
2 for i ← 1 to N do
3     if a_i ≠ i and a_{a_i} ≠ a_i then
4         swap(a_i, a_{a_i})
5         i = i − 1 // repeat with a new value
6 end
```

7.6 Numerical stability

When implementing the FastSLAM algorithm, one needs to pay close attention to numerical issues which may arise as a result of the floating point arithmetic. Floating point is a finite precision arithmetic and especially for very large or very small numbers, the precision may degrade significantly depending on the specific floating point implementation used. FastSLAM contains several spots where numerical instability may effect the behavior and accuracy of the algorithm.

The first two spots relate to the weight computation. In both versions of FastSLAM, the particle weights are computed from a Gaussian PDF. If the particle is highly unlikely, the computed weight will be very small. Depending on the precision, if the weight is small enough, it may be rounded to zero. If all particle weights are rounded this way, the weight renormalization introduces a division by zero. To prevent this, an explicit check for a zero or near-zero sum has to be added. If such an event occurs, the particle weights should be reset to uniform weights. The second spot is related to the weight normalization. To renormalize the weights, the sum has to be computed. As most weights tend to be close to zero, summing will inadvertently introduce round-off errors which will accumulate as all weights are added. The final step of normalization is dividing the weights by this sum. If the sum itself is close to zero, the resulting weight will be imprecise. As a result, the sum of the normalized weights may differ from 1 enough to introduce bias in the resampling step.

Another critical spot is the resampling itself. In this implementation, we opted to use parallel systematic resampling. This resampling algorithm needs to first compute the cumulative sum. Due to the round-off error which is exacerbated by the repeated sums, the partial results may be imprecise, especially towards the end of the cumulative sum. This can again introduce bias in the resampling algorithm.

The last critical spots are matrix inverses. There are several places in the algorithm where the matrix inverse is computed. Namely, the EKF equations and the importance weight. If the original matrix is close to singular, several problems may arise. First, when computing the importance weight, the determinant of the covariance matrix may be negative due to rounding errors. This may pose a challenge since the Gaussian PDF needs to evaluate the square root of the determinant. Second, the resulting matrix inverse may be relatively imprecise or even infinite if the determinant is close to zero.

To prevent or atleast partially decrease the impact of these problems arising from the imprecision of floating point arithmetic, all floating point computations are carried out in double precision IEEE-754 floats. The machine epsilon of a double precision float is approximately 10^{-16} compared to the single precision epsilon of 10^{-8}. The increased precision should be sufficient to prevent any numerical issues. The drawbacks of this decision are increased memory requirements and slower execution of computationally demanding sections of the algorithm.

7.7 Discussion

In this chapter, we described our proposed implementation of a real-time SLAM system using the FastSLAM 1.0 algorithm. The implementation is split into the *frontend* handling data aquisition and preprocessing, and the *backend* which is executed on a GPU and implements the actual SLAM algorithm. We explained the most important modifications to the basic FastSLAM algorithm which was fully described in chapter 6. These modifications include a design of an efficient, yet simple data structure to represent the particles, a data association algorithm based on the nearest neighbor, and a modification to resampling which cuts memory requirements by half. A common theme in this implementation is that simplicity is preferred over complexity as long as the performance loss is not too severe. Specifically, we chose to use less efficient but simpler algorithms for both data association and resampling. This is deliberate as it keeps the codebase simple and allows for rapid changes even by people less familiar with the problem of SLAM. In addition, as the experiments show in the next chapter, the performance of the proposed implementation is more than sufficient for a real-time SLAM system.

Chapter 8

Experimental results

In this chapter, we evaluate the proposed implementation on a variety of simulated and real-world datasets. Specifically, we measure the accuracy of the proposed implementation as a function of the number of particles used. Both known and unknown data association is considered. In addition to the accuracy, we evaluate the performance of the algorithm, the memory requirements, and its suitability for real-time applications. Most importantly, we evaluate the applicability of our implementation for the Formula Student competition. Finally, we investigate the scaling properties of the implementation to identify possible bottlenecks and discuss further improvements.

8.1 Accuracy statistics

To properly compare and evaluate SLAM algorithms, a suitable statistic for measuring accuracy is needed. Unfortunately, a standard statistic does not exist in the SLAM community. Perhaps the most common statistic is the mean square error (MSE) of the estimated robot path. This statistic intentionally ignores the map in assessing the overall accuracy. The reasoning behind this is justified as follows. In general, a small error in the robot pose implies a small error in the map. This is because a precise robot localization largely depends on a correctly estimated map. We can apply this argument in the opposite direction as well. That is, an incorrect map implies an incorrect path. Thus, in principle, statistics that only evaluate the accuracy of the robot path, such as the MSE, ought to be sufficient in providing insight about how well a given algorithm performs. Another reason for preferring statistics that assess only the path error is that it is not at all obvious how to combine the path error with the map error into a single, informative statistic. Moreover, assessing the map quality is difficult in itself. This is because to compute the error, the estimated landmarks need to be associated with the

ground truth. Moreover, the map estimate may contain a different number of landmarks, which necessitates extending the statistic to quantify false positive and false negative landmarks. For these reasons, restricting the error statistic to the path error is usually preferred. Consider the standard MSE definition below:

$$\varepsilon_{MSE}(\boldsymbol{x}_{1:K}) = \frac{1}{N}\sum_{k=1}^{K} ||\boldsymbol{x}_k - \boldsymbol{x}_k^*||^2, \tag{8.1}$$

where $\boldsymbol{x}_{1:K}$ is the estimated robot path $\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_K\}$, $\boldsymbol{x}_{1:K}^*$ is the true robot path $\{\boldsymbol{x}_1^*, \ldots, \boldsymbol{x}_K^*\}$ and $(\boldsymbol{x}_k - \boldsymbol{x}_k^*)$ computes the relative transformation δ_{k,k^*} between the two poses such that

$$\boldsymbol{x}_k + \delta_{k,k^*} = \boldsymbol{x}_k^*. \tag{8.2}$$

If the transformation includes rotations, the angle differences are normalized. Since the robot position and orientation have different scales, it is better to measure the error of the translational component and the rotational component separately. As the authors of [50] point out, statistics that rely on the global reference frame, of which MSE in an example, may be misleading. This is due to the fact that a small mistake at the start of the path will have an additive effect on the error for all subsequent poses. Borrowing an example from [50], consider a robot moving in a straight line. At the time step 1, the robot overestimates its position by e. However, in the subsequent $K-1$ time steps, the robot estimates its relative movement correctly. This situation is depicted in Fig. 8.1. Now, if we measure the MSE of the robot path, the error is $K \times e$, despite the robot only making a mistake at the beginning. Moreover, if we were to measure the error starting at the end and going back, the error would be only e. Similar situation arises with the rotational errors of the robot heading shown in the same figure.

As we have just shown, MSE depends on the coordinate frame and thus tends to compound errors over time, which may skew the results. Authors of [50] instead propose to compare only the relative poses. That is, instead of computing the difference of a pair of poses, we compute the difference between the relative displacements of two pairs of poses. Mathematically, we have

$$\varepsilon_{Rel}(\boldsymbol{x}_{1:K}) = \frac{1}{N}\sum_{i,j \in S} ||\delta_{i,j} - \delta_{i,j}^*||^2, \tag{8.3}$$

where

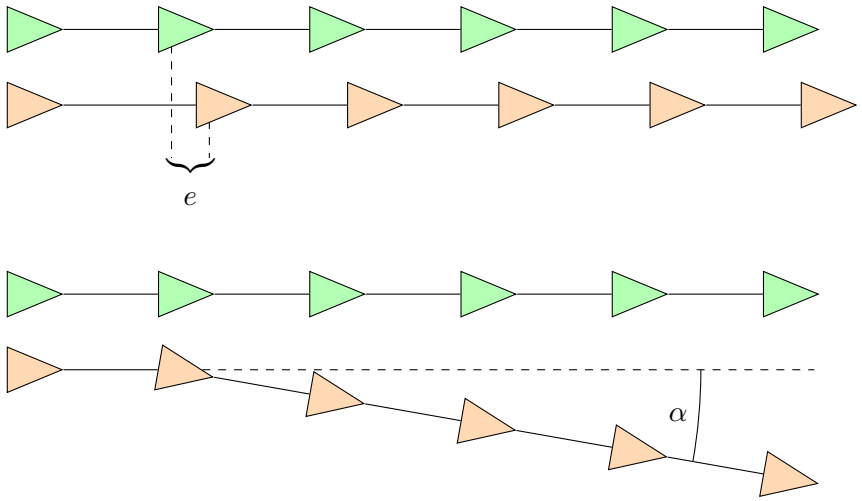


Figure 8.1: An example of the estimated robot path where the MSE statistic is misleading. If the robot makes a translational (**top**) or a rotational error (**bottom**) at the beginning of the path, the error is carried over into the subsequent time steps.

$$\delta_{i,j} = \boldsymbol{x}_j - \boldsymbol{x}_i \tag{8.4}$$
$$\delta^*_{i,j} = \boldsymbol{x}^*_j - \boldsymbol{x}^*_i. \tag{8.5}$$

In the above, $\delta_{i,j}$ is the relative displacement between the pose $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$. Note that the above definition leaves the choice of the relative displacements up to us. The choice of pairs is given by the set S. Depending on which displacements are chosen, different properties of the resulting map can be measured. For example, one can choose to only compare consecutive or nearby poses. Such statistic would in turn measure the local consistency of the map. Adding loop closure links adds information about the quality of loop closures. Finally, adding links between poses that are far away from each other indicates global consistency. These situations are depicted in Fig. 8.2. In the experiments that follow, for every dataset, we report both the MSE and the relative error (8.3) with manually chosen poses. In addition, we report both the translational and rotational error of both of these statistics separately. The translational error is given in meteres and the rotational error is given in degrees.

8.2 Simulated dataset

We start by evaluating our implementation on a simulated dataset. We consider a robot moving in a virtual 2D environment. The dimensions of the

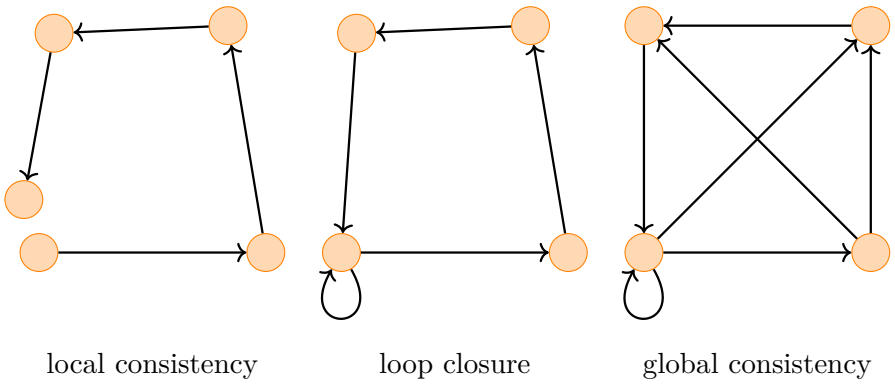


Figure 8.2: Depending on which relative displacements are chosen, different properties of the map may be highlighted.

environment are 20×20 meters. The robot is equipped with a sensor with a limited range. The environment is relatively sparse and consists of twenty landmarks. The robot is given commands to move to random waypoints in the environment and eventually returns back to the starting position. This means that the SLAM algorithm has to correctly handle a loop closure. Fig. 8.3 shows the environment together with the robot path. The robot uses the following nonlinear motion model:

$$\begin{bmatrix} x_k \\ y_k \\ \theta_k \end{bmatrix} = \begin{bmatrix} x_{k-1} + v\cos(x_k)dt \\ y_{k-1} + v\sin(y_k)dt \\ \theta_{k-1} + \omega dt \end{bmatrix}, \tag{8.6}$$

where v is forward velocity and ω is angular velocity of the robot. Zero mean Gaussian error is added to both the robot motion and the sensor which captures the range and bearing of visible landmarks. The sensor range is limited to four meters and the field of view is 135°. We evaluate our implementation using both known and unknown data association. The accuracy and performance are compared with an implementation provided by Python Robotics [84]. Python Robotics is an open-source collection of robotics algorithms, mainly concerned with navigation, localization, mapping, and SLAM, intended to be a source of practical learning material for widely used robotics algorithms. The Python Robotics implementation of FastSLAM 1.0 merely serves as a baseline for the accuracy of our algorithm. Moreover, since it is implemented in Python without the use of parallelization, we can use it to judge the performance of our GPU implementation compared to a purely CPU-bound algorithm.

We report both the MSE and the relative error (8.3). The chosen relative displacements for the relative error are consecutive waypoints of the robot

Particles	MSE(trans)	MSE(rot)	Rel(trans)	Rel(rot)
4	0.35 ±0.28	1.09 ±0.97	0.39 ±0.30	0.85 ±0.32
8	0.18 ±0.14	0.48 ±0.32	0.18 ±0.09	0.40 ±0.23
16	0.14 ±0.11	0.38 ±0.42	0.12 ±0.10	0.31 ±0.18
32	0.12 ±0.11	0.23 ±0.22	0.08 ±0.06	0.20 ±0.14
64	0.07 ±0.04	0.20 ±0.16	0.06 ±0.04	0.12 ±0.05
128	0.08 ±0.08	0.13 ±0.12	0.05 ±0.04	0.13 ±0.07
256	0.09 ±0.06	0.17 ±0.16	0.06 ±0.05	0.10 ±0.06
512	0.05 ±0.03	0.07 ±0.06	0.03 ±0.02	0.09 ±0.05
1024	0.06 ±0.05	0.09 ±0.10	0.04 ±0.03	0.09 ±0.05

Table 8.1: Accuracy on the first simulated dataset with known data association

Particles	MSE(trans)	MSE(rot)	Rel(trans)	Rel(rot)
4	11.74 ±6.99	26.20 ±21.25	7.32 ±5.10	10.61 ±5.72
8	5.60 ±5.96	12.23 ±30.46	3.72 ±5.88	5.61 ±8.44
16	3.12 ±3.95	6.90 ±10.33	2.11 ±2.32	2.63 ±2.12
32	1.86 ±2.33	4.52 ± 5.50	1.64 ±1.86	2.29 ±2.50
64	0.52 ±1.56	1.27 ± 3.63	0.47 ±1.44	0.72 ±1.82
128	0.15 ±0.29	0.26 ± 0.55	0.10 ±0.16	0.18 ±0.14
256	0.06 ±0.05	0.12 ± 0.10	0.06 ±0.05	0.12 ±0.09
512	0.07 ±0.08	0.13 ± 0.16	0.06 ±0.05	0.11 ±0.07
1024	0.08 ±0.07	0.18 ± 0.23	0.06 ±0.06	0.13 ±0.11

Table 8.2: Accuracy on the first simulated dataset with unknown data association

path. We added a loop closing displacement connecting the first and the last waypoint which have the same position. This helps assess how well the algorithm can handle loop closures. Fig. 8.4 shows an example of the estimated robot path and the map using our implementation with unknown data association. The accuracy is measured for an increasing number of particles and then averaged over 20 different runs due to the stochastic nature of the algorithm. Table 8.1 and 8.2 show the complete results for known and unknown data association, respectively. Fig. 8.5 shows the translational MSE of our implementation compared to the Python Robotics implementation. For unknown data association, an increased number of particles helps reduce the error significantly. Notice that for known association, significantly fewer particles are required to achieve the same level of accuracy. Due to the single-threaded nature of the Python Robotics implementation, we were unfortunately unable to measure the accuracy beyond 128 particles due to the time requirements. See 8.5 for a detailed performance analysis.

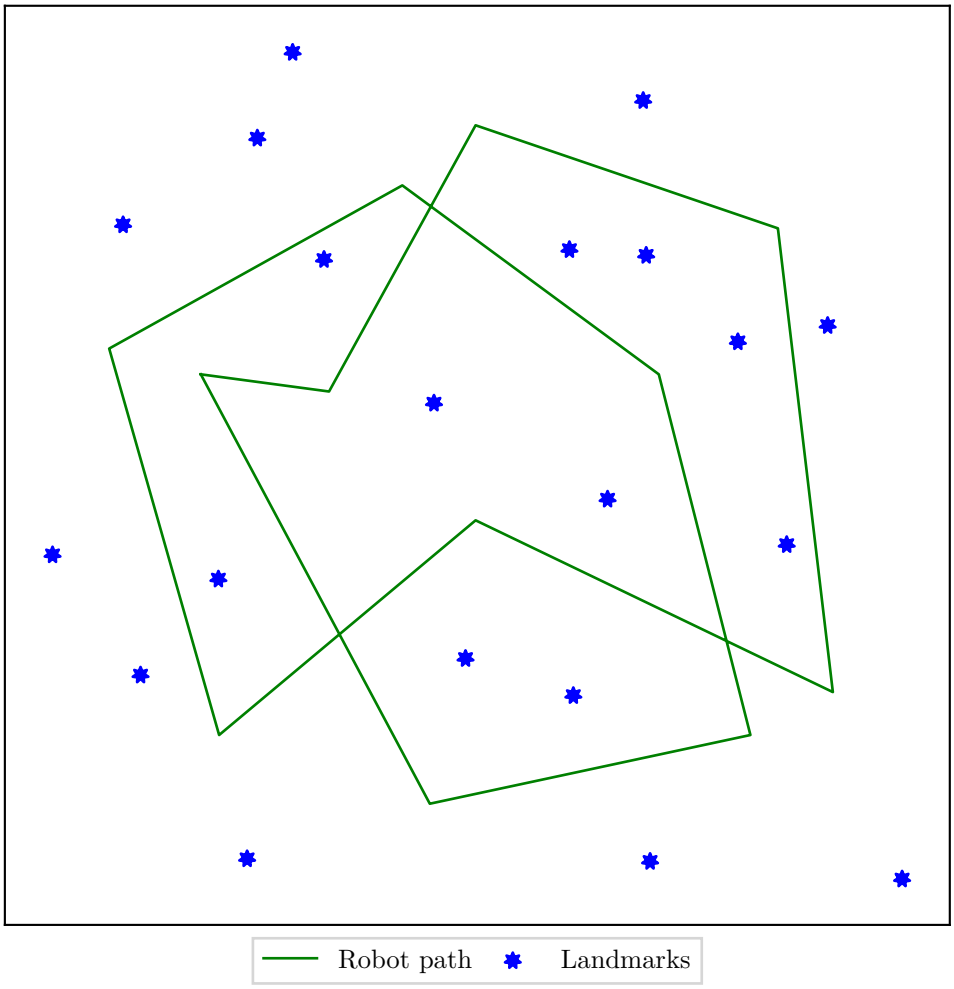


Figure 8.3: The virtual environment of the simulated dataset

8.3 FS Online

The SLAM implementation proposed in the thesis is primarily intended to be used in the Formula Student competition. As such, the implementation should be evaluated on a dataset resembling the racing events of Formula Student. Unfortunately, due to the Covid-19 pandemic, we were not able to collect a comprehensive dataset from our formula. However, following the FS Online competition in 2020, we gained access to the simulation environment provided by the competition organizers. The simulator, called Formula Student Driverless Simulation (FSDS) [1], is built on top of Unreal Engine [86], AirSim [92] and the Robot Operating System [77].

