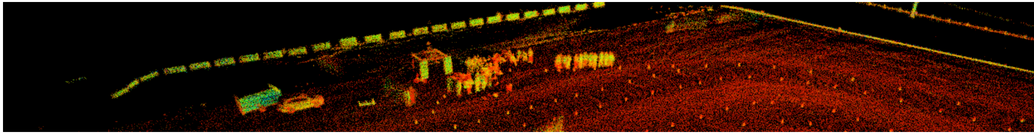
UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Centre de Formació Interdisciplinària Superior

香港科技大學
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

BACHELOR'S THESIS

LIMO-Velo

A Real-Time, Robust, Centimeter-Accurate Estimator for Vehicle Localization and Mapping under Racing Velocities



Andreu Huguet Segarra

Advisor (HKUST):
Qifeng Chen

Tutor (UPC):
Josep R. Casas

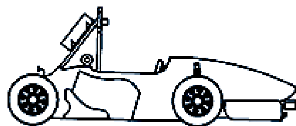
February 2022

In partial fulfilment of the requirements for the
Bachelor's degree in Mathematics
Bachelor's degree in Data Science and Engineering

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat de Matemàtiques i Estadística

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola Tècnica Superior d'Enginyeria de Telecomunicació de Barcelona

telecos BCN



BCN eMOTORSPORT

Acknowledgments

I would like to thank all the institutions that have made this thesis possible:

- La Universitat Politècnica de Catalunya, en especial menció al Centre de Formació Interdisciplinària Superior que m'ha contactat amb el professor Chen i ha fet possible la mobilitat a Hong Kong. També a l'Escola de Telecomunicacións (ETSETB), per haver apostat pel projecte del cotxe autònom, confiant en els estudiants i sense saber si podrien haver-hi resultats. Finalment, al meu tutor Josep R. Casas per sempre donar-me bons consells de com estructurar les meves idees per escriure la tesi.

- Al meu equip BCN eMotorsport, per haver mantingut en bon estat un cotxe que es sortia de la primer corba al Gener i va acabar anant volant a 60km/h tan sols quatre mesos després, al maig. També per seguir amb l'objectiu de continuar sent un pol de talent, arribar a tocar el podi almenys d'una categoria a Alemania i posar a Barcelona a dalt dels rànquings, on li toca estar.

- To the Hong Kong University Science and Technology, giving me the facilities and resources to work on my thesis without difficulties. Especially to my advisor, Dr. Qifeng Chen for his advice and expertise on the academy world, formerly unknown for me. Also for his guidelines on how to adequately prove that our work outperforms other algorithms.

I l'agraïment més important, agraïr a la meva família i amics per tot el seu suport donat en els moments més díficils i el donar el seu màxim en els moments més eufòrics. Ja sabeu qui sou, moltes gràcies per tots aquests anys i pels molts més que venen.

Abstract

Català

Treballs recents sobre localització de vehicles i mapeig dels seus entorns es desenvolupen per a dispositius portàtils o robots terrestres que assumeixen moviments lents i suaus. Contràriament als entorns de curses d'alta velocitat. Aquesta tesi proposa un nou model d'SLAM, anomenat LIMO-Velo, capaç de corregir el seu estat amb una latència extremadament baixa tractant els punts LiDAR com un flux de dades. Els experiments mostren un salt en robustesa i en la qualitat del mapa mantenint el requisit de correr en temps real. El model aconsegueix una millora relativa del 20% en el KITTI dataset d'odometria respecte al millor rendiment existent; no deriva en un sol escenari. La qualitat del mapa a nivell de centímetre es manté amb velocitats que poden arribar a 20 m/s i 500 graus/s. Utilitzant les biblioteques obertes IKFoM i ikd-Tree, el model funciona x10 més ràpid que la majoria de models d'última generació. Mostrem que LIMO-Velo es pot generalitzar per executar l'eliminació dinàmica d'objectes, com ara altres agents a la carretera, vianants i altres.

English

Recent works on localizing vehicles and mapping their environments are developed for handheld devices or terrestrial robots which assume slow and smooth movements. Contrary to high-velocity racing environments. This thesis proposes a new SLAM model, LIMO-Velo, capable of correcting its state at extreme low-latency by treating LiDAR points as a data stream. Experiments show a jump in robustness and map quality while maintaining the real-time requirement. The model achieves a 20% relative improvement on the KITTI Odometry dataset over the existing best performer; it does not drift in a single scenario. Centimeter-level map quality is still achieved under racing velocities that can go up to 20m/s and 500deg/s. Using the IK-

FoM and ikd-Tree open libraries, the model performs x10 faster than most state-of-the-art models. We show that LIMO-Velo can be generalized to work under dynamic object removal such as other agents in the road, pedestrians, and more.

Castellano

Trabajos recientes sobre la localización de vehículos y el mapeo de sus entornos se desarrollan para dispositivos portátiles o robots terrestres que asumen movimientos lentos y suaves. Al contrario de los entornos de carreras de alta velocidad. Esta tesis propone un nuevo modelo SLAM, LIMO-Velo, capaz de corregir su estado en latencia extremadamente baja al tratar los puntos LiDAR como un flujo de datos. Los experimentos muestran un salto en la solidez y la calidad del mapa mientras se mantiene el requisito de tiempo real. El modelo logra una mejora relativa del 20% en el conjunto de datos de KITTI Odometry sobre el mejor desempeño existente; no deriva en un solo escenario. La calidad del mapa de nivel centimétrico todavía se logra a velocidades de carrera que pueden llegar hasta 20 m/s y 500 grados/s. Usando las bibliotecas abiertas IKFoM e ikd-Tree, el modelo funciona x10 más rápido que la mayoría de los modelos de última generación. Mostramos que LIMO-Velo se puede generalizar para trabajar bajo la eliminación dinámica de objetos, como otros agentes en la carretera, peatones y más.

Keywords

Català

SLAM, percepció, estimació, conducció autònoma, Fòrmula Student Driverless, cotxes autònoms, robots autònoms, localització, mapeig, LiDAR, Unitat de Mesures Inercials, filtre de Kalman, temps real, odometria directa.

English

SLAM, perception, estimation, full self-driving, Formula Student Driverless, autonomous cars, autonomous robots, localization, mapping, LiDAR, Inertial Measurement Unit, Kalman Filter, real-time, direct odometry.

Castellano

SLAM, percepción, estimación, conducción autónoma, Fórmula Student Driverless, coches autónomos, robots autónomos, localización, mapeo, LiDAR, Unidad de Mesuras Inerciales, filtro de Kalman, tiempo real, odometria directa.

AMS codes

68T45

Contents

1 Introduction **1**

1.1 Motivation . . . 1

1.1.1 Barcelona's first autonomous car . . . 1

1.1.2 Formula Student . . . 2

1.1.3 An unresolved problem . . . 2

1.2 General problem statement . . . 3

1.2.1 Understanding our environment . . . 3

1.2.2 Localizating ourselves inside the map . . . 3

1.2.3 Simultaneous Localization and Mapping . . . 4

1.2.4 Implementations of the SLAM problem . . . 4

1.3 Problem statement - Formula Student . . . 4

1.4 Objectives . . . 5

1.4.1 Requirements . . . 5

1.4.2 Desired properties . . . 6

2 Preliminaries **9**

2.1 Quaternions and SO(3) . . . 9

2.2 Kalman Filters . . . 10

2.2.1 A two-phase process . . . 10

2.2.2 Error-State Kalman Filters . . . 11

2.2.3 Iterated Kalman Filters . . . 12

2.3 KD-Tree . . . 12

2.4 Sensors' characteristics . . . 12

2.4.1 LiDAR . . . 12

2.4.2 Cameras . . . 13

2.4.3 Inertial Measurement Units (IMUs) . . . 13

2.4.4 Global Navigation Satellite System (GNSS) . . . 14

2.4.5 Ground Speed Sensor (GSS) . . . 15

3 SLAM Implementations **17**
3.1 Feature SLAM 17
3.1.1 EKF SLAM 17
3.1.2 FastSLAM 19
3.1.3 GraphSLAM 20
3.2 LiDAR-specific SLAM methods 21
3.2.1 LiDAR Odometry and Mapping (LOAM) 21
3.2.2 LiDAR-Inertial Odometry (LIO) 22
3.3 Solid-state LiDAR SLAM 24
3.3.1 LOAM-Livox 24

4 Fast-LIO I and II - The mark to beat **25**
4.1 Fast-LIO I: A new computationally efficient Kalman Filter formula 25
4.1.1 IKFoM: Iterated Kalman Filter on Manifolds 26
4.2 Fast-LIO II: Registering raw points to an incremental KD-Tree 26
4.2.1 Direct approach 27
4.2.2 ikd-Tree: Incremental KD-Tree 27

5 LIMO-Velo **29**
5.1 Core idea 29
5.1.1 Localize Intensively 29
5.1.2 Map Offline 30
5.2 Pipeline structure 31
5.2.1 Accumulator: Receiving streams of data 32
5.2.2 Compensator: Adjusting for motion 32
5.2.3 Localizator: Estimating the state 33
5.2.4 Mapper: Building the map 33
5.3 Implementation 33
5.3.1 Modular programming 33
5.3.2 Functional programming 34
5.3.3 Object-oriented programming 34
5.4 Killer app 35
5.4.1 Racing: fast and aggressive motion 35

6 Results **37**
6.1 Robustness 37
6.1.1 Data loss/downsampling 37
6.1.2 Size of partitions (field of view) 38
6.1.3 IMU parameters 38
6.2 Computation performance 39

6.2.1 Speed of computation 39
6.3 Performance 39
6.3.1 KITTI dataset 39
6.3.2 Xaloc's map comparison 45
6.3.3 Xaloc's odometry comparison 45

7 Conclusions 51
7.1 Main key contributions 51
7.1.1 Localize Intensively 51
7.1.2 Map Offline 51
7.2 Derived improvements 51
7.2.1 Improvement over the SOTA 51
7.2.2 Improvement in infrastructure 52
7.3 Achieved objectives 52

8 Future work 55
8.1 Natural step: Dynamic objects' removal 55
8.2 Other possible lines of work 56
8.2.1 Estimation of future actions of moving objects 56
8.2.2 Loop closure detection 56
8.2.3 Multiple LiDARs and IMUs 57
8.2.4 Removing long term drift with GNSS 57
8.3 The next step: LiDAR-Visual-Inertial Odometry and Mapping 58
8.3.1 Adding cameras without correcting pose 58
8.3.2 Adding cameras for correcting pose: R^2LIVE and R^3LIVE 58
8.4 The goal: Visual-Inertial SLAM 59
8.4.1 LiDAR liabilities 59
8.4.2 Difficulties and requirements 59

Chapter 1

Introduction

1.1 Motivation

1.1.1 Barcelona’s first autonomous car

The motivation for this project comes from my time at BCN eMotorsport, Barcelona’s project for designing, building and coding an autonomous real-size single-seater electric car at Universitat Politècnica de Catalunya.