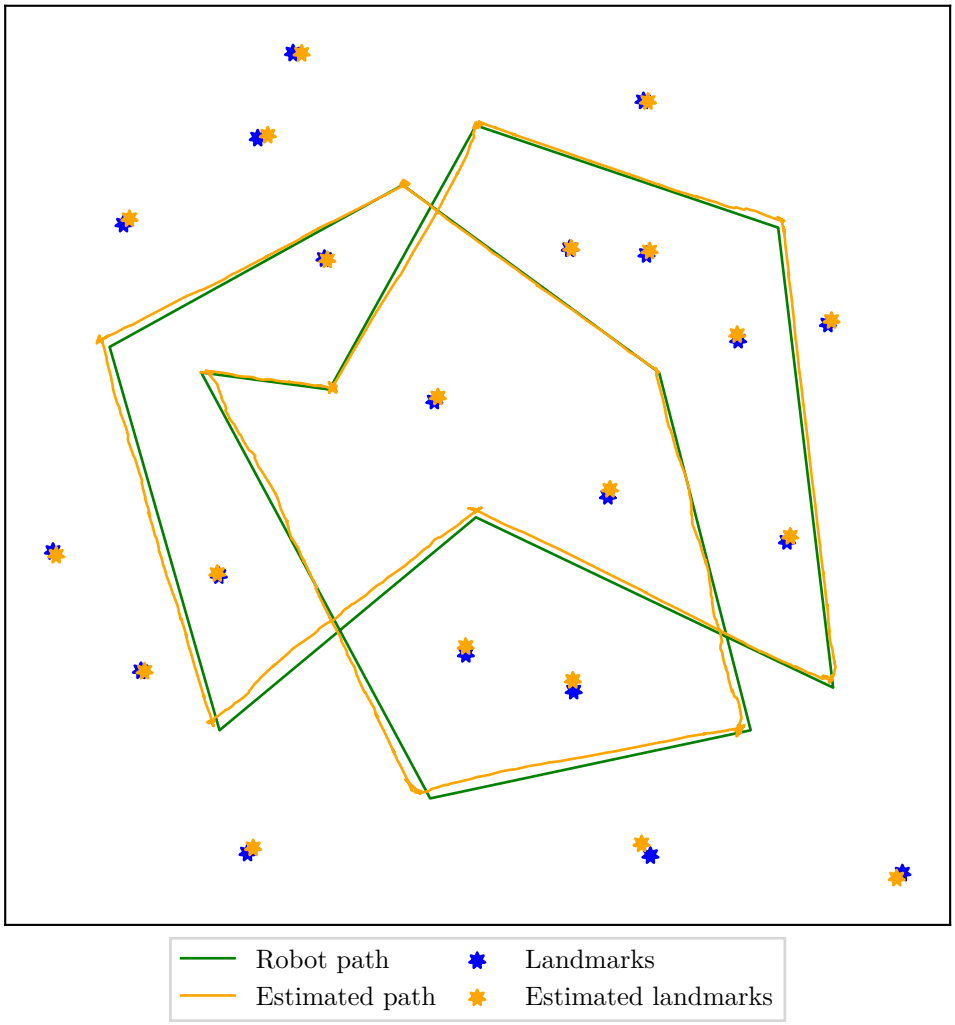


Figure 8.4: An example of the estimated robot path and the final map on the simulated dataset

We used FSDS to collect data from a formula completing one lap in a virtual track which was used in the FS Online competition. The track, shown in Fig. 8.7, is approximately 400 meters long and is delineated by 200 traffic cones. Even though this dataset is simulated, it is of great importance for evaluating the proposed implementation. This is because the virtual track closely resembles the tracks used in real racing events in both size and topology. To collect the data, we used the same sensor setup and algorithms that we competed with in the FS Online event. The virtual sensor suite consists of a ground speed sensor (GSS), an inertial measurement unit (IMU), a GPS unit, and a LiDAR sensor which are all implemented by AirSim based on real sensors. Using a GSS, IMU and GPS, a pose estimate of the formula was computed. Thanks to the information from the GPS, the problem of loop

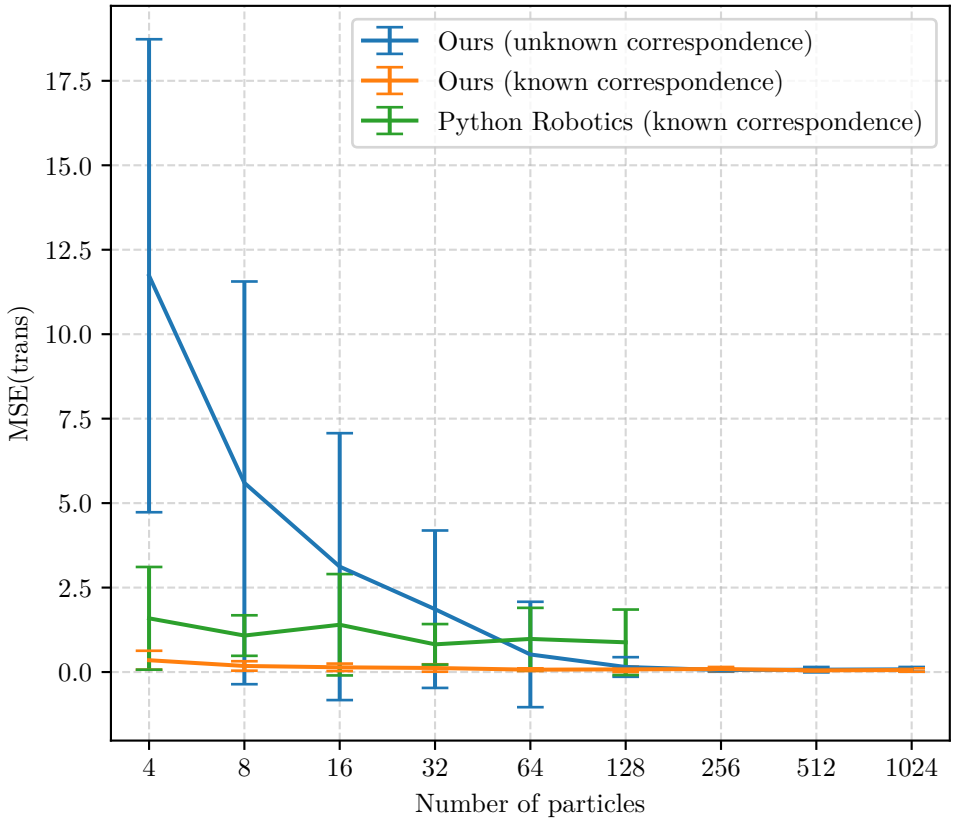


Figure 8.5: A graph showing the translational MSE on the simulated dataset as the number of particles increases

closure and global consistency was made significantly easier. This is because with the GPS, we had access to the (noisy) global pose of the car. This is similar to the real Formula Student events, which do not forbid the use of a GPS and many teams take advantage of this. Once the pose was estimated, a clustering algorithm was used to detect the traffic cones from the LiDAR pointclouds. Due to a large uncertainty in the heading of the formula, the angle component of the measurements was relatively imprecise as is shown in Fig. 8.8. This figure shows the histogram of all measurements taken in this particular area together with the true landmark positions. In the figure, we can see that the range component is relatively much more precise compared to the bearing component.

We again evaluated our implementation for an increasing number of particles, measuring the average MSE and the relative error over 20 runs for every setting. Since global consistency is guaranteed by the GPS, we used only consecutive poses for the computation of the relative error. Fig. 8.9 shows an example of the final path and map estimate. We compare both known and unknown data association. The complete results are summarized in Table

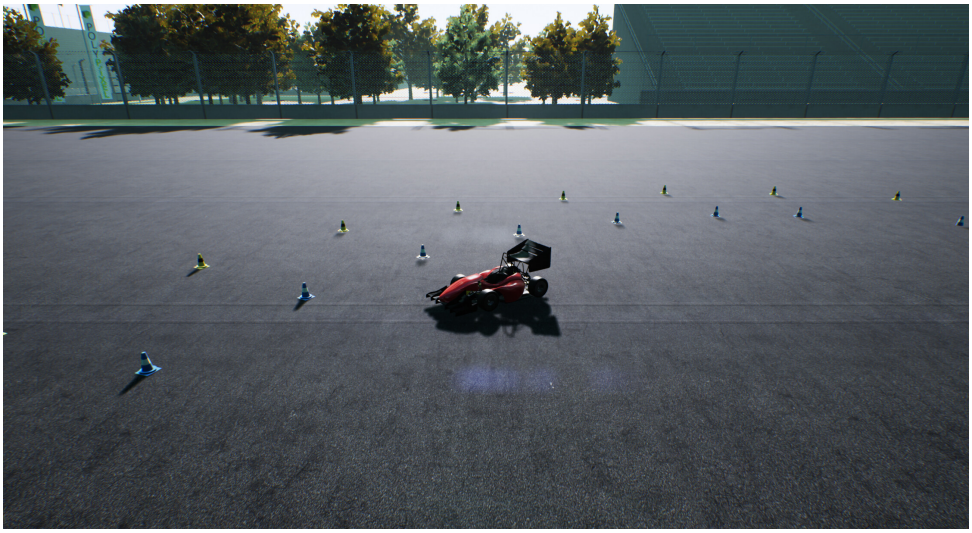


Figure 8.6: The Formula Student Driverless Simulation [1]

Particles	MSE(trans)	MSE(rot)	Rel(trans)	Rel(rot)
4	0.15 ±0.01	0.39 ±0.02	0.16 ±0.01	0.74 ±0.04
8	0.13 ±0.02	0.27 ±0.03	0.10 ±0.01	0.47 ±0.04
16	0.12 ±0.02	0.18 ±0.03	0.06 ±0.01	0.27 ±0.03
32	0.09 ±0.02	0.10 ±0.02	0.04 ±0.00	0.13 ±0.02
64	0.09 ±0.03	0.07 ±0.03	0.03 ±0.00	0.07 ±0.02
128	0.06 ±0.01	0.04 ±0.01	0.02 ±0.00	0.03 ±0.00
256	0.05 ±0.01	0.02 ±0.00	0.02 ±0.00	0.02 ±0.00
512	0.05 ±0.00	0.02 ±0.00	0.02 ±0.00	0.01 ±0.00
1024	0.05 ±0.00	0.01 ±0.00	0.02 ±0.00	0.01 ±0.00

Table 8.3: Accuracy on the FS Online dataset with known data association

8.3 and 8.4. Finally, a graph of the translational MSE is shown in Fig. 8.10. Increasing the number of particles has the effect of decreasing both the MSE and the relative error. Variation with known data association again performs better than the unknown association. In this case, the difference is not as significant as in the previous dataset. This is because the GPS helps bring down the error of the variant with unknown association.

8.4 UTIAS

In addition to the simulated datasets we have shown previously, we evaluate our implementation on real-world data collected by the University of Toronto Institute for Aerospace Studies (UTIAS). The UTIAS dataset collection [53]

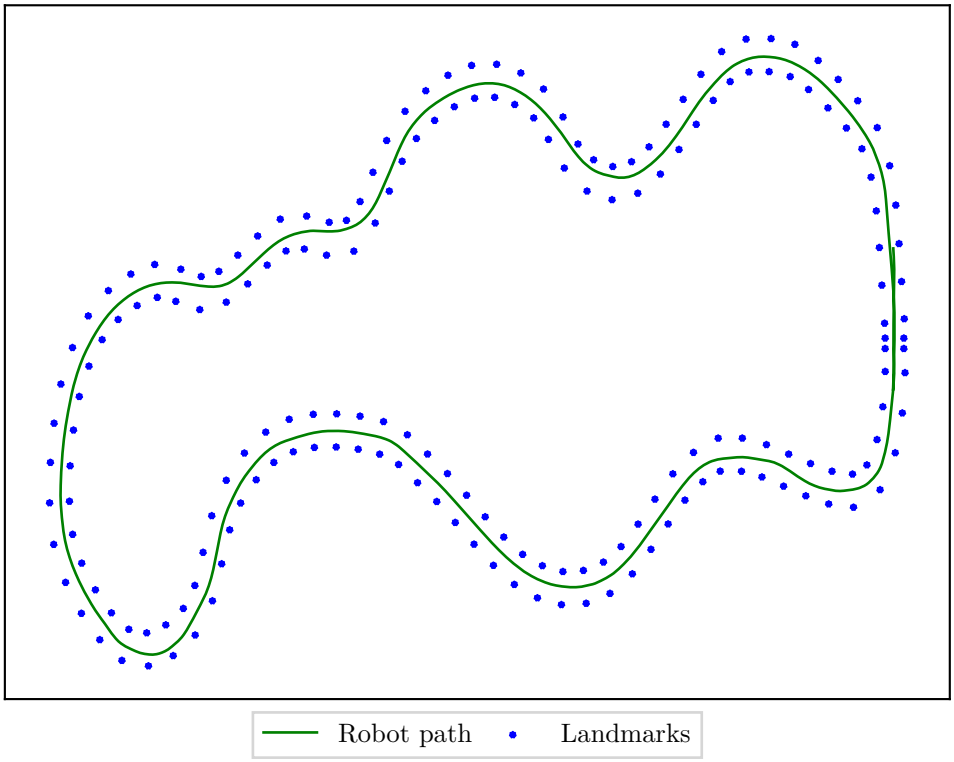


Figure 8.7: The FS Online virtual track used in the experiments

contains 9 separate indoor datasets. Each dataset was recorded with 5 robots simultaneously moving in an environment containing 15 landmarks. The dataset is intended for studying cooperative localization with a known map and cooperative SLAM. However, for the purposes of evaluating our SLAM implementation, the dataset can be adapted for a single robot SLAM by simply considering data from one robot only.

In each dataset, the robots move to random waypoints in the environment. The dimensions of the environment are 15×8 meters. While moving, the robots collect odometry data – forward and angular velocities. In addition, each robot is equipped with a camera for detecting landmarks. The landmarks are cylindrical tubes placed randomly in the environment. Each landmark has a barcode which the robots detect using the onboard camera and report its range and bearing. The dataset also provides accurate robot ground truth poses using a 10-camera Vicon motion capture system.

To estimate the dead-reckoning pose, we use the same kinematic model as described in (8.6). If we look at the measurement histogram depicted in Fig. 8.11, it is apparent that some landmarks have many outliers. This may cause problems with the Extended Kalman filters tracking landmark positions.

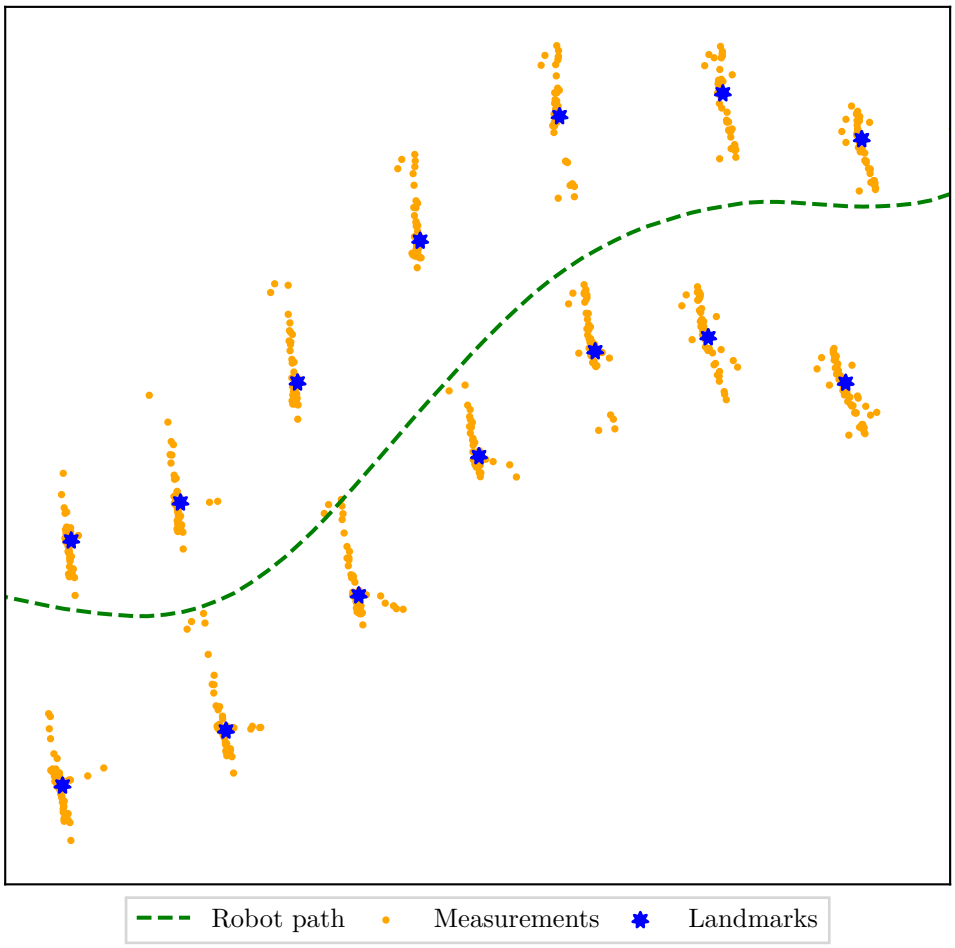


Figure 8.8: A section of the FS Online track showing the histogram of all measurements taken in this area. We can see that the error in the angle component is comparatively much larger.

Outliers can have the effect of shifting the mean of the Gaussian far away from the true position while at the same time decreasing the uncertainty. This makes the EKF more resistant to future correct measurements and at the same time lowers the weight of correct particles. To counteract this, a more conservative estimate of the measurement covariance is needed.

We tested the UTIAS dataset with both known and unknown data association. An example of the final path and map estimate is shown in Fig. 8.12. We again evaluate the accuracy for an increasing number of particles. Each configuration is run twenty times and the average is reported. The poses used for the relative error are a combination of consecutive and far away poses to assess both the local and global consistency. The results are summarized in Table 8.5 and 8.6. Fig. 8.13 shows the translational MSE for both the known and unknown data association. In this dataset, the kinematic model tends to

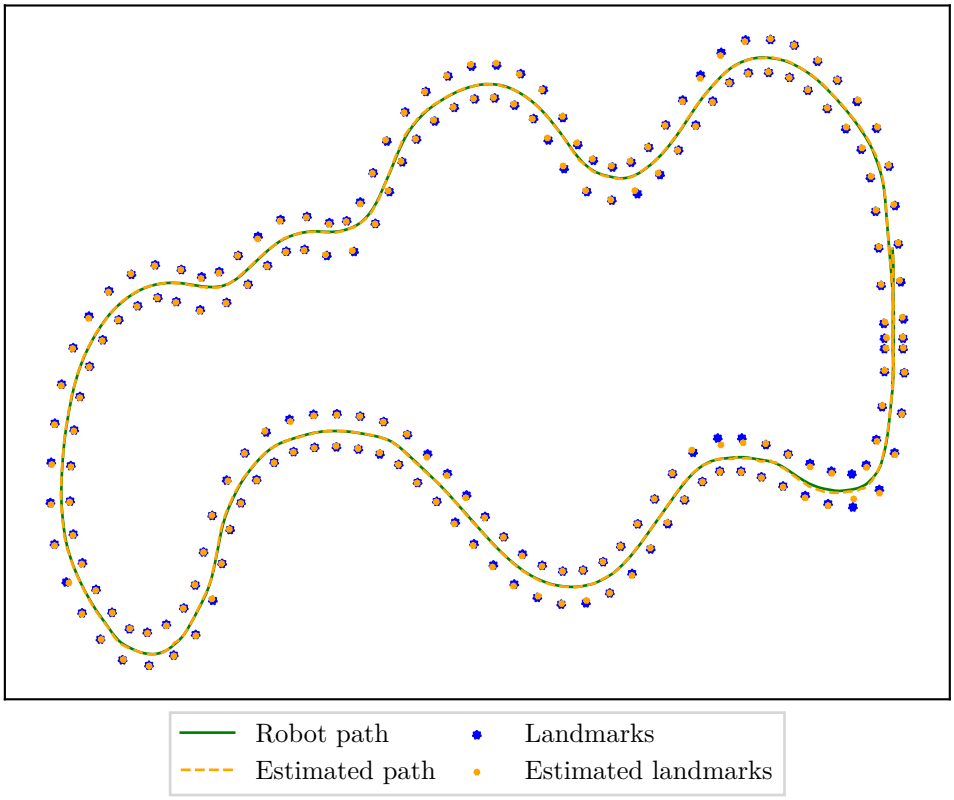


Figure 8.9: An example of the estimated robot path and the final map on the FS Online dataset

drift significantly from the ground truth, especially when the angular velocity is large as shown in Fig. 8.14. Large pose uncertainty is not necessarily problematic on its own. However, in this dataset, it is common that the robots execute a long turn during which no measurements are taken. Due to the large heading uncertainty, during this turn, most particles drift to areas of low likelihood. Because FastSLAM 1.0 has no mechanism to correct the particle trajectory, once new measurements are received, it is possible that no particles are close to the true pose, which inevitably causes filter divergence. An example of filter divergence is shown in Fig. 8.15. For this reason, a sufficient number of particles needs to be used to reach a large enough density which can recover from prolonged intervals without any measurements. We remark that FastSLAM 2.0 can deal with this problem by correcting particles towards areas which agree with the measurements and thus is not effected to the same degree by the lack of sensor input. The results again confirm that the algorithm performs significantly better with an increasing number of particles. Finally, the variant with known association achieves the same error with much fewer particles. This because known association eliminates the need for correct loop closures.

Particles	MSE(trans)	MSE(rot)	Rel(trans)	Rel(rot)
4	0.18 ±0.01	0.35 ±0.04	0.23 ±0.01	0.57 ±0.05
8	0.16 ±0.02	0.30 ±0.05	0.17 ±0.01	0.37 ±0.06
16	0.12 ±0.02	0.25 ±0.05	0.10 ±0.01	0.22 ±0.04
32	0.11 ±0.03	0.19 ±0.04	0.06 ±0.01	0.15 ±0.03
64	0.10 ±0.02	0.16 ±0.05	0.04 ±0.01	0.09 ±0.02
128	0.07 ±0.01	0.08 ±0.02	0.03 ±0.00	0.04 ±0.01
256	0.06 ±0.01	0.06 ±0.02	0.02 ±0.00	0.03 ±0.01
512	0.05 ±0.01	0.05 ±0.01	0.02 ±0.00	0.02 ±0.01
1024	0.05 ±0.01	0.04 ±0.01	0.02 ±0.00	0.02 ±0.00

Table 8.4: Accuracy on the FS Online dataset with unknown data association

Particles	MSE(trans)	MSE(rot)	Rel(trans)	Rel(rot)
4	0.11 ±0.04	1.64 ±0.43	0.12 ±0.04	2.12 ±1.32
8	0.06 ±0.02	1.20 ±0.11	0.06 ±0.02	1.09 ±0.48
16	0.05 ±0.03	1.15 ±0.32	0.06 ±0.03	0.65 ±0.29
32	0.05 ±0.02	1.14 ±0.16	0.05 ±0.02	0.49 ±0.20
64	0.04 ±0.03	1.07 ±0.12	0.04 ±0.02	0.31 ±0.13
128	0.04 ±0.02	0.99 ±0.11	0.04 ±0.01	0.30 ±0.13
256	0.03 ±0.02	1.00 ±0.15	0.04 ±0.01	0.22 ±0.08
512	0.03 ±0.02	0.98 ±0.12	0.04 ±0.01	0.22 ±0.04
1024	0.03 ±0.01	0.94 ±0.12	0.04 ±0.01	0.23 ±0.10
2048	0.03 ±0.01	0.94 ±0.11	0.04 ±0.01	0.17 ±0.05
4096	0.03 ±0.02	0.87 ±0.08	0.04 ±0.01	0.19 ±0.06
8192	0.03 ±0.01	0.88 ±0.09	0.04 ±0.01	0.19 ±0.04

Table 8.5: Accuracy on the UTIAS dataset with known data association

8.5 Performance

Finally, we evaluate the performance of our proposed implementation. We are interested in the scaling properties of the algorithm with respect to both time and memory. All experiments in this section were carried out on a laptop with Intel i7-8550U 1.8GHz and Nvidia GTX 1660 GPU. The accuracy of FastSLAM is directly linked to the number of particles used to approximate the SLAM posterior. It is then important that a FastSLAM implementation is capable of efficiently simulating as many particles as possible. We investigate how the algorithm scales as a function of the number of particles and as a function of the number of simultaneous measurements.

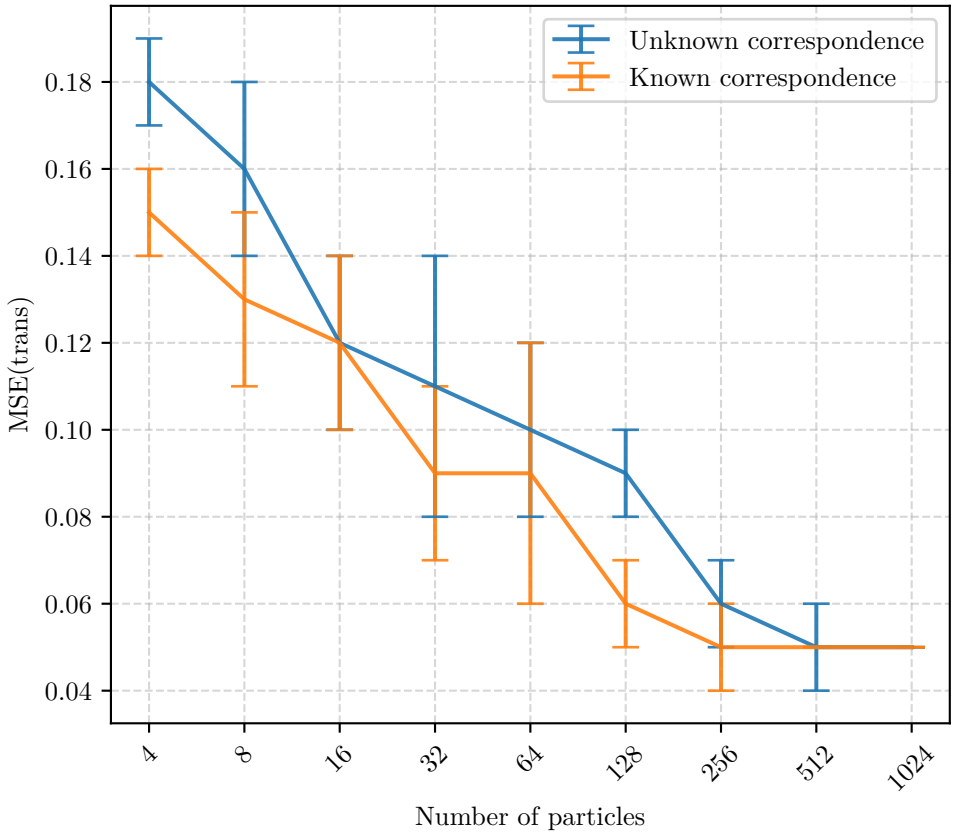


Figure 8.10: A graph showing the translational MSE on the FS Online dataset as the number of particles increases

We first measure the performance on the first simulated dataset. We record the total runtime of the simulation for an increasing number of particles. This is again repeated twenty times and the average is reported. We measure the time for both known and unknown correspondence and compare the execution time with the implementation provided by Python Robotics. We once again remark that this comparison only serves to highlight the potential speedups that are possible by leveraging the massive parallelism of modern GPUs compared to a pure CPU implementation. A logarithmic plot of the results is shown in Fig. 8.16. We can see that even for a small number of particles, our proposed implementation is orders of magnitude faster with better scaling. Due to the time requirements, we were not able to collect data for the Python Robotics implementation beyond 1024 particles.

Next, we evaluate the scaling properties as a function of the number of simultaneous observations. For this evaluation, the robot is placed in a virtual environment containing 1000 landmarks uniformly distributed along the robot path. The robot moves forward at a constant velocity. The simulation lasts 200 time steps after which the robot has incorporated all landmarks.

Particles	MSE(trans)	MSE(rot)	Rel(trans)	Rel(rot)
4	4.10 ±5.36	23.33 ±34.72	5.73 ±6.28	18.43 ±21.75
8	2.08 ±2.78	9.59 ±13.20	2.83 ±3.52	10.22 ±13.45
16	1.18 ±1.45	4.71 ± 4.74	2.25 ±1.95	6.96 ± 8.66
32	2.00 ±3.01	7.94 ±10.61	2.47 ±3.24	7.14 ± 9.19
64	2.40 ±2.89	8.82 ± 9.72	2.78 ±2.96	11.90 ±19.21
128	1.02 ±2.00	4.99 ± 8.02	1.42 ±2.14	6.24 ±12.75
256	0.60 ±1.76	3.02 ± 5.89	1.03 ±1.96	2.74 ± 5.06
512	0.82 ±1.89	3.88 ± 7.02	1.16 ±2.09	4.33 ± 9.41
1024	0.30 ±0.79	1.43 ± 1.40	0.44 ±1.12	0.88 ± 1.77
2048	0.06 ±0.11	1.13 ± 0.83	0.30 ±0.93	0.90 ± 2.26
4096	0.03 ±0.02	0.90 ± 0.11	0.03 ±0.01	0.22 ± 0.08
8192	0.03 ±0.03	0.86 ± 0.09	0.03 ±0.01	0.22 ± 0.07

Table 8.6: Accuracy on the UTIAS dataset with unknown data association

By increasing the sensor range of the robot, we can control the number of simultaneous measurements that the robot observes. We then measure the total execution time of the simulation as the number of simultaneous measurements increases. Here, we again compare both known and unknown correspondence for 128 and 1024 particles. The results are shown in Fig. 8.17. Based on the results, which are confirmed by profiling the GPU code, we conclude that the most time is spent in the data association step. This is given by the time complexity of the data association algorithm. Recall that the time complexity is $\mathcal{O}(M + ZR)$, where M is the map size, Z is the number of measurements and R is the number of landmarks in the sensor range. Thus, when the robot observes more landmarks, both Z and R increase simultaneously. As we noted in the previous chapter, by using K-D-B trees [82] or Balanced k-d trees [12], the complexity can be brought down to $\mathcal{O}(ZlogM)$. However, the extra overhead and hidden constants may make it less efficient when the number of simultaneous observations is relatively low, as is the case in Formula Student.

In the previous paragraph, we evaluated the scaling properties of our implementation for extreme cases. However, we are also interested in the expected performance on an average instance. Since the FS Online dataset closely resembles real events in terms of the available sensors and the map size, we evaluated the performance on this dataset as well. Fig. 8.18 shows the total runtime of the simulation as the number of particles increases. To get a better intuition behind these numbers, Fig. 8.19 shows the update frequency rather than the total runtime. The update frequency denotes how many times per second a single step of the FastSLAM algorithm was performed. The update frequency counts the whole algorithm, that is, prediction, correction, data association, and resampling. For 1024 particles and unknown correspondence,

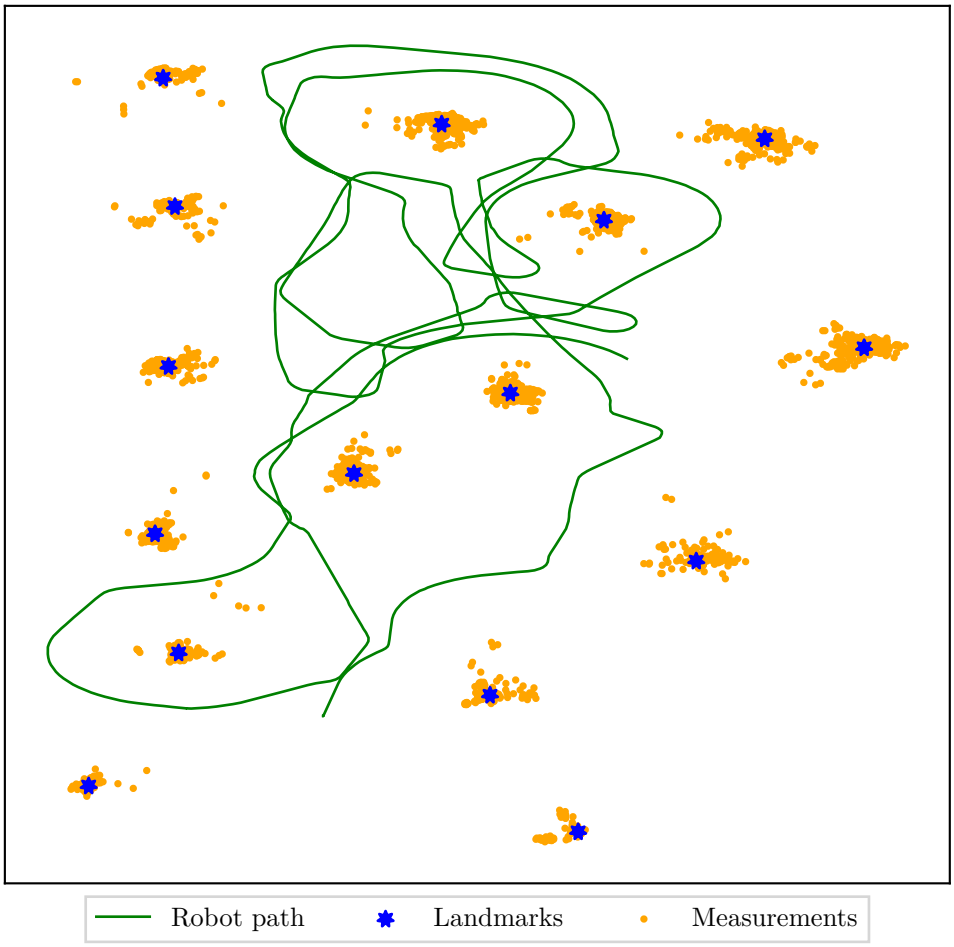


Figure 8.11: Histogram of all measurements taken by one robot in the UTIAS dataset. Notice that for some landmarks, the robot generates distant outliers.

the algorithm runs at approximately 370Hz.

Another point related to performance, is the execution configuration of the GPU kernels. CUDA requires the program to specify the block and grid dimensions that the kernels are launched with. Suboptimal configurations can significantly degrade the performance and thus it is worth to invest time to find an optimal configuration. By methodically comparing different configurations, we were able to further achieve a 2x speedup. This speedup comes from deliberately launching kernels with significantly smaller block sizes than what the GPU Streaming multiprocessors (SM) can execute. The most likely explanation for why this leads to an increase in speed is that the algorithm exhibits a large degree of warp divergence caused mainly by data association. This makes different particles execute different branches of the code even in a single warp. Since a block can only be removed from an SM when all of its threads have finished, diverging threads keep the SM from

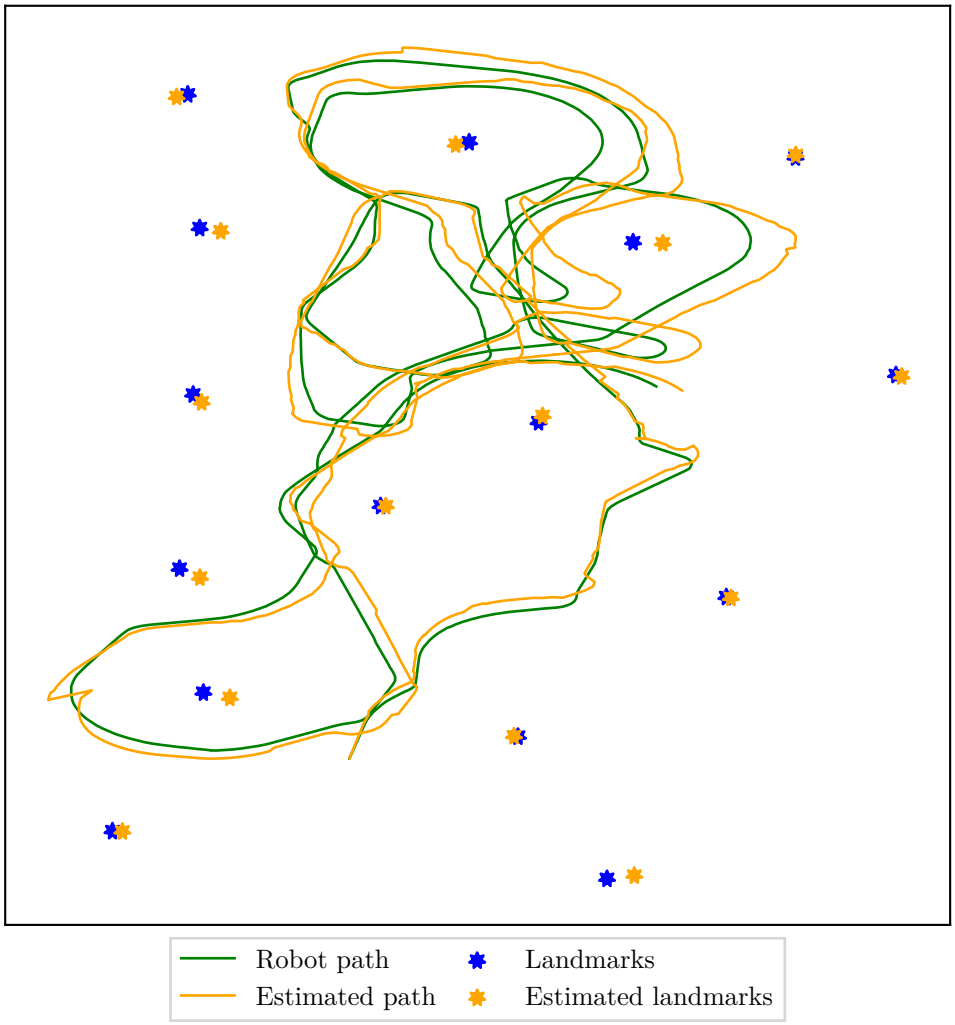


Figure 8.12: An example of the final path and map estimate on the UTIAS dataset

scheduling new blocks. Thus, by keeping the block size smaller, the SM can cycle through blocks more efficiently.