Figure 1.1: Barcelona’s first ever autonomous car, ”Xaloc” (2018-2021).

I was part of Xaloc’s Perception team and later in the season I became the main lead of its ”Localization and Mapping”. ”Localization and Mapping” is a concept used to describe the algorithms that make sure the car is aware of its environment, avoids collisions and detects the track it can take.

The team was lagging behind in that sense (as most Formula Student teams do) since it: depended on a strong GPS signal, had stability issues and didn’t filter appropriately noisy observations which complicated identifying the track.

1.1.2 Formula Student

Formula Student is a international student's engineering competition that aims to promote excellence in engineering via organizing events around the globe where university students design and construct a single-seater electric or combustion car and compete with it in dynamic tests. Points are also awarded for its technical design and economic cost. [5]

Formula Student Driverless

Since 2017, many Formula Student competitions such as F.S. Germany, F.S. Spain, F.S. East (East Europe) or F.S. Czech have added a new variant a part from combustion cars and electric cars: autonomous cars. Teams are expected to design, build and code single-seater cars that can be both driven by a human pilot and driven autonomously.

This new layer of complexity has attracted few teams to join but UPC Barcelona decided to take on the challenge on 2018, making it the first and only Spanish team to compete with the likes of Zurich, Karlsruhe, Ausburg, Roma or Trondheim until this date. In 2022, a new regulation has been passed that electric or combustion teams are expected to go driverless if they want the full score and that's flooded the competition with newcomers. A new and exciting chapter in the Formula Student has begun and it is only the beginning.

Control and Perception

Driverless teams usually split the coding in two: Control and Perception.

- **Perception** is focused on detecting the cones that determine the track and remember their location for the next laps.
- **Control** is focused on deciding which actions does the car have to take to drive around the track minimizing lap time without hitting any cone.

1.1.3 An unresolved problem

Finally, the last motivation I had to take on this thesis was the fact that knowing your surroundings on a racing environment did not have any public implementation that was good enough for us. State-of-the-art algorithms all had different setbacks (stability, accuracy, computation cost) that created a gap which to improve on.

The "Localization and Mapping" field is on a tremendous pace of improvement these last few years with multiple implementations that use different

sensors and algorithms and it is getting better and better to the point where it not only can be used on university laboratories, but also can be used on open areas, city streets, roads and yes, even racing circuits.

1.2 General problem statement

1.2.1 Understanding our environment

Autonomous vehicles require to know their surroundings to move avoiding collisions. This is a simple idea can be extended to: creating a three-dimensional map, stitching these local surroundings through time to the global frame, and at the same time estimating where the car is located inside this global map at every time.

Maps bring us better understanding of our environment, what can we expect to happen (such as children appearing behind parked cars or car doors opening) or what other agents (such as vehicles, people or animals) are able to do.

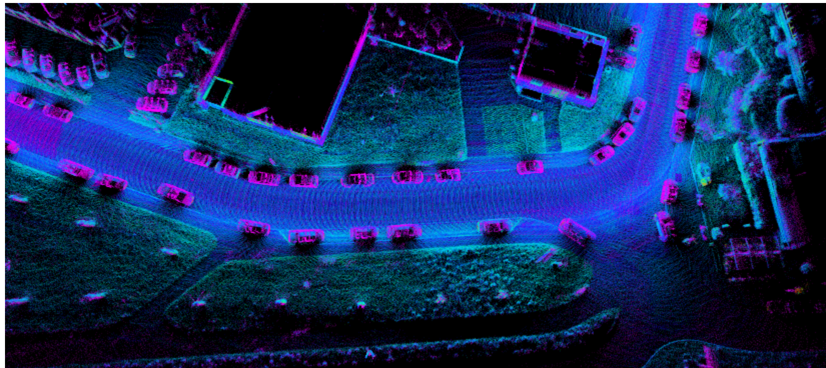


Figure 1.2: Our solution's three-dimensional map of a street

1.2.2 Localizating ourselves inside the map

Sensors such as cameras, LiDARs or radars give us data about our local surroundings: distance from objects to the sensor, shapes, colors... However, if we are to stitch local surroundings together to create a global map, we need to know the locations of where we took each measurement.

A new problem arises: where did we take each measurement? A solution is found when we assume that probably (assuming high frequency of the sensors and smooth movement) a big majority of the surroundings (walls,

objects...) we see in the new measurements is things we already measured at a previous location. Therefore, we can associate the new measurements with old measurements.

Then a recursive strategy is laid out: if we know the location from where we took the old measurements, we can find them in the map, associate them with the new ones and triangulate our new location.

1.2.3 Simultaneous Localization and Mapping

Knowing how to localize ourselves given a map and knowing how to expand the map once we know our location. This idea is what gave birth to the Simultaneous Localization and Mapping (SLAM) field: localizing itself with known measurements, mapping the new ones and back to localizing again.

The term was coined back in the 1990s by H. Durrant-Whyte's research group [4] but it gained international attention after S. Thurn's Standford team crossed a desert without a driver [25], winning the first Autonomous Driving DARPA challenge a decade later in 2005. After that, Thurn created the Google's self-driving project in 2009 which is now Waymo; one of the major players in the self-driving scene.

1.2.4 Implementations of the SLAM problem

Lots of implementations of the SLAM problem have been proposed. Many of them try to find effective map representations and mapping techniques taking advantage of the usual errors each sensor has. Others, try to fuse different sensors to make up for the weaknesses the other sensor has.

This thesis is going to focus first on surveying different implementations of the SLAM problem. In particular, solutions using LiDARs sensors. LiDARs offer accurate distances and dense definitions of close objects so its easier to triangulate our location and stitch local surroundings than other sensors like cameras.

1.3 Problem statement - Formula Student

The Formula Student competition has specific rules and requirements that make the SLAM problem unresolved for this environment. We will now dive into which are these requirements that make current best performers, unusable.

No margin for error

- Rule D6.1.1 [7] determines the minimum track width to be 3 meters. Usual formula cars are about 1.5m wide, leaving a **error margin of 75cm** to not collision with each side of the track.
- Competitions' circuits (usually the official F1 circuits) may have **weak GPS signals**.
- To be competitive, a car has to reach speeds of **20m/s** and turn speeds of **500deg/s**.
- **Real-time** decision making is necessary.
- Reaction times are of less than **50ms**.
- The track layout is **unknown**, there are multiple competitions around the globe and all circuits have different surroundings.
- The team **budget** is limited, we have to minimize sensor spending.
- The **CPU usage** is also limited since we are running all the autonomous pipeline.

1.4 Objectives

1.4.1 Requirements

When designing an algorithm, we must ask ourselves what key requirements should this algorithm have. Considering the problem as set in the Formula Student Driverless competition, we will separate the requirements into three groups.

A - Hard requirements

1. Drive close to the track limits without crossing them.
2. Output information more frequently than the time of reaction of 50ms.
3. Handle fast and aggressive motion.

B - Soft requirements

1. Ability to detect cones at 40 meters.
2. Real-time sensor calibration.
3. Stable to occasional sensor failure or degeneracy.

C - Design requirements

1. Works anywhere and anytime.
2. Easy and modular design to be easily maintained.
3. Seamless integration with the full pipeline.

1.4.2 Desired properties

We can ask which are good properties for a SLAM algorithm. Then, see which ones are the most critical and which ones are feasible to add.

Determining the state of a vehicle means estimating its position, orientation, velocities and calibration parameters at every point in time. The state then is a collection of vectors and matrices of real scalars. We will consider a well defined distance and norm $d(s_{t_k}, s_{t_l}) = \|s_{t_k} - s_{t_l}\|$ that give us the difference between two states. This will be helpful when talking about properties. Consider s_t the estimated state and g_t the ground truth state at time t.

We first will define a set of base properties to fulfill our requirements (A, B, C) to later derive which properties should our algorithm aim to have.

Desired SLAM properties

- **Stable**: position errors should be bounded. ($\exists \varepsilon\ \forall t, \|s_t - g_t\| < \varepsilon$) [A1, A3, C1]
- **Accurate**: the position error bound ε should be less than the minimum distance of collision with the environment. [A1]
- **Real-time**: for every time t we should have calculated the solution s_t before the new data for estimating s_{t+1} is available. [A2]
- **Fine**: the time between the estimated solution s_t and s_{t+1} should be less than the time of reaction of our vehicle. [A2]

- **Dense**: objects in the map should be clearly differentiated. Formally, given two areas of the map A and B containing each an object and an ideal segmentation algorithm g that maps an area to an object identifier, $g(A) = g(B) \iff A$ and B contain the same object. [B1]

- **Light**: the CPU usage of the algorithm should be less than the one left while the other algorithms on the pipeline are running. [C3]

- **Minimal**: the sum of all sensors' price, space and dependency needed to make the algorithm work should be minimized. [B3]

- **Modular**: should be able to fuse different sensors and its respective calibration seamlessly. [B2, C2]

Derived SLAM properties

We can derive the following properties as a corollary.

- **Weather-resistant**: with a stable algorithm, weather cannot cause our algorithm to drift.

- **Works on open areas**: with a stable and accurate algorithm, featureless environments such as open areas cannot cause trouble.

- **Able to detect cones**: with a dense and modular algorithm, cone detection can be performed aided by camera data.

- **Can run for 10 laps**: with a stable and real-time algorithm, the car has to yield feasible solutions continuously for (more than) 10 laps.

Chapter 2

Preliminaries

2.1 Quaternions and SO(3)

Localizing our vehicle will require knowing two things: its position and orientation (also known as "attitude") on the track. The position of our vehicle will be represented by a $\mathbb{R}^3$ vector (x, y, z). Attitude, however, is a different story. First thing that comes to our mind are Euler Angles (yaw, pitch, roll), introduced by Euler in 1775. But Euler Angles are well known in the literature to suffer from singularities and not being uniquely defined. For our case, we will need to find an alternative representation without singularities and with nice composition properties to lead to smooth estimations.

Definition 2.1.1 (SO(3) - 3D rotation group)**.** We define SO(3) as the group of all rotations about the origin of the three-dimensional Euclidean space $\mathbb{R}^3$ under the operation of composition.

Definition 2.1.2 (Quaternion)**.** Quaternions, introduced by Hamilton in 1843, are an extension of complex numbers in the three-dimensional space. Apart from i, we have to consider additional axes j, k.

Proposition 2.1.1. Unit quaternions form a group structure isomorphic to SO(3):

$$q = a + bi + cj + dk \in \mathrm{SO}(3), \quad \|q\|_2 = 1$$

With the following relationships:

$$i^2 = j^2 = k^2 = ijk = -1, \quad a, b, c, d \in \mathbb{R}$$

Proof. Refer to [9]. □

So, attitude can be represented as a unit quaternion and enjoy their nice composition properties. These will come handy for the next section: estimating the least-squares position/attitude given a stream of measurements.

2.2 Kalman Filters

We can consider that the readings from our sensors will be noisy. We model this noise as a normal distribution: with a mean vector and a covariance matrix. Given these noisy measurements from different sensors over time, we wish to estimate the least-square attitude and position. The Kalman filter, introduced by Kálmán around 1960 [10] to estimate the trajectory of NASA's Apollo rockets, give least-square estimates of unknown parameters under the Gaussian assumption. Kalman filters work in a two-phase process: we first predict what movement we did and then correct this movement to fit observations.

2.2.1 A two-phase process

Prediction phase

We consider a movement function f_k that outputs where our car will be in time step k given the state in x_{k-1} and the latest control measures u_k (such as accelerations or velocities). Then, we can predict the state at time k (including position, attitude, velocities...) x_k given the observations up until $k-1$, as $\hat{x}_{k|k-1}$:

$$\hat{x}_{k|k-1} = f_k(\hat{x}_{k-1|k-1}, u_k) \approx F_k \hat{x}_{k-1|k-1} + B_k u_k + w_k, \quad w_k \sim N(0, \sigma_k^2)$$

and its predicted covariance estimate as:

$$P_{k|k-1} = F_k P_{k|k-1} F_k^T + Q_k$$

The vanilla Kalman filter assumes the movement function f_k to be linear. However, in the general case, it's approximated by linear maps $F_k = \frac{\partial f_k}{\partial x}(\hat{x}_{k-1|k-1}, u_k)$ and $B_k = \frac{\partial f_k}{\partial u}(\hat{x}_{k-1|k-1}, u_k)$. This is called an "Extended Kalman Filter", since the movement function f_k is not assumed to be linear but approximated well enough by a first Taylor expansion in a tiny time span $\Delta t_k = t_k - t_{k-1}$. This "tiny time span" assumption will play a major role in the solution we develop in this thesis, as described in Chapter 5.

Update/Correction phase

Now that we have a rough estimation of where our car has moved to, we can correct it by associating our new measurements with known old ones. Consider h_k^j, a smooth function for each observation j in time k satisfying $0 =$

$h_k^j(x_k)$. We can approximate this function by evaluating it on a previously predicted state $\hat{x}_{k|k-1}$.

$$0 = h_k^j(x_k) \approx h_k^j(\hat{x}_{k|k-1}) + H_k^j \cdot (x_k - \hat{x}_{k|k-1}) + v_k, \quad v_k \sim N(0, \nu_k^2)$$

These functions h_k^j can also be approximated by a first-order Taylor approximation considering $H_k^j = \frac{\partial h_k^j}{\partial x}(\hat{x}_{k|k-1})$.

Considering the new predicted state $\hat{x}_{k|k-1}$ as a close solution to the real one x_k, we calculate close solutions for each h_k^j we call "residuals" $z_k^j := h_k^j(\hat{x}_{k|k-1})(\approx 0)$ and want to find the x_k that minimizes:

$$\min_{x_k} \left(\|x_k - \hat{x}_{k|k-1}\|^2_{P_k^{-1}} + \sum_{j=1}^{N} \left\| z_k^j + H_k^j \cdot (x_k - \hat{x}_{k|k-1}) \right\|^2_{R_j^{-1}} \right)$$

where $\|x\|_M^2 = x^T M x$. We can interpret this as finding the state x_k that gives the least-squared error considering the covariance of the previous state and the covariance of the observations. To find the least-squares x_k, we will consider the following equations

$$\tilde{\mathbf{y}}_k = \mathbf{z}_k - \mathbf{H}_k \, \hat{x}_{k|k-1} \qquad \text{(A priori "innovation")}$$
$$\mathbf{K}_k = \mathbf{P}_{k|k-1} \mathbf{H}_k^T \left(\mathbf{H}_k \mathbf{P}_{k|k-1} \mathbf{H}_k^T + \mathbf{R}_k \right)^{-1} \qquad \text{(Optimal Kalman gain)}$$

$$\hat{\mathbf{x}}_{k|k} = \mathbf{x}_{k|\hat{k}-1} + \mathbf{K}_k \tilde{\mathbf{y}}_k \qquad \text{(A posteriori mean)}$$
$$\mathbf{P}_{k|k} = \left(\mathbf{I} - \mathbf{K}_k \mathbf{H}_k \right) \mathbf{P}_{k|k-1} \qquad \text{(A posteriori covariance)}$$

and repeat them for M iterations updating the residuals $\mathbf{z}_k = \mathbf{h}_k(\hat{\mathbf{x}}_{k|k})$ using the new-found corrected state and using $\hat{\mathbf{x}}_{k|k}$, $\mathbf{P}_{k|k}$ as the new $\hat{\mathbf{x}}_{k|k-1}$, $\mathbf{P}_{k|k-1}$.

2.2.2 Error-State Kalman Filters

In the 90s, three decades later after the Kalman Filter was introduced researchers began noticing that the errors $\delta_k = x_k - \hat{x}_{k|k}$ behaved in a less complex manner than the states. While the states can have highly non-linear trajectories, the state's errors with the filter estimates were seen to behave almost linearly.

That gave birth to the error-state formulation of the Kalman Filter [22]. Instead of calculating the *a posteriori* mean directly, we update the error estimate $e_k = \hat{x}_{k|k} - \hat{x}_{k-1|k-1}$.

$$e_k = \mathbf{K}_k \tilde{\mathbf{y}}_k$$
$$\hat{\mathbf{x}}_{k|k} = \hat{\mathbf{x}}_{k-1|k-1} + e_k$$

2.2.3 Iterated Kalman Filters

Iterated Kalman Filters are yet another solution for improving the linearization of the EKF. The idea is to perform the update phase for M iterations in the hopes of improving the $h_k^j(x_k) \approx h_k^j(\hat{x}_k) + H_k^j \cdot (x_k - \hat{x}_k)$ Taylor estimation by applying it to the m-iterated $\hat{x}_{k|k-1}^m$ closer to the ground truth x_k.

Accidentally (or not), this formulation makes a lot of sense for SLAM because not only can we improve the linearization of each h_k^j, we can also rematch the features using the new updated state $\hat{x}_{k|k-1}^m$, changing the observation functions h_k^j and getting rid of erroneous data associations.

This formulation does not interfere with the error-state formulation, and we can mix them both in what is called an Error-State Iterated (Extended) Kalman Filter (ESIKF). This is relevant since we are going to be using this formulation for LIMO-Velo [8].

2.3 KD-Tree

Now that we now what we are going to use to estimate our state, we need to talk about how we are going to represent our map. Usual SLAM algorithms work by associating new measurements with known ones in the map that are close to the ones projected with predicted state. Therefore, we need a data structure optimized for searching in a queried position's neighborhood.

K-dimensional trees (or KD-Trees), introduced by Bentley in 1975 [1], are exactly this type of structure. In these, searching the closest neighbour of a three-dimensional point p has a best-case scenario time complexity of $O(\log N)$; being N the number of points in the tree.

2.4 Sensors' characteristics

As a preliminary, we also have to understand what different characteristics each sensor brings; what are their weaknesses and how can they be combined with one another.

2.4.1 LiDAR

What LiDAR offers us are highly accurate distances in a semi-dense format. That is, the point cloud's density decreases quadratically with distance. These accurate distances give us clear picture of our environment: walls, other cars, the ground, traffic signals and even can detect close pedestrians.

However, if we want to see finer details, like: recognizing objects, recognizing agents at large distances or high-frequency features; this semi-density is not dense and feature-rich enough.

Pros

+ Accurate distances

+ High range

Cons

- Expensive

- Not good for details

2.4.2 Cameras

Vision gives all the details and information we humans need to drive for ourselves. We can put many cameras in our vehicle since they are cheap and overlapping their fields of view can give us stereo vision to better estimate close objects (just as humans have two eyes).

The human brain, however, still works in ways much more advanced than any machine can and emulating it is the hard part when using cameras. Also, vision is very resource-heavy and requires a lot processing power.

Pros

+ Accurate and abundant details

+ Cheap

Cons

- Require complex algorithms and resources to treat their data

- Easily blocked by weather conditions or direct sunlight

2.4.3 Inertial Measurement Units (IMUs)

IMUs are electronic devices that combine accelerometers, gyroscopes and magnetometers to measure forces, angular rates and orientation of the car's

body. These measurements give us a reading that is independent of the environment we are in (feature-independent).

Integrating gyroscope velocities gives us an accurate prediction of our orientation (error of ± 5 deg/h) and integrating accelerometer accelerations with the car current position and velocity can bring rough estimates for the new position and velocity.

IMUs are particularly interesting for the prediction phase of Kalman Filters since they act as the control input u_k that we later correct with feature-dependent sensors.

Pros

+ Independent of its environment

+ Wide price range and accuracies

Cons

- Only for prediction purposes

- Integrated error increases quickly in the long term

2.4.4 Global Navigation Satellite System (GNSS)

GNSS (known for it's North American system, GPS) is a sensor thought for correcting long-term drift since it's also independent of its environment but has too high of an error to be used in the short-term.

Figure 2.1: A naïve team's out-of-track autonomous car thinking GNSS as an initial solution would "kind of work", Formula Student Germany 2021.

Pros

+ Independent of its environment

+ Wide price range and accuracies

Cons

- Only for correction purposes

- Error too high for the short-term

2.4.5 Ground Speed Sensor (GSS)

GSS is a relatively unknown sensor that only one company has its patent (and subsequent monopoly) [11]. It offers near-perfect velocity estimates in dry ground. Its prices are in the thousands of euros and are affordable for only select teams/organizations in limited vehicles.

Pros

+ High accuracy

+ Independent of environment features

Cons

- Has to be placed just above the ground at all times

- Only for prediction purposes

- Problems on wet ground

- Prohibitively expensive

Chapter 3

SLAM Implementations

The Simultaneous Localization and Mapping problem is an abstract problem that can come to live in so many ways. However, most SLAM implementations share a common structural order [23], inspired in the two-phase process of the Kalman Filter:

Step 1 Predict the robot's motion.

Step 2 Associate new data with known data.

Step 3 Correct the motion given the associations.

Step 4 Update the data in the map.

Step 5 Repeat.

3.1 Feature SLAM

Feature SLAM implementations start with the premise that the full data given by the cameras or LiDAR is too abundant and it needs to be processed (and downsized) in favor of extracting the relevant parts of it: the features. These were the first working implementations in the literature because of their lightweight computational load given the hardware requirements 20 years ago. Features are usually known objects or object properties that can be easily segmented from the environment.

3.1.1 EKF SLAM

The Extended Kalman Filter SLAM implementation was the natural way of thinking of the prediction/correction duality, and it was the de facto method for SLAM until the introduction of FastSLAM in 2002 [16]. Its structure is:

Step 1 Predict the robot's next state x and next covariance matrix P with a motion function f and its Jacobian matrices with respect to the control input (u) and with respect to the noise (v): F_u, F_v.

Step 2 Project the extracted local features at time k to the global frame and associate them with the old global features (also known as "landmarks") using a maximum likelihood approach with a threshold. If the threshold is not surpassed, we consider the observation as new.

Step 3 Given the associations, we first correct the motion using the Kalman Filter's observation motion h update method minimizing the Mahalanobis distance between the projected feature and the landmark.