Lastly, we look at the memory scaling of the GPU implementation. Our implementation scales linearly with respect to both the number of particles and the map size. Apart from several auxiliary buffers and scratchpad memory reserved for data association, most memory is taken up by the particles themselves. One particle takes up approximately $8 \times 6 \times M$ bytes, where 8 is given by the size of a double precision float, 6 is the number of floats needed to store the mean and covariance of a landmark, and M is the map size. Fig. 8.20 shows a graph of the memory usage for known and unknown correspondence. The memory requirements for known correspondence are

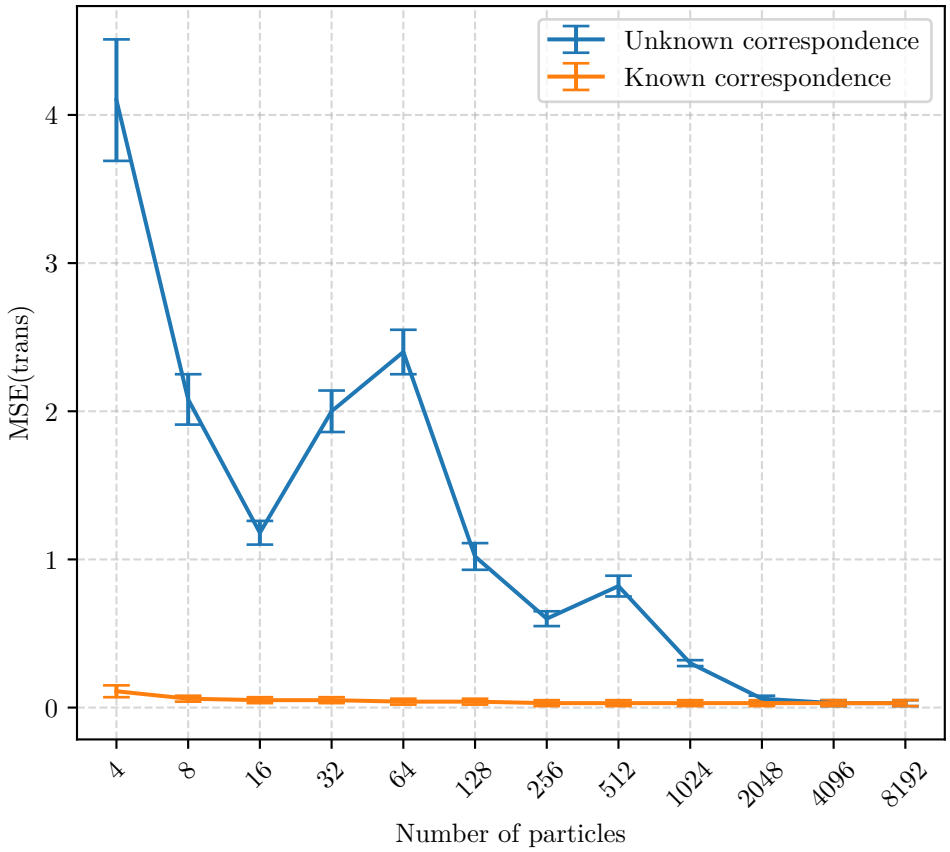


Figure 8.13: Comparison of the translational MSE of known and uknown data association on the UTIAS dataset

slightly lower as no extra memory for data association is needed. Fig. 8.20 shows the memory requirements for increasing map size for 128 and 1024 particles. To store 1024 particles each with 10^5 landmarks, the algorithm needs approximately 700MB. In the case of FS Online with 1024 particles and 200 landmarks, the required memory is approximately 15MB.

8.6 Discussion

The experimental results show that our implementation achieves a good accuracy on both simulated and real-world datasets. The accuracy was first evaluated on a simulated dataset and compared with an implementation provided by Python Robotics [84]. Following, we evaluated the implementation on data collected from the Formula Student Driverless Simulation which represents a typical application of this implementation. Finally, a dataset collected by the University of Toronto Institute for Aerospace Studies [53] was used to

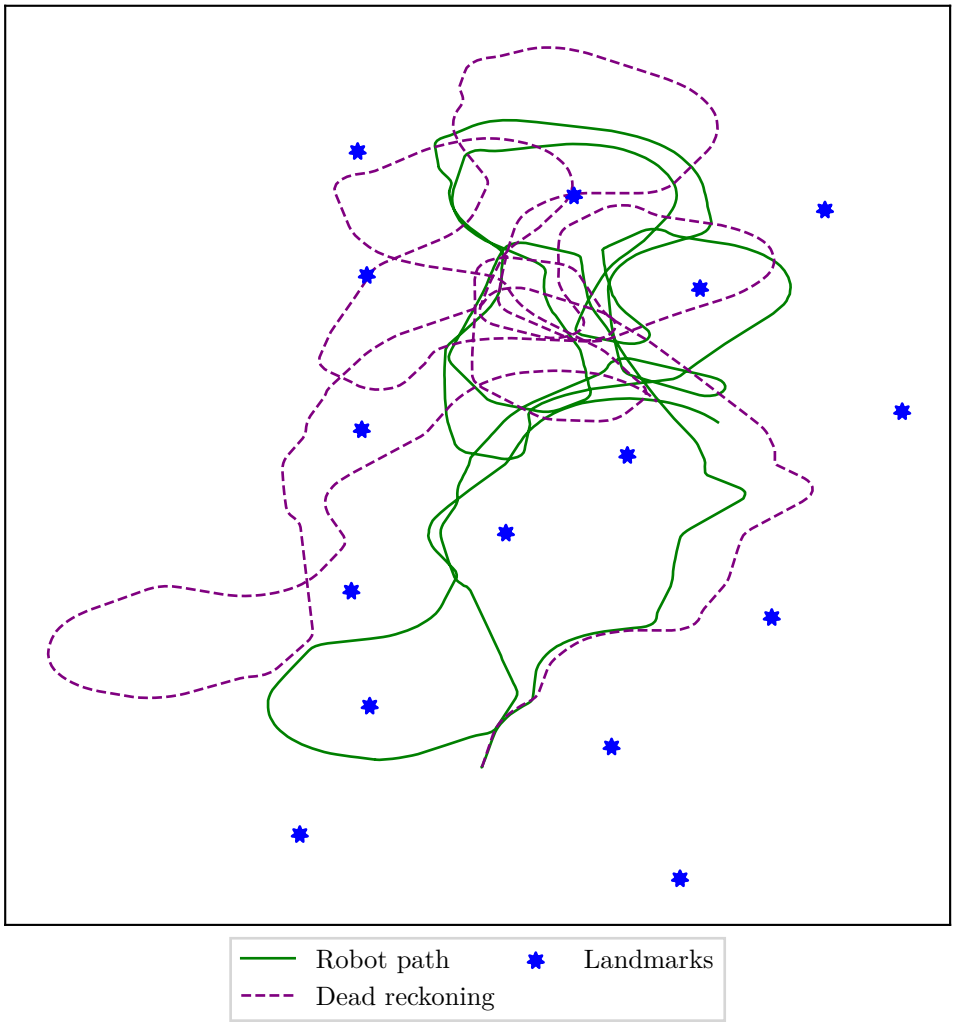


Figure 8.14: A depiction of the drift in odometry in the UTIAS dataset due to the motion noise

evaluate the algorithm on real-world data. By evaluating the performance, we showed that by leveraging GPUs, we can increase the performance by several orders of magnitude compared to a CPU implementation. Furthermore, the required memory scales linearly in both the number of particles and the map size. For most practical use cases, the used memory will not exceed 100MB. By evaluating the scaling properties of the implementation, we identified the data association step as the most time-consuming part. This bottleneck can be largely circumvented by considering more suitable data structures to represent the map. However, taking FS Online as a prototypical example of a real-world application, we showed that the performance far exceeds the minimum requirements and the memory consumption is miniscule. Thus we conclude that our proposed implementation is indeed sufficiently accurate

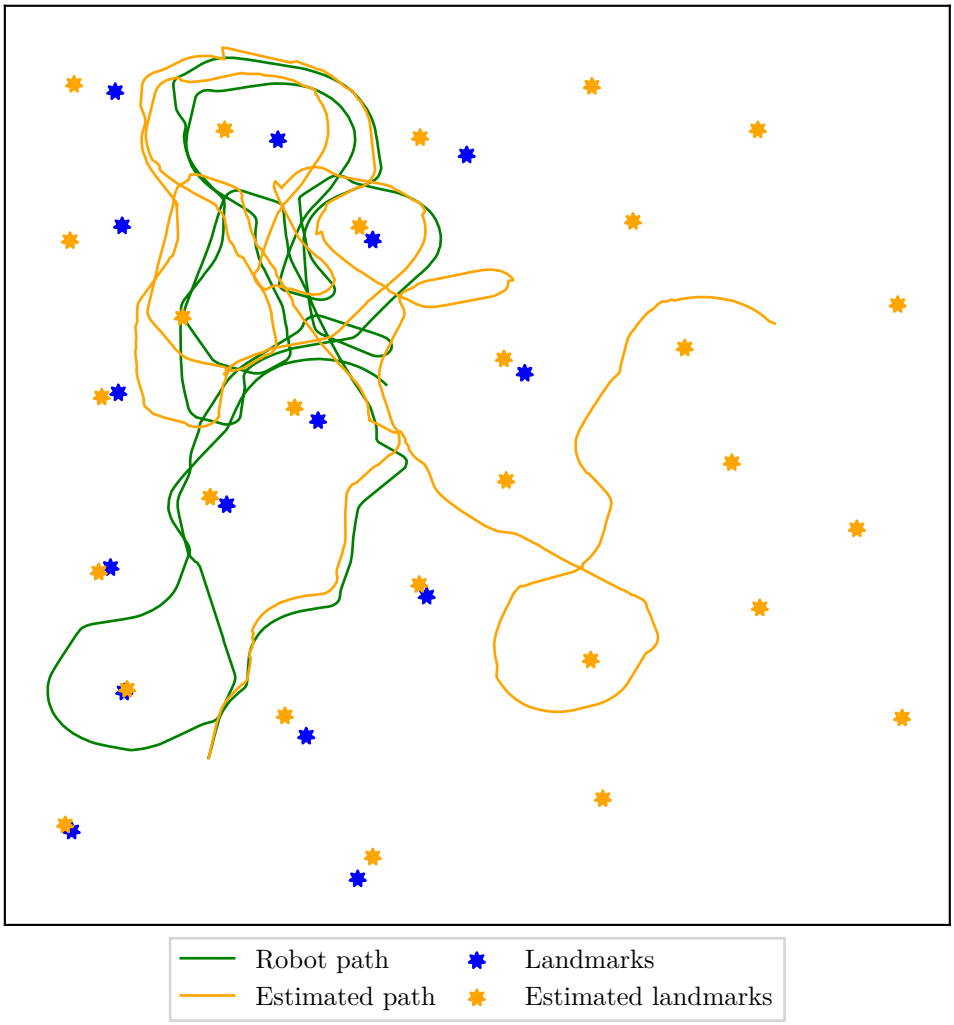


Figure 8.15: Divergence in the UTIAS dataset may occur due to prolonged intervals without sensor input combined with odometry noise.

and performant for an autonomous racing formula for the Formula Student competition.

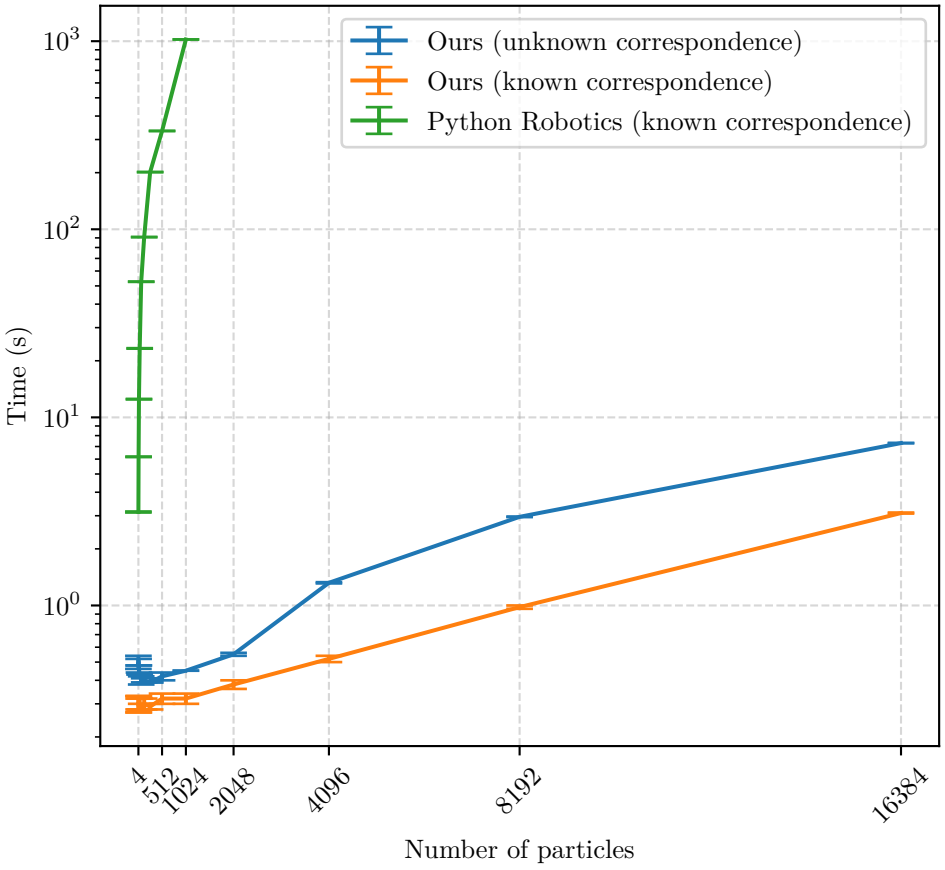


Figure 8.16: A graph of the total execution time on the simulated dataset. The x-axis shows the number of particles and the y-axis is the total time in seconds on a logarithmic scale. Note that the GPU implementation is orders of magnitude faster compared to the CPU implementation even for a relatively small number of particles.

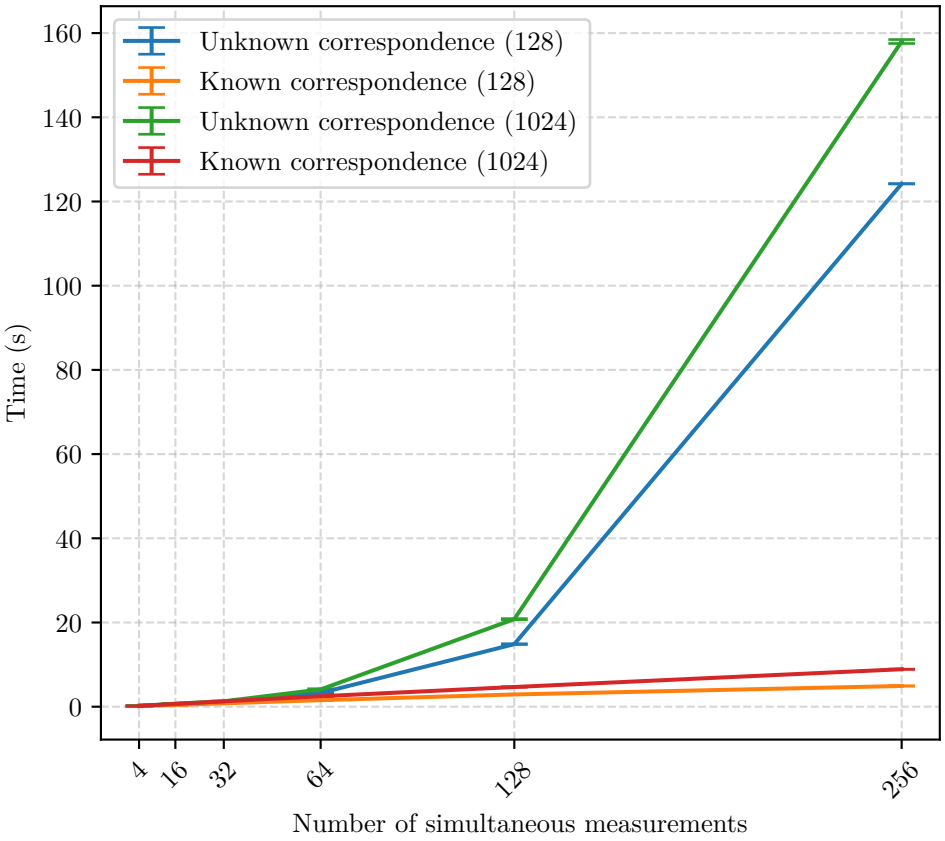


Figure 8.17: The scaling properties of the algorithm as the number of measurements increases. The x-axis shows the number of simultaneous measurements while the y-axis shows the total execution time in seconds. This graph implies that the most time in the algorithm is spent in the data association step which was confirmed by profiling the code.