Step 4 With the new updated state, we correct every landmark's position mean and covariance to fit the new updated state.

Step 5 Additionally, we can delete corrupted features in the map if the Mahalanobis distance between the new feature and known feature surpasses a threshold.

The map is a large state vector $x = (\mathcal{R}\ \mathcal{M})^T$ being $\mathcal{R}$ the robot state and $\mathcal{M}$ a collection of landmark states $(\mathcal{L}_1, \ldots, \mathcal{L}_N)$. EKF SLAM assumes the landmarks' states depend on the robot state and depend on the states of the other landmarks. Therefore, making every covariance update on a landmark, a covariance update on all the other landmarks.

Pros and cons

+ Proven convergence if the associations are all correct.

+ Smooth correction process, the state will not jump around across time.

- The EKF covariance update complexity for k landmarks is of $O(kN^2)$, being N the total number of landmarks.

- Data association failures can cause the algorithm to fail.

- The "Gaussian error" assumption requires the observation model h to be well approximated by its linearization.

3.1.2 FastSLAM

FastSLAM (Montemerlo et. al, 2002) [16] was the first implementation of a particle filter in SLAM, dropping the "Gaussian error" assumption in favor of a Rao-Blackwellized non-linear "particle" filter. Particle filters consider the robot's current state as the maximum likelihood state x in a collection of probable states $\mathcal{X}$ (called "particles"). The distribution of these particles for the next iteration is updated on a process called "resampling", meaning we keep and reproduce variations of the most likely states.

FastSLAM also has another assumption that makes it way faster than EKF, it assumes every landmark is independent of the others. Therefore, each landmark is updated with its own independent Kalman Filter with $O(1)$ update time. With a tree-structure, the computational complexity is $O(M \log K)$ being M the number of particles and K the number of landmarks. Hence, the name "FastSLAM". Its structure is:

Step 1 For each particle m, sample the robot's next state from the probabilistic motion model's distribution: $s_k^{[m]} \sim p(s_k|u_k, s_{k-1}^{[m]})$.

Step 2 Project the extracted local features at time k to the global frame and associate them with landmarks using either a maximum likelihood approach (like EKF). So, each particle has its own (possibly different) associations.

Step 3 Update the state's EKF given the associations with the landmarks.

Step 4 Update the landmarks' EKFs given the updated state.

Step 5 Additionally, delete corrupted landmarks if needed.

Step 6 Re-sample the particles, weighting each particle in favor of maximizing the likelihood of each observation being assigned the landmark it has been assigned to after the updates.

The main difference from the EKF SLAM implementation is that now each particle acts as its own EKF SLAM, we can evaluate each particle's performance and finally propagate the best performers (based on the residuals of their data associations). This allows us to follow multiple hypothesis throughout the process. Not only are we propagating the better looking hypothesis but we are also propagating less-good looking hypotheses that may turn out to be correct and replace the best-looking ones of a previous iteration.

Pros and cons

+ Can follow multiple hypothesis so it's resistant to bad data associations.

+ Scales logarithmically in number of landmarks.

+ We can trade-off performance for accuracy (nº of hypotheses) by changing the number of particles.

- The state throughout time is not smooth. It can jump around through hypotheses.

- Different hypotheses can have big differences; risking big sudden errors with respect to the ground-truth.

3.1.3 GraphSLAM

For the final feature SLAM implementation, we have GraphSLAM (Thurn et. al, 2006) [24]. GraphSLAM was the first widely used implementation of the idea of a factor-graph in the SLAM field. The idea behind GraphSLAM is to use standard least-square optimization techniques used in other more mature areas on the SLAM problem. In GraphSLAM, the map is a graph and each state (node) is connected with its previous and next state via edges they call "motion arcs". Similarly, each state is connected with its associated landmarks (nodes) used to update the state via edges they call "measurement arcs". Every edge represents a non-linear constraint associated with the likelihood of the measurement and motion models, and the sum of all constraints results in a non-linear least squares problem that aims to maximize the state's and landmark's position likelihood.

The idea of a factor-graph became really relevant with the introduction of "loop closures", with we can also associate our current state with an older state if the two measurements are similar enough and create a "loop". With a factor-graph it is really natural to then re-optimize the full graph using this new constraint, leading to a drift (propagated error) reducing technique.

GraphSLAM's structure is the following:

Step 1 Add a new state x_{k+1} node depending on the previous states $x_{1:k}$ and the control input u_k. Then, create a "motion arc" with the previous state.

Step 2 For every feature, create a new landmark node and a "measurement arc" connecting it on the state node x_{k+1} just created.

Step 3 Optimize the new state and landmarks (mean and covariance) with all this new information.

Step 4 Perform a correspondence test to match the new landmarks with old landmarks, merge them if passed and optimize the state and landmarks positions again.

Step 5 Repeat the test until no matches are found.

Pros and cons

+ Novel architecture that allows for optimization of older states and landmarks as well as current ones.

+ The computations can be handled by specialized least-squares optimizers, one of the most common optimization problem there is.

+ More stable than EKF SLAMs since it considers past states and more consistent than Particle Filter SLAMs since it does not jump through different hypotheses.

- Higher memory and computational load.

- More complex code and need for third-party solvers.

3.2 LiDAR-specific SLAM methods

Up until now, we have discussed feature-based SLAM methods that can work either with LiDAR, cameras or any other type of sensor. Now, we will dive in the most widely used methods for LiDAR-specific solutions.

3.2.1 LiDAR Odometry and Mapping (LOAM)

Zhang et. al (2014) [29] proposed LiDAR Odometry and Mapping (LOAM), a SLAM method that extracted an abundant number of features in any unstructured environment using LiDAR. This lead to the explosion of so-called LOAM-based solutions, solutions that built on this idea of having lots of features available to match.

The feature extraction is fairly simple, extract two types of features: planes and edges.

- Planes are calculated from a collection of close points in the current scan that form a plane.

- Edges are calculated from a collection of close points in the current scan that conform a line.

The method tries to minimize the total point-to-plane or point-to-line distance between a mapped plane or edge and our matched point in the current scan. Since we assume our environment is locally smooth, there will be an abundant number of planes and edges in every LiDAR scan.

LOAM's structure is the following:

Step 1 Detect points that belong to planes and edges in the current scan. Save them separately in two vectors: one for plane points and one for edge points.

Step 2 Match current scan planes/edges with known planes/edges in the map, calculate their distance with respect to the projected point (using the latest state) and add a new constraint to optimize.

Step 3 Optimize the least-square problem minimizing the distances.

Step 4 Repeat steps 2-3 with the new updated state until convergence.

Step 5 Map the projected points as plane or edge points with the latest state.

Pros and cons

+ High accuracy caused by an abundant number of constraints to minimize, one for each extracted feature.

+ Independence of control input.

- There is no prediction step, the predicted state is estimated to be the previous one. Fails under fast and aggressive movement.

- High memory and computational load.

3.2.2 LiDAR-Inertial Odometry (LIO)

Zhang et. al already mention in the LOAM (2014) paper that an IMU can be used to have a better prediction for the next state. Shan et. al bring this idea to another level of robustness and accuracy in LeGO-LOAM (2018) [20] and more importantly in LIO-SAM (2020) [21] which they provide an open-source factor-graph based method that can do long drives without drift, handles relatively fast rotations and can perform loop closures leading to great mapping accuracy.

Tightly-coupled vs. loosely-coupled methods

The main difference between the two approaches is that LIO-SAM (2020) is a tightly-coupled vs. LeGO-LOAM a loosely-coupled (2018) method. The difference between these two concepts is that loosed-coupled methods use the IMU (or other sensors) as a first solution to then correct with the LiDAR (or other sensors) observations. Tightly-coupled methods view the different sensors as capable of delivering their own odometry solution (with different levels of accuracy) and corrections are made every X frames (known as keyframes). This can make up for the slow computation of the LiDAR correction and can output odometry at IMU rate (usually 200Hz or more).

LIO-SAM (2020) marks a step forward towards centimeter-level accuracy but its computational load still makes it fail under fast and aggressive motion.

Motion compensation

LeGO-LOAM and LIOSAM not only use the IMU as a prediction method, they also use it to remove the distortion caused by the robot's motion on the LiDAR observations. When our robot is moving, the pointcloud result of accumulating a full LiDAR rotation has points extracted from a time span (t_1, t_2). That means, points extracted at t_1 and t_2 will have been extracted from different states x_{t_1} and x_{t_2}. This means that the pointcloud will be skewed by the motion of going from x_{t_1} to x_{t_2}, and it would be wrong to project the (t_1, t_2) point cloud with just one state.

To fix this distortion, we can use the IMU readings (usually at a $\times 20$ more frequency than the LiDAR point clouds) to estimate this motion and undistort the points in (t_1, t_2) to a common time frame. Without loss of generality, we choose this common time frame to be t_2 since we are trying to estimate the latest state possible. Being $\{\hat{x}_t\}_{t=t_1,\ldots,t_2}$ the estimated states between (t_1, t_2) by the IMU, we can transport local points measured in time t to the local frame at t_2 via the following formula:

$$p_{t_2}^L = \hat{X}_{t_2}^{-1} \cdot \hat{X}_t \cdot p_t^L, \quad p \in \mathbb{R}^3, X \in \text{Aff}(3)$$

The points $p_{t_2}^L$ are now ready to be brought to the global frame by left multiplying $\hat{X}_{t_2}$ to them. Note that these are the points used to find the corrected x_{t_2}. Therefore, the difference x_{t_2} and $\hat{x}_{t_2}$ has to be really low if we don't want non-accurate results. There are two ways to reduce this difference: having a better $\hat{x}_{t_2}$ estimation (better IMU quality/processing) or shortening the difference between t_1 and t_2, which we will discuss in Chapter 5.

3.3 Solid-state LiDAR SLAM

Solid-state LiDARs are a low-cost alternative to traditional spinning LiDARs. Instead of having multiple channels as spinning LiDARs have, solid-state LiDARs have a single moving laser head. They have various difficulties that make SLAM harder:

1. Smaller FoV: less field of view will lead to fewer features, making it prone to degeneracy and moving objects.
2. Irregular scanning patterns: irregular non-rotating patterns are harder to extract features from.
3. Non-repetitive scanning: to maximize coverage even with the LiDAR static, the scanning process tries to not pass to points it already has via non-repetitive patterns.
4. Motion blur: the irregular and non-repetitive scanning allows for close points in the same scan have a big time difference, creating a blur under fast and aggressive motion.

3.3.1 LOAM-Livox

Parallelly to LeGO-LOAM (2018) and LIO-SAM (2020), Lin et. al developed loam_livox (2019) [13] that set the foundations of Fast-LIO's (2020) and Fast-LIO2's (2021) successes. Their main contributions are:

1. The idea of extracting planes and edges directly from the map (given the difficulty of extracting features from the current scan) started to take place.
2. Pointing out that an improvement of the mapping routine's computational cost was needed and feasible.
3. Pointing out that not all sections of the scan are equally informative, a routine for downsampling irrelevant parts was needed.

Chapter 4

Fast-LIO I and II - The mark to beat

The Fast-LIO's papers set a new era of possibilities via extremely improved computation costs (x10 improvement with respect to other LOAM-based methods) and a type-of-LiDAR-agnostic solution that works independently of the pattern's regularity or the different sizes of the field of view.

4.1 Fast-LIO I: A new computationally efficient Kalman Filter formula

Xu et. al present Fast-LIO (2020) [26], a method inspired in loam_livox (2019) but instead of an optimization problem like other LOAM-based solutions, it is formulated as an ESIKF SLAM problem. With a twist: a new Kalman Filter formulation is proposed that can calculate the Kalman Gain quadratically on the dimensionality of the state vector ($O(1)$) instead of quadratically on the number of landmarks $O(N^2)$. This allows the usage of Extended Kalman Filters (EKFs) with an abundant number of features, making it the fastest centimeter-level method yet.

However, the Fast-LIO's filter formulation is far more complex than older EKF SLAM solutions that usually only estimated the robot's 2D pose and orientation: (x, y, θ). In this case, the state x accounts for the robot's 3D pose and orientation, its velocity, the gravity vector and calibration parameters for the IMU.

$$x \coloneqq [\ {}^{G}R_I^T \quad {}^{G}p_I^T \quad {}^{G}v_I^T \quad b_\omega^T \quad b_\omega^T \quad {}^{G}g^T]^T$$

This is by no means a trivial state to estimate. The orientation and gravity vector belong to $SO(3)$ and $\mathbb{S}^2$, respectively. A Kalman Filter formulation

had to be defined in this two spaces, and that's what they did in the paper "Kalman Filters on Differentiable Manifolds" (2021) by He et. al [8].

Pros and cons

+ $\times 2$ faster than other LOAM-based solutions.

+ Smooth, iterated and robust estimation of the full robot's state (pose, orientation, velocity, calibration parameters and gravity direction).

+ Handles fast and aggressive movements given a small (but low-latency) field of view.

- Mapping time can still be improved.

- The number of features extracted from such a small FoV can be low and lead to degeneracy.

4.1.1 IKFoM: Iterated Kalman Filter on Manifolds

In this work, He et. al (2021) [8] propose a canonical representation of robot systems and develop a symbolic error-state iterated Kalman filter (ESIKF) for it. The spaces considered are: $\mathbb{R}^n$, $SO(3)$ and $\mathbb{S}^2$. They additionally develop a C++ toolkit used and tested in Fast-LIO (2020) [26] for quick deployment of generic robotic solutions using Kalman Filters.

4.2 Fast-LIO II: Registering raw points to an incremental KD-Tree

Xu et. al are back presenting Fast-LIO2 (2021) [27] with a bold and exciting claim: a 10-fold decrease in computational time with respect to other state-of-the-art algorithms and the highest-seen accuracy under fast and aggressive movements.

The accuracy increase, as they say in the paper, is related to being able to utilize more points in the odometry as they have more mapping efficiency and have removed the feature extraction module in favor of what they call a "direct approach".

+ Faster than any other approach. Generally $\times 10$ faster than other LOAM-based approaches.

+ More accurate than any other approach under fast and aggressive movement.

+ Generalizable to any LiDAR pattern.

- Struggles under big fields of view.

- Lacks a degeneracy module.

4.2.1 Direct approach

Inspired by visual SLAM's direct approaches (Matthies '88, Hanna '91, Comport '06, Newcombe '11, Engel '13...) that register the full image in the map and minimize the photometric error, Fast-LIO2 (2021) matches its current scan points to local planes it finds in the map; and minimizes the point-to-plane distance.

That is,

1. A point in the current scan is projected to the map frame using the estimated state.

2. We get the m closest points with respect to our projected point in the map. If the closest points form a plane, then we match our projected point with such plane.

3. The point-plane pairs then are sent as observations to optimize in an error-state iterated Kalman filter.

This allows us to not care about extracting features and directly registering raw points to the map. That means a sparse input is also welcome and no matter the type of LiDAR pattern we are using, this will also work.

4.2.2 ikd-Tree: Incremental KD-Tree

To have a successful direct approach, skipping scans to map is not an option, since we want the map to be as dense as possible. That means the mapping routine's computational time has to be improved by a significant amount. This is where ikd-Tree comes in.

LOAM-based algorithms use a KD-tree [1] to represent the map for its fast logarithmic lookup time. However, KD-trees are to be built once and are not meant to be modified; since that causes them to worsen its look-up time until ending up with linear look-up time. The Fast-LIO authors also publish "ikd-Tree: An Incremental K-D Tree for Robotic Applications" (2021) by Cai

et. al [2], a data structure that offers two orders of magnitude improvement with respect to the vanilla KD-tree.

An incremental solution: re-balancing

The ikd-Tree can add/remove points incrementally to the tree via performing the add/remove operation and then re-balancing the tree. It also actively monitors a balance criterion and dynamically re-builds it partially when needed to.

Time complexity

- Point-wise operations: an incremental operation with a point to a N-sized ikd-Tree costs $O(\log N)$.
- Box-wise operations: the insertion of M points in a N-sized ikd-Tree costs $O(M \log N)$.
- Re-building: rebuilding K points of an ikd-Tree with two threads costs $O(K)$ and $O(K \log K)$ with a single thread.
- Nearest search: the average cost for searching the nearest neighbour of a point is $O(\log N)$.

Space complexity

Each node on the ikd-Tree records point information, tree size, invalid point number and point distribution of the tree. Even though the space complexity is $O(N)$, it is a few times larger than a static KD-tree [1].

Chapter 5

LIMO-Velo

Knowing the perks and flaws of the other state-of-the-art solutions. Now we are searching a solution for fast and aggressive motion with spinning LiDARs.

5.1 Core idea

5.1.1 Localize Intensively

Smaller steps work better

The only method able to handle fast and aggressive motion to date was Fast-LIO2 (2021). The reason: its low latency. With smaller steps (Δt), state estimates are closer together, providing two key benefits:

1. Closer steps means linearization will approximate better: $\Delta t \to 0 \implies O(\Delta t^2) \to 0$ in the Taylor expansion of the motion model f and more importantly in the Taylor expansion of the functions $h_k^j, \forall j$. Being closer to the ground truth state will inevitably lead to better matchings, abstractly speaking, defining better the functions h_k^j.

2. The predicted state $\hat{x}_k$ will have less integrated error ("dead-reckoning") caused by having to integrate less noisy IMU measurements.

So the first key concept of LIMO-Velo is "Localizing Intensively" (L.I.). To obtain better results under fast and aggressive movements, we have to be able to get smaller - but more frequent - fields of view. The way we do this is by treating the received point clouds as a stream of timestamped points and updating our state every Δt with the points in $(t - \Delta t, t]$.

Dealing with degeneracy

Smaller fields of view are prone to degenerate scenarios, we have to be aware of that and implement a module that detects and fixes the degraded DOFs. Zhang et. al (2016) [28] proposes a method that given a function to minimize f and its Jacobian $J := \partial f/\partial x$, identifies the eigenvectors associated to the smallest eigenvalues of the $J^T J$ matrix as the axis of degeneracy of the solution x^*. In our case, our Kalman Filter aims to minimize the distance between the projected points and the matched planes, giving us as Jacobian, the observation Jacobian $H := \partial h/\partial x$. Following Zhang et. al (2016)'s method, we identify the axis of degeneracy in our updated solution (x_u) with a threshold and fill those m directions with the integrated IMU predicted estimate (x_p):

$$\begin{aligned} V_p &= [v_1, \ldots, v_m, 0, \ldots, 0]^T \\ V_u &= [0, \ldots, 0, v_{m+1}, \ldots, v_n]^T \\ x^* &= V_f^{-1} V_p x_p + V_f^{-1} V_u x_u \end{aligned}$$

Ensuring true real-time performance

A third and last requirement for ensuring the success of localizing intensively is that we need to ensure real-time performance. That means we will have to update our state with the latest $(t_k - \Delta t, t_k]$ points, losing the points in $(t_{k-1}, t_k - \Delta t)$ in the case our pipeline runs longer than Δt, if it runs shorter we will always have the estimated state at $< \Delta t \sim 10^{-2}$ seconds of real-time. Having true real-time performance is essential for split-second decisions.

5.1.2 Map Offline

A lossless decrease of frequency

Localizing intensively can cause degeneracy cases and also can be affected by moving objects (caused by corrupted point-plane matches). We have seen how to fix the first issue and we explain the solution for the second issue for Chapter 8.

On the other hand, there's a problem with mapping intensively - specially in new unexplored areas where there's little map information. Mapping intensively can cause the algorithm to hold onto the only points it sees and try to match points to planes from the same scan. In the general case, in a rotating LiDAR, points from the same scan do not intersect and therefore should not be matched together. This calls for a fix. LIMO-Velo's second key idea is "Mapping Offline (M.O.)".

The idea of mapping offline is to use the fact that points from the same rotating scan do not intersect. Therefore, mapping a point before its rotation ends is not only useless, but dangerous. Now, instead of mapping intensively we will choose to wait until the rotation ends to map. This will lead to a - lossless - decrease in mapping frequency.

Processing before mapping

Now, we know that we can wait to map the points we are accumulating without losing performance, as long as we map them when the rotation ends. That opens a gap that we can exploit by processing the points waiting to be added before we register them to the map.

In the case of a dynamic environment, we could identify moving objects, remove them from the pointcloud, map the pointcloud without the moving object and then add the moving object again as a new independent layer on the map (not to be used for localization). Chen et al. (2021) present LMNet (or LiDAR-MOS) [3], a CNN that works at 20Hz capable of segmenting moving objects given a ranged image of a 10Hz pointcloud. We could use this neural network on the points we are about to map and remove cars, pedestrians, bicycles...

Masking the localization

We can go deeper on the details of how to not let these corrupt parts of the map (such as moving objects) corrupt our localization: we can use a mask. We define a mask as a 3D grid with binary values. This grid's cells will have 0 as value if at least one corrupted point is in it or 1 if it doesn't contain corrupted points. Then, we can decide if we want to use or not a certain point for localization if it belongs or not on a corrupted cell.

Using a hashmap, the mask's spatial complexity is $O(N)$ (being N the number of points in a full rotation) if we assume that the points are distributed on the space and the cells are little enough to convey detail. Since it is a hashmap, it can run $O(1)$ in time. For racing, we will not use the mask so we will leave this idea for Future Work in Chapter 8.

5.2 Pipeline structure

In this section, we are going to look into the different parts of the pipeline, what they do and how they communicate with one another.