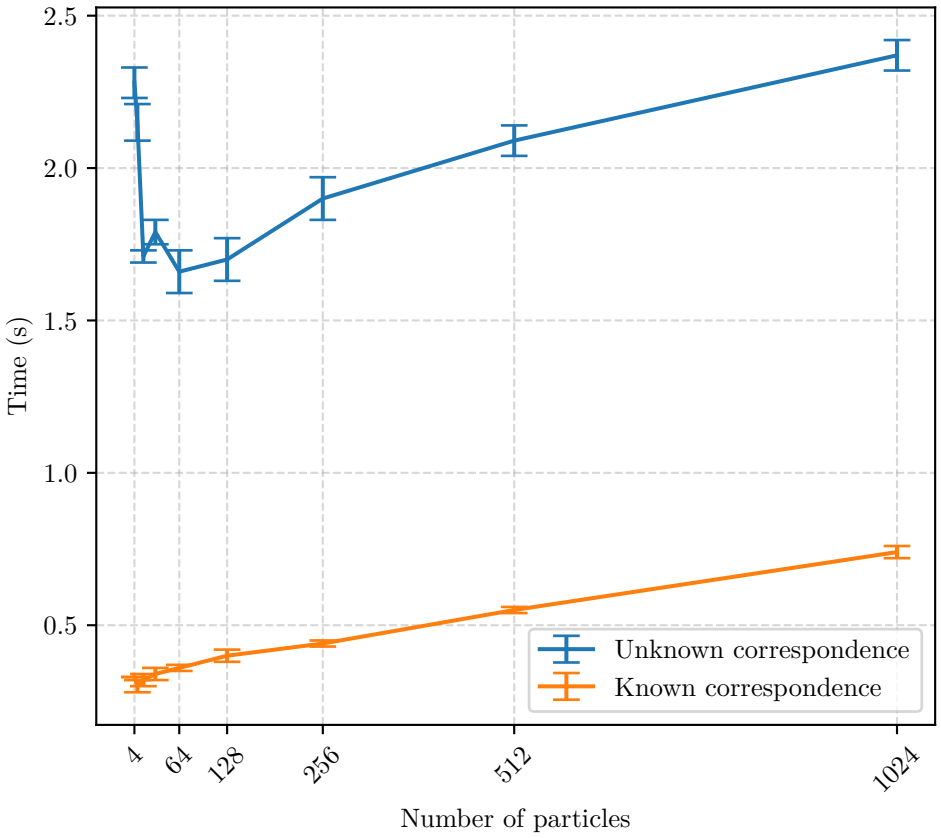


Figure 8.18: A graph of the total execution time on the FS Online dataset as a function of the number of particles

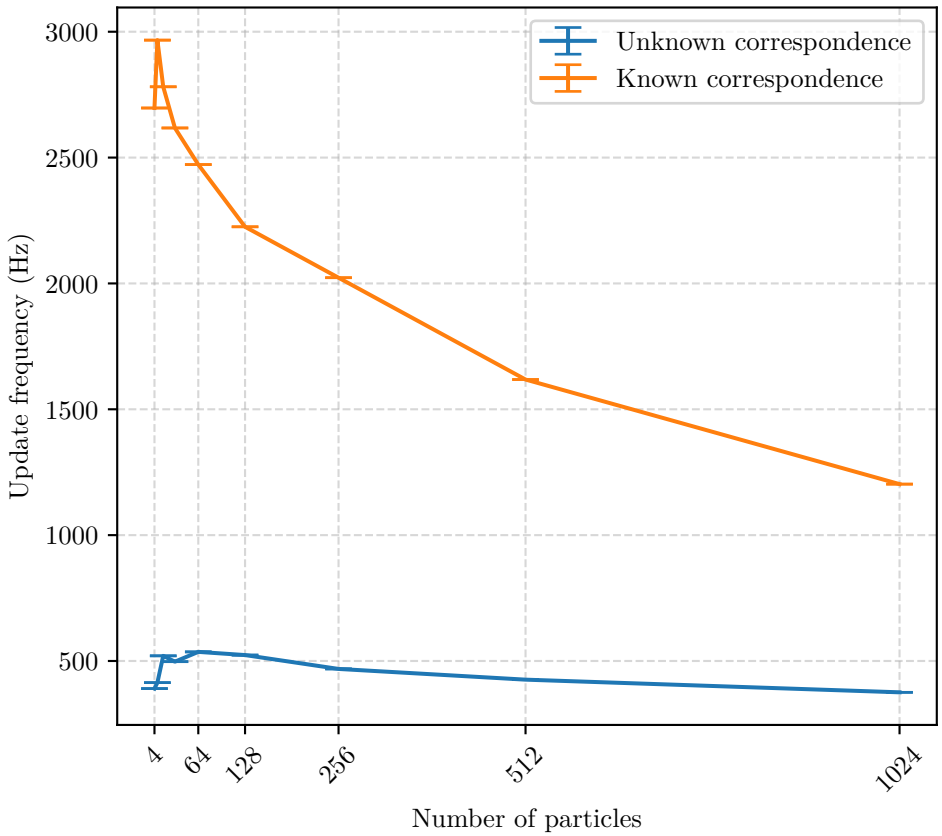


Figure 8.19: A graph of the update frequency on the FS Online dataset. For 1024 particles and uknown association, the algorithm operates at approximately 370Hz.

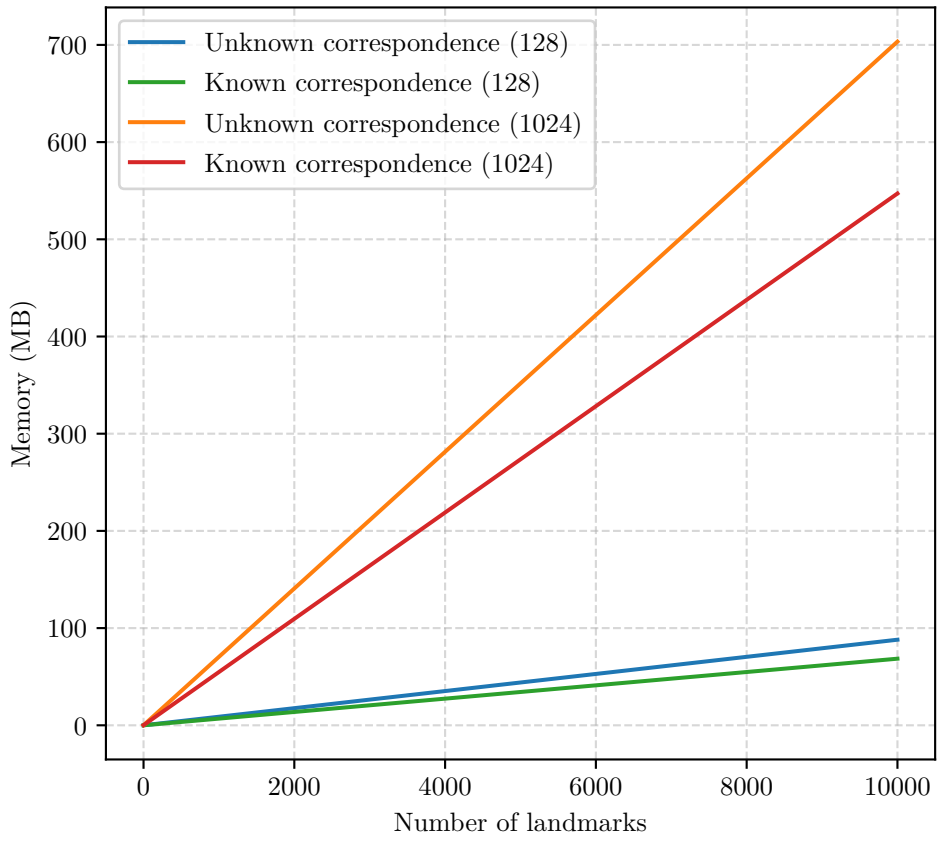


Figure 8.20: A graph showing the memory scaling of the GPU implementation for an increasing map size

Chapter 9

Conclusion

In this thesis, we have discussed the design and implementation of a Simultaneous localization and mapping system for an autonomous student formula participating in the Formula Student competition. Simultaneous localization and mapping (SLAM) is a problem in which a robot traverses an unknown environment and uses the information from its exteroceptive sensor together with odometry to construct a virtual map of the environment and simultaneously localizes itself within it. In the theoretical section of the thesis, we provide a thorough overview of the evolution of SLAM in the last decades. We discussed the theory behind recursive Bayesian estimation and recursive filtering, which forms the basis of many state estimation techniques including the probabilistic formulation of SLAM. Bayesian filtering estimates the posterior density of a system using an estimate from a previous time step, which is projected forward using the process model and subsequently corrected using the measurement model. Next, we show a special instance of Bayesian filtering, which is the Kalman filter. The Kalman filter is an optimal state estimator for linear/Gaussian systems and plays a crucial role in many SLAM algorithms. As a last step before SLAM, we described the theory of particle filters and discussed their use in robotics. Particle filters overcome many limitations of Kalman filters thanks to their ability to represent arbitrary distributions by using Monte Carlo techniques. This property makes particle filters suitable for modeling highly nonlinear systems.

This lays the necessary foundation in order to introduce the three main SLAM algorithms most commonly found in literature – EKF-SLAM, FastSLAM, and Graph SLAM. EKF-SLAM can be thought of as an extension of a Kalman filter where the state is extended to include the map estimate. EKF-SLAM was the first SLAM algorithm successfully deployed in real applications and remains widely used to this day. The FastSLAM algorithm supersedes EKF-SLAM in many regards. FastSLAM exploits the conditional independence between landmarks to factor the SLAM posterior. Thanks

to this factorization, every landmark can be tracked using an independent low-dimensional EKF, which reduces the time complexity of updating the estimate from $\mathcal{O}(M^2)$ to $\Theta(1)$ compared to EKF-SLAM. Moreover, FastSLAM uses a particle filter to estimate the robot pose, making it more resilient to nonlinear motion. Finally, we describe Graph SLAM, which is the most recent state-of-the-art formulation of offline SLAM. Graph SLAM uses the fact that SLAM can be modeled as a dynamic Bayesian network to build a constraint graph induced by the robot odometry and sensor measurements. This graph is then optimized using nonlinear least-squares to find the most likely path and map configuration. We also discussed the problem of data association. Data association is a general problem in SLAM in which the most recent sensor measurements need to be associated with previously seen landmarks. A robust data association algorithm is crucial for detecting loop closures in the map. Data association is a difficult problem on its own. In the thesis, we show a non-exhaustive list of common data association algorithms.

The thesis describes a real-time implementation of FastSLAM 1.0 that is suitable for use in an autonomous racing formula. We describe how FastSLAM can be efficiently parallelized by leveraging GPUs. We discuss the differences in CPU and GPU programming paradigms and give a brief overview of CUDA. We provide a detailed description of our implementation encompassing the whole architecture and data structures used. The frontend of the algorithm which handles data preprocessing and visualization is written in Python, while the backend is written in CUDA. We use simple linear data structures which provide a good tradeoff between complexity and speed. Finally, we evaluate our implementation on an array of simulated and real datasets. We verify the accuracy compared to a baseline implementation provided by Python Robotics [84]. Furthermore, by comparing the performance with a CPU implementation, we find our GPU implementation to be several orders of magnitude faster and to have significantly better scaling properties. Based on this, we conclude that our implementation is suitable for real-time SLAM in an autonomous racing formula.

9.1 Future work

The thesis proposes a implementation of FastSLAM 1.0, however, an improved version known as FastSLAM 2.0 [63] exists and is also discussed in the thesis. Unlike FastSLAM 1.0, FastSLAM 2.0 has been shown to converge in the linear/Gaussian case. Moreover, experimental results show that FastSLAM 2.0 can achieve the same level of accuracy while needing significantly fewer particles. As such, future work on this implementation should be focused on extending the original FastSLAM 1.0 formulation to include the proposal distribution used by FastSLAM 2.0. To make our implementaion even more efficient, an attention should also be given to data association. Based on

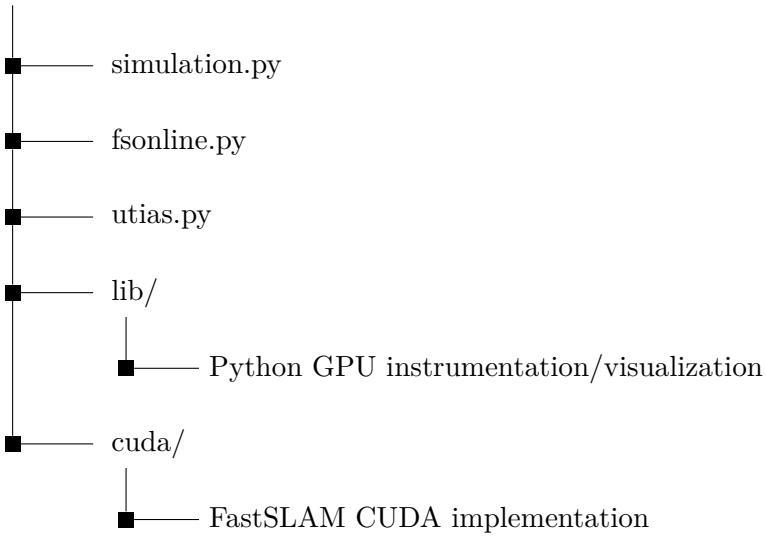
the information gained by profiling the algorithm, the vast majority of time is indeed devoted to data association. This is because in order to keep the implementation relatively simple, the data association algorithm uses linear data structures resulting in time complexity linear as a function of the map size. By using proper tree data structures, a logarithmic time complexity can be achieved.

Source code

The source code of the proposed implementation is enclosed with the thesis. It also available in a Github repository [1]. The code contains the library together with several examples based on the experiments shown in the thesis. To run them, a CUDA-capable GPU is required and CUDA has to be installed on the host computer. The experiments also require Python3 and several Python packages. The packages can be installed by running the following command:

```
$ python3 -m pip install -r requirements.txt
```

The code structure is shown below. There are three prepared examples, *simulation.py*, *fsonline.py*, and *utias.py*, which can be run by executing the corresponding files with Python.



[1] https://github.com/tomasr8/fastslam64/tree/thesis

Appendix A

Supplemental material

A.1 Recursive Bayesian filter

A.1.1 Derivation of the MMSE estimator

The MMSE estimator minimizes the following loss function:

$$\hat{\boldsymbol{x}}_k^{MMSE} = \arg\min_{\hat{\boldsymbol{x}}_k} \ \mathbb{E}\left[||\boldsymbol{x}_k - \hat{\boldsymbol{x}}_k||^2 | \boldsymbol{z}_{1:k}\right] \tag{A.1}$$

$$= \arg\min_{\hat{\boldsymbol{x}}_k} \ \int ||\boldsymbol{x}_k - \hat{\boldsymbol{x}}_k||^2 \ p(\boldsymbol{x}_k|\boldsymbol{z}_{1:k}) d\boldsymbol{x}_k. \tag{A.2}$$

Taking the derivative of the loss, we obtain

$$\frac{\partial \mathcal{L}}{\partial \hat{\boldsymbol{x}}_k} = \frac{\partial}{\partial \hat{\boldsymbol{x}}_k} \int ||\boldsymbol{x}_k - \hat{\boldsymbol{x}}_k||^2 \ p(\boldsymbol{x}_k|\boldsymbol{z}_{1:k}) d\boldsymbol{x}_k \tag{A.3}$$

$$= \int \frac{\partial}{\partial \hat{\boldsymbol{x}}_k} ||\boldsymbol{x}_k - \hat{\boldsymbol{x}}_k||^2 \ p(\boldsymbol{x}_k|\boldsymbol{z}_{1:k}) d\boldsymbol{x}_k \tag{A.4}$$

$$= \int -2(\boldsymbol{x}_k - \hat{\boldsymbol{x}}_k)^T p(\boldsymbol{x}_k|\boldsymbol{z}_{1:k}) d\boldsymbol{x}_k \tag{A.5}$$

$$= -2 \int \boldsymbol{x}_k^T p(\boldsymbol{x}_k|\boldsymbol{z}_{1:k}) d\boldsymbol{x}_k + 2\hat{\boldsymbol{x}}_k^T \int p(\boldsymbol{x}_k|\boldsymbol{z}_{1:k}) d\boldsymbol{x}_k \tag{A.6}$$

$$= -2\mathbb{E}\left[\boldsymbol{x}_k|\boldsymbol{z}_{1:k}\right]^T + 2\hat{\boldsymbol{x}}_k^T \tag{A.7}$$

Setting the last expression to zero, we find that $\hat{\boldsymbol{x}}_k^{MMSE} = \mathbb{E}\left[\boldsymbol{x}_k|\boldsymbol{z}_{1:k}\right]$.

A.2 Kalman filter

A.2.1 Intuition behind the Kalman gain

One can *gain* more intuition about the Kalman gain by considering what happens when either the prediction or measurement error goes to zero. Consider first the case when $\boldsymbol{P}_{k|k-1} \to 0$, i.e., there is a small prediction error. Then,

$$\lim_{\boldsymbol{P}_{k|k-1}\to 0} \boldsymbol{K}_k = 0 \;\Rightarrow\; \hat{\boldsymbol{x}}_{k|k} = \hat{\boldsymbol{x}}_{k|k-1} + \boldsymbol{K}_k \boldsymbol{y}_k \tag{A.8}$$

$$= \hat{\boldsymbol{x}}_{k|k-1}. \tag{A.9}$$

In other words, the filter completely disregards the available measurement in favor of the prediction. Conversely, when $\boldsymbol{R}_k \to 0$ instead, we have

$$\lim_{\boldsymbol{R}_k\to 0} \boldsymbol{K}_k = \boldsymbol{H}_k^{-1} \;\Rightarrow\; \hat{\boldsymbol{x}}_{k|k} = \hat{\boldsymbol{x}}_{k|k-1} + \boldsymbol{K}_k \boldsymbol{y}_k \tag{A.10}$$

$$= \hat{\boldsymbol{x}}_{k|k-1} + \boldsymbol{H}_k^{-1}\left(\boldsymbol{z}_k - \boldsymbol{H}_k \hat{\boldsymbol{x}}_{k|k-1}\right) \tag{A.11}$$

$$= \boldsymbol{H}_k^{-1} \boldsymbol{z}_k. \tag{A.12}$$

Here, the filter estimates its position using only the measurement while ignoring the previous estimate. The Kalman gain is simply a measure of how much the filter trusts the prediction over the measurement and vice versa.

A.3 Particle filter

A.3.1 Resampling strategies

In this section, we describe some common resampling strategies for particle filters. Namely, we introduce multinomial, stratified and systematic resampling. Later we also show algorithms better suited for parallel resampling.

Multinomial resampling

Multinomial resampling is a resampling algorithm in which the new particles are sampled directly from the multinomial distribution with parameters N and $\boldsymbol{w} = \{w^1, \ldots, w^N\}$. This by definition satisfies the unbiasedness condition (5.21). Multinomial sampling can be realised by using the cumulative distribution function (CDF) induced by the particle weights. The CDF is a partial sum defined as:

$$P(\boldsymbol{x}^j) = \sum_{i=1}^{j} w^i. \tag{A.13}$$

The inverse CDF for a real number $u \in [0, 1]$ is then

$$P(u)^{-1} = \begin{cases} \boldsymbol{x}^1, & P(\boldsymbol{x}^1) \leq u \\ \boldsymbol{x}^i, & P(\boldsymbol{x}^i) \geq u \text{ and } P(\boldsymbol{x}^{i-1}) < u \end{cases}. \tag{A.14}$$

Since the particle weights are discrete values, the CDF is a step function as shown in Fig. A.1. One can now implement multinomial resampling by first drawing a random number $u \sim \mathcal{U}[0, 1]$ and then computing the inverse CDF to get find the corresponding particle. Mathematically, the ancestor a^i is defined as $a^i = P^{-1}(u^i)$, where u^i is the ith random number. This step is repeated N times until the ancestor vector is filled. The pseudocode of multinomal resampling is shown in Algorithm 13.

Algorithm 13: Multinomial resampling

input : An array of particle weights $\{w_i,\ i = 1, 2, ..., N\}$
output : An ancestor vector $\{a_i,\ i = 1, 2, ..., N\}$ mapping new particle indices to old indices

```
c_1 = w_1          // Compute cummulative sum
for i ← 1 to N do
    c_i = c_{i−1} + w_i
end

for i ← 1 to N do
    Draw u ∼ U(0, 1)
    a_i = P^{−1}(u)
end
```

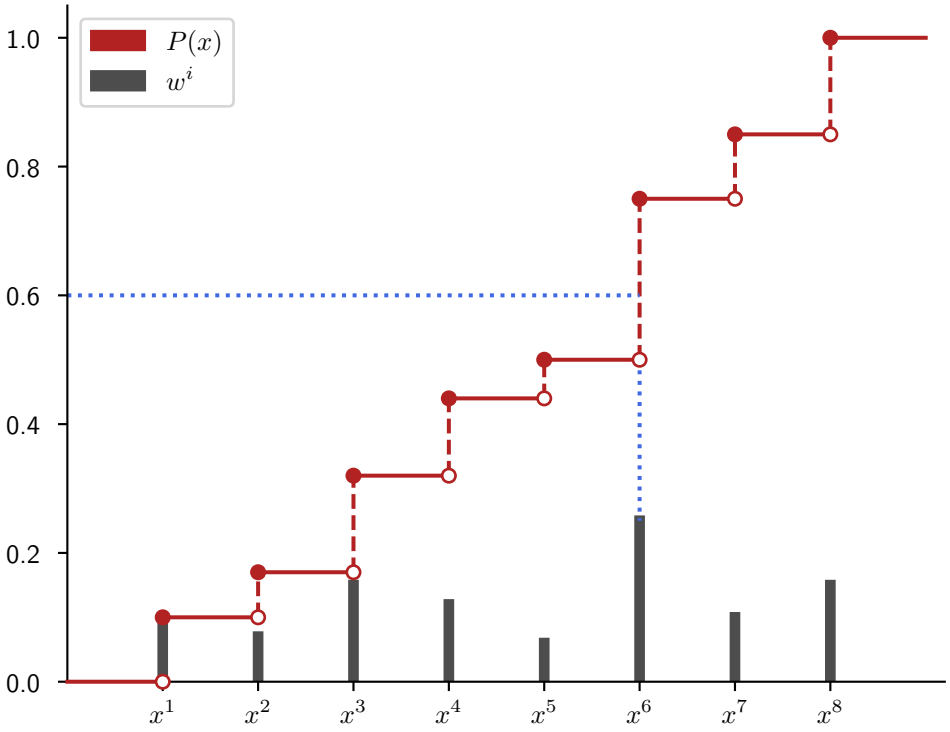


Figure A.1: A visualization of the CDF $P(\boldsymbol{x})$ computed from particle weights $\{w^1, ..., w^N\}$. The inverse $P^{-1}(u)$ can be found visually by intersecting a line with the CDF. In this example, $P^{-1}(0.6) = \boldsymbol{x}^6$.

Stratified resampling

Stratified resampling is another common resampling method [45]. Similarly to multinomial sampling, this method first constructs a CDF $P(x^i)$ of the particle weights. Next, the unit interval $[0, 1)$ is partitioned into N equally sized subintervals N^i such that

$$N^i = \left[\frac{i-1}{n}, \frac{i}{n}\right). \tag{A.15}$$

Subsequently, for each subinterval N^i, a random number u^i is drawn from the uniform distribution

$$u^i \sim \mathcal{U}\left[\frac{i-1}{n}, \frac{i}{n}\right). \tag{A.16}$$

The ancestor vector is then defined as $a^i = P^{-1}(u^i)$. As shown in [21],

stratified resampling also satisfies the unbiasedness condition. Furthermore, it can be shown that the replication counts differ from the expected value Nw^i by at most one in absolute value. Formally, we have $|\boldsymbol{o}^i - Nw^i| \leq 1$. This so-called *set restriction* technique leads to a reduction in variance in the offspring vector. Authors of [21] prove that the variance in the offspring vector is always lower than that of multinomial resampling, meaning that stratified sampling is less sensitive to the drawn random numbers. A more detailed comparison is provided in [67]. Finally, an implementation of stratified resampling is provided in Algorithm 14.

Algorithm 14: Stratified resampling

input : An array of particle weights $\{w_i,\ i = 1, 2, ..., N\}$
output : An ancestor vector $\{a_i,\ i = 1, 2, ..., N\}$ mapping new particle indices to old indices

$c_1 = w_1$ // Compute cummulative sum
for $i \leftarrow 1$ **to** N **do**
    $c_i = c_{i-1} + w_i$
end

Draw $u \sim \mathcal{U}[0, 1)$
$i = 0$
$j = 0$
while $i < N$ **do**
    if $(i + u)/N < c_j$ **then**
        $a_i = j$
        $i = i + 1$
        Draw $u \sim \mathcal{U}[0, 1)$
    else
        $j = j + 1$
    end
end

Systematic resampling

Systematic resampling is very similar in principle to stratified resampling. In particular, the unit interval $[0, 1]$ is again partitioned into equally-sized subintervals

$$N^i = \left[\frac{i-1}{n}, \frac{i}{n}\right). \tag{A.17}$$

However, unlike the stratified resampling, only a single random number u

is drawn from $\mathcal{U}[0, 1)$. The ancestor a^i of a particle $\boldsymbol{x}^i$ is then chosen as

$$\boldsymbol{a}^i = P^{-1}\left(\frac{i-1}{n} + \frac{u}{n}\right). \tag{A.18}$$

The difference from stratified sampling is that the offset in every subinterval is the same and given by u. As shown in [21], systematic resampling is also unbiased. However, unlike stratified sampling, systematic resampling does not provably attain a lower variance in the offspring vector compared to multinomial resampling. Although in practise, the variance of this method is lower than multionomial resampling and comparable to stratified resampling [67]. A nice property of systematic resampling is that one only needs to generate a single random number compared to N random numbers of the previous two methods. The implementation of systematic resampling is completely analogous to that of stratified resampling shown in Algorithm 14. A visual representation of multinomial, stratified and systematic resampling is shown in Fig. A.2.

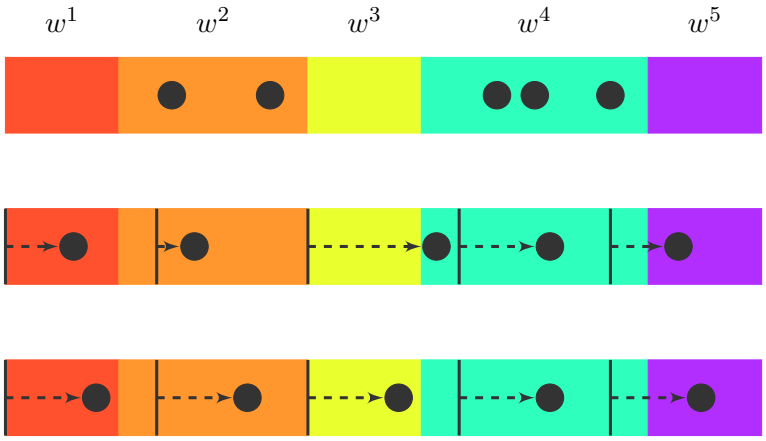


Figure A.2: A visual comparison of multinomial (**top**), stratified (**middle**) and systematic resampling (**bottom**). The length of the color segments corresponds to the particle weight. The black circles represent the random samples for every method. The black horizontal lines show the partitioning of the weights into equal subintervals. Note that for stratified and systematic resasmpling, there is exactly one sample in each subinterval. The difference being that for systematic resampling, the offset in each partition is the same, given by just one randomly drawn number.

A.3.2 Parallel resampling

Recalling the SIR filter from Algorithm 2, one can notice that the first two steps, namely, the sampling of particles and weight computation, are easy to parallelize. This is because in both steps, the particles have no intra-particle dependencies and thus can be handled separately. With the advent of general purpose GPUs, taking advantage of parallel computation can offer immense speedups in the runtime of a particle filter. The potential speedups are even more enticing when the number of particles is very large. It is therefore desirable to parallelize the resampling step as well. The sampling algorithms we have introduced thus far all have collective operations which are difficult to parallelize efficiently. In this section, we show two resampling algorithms which, unlike the algorithms shown so far, can be easily parallelized and thus are well suited for use cases where performance is critical. For a more detailed review of resampling algorithms, see [67].

Metropolis-Hastings sampling

The Metropolis algorithm, also called Metropolis-Hastings (M-H) [60, 17], is a method to obtain samples from a distribution which is difficult to sample from directly. This problem is circumvented by sampling from an entirely different distribution. Specifically, The M-H sampling is a Markov chain method that is capable of indirectly sampling from the target distribution $f(x)$ using an arbitrary proposal distribution $g(x_k|x_{k-1})$. For the M-H sampler to work, it is required that the target density can be evaluated pointwise up to a constant factor. The iterative sampling procedure is given below:

1. First, we choose a proposal distribution $g(x_k|x_{k-1})$. This distribution proposes a new sample given the sample from a previous iteration.
2. Second, the algorithm picks an initial sample x_1 at random.
3. Finally, at time k, the algorithm generates a new sample $x' \sim g(x_k|x_{k-1})$ and computes the acceptance ratio $\alpha = f(x')/f(x_{k-1})$. If $u \leq \alpha$, where $u \sim \mathcal{U}(0,1)$ is drawn from a uniform distribution, we accept the sample and set $x_{k+1} = x'$. If $u \geq \alpha$, we reject x' and set $x_{k+1} = x_k$. Then, the sequence $\{x_1, ..., x_n\}$ approaches the target distribution $f(x)$ in the limit.

The pseudocode of the M-H sampler is given in Algorithm 15. This method works for general multivariate distributions but can be simplified greatly in the context of particle resampling. Our target distribution $f(x)$ is $f(x^i) = w_i$. The proposal distribution is chosen to be $g(x_k|x_{k-1}) = \mathcal{U}(\{1, \ldots, N\})$. In

other words, the new sample is chosen randomly from a uniform distribution over all samples.

Since we wish to parallelize the sampling algorithm, we do not generate a sequence $\{x_1, ..., x_n\}$, but use a modification given in Algorithm 16 that generates only a single sample. If we then run N instances of this algorithm in parallel, we will obtain N samples. The modified algorithm is parametrized by B, specifying the number of iterations before it is assumed to have converged to the target distribution. Since any setting of B must be finite, the Metropolis sampler is always biased. The choice of B is directly linked to the sampling bias and is described in [67].

Algorithm 15: General Metropolis-Hastings sampling

input : A density $f(x)$ proportional to the tartget distribution $P(x)$, the proposal distribution $g(x_k|x_{k-1})$ and the number of samples to generate given as K
output : An array of samples $\{x_i,\ i = 1, 2, ..., K\}$ sampled from $P(x)$

$x_1 \leftarrow$ Select arbitrary x_i
for $k \leftarrow 2$ **to** K **do**
    $x' \sim g(x_k|x_{k-1})$
    $\alpha = f(x')/f(x_{k-1})$
    $u \sim \mathcal{U}(0, 1)$
    if $u \leq \alpha$ **then**
        $x_k = x'$
    else
        $x_k = x_{k-1}$
end

Algorithm 16: Metropolis-Hastings resampling

input : An array of particle weights $\{w_i,\ i = 1, 2, ..., N\}$ Number of iterations B
output : An ancestor vector $\{a_i,\ i = 1, 2, ..., N\}$ mapping new particle indices to old indices

for $i \leftarrow 1$ **to** N **do**
    $k = i$
    for $n \leftarrow 1$ **to** B **do**
        $u \sim \mathcal{U}(0, 1)$
        $j \sim \mathcal{U}(\{1, \ldots, N\})$
        if $u \leq w_j/w_k$ **then**
            $k = j$
    end
    $a_i = j$
end

Acceptance-rejection sampling

Another sampling method well-suited for parallelization is Acceptance-rejection sampling or simply rejection sampling [83, 26, 17]. Similarly to M-H sampling, rejection sampling can be used to sample from an arbitrary distribution provided we can evaluate its probability density $f(x)$. Moreover, compared to the M-H sampling, rejection sampling is a conceptually much simpler method.

In particular, assuming we have the target density $f(x)$ which in general may be difficult to sample from, we search for a density $h(x)$ which can be efficiently sampled and satisfies $f(x) \leq ch(x)\ \forall x$ for some $c \in \mathbb{R}$. The sampling algorithm is then as follows:

1. First, x is sampled from $h(x)$.
2. Second, we sample $u \sim \mathcal{U}(0, 1)$.
3. Finally, if $u \leq f(x)/ch(x)$, then we accept x and it holds that x comes from the target distribution $f(x)$. Othwerise if $u \geq f(x)/ch(x))$, we reject the sample and repeat the process until a sample is accepted.

Algorithm 17 illustrates this in detail. It is important to note that if the bound on $f(x)$ given by $ch(x)$ is not tight, the algorithm may produce many rejections before accepting a sample. We can easily adapt this general algorithm for sampling from discrete particle weights as shown in Algorithm 18. Our target $f(x)$ is again the discrete density $f(x^i) = w^i$. The bounding density is chosen to be the uniform discrete distribution $h(x) = 1/N$. The constant c should be chosen such that the bound is tight to minimize the number of rejections. The obvious choice is to set $ch(x) = w_{max}$ where $w_{max} = \max_i w^i$. Unfortunately, computing the maximum of a set of weights is difficult to efficiently parallelize and goes against the idea of parallel sampling. The choice of a suitable heuristic bound is elaborated in [67, 66]. As a final note, in order to further reduce the variance of the new particle set, the initial guess may be set to i instead of sampling it from $\mathcal{U}(\{1, \ldots, N\})$.

Parallel stratified & systematic resampling

As we discussed in previous sections, both stratified and systematic resampling attain a lower variance compared to other resampling schemes [67]. This means that they produce more consistent results and are less sensitive to the drawn random numbers. Due to this fact, either stratified or systematic resampling is usually the preferred resampling algorithm. However, these algorithms are not easily parallelizable due to their data dependencies apparent

Algorithm 17: General Acceptance-rejection sampling

input : A target density $f(x)$, bounding density $h(x)$ and a constant c s.t. $f(x) \leq ch(x)\ \forall x$
output : A sample x sampled from $f(x)$

```
x ~ h(x)
u ~ U(0, 1)
while u ≥ f(x)/ch(x) do
    x ~ h(x)
    u ~ U(0, 1)
end
```

Algorithm 18: rejection resampling

input : An array of particle weights $\{w_i,\ i = 1, 2, ..., N\}$
output : An ancestor vector $\{a_i,\ i = 1, 2, ..., N\}$ mapping new particle indices to old indices

```
for i ← 1 to N do
    j ~ U{1, N}
    u ~ U(0, 1)
    while u > w_j / w_max do
        j ~ U({1, ..., N})
        u ~ U(0, 1)
    end
    a_i ← j
end
```

in Algorithm 14. Luckily, the data dependencies can be removed [28, 69]. This is based on the fact that it is possible to compute the replication count for every particle directly using the following formula:

$$N_i = \lceil Nc_i - u_{\lceil Nc_i \rceil} \rceil. \tag{A.19}$$

Here, N denotes the total number of particles, u_i is the ith random number sampled from $\mathcal{U}(0, 1]$, $c_i = \sum_{j=1}^{i} w^i$ is a partial sum of weights and N_i is the number of particles selected upon reaching the ith particle. With this expression, we can calculate the number of offsprings for a particle $\boldsymbol{x}^i$ simply as $o_i = N_i - N_{i-1}$. The expression for N_i can be adapted to systematic resampling by replacing $u_{\lceil Nc_i \rceil}$ in the expression with just u as systematic resampling draws only a single random number.

Note that in the above we need to compute the cumulative sum of weights c_i. Cumulative sum, also known as a prefix sum or scan, is a well-known elemen-

tary algorithm in the field of parallel computing, and there exist work-efficient algorithms to compute it [34]. Pseudocode of the parallel implementation of stratified and systematic resampling is provided in Algorithm 19 and 20.