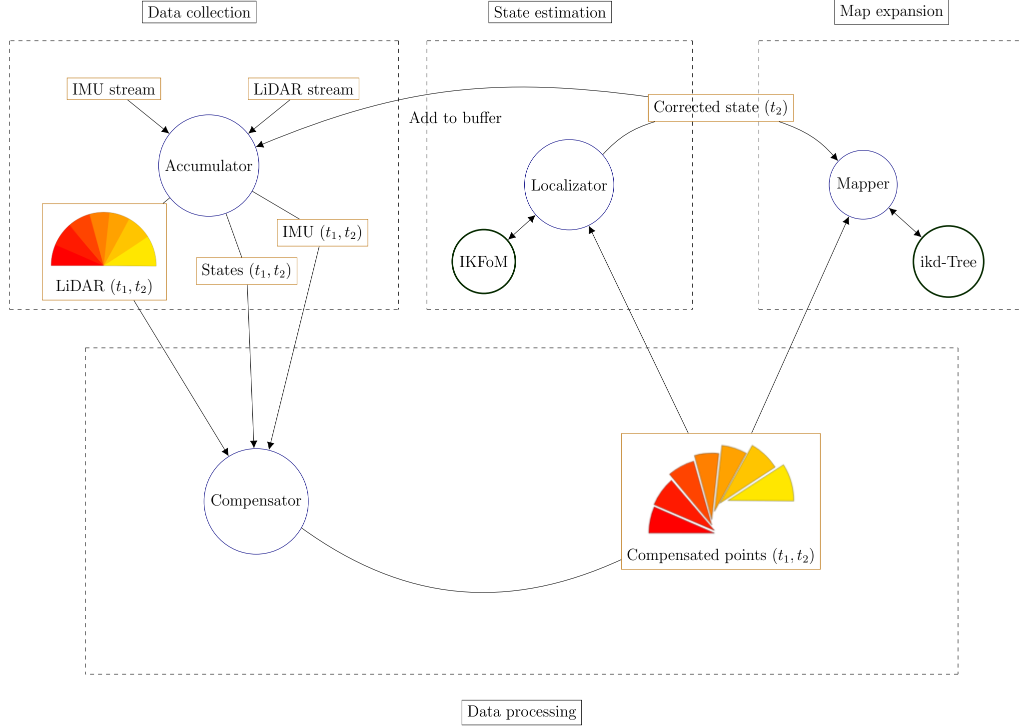


Figure 5.1: LIMO-Velo's pipeline.

5.2.1 Accumulator: Receiving streams of data

The Accumulator's job is to receive the sensor data, store it and deliver to whoever needs it. It also accumulates all the corrected states we calculate. Basically, if we want data: we call the Accumulator.

- Input: timestamps $t1$, $t2$ and type of data (points/IMU measurements/states).
- Output: collection of chosen data between $t1$ and $t2$.

5.2.2 Compensator: Adjusting for motion

The compensator is in charge of correcting the motion distortion of a collection of timestamped points knowing what position, velocity and acceleration was the robot having at those timestamps.

- Input: timestamps $t1$, $t2$.
- Output: collection of motion compensated points between t_1 and t_2 in the local frame.

5.2.3 Localizator: Estimating the state

The Localizator updates the predicted state and turns it to a "corrected" state using IKFoM's Kalman Filter [8] toolkit.

- Input: local observations between $t1$ and $t2$, predicted state at $t2$.
- Output: corrected state at $t2$.

5.2.4 Mapper: Building the map

The Mapper registers global points onto the map (an ikd-Tree).

- Input: points on the global frame.
- Output: None. It updates itself.

5.3 Implementation

A hard requirement needed for this project was to for it to be a long-term resource to Barcelona's Formula Student team. Therefore, it needed to be clearly readable, easy to understand, easy to maintain and easy to improve. We followed three key software design methodologies to satisfy these objectives:

5.3.1 Modular programming

Modular programming emphasizes on separating the functionality of a program into independent, interchangeable modules such that each contains everything necessary to execute only one aspect of the desired functionality. Benefits to this are:

- Ease to debug: one can try every module for itself to see if the output corresponds to the one expected.
- Ease to improve: if one wants to change a module, one doesn't have to worry about it affecting the overall code since it's independent of them.
- Ease to understand: every module has one specific functionality self-explained in the function name. Makes the main code read like pseudo-code.

5.3.2 Functional programming

Functional programming is a programming paradigm on which programs are constructed by applying and composing mathematical functions, clearly specifying input and output. Functional programming does not accept the use of global variables if they are not universal constants: functions take an input and give an output. Neither the input changes nor any other global variable changes, a function only creates an output. Benefits of this are:

- Ease to debug: there are no unexpected side-effects when applying a function.

- Ease to understand: functions are simple and the input/output concept is something any engineering student is already used to after their first two years of university.

5.3.3 Object-oriented programming

Object-oriented programming is a programming paradigm based on the concept of "objects". Every object has properties (data) and methods (functions). Objects have public methods which are clearly readable (close to pseudo-code) that call private methods that are the raw implementation (usage of specific data-structures or formulas to convey an output). This distinction separates the methods' design (the what) from the actual implementation of them (the how). Benefits of this are:

- Ease to improve: whenever we identify bottlenecks in design (order of operations, data flow...) or bottlenecks in implementation (data structures used, formulas used, packages used...) we know where to look and what to change.

- Ease to understand: if we want to understand what a method does, we read its public part, if we want to understand how a method does it we read the private part.

- Ease to maintain: if a private method uses a package that needs to be changed, we don't need to change anything from the public method.

- Self-contained code: defining canonical objects and their relations between them, we let ourselves go of third party objects that have incompatibility issues and lack of methods that combine them.

5.4 Killer app

Now we are going to talk about which is the specific case on where LIMO-Velo shines. Every good SLAM implementation is thought and designed to improve on one specific (but broad enough) case. So let's see where LIMO-Velo works when no other solution does.

5.4.1 Racing: fast and aggressive motion

Under a racing environment, there are no open-source methods to this date - to the best of the author's knowledge - that can handle racing speeds and turns with a spinning LiDAR. The main reasons are: high latency in localization and high latency in mapping.

1. High latency in localization causes jumps to up to 2 meters (100ms at 20m/s) from correction to correction and that inevitably causes bad data associations.

2. High latency in mapping causes a lack of information near the new state of the robot, leading to worse corrections and uncertainty.

LIMO-Velo addresses this two issues by:

1. Achieving low-latency localization: Using the Accumulator to select the desired localization time span $(t1, t2]$ and using IKFoM's Kalman Filter constant time computation. Results show that the algorithm works on time spans as short as $t2 - t1 = 0.0001$.

2. Achieving low-latency mapping: Using an incremental KD-tree as the map's data structure, mapping is two orders of magnitude faster than a static KD-tree.

Additionally, LIMO-Velo re-calculates matches for each Kalman Filter iteration arguing that fast and aggressive movement can lead to big enough differences even with short time spans. In constrast, Fast-LIO does reuse matches if the iterated Kalman Filter says it has converged (sometimes wrongly due to degeneration). Fast-LIO at 10Hz fails in aggressive scenarios, LIMO-Velo does not.

Chapter 6

Results

Note 6.0.1. When comparing with other algorithms, all overlapping parameters have been set to the same ones to not favor any.

6.1 Robustness

LIMO-Velo takes into consideration different parameters that control how much data is processed, the level of refinement we want, estimates of the noise covariance, calibration extrinsics... A truly robust model should not depend on exact parameters to work and should have a certain margin for variation without failing, even in the most adverse scenarios.

6.1.1 Data loss/downsampling

We now study how much LIMO-Velo depends on dense and stable sensor readings. First we consider data downsampling, arguing other sensors may have different data densities.

- **LiDAR downsampling**: Results on the KITTI dataset [6] show that the first rate of downsampling to fail at least in one run is 64. Compared to Fast-LIO2's, which is 8.

- **IMU downsampling** Results show that the IMU can be downsampled down to 50Hz and still work on racing conditions. Usual IMUs work at more than 100Hz, at least.

Then, we consider occasional data loss, arguing that our algorithm cannot fail if a hardware issue causes it to lose sensor data for a small amount of time.

- **IMU loss** Results show that losing IMU data completely for a short period of time ($< 1s$) when there's no aggressive movement taking place, the algorithm still recovers.

- **LiDAR loss** Results show that losing LiDAR data for a short period of time ($< 1s$), the algorithm still recovers.

6.1.2 Size of partitions (field of view)

Different sizes of point cloud partitions (or field of view) should not cause our algorithm to fail. Bigger size of partitions means more stability in a complex scenario but in fast and aggressive motion scenarios, smaller partitions offer more frequent corrected estimations.

Results show that on complex scenarios on the KITTI dataset [6] such as a drive besides a forest (runs 27, 28 of the KITTI odometry dataset) need big fields of view because it's a high-frequency adverse environment where bad data associations are common. Alternatively, results show that in a racing scenario, big fields of view work worse because the bigger differences between predicted and corrected cause the data associating algorithm to be caught in local minima.

6.1.3 IMU parameters

When integrating IMU measurements, we are propagating their noise. Therefore, we need to detect and model the IMU measurement's noise in order to know what amount can we expect to correct. We do that by assuming that the accelerometer and gyroscope error from the ground truth both come from a multivariate $\mathbb{R}^3$ normal with means μ_a, μ_g and covariance matrices $\sigma_a \mathbb{I}_{3\times3}, \sigma_g \mathbb{I}_{3\times3}$.

We will assume the biases μ_a, μ_g come from another multivariate $\mathbb{R}^3$ normal with mean $\bar{0} \in \mathbb{R}^3$ and covariance matrices $\nu_a \mathbb{I}_{3\times3}, \nu_g \mathbb{I}_{3\times3}$. Therefore, we have 4 parameters to tune: two error biases variances ν and two error variances σ.

Error biases variances

Biases usually are very small, in the order of 10^{-4}. Results show that modelling the bias error variance under its true amount causes a slight drop in map quality but it doesn't cause it to drift. Modelling it over its true amount, causes the algorithm to estimate oversized biases and eventually fail.

Error variances

Variances are usually bigger, specially the accelerometer variance, in the order of 10^{-2}. Results show that modelling the error variance under its true amount causes the algorithm to trust too much the IMU and fail. Modelling it over, causes an over-calculated propagated covariance that doesn't cause failure but we are not able to get accurate intervals of confidence of our estimated state.

6.2 Computation performance

6.2.1 Speed of computation

Results show that on the KITTI dataset [6], LIMO-Velo is about $\times 1.35$ slower than Fast-LIO2 [27]. However, thanks to the results of the Fast-LIO2 paper, we can conclude that LIMO-Velo is about $\times 8$ faster than other state-of-the-art algorithms LIO-SAM [21], LILI-OM [12] and LINS [18]. Setting the field of view smaller, however, makes the algorithm run slightly slower.

6.3 Performance

6.3.1 KITTI dataset

From the results in Table 6.1 we see LIMO-Velo having a more consistent performance than Fast-LIO2 [27] with a total averaged (total error divided by total length) relative improvement of $-\mathbf{20.04}\%$ (absolute improvement of $-\mathbf{1.09}\%$).