Algorithm 19: Parallel stratified resampling

input : An array of particle weights $\{w_1, ..., w_N\}$
output : An ancestor vector $\{a_1, ..., a_N\}$ mapping new particle indices to old indices

$\boldsymbol{c} \leftarrow ParallelCummulativeSum(\boldsymbol{w})$

for $i \leftarrow 1$ **to** N **do in parallel**
    Draw $u_i \sim \mathcal{U}(0, 1)$
end

for $i \leftarrow 1$ **to** N **do in parallel**
    $l = \lceil((c_i - w_i) \times N) - u_{\lceil (c_i - w_i) \times N \rceil}\rceil$
    $r = \lceil(c_i \times N) - u_{\lceil c_i \times N \rceil}\rceil$
    for $j \leftarrow l$ **to** r **do**
        $a_j = i$
    end
end

A.3.3 Obtaining the marginal posterior

The recursive formula of the full posterior is given as

$$p(\boldsymbol{x}_{1:k}|\boldsymbol{z}_{1:t}) = \frac{p(\boldsymbol{z}_k|\boldsymbol{x}_k)p(\boldsymbol{x}_{1:k}|\boldsymbol{x}_{k-1})p(\boldsymbol{x}_{1:k-1}|\boldsymbol{z}_{1:k-1})}{p(\boldsymbol{z}_k|\boldsymbol{z}_{1:k-1})}. \tag{A.20}$$

We can derive the marginal formula (3.16) by marginalizing out $\boldsymbol{x}_{1:k-1}$:

Algorithm 20: Parallel systematic resampling

input : An array of particle weights $\{w_1, ..., w_N\}$
output : An ancestor vector $\{a_1, ..., a_N\}$ mapping new particle indices to old indices

$\boldsymbol{c} \leftarrow ParallelCummulativeSum(\boldsymbol{w})$

Draw $u \sim \mathcal{U}(0, 1)$

for $i \leftarrow 1$ **to** N **do in parallel**
  $l = \lceil((c_i - w_i) \times N) - u\rceil$
  $r = \lceil(c_i \times N) - u\rceil$
  for $j \leftarrow l$ **to** r **do**
    $a_j = i$
  end
end

$$p(\boldsymbol{x}_k|\boldsymbol{z}_{1:t}) = \int p(\boldsymbol{x}_{1:k}|\boldsymbol{z}_{1:t})d\boldsymbol{x}_{1:k-1} \tag{A.21}$$

$$= \int \frac{p(\boldsymbol{z}_k|\boldsymbol{x}_k)p(\boldsymbol{x}_k|\boldsymbol{x}_{k-1})p(\boldsymbol{x}_{1:k-1}|\boldsymbol{z}_{1:k-1})}{p(\boldsymbol{z}_k|\boldsymbol{z}_{1:k-1})}d\boldsymbol{x}_{1:k-1} \tag{A.22}$$

$$= p(\boldsymbol{z}_k|\boldsymbol{x}_k)\frac{\int p(\boldsymbol{x}_k|\boldsymbol{x}_{k-1})p(\boldsymbol{x}_{1:k-1}|\boldsymbol{z}_{1:k-1})d\boldsymbol{x}_{1:k-1}}{p(\boldsymbol{z}_k|\boldsymbol{z}_{1:k-1})} \tag{A.23}$$

$$= p(\boldsymbol{z}_k|\boldsymbol{x}_k)\frac{\int p(\boldsymbol{x}_k|\boldsymbol{x}_{k-1})\left(\int p(\boldsymbol{x}_{1:k-1}|\boldsymbol{z}_{1:k-1})d\boldsymbol{x}_{1:k-2}\right)d\boldsymbol{x}_{k-1}}{p(\boldsymbol{z}_k|\boldsymbol{z}_{1:k-1})} \tag{A.24}$$

$$= p(\boldsymbol{z}_k|\boldsymbol{x}_k)\frac{\int p(\boldsymbol{x}_k|\boldsymbol{x}_{k-1})p(\boldsymbol{x}_{k-1}|\boldsymbol{z}_{1:k-1})d\boldsymbol{x}_{k-1}}{p(\boldsymbol{z}_k|\boldsymbol{z}_{1:k-1})} \tag{A.25}$$

$$= \frac{p(\boldsymbol{z}_k|\boldsymbol{x}_k)p(\boldsymbol{x}_k|\boldsymbol{z}_{1:k-1})}{p(\boldsymbol{z}_k|\boldsymbol{z}_{1:k-1})}. \tag{A.26}$$

A.3.4 Optimal sampling distribution

This proof was adapted from [23]. In a particle filter, the new weight is calculated as

$$w_k^i = w_{k-1}^i \frac{p(\boldsymbol{z}_k|\boldsymbol{x}_k^i)p(\boldsymbol{x}_k^i|\boldsymbol{x}_{k-1}^i)}{q(\boldsymbol{x}_k^i|\boldsymbol{x}_{k-1}^i, \boldsymbol{z}_k)}. \tag{A.27}$$

We want to show that the choice of $p(\boldsymbol{x}_k^i|\boldsymbol{x}_{k-1}^i, \boldsymbol{z}_k)$ as the proposal density is optimal in the sense that it minimizes the variance of the weights conditioned on $\boldsymbol{x}_{k-1}^i$ and $\boldsymbol{z}_k$. The variance can be expressed as

$$Var_q[w_k^i] = \left(w_{k-1}^i\right)^2 Var_q\left[\frac{p(\boldsymbol{z}_k|\boldsymbol{x}_k^i)p(\boldsymbol{x}_k^i|\boldsymbol{x}_{k-1}^i)}{q(\boldsymbol{x}_k^i|\boldsymbol{x}_{k-1}^i, \boldsymbol{z}_k)}\right] \tag{A.28}$$

$$= \left(w_{k-1}^i\right)^2 \left[\int \frac{(p(\boldsymbol{z}_k|\boldsymbol{x}_k^i)p(\boldsymbol{x}_k^i|\boldsymbol{x}_{k-1}^i))^2}{q(\boldsymbol{x}_k^i|\boldsymbol{x}_{k-1}^i, \boldsymbol{z}_k)} d\boldsymbol{x}_k - \left(\int p(\boldsymbol{z}_k|\boldsymbol{x}_k^i)p(\boldsymbol{x}_k^i|\boldsymbol{x}_{k-1}^i)d\boldsymbol{x}_k\right)^2\right] \tag{A.29}$$

$$= \left(w_{k-1}^i\right)^2 \left[\int \frac{(p(\boldsymbol{z}_k|\boldsymbol{x}_k^i)p(\boldsymbol{x}_k^i|\boldsymbol{x}_{k-1}^i))^2}{q(\boldsymbol{x}_k^i|\boldsymbol{x}_{k-1}^i, \boldsymbol{z}_k)} d\boldsymbol{x}_k - p(\boldsymbol{z}_k|\boldsymbol{x}_{k-1}^i)^2\right]. \tag{A.30}$$

Setting $q(\boldsymbol{x}_k^i|\boldsymbol{x}_{k-1}^i, \boldsymbol{z}_k) = p(\boldsymbol{x}_k^i|\boldsymbol{x}_{k-1}^i, \boldsymbol{z}_k)$, we obtain

$$= \left(w_{k-1}^i\right)^2 \left[\int \frac{(p(\boldsymbol{z}_k|\boldsymbol{x}_k^i)p(\boldsymbol{x}_k^i|\boldsymbol{x}_{k-1}^i))^2}{p(\boldsymbol{x}_k^i|\boldsymbol{x}_{k-1}^i, \boldsymbol{z}_k)} d\boldsymbol{x}_k - p(\boldsymbol{z}_k|\boldsymbol{x}_{k-1}^i)^2\right] \tag{A.31}$$

$$\overset{Bayes}{=} \left(w_{k-1}^i\right)^2 \left[\int \frac{(p(\boldsymbol{z}_k|\boldsymbol{x}_k^i)p(\boldsymbol{x}_k^i|\boldsymbol{x}_{k-1}^i))^2}{\frac{p(\boldsymbol{z}_k|\boldsymbol{x}_k^i, \boldsymbol{x}_{k-1}^i)p(\boldsymbol{x}_k^i|\boldsymbol{x}_{k-1}^i)}{p(\boldsymbol{z}_k|\boldsymbol{x}_{k-1}^i)}} d\boldsymbol{x}_k - p(\boldsymbol{z}_k|\boldsymbol{x}_{k-1}^i)^2\right] \tag{A.32}$$

$$\overset{Markov}{=} \left(w_{k-1}^i\right)^2 \left[\int \frac{(p(\boldsymbol{z}_k|\boldsymbol{x}_k^i)p(\boldsymbol{x}_k^i|\boldsymbol{x}_{k-1}^i))^2}{\frac{p(\boldsymbol{z}_k|\boldsymbol{x}_k^i)p(\boldsymbol{x}_k^i|\boldsymbol{x}_{k-1}^i)}{p(\boldsymbol{z}_k|\boldsymbol{x}_{k-1}^i)}} d\boldsymbol{x}_k - p(\boldsymbol{z}_k|\boldsymbol{x}_{k-1}^i)^2\right] \tag{A.33}$$

$$= \left(w_{k-1}^i\right)^2 \left[\int p(\boldsymbol{z}_k|\boldsymbol{x}_k^i)p(\boldsymbol{x}_k^i|\boldsymbol{x}_{k-1}^i)p(\boldsymbol{z}_k|\boldsymbol{x}_{k-1}^i)d\boldsymbol{x}_k - p(\boldsymbol{z}_k|\boldsymbol{x}_{k-1}^i)^2\right] \tag{A.34}$$

$$= \left(w_{k-1}^i\right)^2 p(\boldsymbol{z}_k|\boldsymbol{x}_{k-1}^i)\left[\int p(\boldsymbol{z}_k|\boldsymbol{x}_k^i)p(\boldsymbol{x}_k^i|\boldsymbol{x}_{k-1}^i)d\boldsymbol{x}_k - p(\boldsymbol{z}_k|\boldsymbol{x}_{k-1}^i)\right] \tag{A.35}$$

$$= \left(w_{k-1}^i\right)^2 p(\boldsymbol{z}_k|\boldsymbol{x}_{k-1}^i)\left[p(\boldsymbol{z}_k|\boldsymbol{x}_{k-1}^i) - p(\boldsymbol{z}_k|\boldsymbol{x}_{k-1}^i)\right] \tag{A.36}$$

$$= 0 \tag{A.37}$$

As the variance is equal to zero, the choice of $p(\boldsymbol{x}_k^i|\boldsymbol{x}_{k-1}^i, \boldsymbol{z}_k)$ as the proposal density is optimal.

A.4 Simultaneous localization and mapping

A.4.1 Extracting the map

In this section, we provide a detailed description of three heuristic approaches that are suitable to extract the current map estimate from the FastSLAM algorithm.

Maximum likelihood

The simplest approach is to only consider the map provided by the particle with the highest weight. As the weight of a particle provides a certain measure of fit or likelihood, it is reasonable to assume that such a particle will contain the most likely map. The issue with this approach is its lack of robustness. In a given iteration, a substandard particle can be assigned the highest weight due to a higher measurement error or simply the probabilistic nature of the problem. Thus, a map which may differ a lot from the ground truth will be extracted. By considering only the best particle, we discard the information from all other particles. This becomes less of a problem the longer robot explores. Due to resampling, all particles will eventually share a common history which grows over time. Thus, if the robot does not continuously discover new areas, the particle maps will be largely the same.

K-Means

To incorporate multiple particles in the final map estimate, two questions need to be answered. First, how many landmarks should be in the final map, and second, how to associate landmarks from different particles. Here we remark that this landmark association differs from the problem of data association we have described previously. Data association deals with pairwise associations. In contrast, here we need to associate across potentially thousands of different particles with hundreds of landmarks each, which is not tractable in general.

The problem of associating landmarks from different particles can be solved heuristically using weighted K-Means. K-Means is an unsupervised learning technique for estimating clusters in a dataset. It operates in two steps which are repeated until the algorithm converges. First, the assignment of each data point is computed based on the distance from the cluster centers. Second, the new cluster centers are recomputed as the average of all data points assigned to it in step one. The input to a standard K-Means algorithm is a set of

data points $D = \{x_1, ..., x_N\}$ and the number of clusters k. The output is an assignment of each data point to a cluster which minimizes the total distance of points to the cluster centers. Formally, K-Means finds cluster centers μ_i by minimizing the following cost function:

$$C^* = \arg\min_C \sum_{i=1}^{k} \sum_{x \in C_i} ||x - \mu_i||^2, \tag{A.38}$$

where $C = \{C_1, ..., C_k\}$ satisfying

$$\bigcup_{i=1}^{k} C_i = D \tag{A.39}$$

$$C_i \cap C_j = \emptyset \ \forall i, j \in \{1, ..., k\}, i \neq j. \tag{A.40}$$

The cluster centers are computed as a simple average of the points belonging to the cluster:

$$\mu_i = \frac{1}{|C_i|} \sum_{x \in C_i} x \tag{A.41}$$

K-Means can also be weighted, i.e., every point x has an associated weight w_x:

$$C^* = \arg\min_C \sum_{i=1}^{k} \sum_{x \in C_i} w_x ||x - \mu_i||^2. \tag{A.42}$$

Analogously, the cluster centers can then be computed as a weighted average:

$$\mu_i = \frac{\sum_{x \in C_i} w_x x}{\sum_{x \in C_i} w_x}. \tag{A.43}$$

To cast the map extraction problem as an instance of K-Means, we first set D as the set of all landmarks from all particles. The weight of a data point is the weight of the particle it came from. Since K-Means needs to know the number of clusters k beforehand, we estimate it as the number of landmarks in the particle with the highest weight. Moreover, the landmarks belonging to the most likely particle are set as the initial guess for K-Means. The final map is then taken as the computed cluster centers.

Gaussiam Mixture models

While K-Means is a good clustering method, it makes several assumptions that usually do not hold in the context of SLAM. K-means assumes that the landmark coordinates are uncorrelated and that the noise is identical for every landmark cluster. This is problematic, for example, in the case of a sensor providing range & bearing measurements which typically results in a high correlation between the landmark coordinates. Moreover, landmarks that were observed more often will tend to have a lower uncertainty than landmarks observed only a handful of times. For the purposes of extracting a map, a clustering algorithm should take these concerns into account.

Gaussiam Mixture models (GMM) can model this problem by not making any assumptions on the structure of the covariance matrix. For this reason, GMM can be thought of as a generalization of K-Means. Formally, GMM assumes that the underlying model that the data comes from is of the form

$$p(\theta) = \sum_{i=1}^{k} \phi_i \mathcal{N}(\mu_i, \Sigma_i). \tag{A.44}$$

The model is a mixture of k Gaussian distributions with mean μ_i and covariance Σ_i. The cluster weight is given by ϕ_i. GMM, similarly to K-Means, can be solved iteratively using the Expectation-Maximization (E-M) algorithm. Just like K-Means, the algorithm is a two-step iterative minimization that is repeated until convergence. In the expectation step, every data point x_j is assigned the likelihood of being generated by every Gaussian:

$$p(x_j|\theta_i) = \mathcal{N}(x_j; \mu_i, \Sigma_i). \tag{A.45}$$

In the maximization step, a data point is assigned to a cluster which maximizes the above likelihood:

$$C_i = \{x_j \in D \mid \forall k : p(x_j|\theta_i) > p(x_j|\theta_k)\} \tag{A.46}$$

With this new assignment, the mean and covariance of the Gaussians are recomputed as follows:

$$\mu_i = \frac{1}{|C_i|} \sum_{x \in C_i} x, \tag{A.47}$$

$$\Sigma_i = \frac{1}{|C_i|} \sum_{x \in C_i} (x - \mu_i)(x - \mu_i)^T. \tag{A.48}$$

Both steps are repeated until convergence. Convergence is assumed when the gain in log-likelihood is below a specified threshold. Similarly to K-Means, GMMs can be adapted to work with a weighted dataset. The equations (A.47), (A.48) are then adapted as follows:

$$\mu_i = \frac{\sum_{x \in C_i} w_x x}{\sum_{x \in C_i} w_x}, \tag{A.49}$$

$$\Sigma_i = \frac{\sum_{x \in C_i} w_x (x - \mu_i)(x - \mu_i)^T}{\sum_{x \in C_i} w_x}. \tag{A.50}$$

The FastSLAM map estimation is analogous to K-Means. The number of clusters is again set as the number of landmarks in the most likely particle. The initial mean and covariance of each are again either taken from the most likely particle or they can also be estimated by first running K-Means and using its cluster centers to initialize the means and the within-cluster covariance to initialize the covariance matrices. The initial cluster weights ϕ_i are set to $1/k$.

A.5 FastSLAM GPU implementation

A.5.1 Resampling

The parallel modification of the permute algorithm consists of two procedures. The algorithm also uses an auxilliary vector $\boldsymbol{d}$ and an output vector $\boldsymbol{c}$ which is the resulting conflict-free ancestor vector. In the first procedure, called *prepermute*, the vector $\boldsymbol{d}$ is used to claim positions in the output vector $\boldsymbol{c}$. If there are multiple copies of the same particle in $\boldsymbol{a}$, only one of them will succeed in claiming a space. The prepermute procedure ensures that the condition (7.1) holds. However, the resulting ancestor vector is not yet valid as duplicate particles have not yet claimed a position. This is addressed in the main permute procedure. By constructing a unique sequence given by the rule $x \leftarrow d_x$, an unclaimed position is eventually found for every particle. It can be shown that every particle will find a different unclaimed position [66]. A visual representation of this algorithm is shown in Fig. A.3. The pseudocode of the two procedures is provided in Algorithm 21 and 22.

Note that the atomic minimum operation in the prepermute procedure merely ensures a deterministic behaviour of the algorithm. If determinism is not a concern, then atleast in CUDA, the atomic operation can be replaced with a simple assignment. The CUDA specification guarantees that in the case of a write conflict, at least one write transaction is guaranteed to be executed, although it is undefined which one. The prepermute procedure only needs one particle to succeed in claiming the space, thus this is safe to do. Removing the atomic operation may increase performance in cases when the ancestor vector contains many duplicates contesting a write lock for the same location in $\boldsymbol{d}$.

Algorithm 21: Prepermute

input : An ancestor vector $\{a_1, \ldots, a_N\}$
An auxilliary vector $\{d_1, \ldots, d_N\}$

```
1
2  // Mark free positions
3  for i ← 1 to N do in parallel
4      d_i = N + 1
5  end
6
7  // Claim positions
8  for i ← 1 to N do in parallel
9      atomic d_i ← min(d_{a_i}, i)
10 end
```

Algorithm 22: Parallel permute

input : An ancestor vector $\{a_1, \ldots, a_N\}$
An output vector $\{c_1, \ldots, c_N\}$

```
1
2  d ← Prepermute(a)
3
4  // Claim leftover positions
5  for i ← 1 to N do in parallel
6      x = d_{a_i}
7      if x ≠ i then
8          // Claim unsuccessful in Prepermute
9          x = i
10         while d_x < N + 1 do
11             x = d_x
12         end
13         d_x = i
14 end
15
16 // Write to output vector
17 for i ← 1 to N do in parallel
18     c_i = a_{d_i}
19 end
```

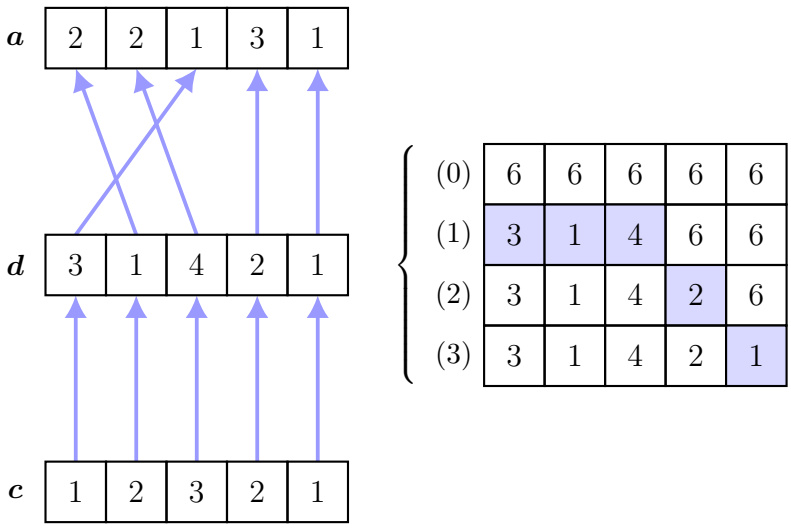


Figure A.3: An example of the parallel permute procedure for a case when the ancestor vector contains duplicate entries. With duplicate entries, only some particles are able to claim positions in the prepermute procedure, while the rest have to claim the remaining positions in the main procedure. To illustrate how the algorithm operates, the content of the auxilliary $\boldsymbol{d}$ vector is shown on the right side. First, the vector is preffiled with the value $N + 1 = 6$ which denotes an unclaimed position. Subsequently, (1) shows the contents after the particles claim positions in the prepermute procedure. We can see that a^3, a^1 and a^4 succeed. In step (2), a^2 claims a position and finally in (3), a^5 claims the final position. Note that that steps (2) and (3) are executed in parallel.

Figure A.4: The current (2021) autonomous (**foreground**) and electric (**background**) formulas

Appendix B

Bibliography

[1] Formula student driverless simulation. `https://github.com/FS-Driverless/Formula-Student-Driverless-Simulator`. Accessed: 2021-05-12.

[2] S. Ahn, J. Choi, N. L. Doh, and W. K. Chung. A practical approach for EKF-SLAM in an indoor environment: fusing ultrasonic sensors and stereo camera. *Autonomous robots*, 24(3):315–335, 2008.

[3] P. F. Alcantarilla, L. M. Bergasa, and F. Dellaert. Visual odometry priors for robust EKF-SLAM. In *2010 IEEE International Conference on Robotics and Automation*, pages 3501–3506, 2010.

[4] M. S. Arulampalam, S. Maskell, N. Gordon, and T. Clapp. A tutorial on particle filters for online nonlinear/non-Gaussian Bayesian tracking. *IEEE Transactions on Signal Processing*, 50(2):174–188, 2002.

[5] F. Auger, M. Hilairet, J. M. Guerrero, E. Monmasson, T. Orlowska-Kowalska, and S. Katsura. Industrial applications of the Kalman filter: A review. *IEEE Transactions on Industrial Electronics*, 60(12):5458–5471, Dec. 2013.