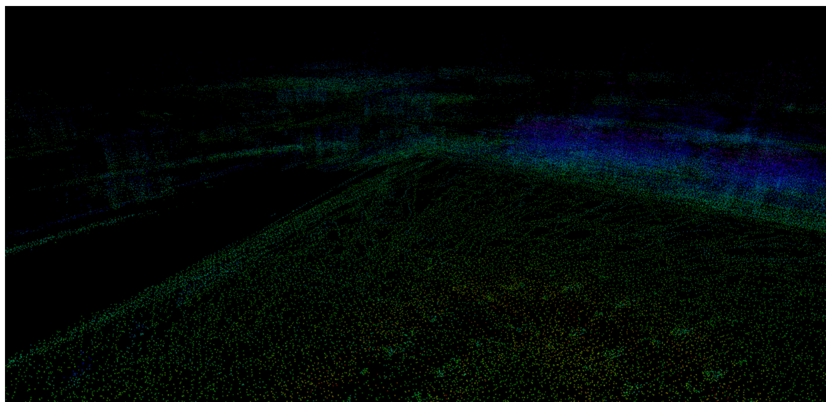
Error is calculated by getting the difference of position at time $t+1$: $p_{t+1} - p_t$ and the orientation rotation matrix at time t: Q_t of the algorithm output and the ground truth data. Then, the error is the sum of:

$$e_t := Q_{t-1,a}^{-1} \cdot (p_t^a - p_{t-1}^a) - Q_{t-1,g}^{-1} \cdot (p_t^g - p_{t-1}^g)$$

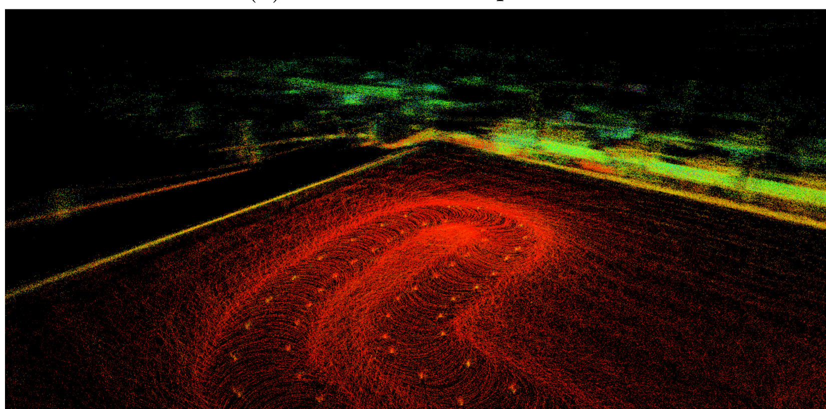
divided by the total sum of the total length of ground truth data: $dp_t^g := \|p_t^g - p_{t-1}^g\|$ and multiplied by 100. The idea around this is that we are comparing locally the position increments at each time t allowing us to detect drift at every incremental dt instead of comparing the accumulated drift at the end.

Run name	LV error ↓	FL error ↓	Relative difference
kitti_2011_09_26_drive_0001	**0.95%**	0.97%	-1.21%
kitti_2011_09_26_drive_0002	1.16%	**1.04%**	9.75%
kitti_2011_09_26_drive_0009	**5.15%**	7.42%	-44.14%
kitti_2011_09_26_drive_0011	2.05%	**2.00%**	2.29%
kitti_2011_09_26_drive_0013	**1.20%**	1.25%	-4.22%
kitti_2011_09_26_drive_0014	1.00%	**0.89%**	10.66%
kitti_2011_09_26_drive_0015	**0.57%**	3.59%	-522.97%*
kitti_2011_09_26_drive_0019	**1.86%**	3.04%	-63.07%
kitti_2011_09_26_drive_0022	**6.81%**	13.19%	-93.61%
kitti_2011_09_26_drive_0027	**0.46%**	0.71%	-53.33%
kitti_2011_09_26_drive_0028	**2.75%**	2.79%	-1.75%
kitti_2011_09_26_drive_0029	**3.09%**	6.89%	-122.85%*
kitti_2011_09_26_drive_0032	**0.60%**	1.16%	-90.49%
kitti_2011_09_26_drive_0036	6.28%	**6.20%**	1.22%
kitti_2011_09_26_drive_0039	2.49%	**2.37%**	4.86%
kitti_2011_09_26_drive_0051	**1.89%**	3.65%	-93.11%
kitti_2011_09_26_drive_0056	0.60%	**0.54%**	9.27%
kitti_2011_09_26_drive_0059	**2.08%**	2.13%	-2.04%
kitti_2011_09_26_drive_0061	**3.40%**	10.88%	-219.52%*
kitti_2011_09_26_drive_0064	**2.23%**	2.38%	-6.50%
kitti_2011_09_26_drive_0070	**0.77%**	4.81%	-519.78%*
kitti_2011_09_26_drive_0084	2.42%	**2.27%**	6.07%
kitti_2011_09_26_drive_0086	**9.79%**	9.96%	-1.69%
kitti_2011_09_26_drive_0087	**14.9%**	18.79%	-25.76%
kitti_2011_09_26_drive_0091	**4.31%**	8.50%	-97.06%
kitti_2011_09_26_drive_0095	2.95%	**2.86%**	2.96%
kitti_2011_09_26_drive_0101	**0.71%**	0.77%	-8.77%
kitti_2011_09_26_drive_0104	**2.32%**	7.72%	-232.49%*
kitti_2011_09_26_drive_0106	3.51%	**3.28%**	6.38%
kitti_2011_09_26_drive_0117	**11.4%**	15.46%	-35.45%

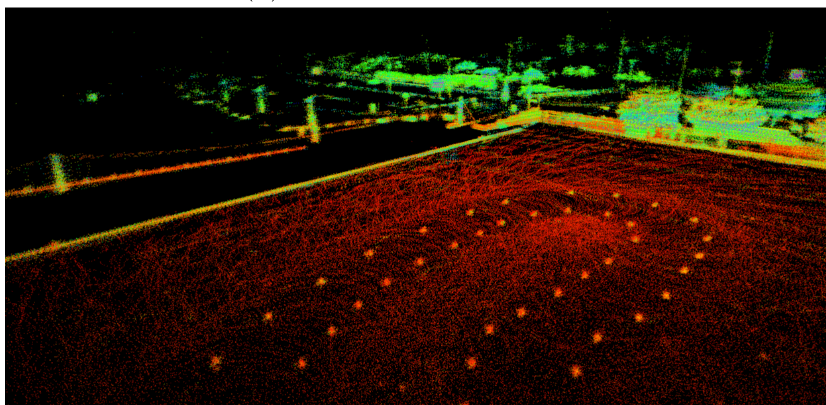
Table 6.1: LIMO-Velo (LV) vs. Fast-LIO (FL) error comparison on the KITTI dataset. LV does better in 21/30 runs. Starred runs (*) are runs with a jump in improvement.



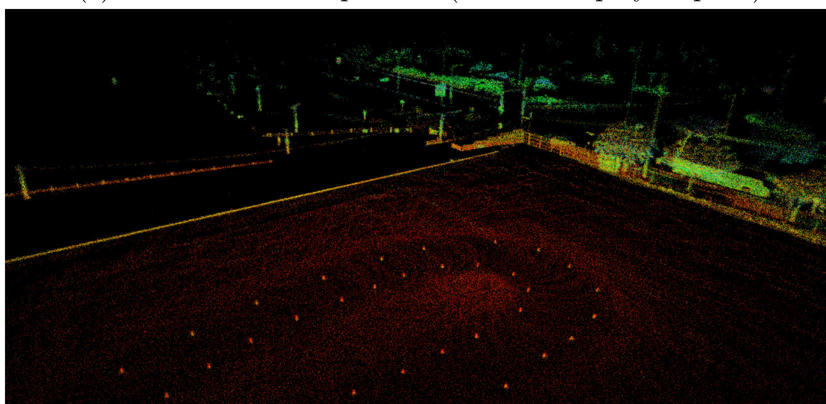
(a) ALOAM - Viewpoint A



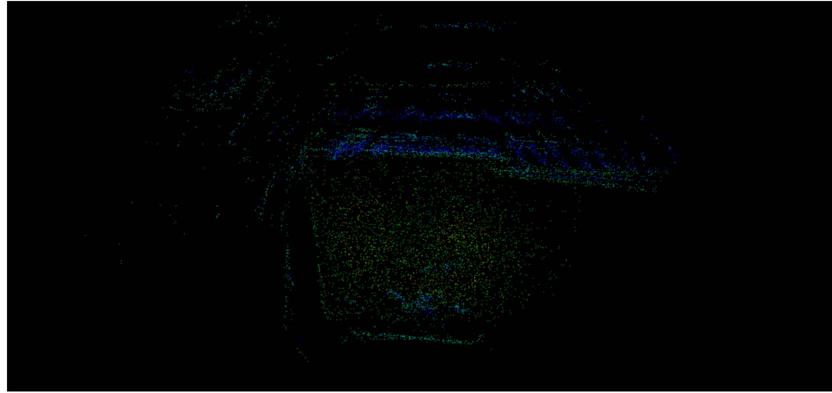
(b) Fast-LIO - Viewpoint A



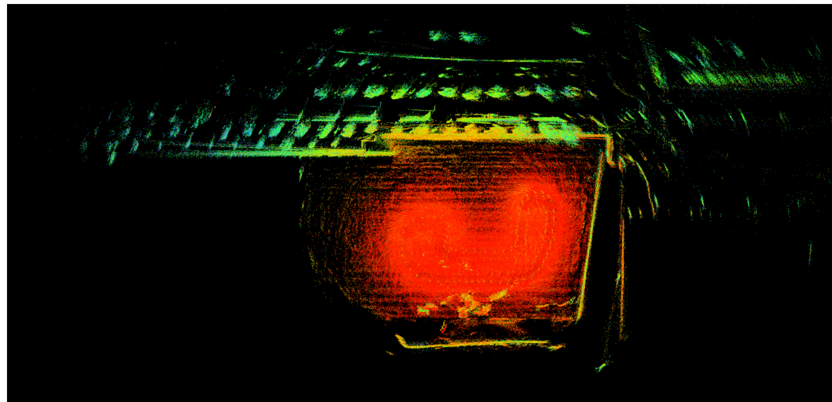
(c) LIO-SAM - Viewpoint A (with $\times 0.5$ player speed)