[6] T. Bailey and H. Durrant-Whyte. Simultaneous localization and mapping (SLAM): part ii. *IEEE Robotics Automation Magazine*, 13(3):108–117, 2006.

[7] T. Bailey, J. Nieto, J. Guivant, M. Stevens, and E. Nebot. Consistency of the EKF-SLAM algorithm. In *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3562–3568, 2006.

[8] T. Bailey, J. Nieto, and E. Nebot. Consistency of the FastSLAM algorithm. In *Proceedings 2006 IEEE International Conference on Robotics and Automation, 2006. ICRA 2006.*, pages 424–429. IEEE, 2006.

[9] N. Bergman. *Recursive Bayesian estimation: Navigation and tracking applications*. PhD thesis, Linköping University, 1999.

[10] M. Birsan. Recursive Bayesian method for magnetic dipole tracking with a tensor gradiometer. *IEEE Transactions on Magnetics*, 47(2):409–415, 2011.

[11] A. Bochkovskiy, C.-Y. Wang, and H.-Y. M. Liao. Yolov4: Optimal speed and accuracy of object detection. *arXiv preprint arXiv:2004.10934*, 2020.

[12] R. A. Brown. Building a balanced kd tree in o (kn log n) time. *arXiv preprint arXiv:1410.5420*, 2014.

[13] C. Cadena, L. Carlone, H. Carrillo, Y. Latif, D. Scaramuzza, J. Neira, I. Reid, and J. J. Leonard. Past, present, and future of simultaneous localization and mapping: Toward the robust-perception age. *IEEE Transactions on Robotics*, 32(6):1309–1332, 2016.

[14] B. Carlin, P. Carlin, N. Polson, and D. Stoffer. A Monte Carlo approach to nonnormal and nonlinear state-space modeling. *Journal of the American Statistical Association*, 87, 01 2003.

[15] P. Cheeseman, R. Smith, and M. Self. A stochastic map for uncertain spatial relationships. In *4th International Symposium on Robotic Research*, pages 467–474. MIT Press Cambridge, 1987.

[16] Z. Chen. Bayesian filtering: From Kalman filters to particle filters, and beyond. *Statistics*, 182, 01 2003.

[17] S. Chib and E. Greenberg. Understanding the Metropolis-Hastings algorithm. *The American Statistician*, 49(4):327–335, 1995.

[18] D. Crisan and A. Doucet. Convergence of sequential Monte Carlo methods. *Signal Processing Group, Department of Engineering, University of Cambridge, Technical Report CUEDIF-INFENGrrR38*, 1, 2000.

[19] F. Dellaert. Factor graphs and gtsam: A hands-on introduction. Technical report, Georgia Institute of Technology, 2012.

[20] F. Dellaert, D. Fox, W. Burgard, and S. Thrun. Monte Carlo localization for mobile robots. In *Proceedings 1999 IEEE International Conference on Robotics and Automation (Cat. No. 99CH36288C)*, volume 2, pages 1322–1328. IEEE, 1999.

[21] R. Douc and O. Cappe. Comparison of resampling schemes for particle filtering. In *ISPA 2005. Proceedings of the 4th International Symposium on Image and Signal Processing and Analysis, 2005.*, pages 64–69, 2005.

[22] A. Doucet, N. De Freitas, and N. Gordon. An introduction to sequential Monte Carlo methods. In *Sequential Monte Carlo methods in practice*, pages 3–14. Springer, 2001.

[23] A. Doucet, S. Godsill, and C. Andrieu. On sequential Monte Carlo sampling methods for Bayesian filtering. *Statistics and Computing*, 10, 04 2003.

[24] H. Durrant-Whyte and T. Bailey. Simultaneous localization and mapping: part i. *IEEE robotics & automation magazine*, 13(2):99–110, 2006.

[25] R. Fitzgerald. Divergence of the Kalman filter. *IEEE Transactions on Automatic Control*, 16(6):736–747, 1971.

[26] B. D. Flury. Acceptance–rejection sampling made easy. *SIAM Review*, 32:474–476, 1990.

[27] T. Furukawa, F. Bourgault, B. Lavis, and H. F. Durrant-Whyte. Recursive Bayesian search-and-tracking using coordinated uavs for lost targets. In *Proceedings 2006 IEEE International Conference on Robotics and Automation, 2006. ICRA 2006.*, pages 2521–2526, 2006.

[28] P. Gong, Y. O. Basciftci, and F. Ozguner. A parallel resampling algorithm for particle filtering on shared-memory architectures. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum*, pages 1477–1483, 2012.

[29] N. Gordon, B. Ristic, and S. Arulampalam. Beyond the Kalman filter: Particle filters for tracking applications. *Artech House, London*, 830(5):1–4, 2004.

[30] N. J. Gordon, D. J. Salmond, and A. F. Smith. Novel approach to nonlinear/non-Gaussian Bayesian state estimation. In *IEE proceedings F (radar and signal processing)*, volume 140, pages 107–113. IET, 1993.

[31] G. Grisetti, R. Kümmerle, C. Stachniss, and W. Burgard. A tutorial on graph-based SLAM. *IEEE Transactions on Intelligent Transportation Systems Magazine*, 2:31–43, 12 2010.

[32] F. Gustafsson. Particle filter theory and practice with positioning applications. *IEEE Aerospace and Electronic Systems Magazine*, 25(7):53–82, 2010.

[33] F. Gustafsson, F. Gunnarsson, N. Bergman, U. Forssell, J. Jansson, R. Karlsson, and P.-J. Nordlund. Particle filters for positioning, navigation, and tracking. *IEEE Transactions on signal processing*, 50(2):425–437, 2002.

[34] M. Harris, S. Sengupta, and J. D. Owens. Parallel prefix sum (scan) with CUDA. *GPU gems*, 3(39):851–876, 2007.

[35] G. Hendeby, R. Karlsson, and G. Fredrik. Particle filtering: The need for speed. *EURASIP Journal on Advances in Signal Processing*, 2010, 02 2010.

[36] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012.

[37] F. Hidalgo and T. Bräunl. Review of underwater SLAM techniques. In *2015 6th International Conference on Automation, Robotics and Applications (ICARA)*, pages 306–311. IEEE, 2015.

[38] J. D. Hol, T. B. Schon, and F. Gustafsson. On resampling algorithms for particle filters. In *2006 IEEE nonlinear statistical signal processing workshop*, pages 79–82. IEEE, 2006.

[39] Honghui Qi and J. B. Moore. Direct Kalman filtering approach for GPS/INS integration. *IEEE Transactions on Aerospace and Electronic Systems*, 38(2):687–693, April 2002.

[40] C. Hu, W. Chen, Y. Chen, and D. Liu. Adaptive Kalman filtering for vehicle navigation. *Journal of Global Positioning Systems*, 2(1):42–47, June 2003.

[41] S. Huang, Z. Wang, and G. Dissanayake. Sparse local submap joining filter for building large-scale maps. *IEEE Transactions on Robotics*, 24(5):1121–1130, 2008.

[42] S. Huh, D. H. Shim, and J. Kim. Integrated navigation system using camera and gimbaled laser scanner for indoor and outdoor autonomous flight of UAVs. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3158–3163. IEEE, 2013.

[43] R. E. Kalman. A new approach to linear filtering and prediction problems. *Transactions of the ASME–Journal of Basic Engineering*, 82(Series D):35–45, 1960.

[44] R. Karlsson and F. Gustafsson. Recursive Bayesian estimation: bearings-only applications. *IEE Proceedings - Radar, Sonar and Navigation*, 152(5):305, 2005.

[45] G. Kitagawa. Monte Carlo filter and smoother for non-Gaussian nonlinear state space models. *Journal of Computational and Graphical Statistics*, 5:1–25, 1996.

[46] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih. PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation. *Parallel Computing*, 38(3):157–174, 2012.

[47] A. Kong, J. S. Liu, and W. H. Wong. Sequential imputations and Bayesian missing data problems. *Journal of the American Statistical Association*, 89(425):278–288, Mar. 1994.

[48] H. W. Kuhn. The Hungarian method for the assignment problem. *Naval research logistics quarterly*, 2(1-2):83–97, 1955.

[49] R. Kümmerle, G. Grisetti, H. Strasdat, K. Konolige, and W. Burgard. G2o: A general framework for graph optimization. *2011 IEEE International Conference on Robotics and Automation*, pages 3607–3613, 2011.

[50] R. Kümmerle, B. Steder, C. Dornhege, M. Ruhnke, G. Grisetti, C. Stachniss, and A. Kleiner. On measuring the accuracy of SLAM algorithms. *Autonomous Robots*, 27(4):387–407, Sept. 2009.

[51] T. Lefebvre, H. Bruyninckx, and J. De Schutter. Kalman filters for non-linear systems: a comparison of performance. *International journal of Control*, 77(7):639–653, 2004.

[52] F. Lehmann. Recursive Bayesian filtering for multitarget track-before-detect in passive radars. *IEEE Transactions on Aerospace and Electronic Systems*, 48(3):2458–2480, 2012.

[53] K. Leung, Y. Halpern, T. Barfoot, and H. Liu. The UTIAS multi-robot cooperative localization and mapping dataset. *I. J. Robotic Res.*, 30:969–974, 07 2011.

[54] K. Y. Leung, T. D. Barfoot, and H. H. Liu. Decentralized localization of sparsely-communicating robot networks: A centralized-equivalent approach. *IEEE Transactions on Robotics*, 26(1):62–77, 2009.

[55] T. Li, S. Sun, T. Sattar, and J. Corchado Rodríguez. Fight sample degeneracy and impoverishment in particle filters: A review of intelligent approaches. *Expert Systems with Applications*, 41:3944–3954, 06 2014.

[56] J. S. Liu. Metropolized independent sampling with comparisons to rejection sampling and importance sampling. *Statistics and Computing*, 6(2):113–119, June 1996.

[57] F. Lu and E. Milios. Globally consistent range scan alignment for environment mapping. *Autonomous robots*, 4(4):333–349, 1997.

[58] H. Masnadi-Shirazi, A. Masnadi-Shirazi, and M.-A. Dastgheib. A step by step mathematical derivation and tutorial on Kalman filters, 2019.

[59] P. Maybeck. *Stochastic models, estimation and control*. Academic Press, New York, 1979.

[60] N. Metropolis, A. W. Rosenbluth, M. Rosenbluth, A. H. Teller, and E. Teller. Equation of state calculations by fast computing machines. *Journal of Chemical Physics*, 21:1087–1092, 1953.

[61] A. H. Mohamed and K. P. Schwarz. Adaptive Kalman filtering for INS/GPS. *Journal of Geodesy*, 73(4):193–203, May 1999.

[62] M. Montemerlo. *FastSLAM: A Factored Solution to the Simultaneous Localization and Mapping Problem with Unknown Data Association*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, July 2003.

[63] M. Montemerlo, S. Thrun, D. Koller, and B. Wegbreit. Fastslam 2.0: An improved particle filtering algorithm for simultaneous localization and mapping that provably converges. *Proc. IJCAI Int. Joint Conf. Artif. Intell.*, 06 2003.

[64] M. Montemerlo, S. Thrun, D. Koller, B. Wegbreit, et al. FastSLAM: A factored solution to the simultaneous localization and mapping problem. *Aaai/iaai*, 593598, 2002.

[65] A. I. Mourikis and S. I. Roumeliotis. A multi-state constraint Kalman filter for vision-aided inertial navigation. In *Proceedings 2007 IEEE International Conference on Robotics and Automation*, pages 3565–3572, 2007.

[66] L. M. Murray, A. Lee, and P. E. Jacob. Rethinking resampling in the particle filter on graphics processing units. *arXiv preprint arXiv:1301.4019*, 135, 2013.

[67] L. M. Murray, A. Lee, and P. E. Jacob. Parallel resampling in the particle filter. *Journal of Computational and Graphical Statistics*, 25(3):789–805, 2016.

[68] J. Neira and J. Tardos. Data association in stochastic mapping using the joint compatibility test. *IEEE Transactions on Robotics and Automation*, 17(6):890–897, 2001.

[69] M. A. Nicely and B. E. Wells. Improved parallel resampling methods for particle filtering. *IEEE Access*, 7:47593–47604, 2019.

[70] NVIDIA, P. Vingelmann, and F. H. Fitzek. CUDA, release: 10.2.89, 2020.

[71] F. Orderud. Comparison of Kalman filter estimation approaches for state space models with nonlinear measurements. In *Proc. of Scandinavian Conference on Simulation and Modeling*, pages 1–8. Citeseer, 2005.

[72] G. Pasricha. Kalman filter and its economic applications. MPRA paper, University Library of Munich, Germany, 2006.

[73] L. Paull, G. Huang, M. Seto, and J. J. Leonard. Communication-constrained multi-AUV cooperative SLAM. In *2015 IEEE international conference on robotics and automation (ICRA)*, pages 509–516. IEEE, 2015.

[74] L. M. Paz, J. D. Tardós, and J. Neira. Divide and conquer: EKF SLAM in $o(n)$. *IEEE Transactions on Robotics*, 24(5):1107–1120, 2008.

[75] D. Pollock. Recursive estimation in econometrics. *Computational Statistics & Data Analysis*, 44(1-2):37–75, Oct. 2003.

[76] K. Punithakumar, I. B. Ayed, I. G. Ross, A. Islam, J. Chong, and S. Li. Detection of left ventricular motion abnormality via information measures and Bayesian filtering. *IEEE Transactions on Information Technology in Biomedicine*, 14(4):1106–1113, 2010.

[77] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, A. Y. Ng, et al. ROS: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, Japan, 2009.

[78] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788, 2016.

[79] J. Redmon and A. Farhadi. Yolo9000: better, faster, stronger. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7263–7271, 2017.

[80] J. Redmon and A. Farhadi. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767*, 2018.

[81] L. Riazuelo, J. Civera, and J. M. Montiel. C2tam: A cloud framework for cooperative tracking and mapping. *Robotics and Autonomous Systems*, 62(4):401–413, 2014.

[82] J. T. Robinson. The k-d-b-tree. In *Proceedings of the 1981 ACM SIGMOD international conference on Management of data - SIGMOD '81*. ACM Press, 1981.

[83] R. Y. Rubinstein. *Simulation and the Monte Carlo Method*. John Wiley & Sons, Inc., New York, 1981.

[84] A. Sakai, D. Ingram, J. Dinius, K. Chawla, A. Raffin, and A. Paques. PythonRobotics: a Python code collection of robotics algorithms. *arXiv preprint arXiv:1808.10703*, 2018.

[85] R. Sameni, M. B. Shamsollahi, C. Jutten, and G. D. Clifford. A nonlinear Bayesian filtering framework for ECG denoising. *IEEE Transactions on Biomedical Engineering*, 54(12):2172–2185, 2007.

[86] A. Sanders. *An introduction to Unreal engine 4*. CRC Press, 2016.

[87] T. D. Sanger. Bayesian filtering of Myoelectric signals. *Journal of Neurophysiology*, 97(2):1839–1845, Feb. 2007.

[88] S. F. Schmidt. The Kalman filter - its recognition and development for aerospace applications. *Journal of Guidance and Control*, 4(1):4–7, Jan. 1981.

[89] R. Schneider and C. Georgakis. How to not make the extended Kalman filter fail. *Industrial & Engineering Chemistry Research*, 52(9):3354–3362, 2013.

[90] M. Schütz, N. Appenrodt, J. Dickmann, and K. Dietmayer. Occupancy grid map-based extended object tracking. In *2014 IEEE Intelligent Vehicles Symposium Proceedings*, pages 1205–1210. IEEE, 2014.

[91] F. Schwiegelshohn, E. Ossovski, and M. Hübner. A resampling method for parallel particle filter architectures:. *Microprocessors and Microsystems*, 47, 07 2016.

[92] S. Shah, D. Dey, C. Lovett, and A. Kapoor. Airsim: High-fidelity visual and physical simulation for autonomous vehicles. In *Field and Service Robotics*, 2017.

[93] B. Siciliano and O. Khatib. *Springer handbook of robotics*. springer, 2016.

[94] F. H. Sschlee, C. J. Standish, and N. F. Toda. Divergence in the Kalman filter. *AIAA Journal*, 5(6):1114–1120, June 1967.

[95] J. E. Stone, D. Gohara, and G. Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in Science Engineering*, 12(3):66–73, 2010.

[96] M. Szeles. Founding of a student research team developing an autonomous electric vehicle. Master's thesis, Czech Technical University in Prague, 2020.

[97] S. Thrun. Particle filters in robotics (invited talk). *arXiv preprint arXiv:1301.0607*, 2012.

[98] J. Walrand and A. Dimakis. Random processes in systems—lecture notes (department of electrical engineering and computer sciences). *University of California, Berkeley CA*, 94720, 2006.

[99] N. Waniek, J. Biedermann, and J. Conradt. Cooperative SLAM on small mobile robots. In *2015 IEEE International Conference on Robotics and Biomimetics (ROBIO)*, pages 1810–1815. IEEE, 2015.

[100] D. Whitley. A genetic algorithm tutorial. *Statistics and computing*, 4(2):65–85, 1994.

[101] S. Wienke, P. Springer, C. Terboven, and D. an Mey. OpenACC — first experiences with real-world applications. In *Euro-Par 2012 Parallel Processing*, pages 859–870. Springer Berlin Heidelberg, 2012.

[102] K. Yousif, A. Bab-Hadiashar, and R. Hoseinnezhad. An overview to visual odometry and visual SLAM: Applications to mobile robotics. *Intelligent Industrial Systems*, 1(4):289–311, 2015.

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Roun** Jméno: **Tomáš** Osobní číslo: **457086**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Specializace: **Umělá inteligence**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Navigační systém pro autonomní studentskou formuli

Název diplomové práce anglicky:

Navigation System for Autonomous Student Formula

Pokyny pro vypracování:

Seznam doporučené literatury:

1. Michael Montemerlo, Sebastian Thrun, Daphne Koller, Ben Wegbreit. FastSLAM: A Factored Solution to the Simultaneous Localization and Mapping Problem. Proc. AAAI, 2003.
2. T. Pire, T. Fischer, G. Castro, P. De Cristóforis, J. Civer, J. J. Berlles. S-PTAM: Stereo Parallel Tracking and Mapping. Robotics and Autnomous Systems. Vol. 93, 2017.
3. Guillaume Bresson, Zayed Alsayed, Li Yu, and Sebastien Glaser. Simultaneous Localization and Mapping: A Survey of Current Trends in Autonomous Driving. IEEE Trans. on Intelligent Vehicles, Vol. 2, No. 3, 2017.

Jméno a pracoviště vedoucí(ho) diplomové práce:

Ing. Jan Čech, Ph.D., skupina vizuálního rozpoznávání FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **21.02.2021** Termín odevzdání diplomové práce: **21.05.2021**

Platnost zadání diplomové práce: **19.02.2023**

Ing. Jan Čech, Ph.D. podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

. | Datum převzetí zadání

Podpis studenta