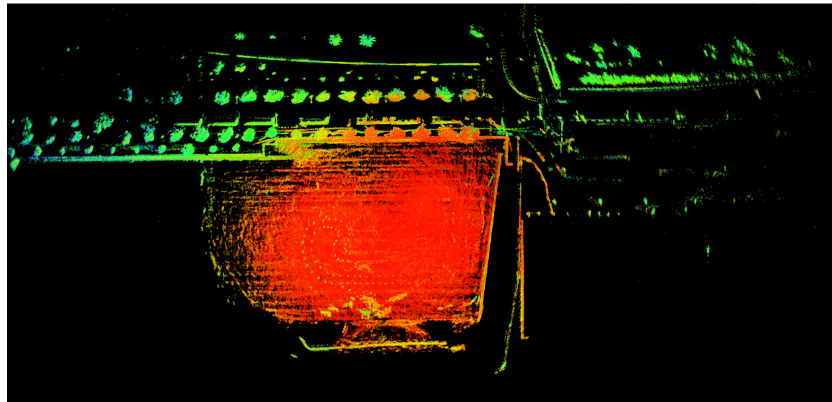
(d) LIMO-Velo - Viewpoint A



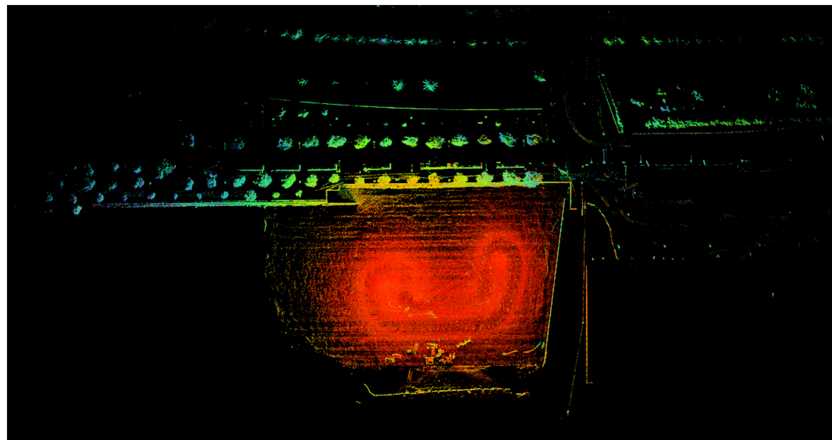
(a) ALOAM - Viewpoint B



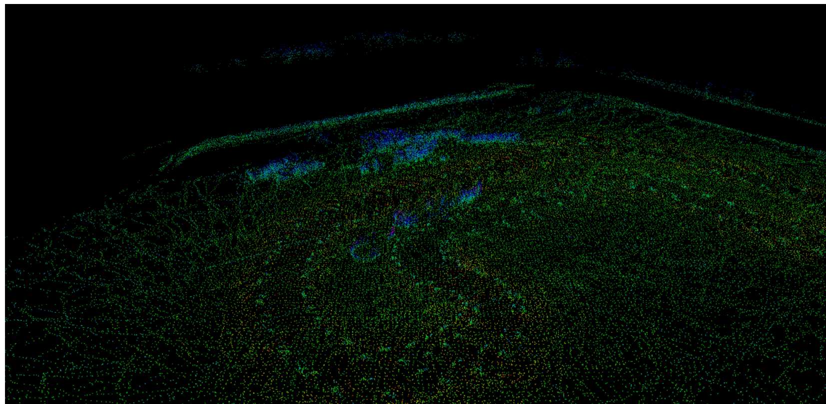
(b) Fast-LIO - Viewpoint B



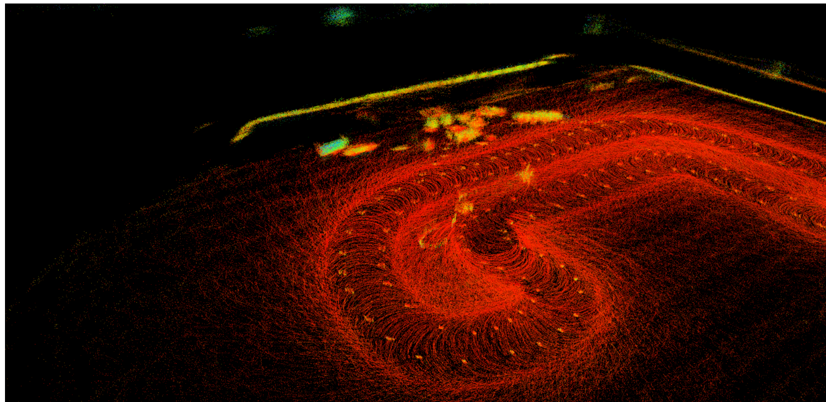
(c) LIO-SAM - Viewpoint B (with $\times 0.5$ player speed)



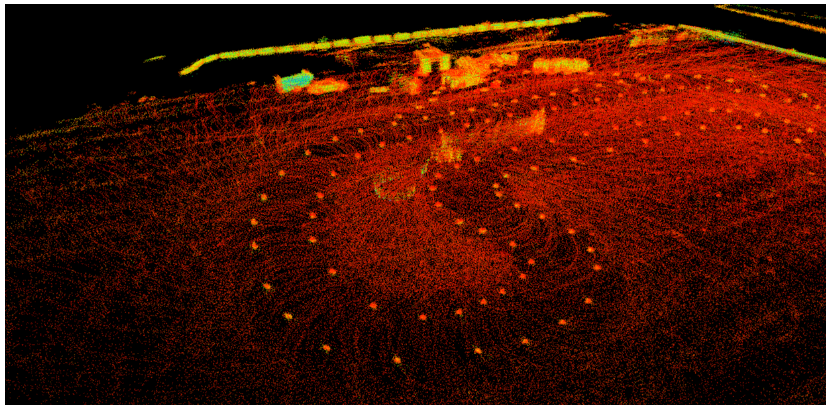
(d) LIMO-Velo - Viewpoint B



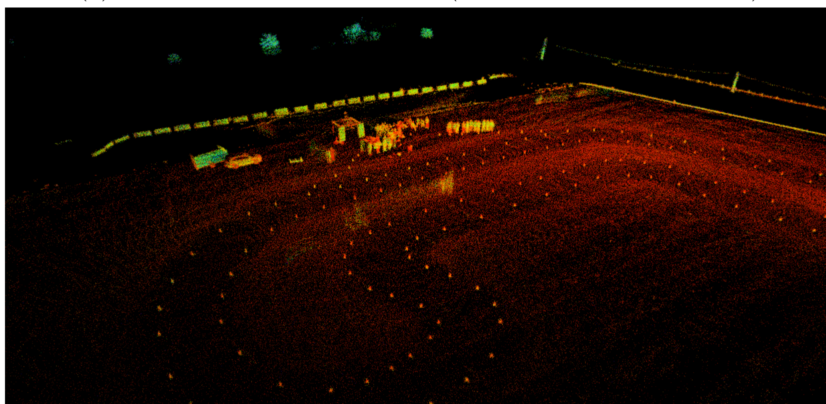
(a) ALOAM - Viewpoint C



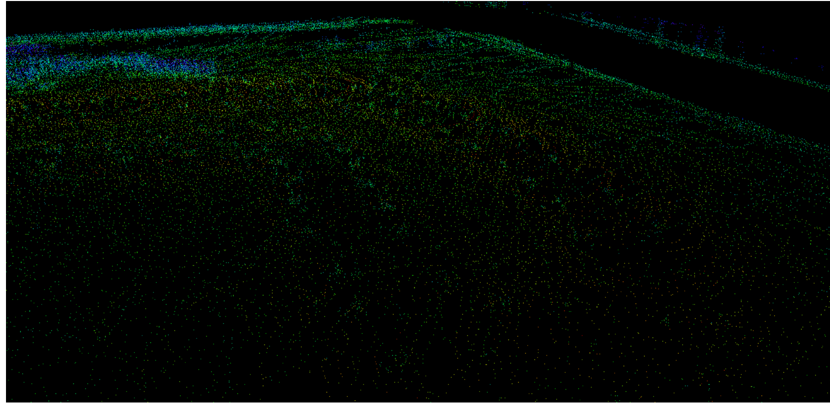
(b) Fast-LIO - Viewpoint C



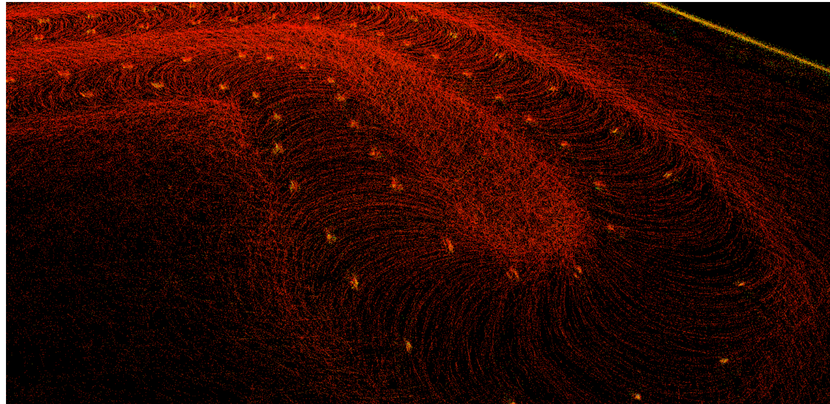
(c) LIO-SAM - Viewpoint C (with $\times 0.5$ player speed)



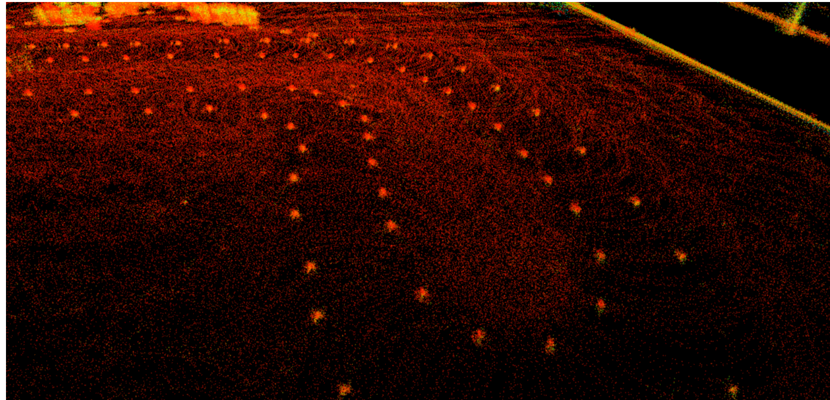
(d) LIMO-Velo - Viewpoint C



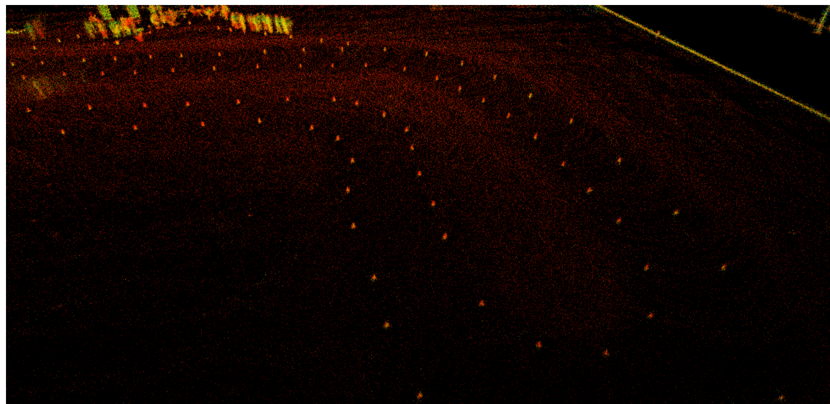
(a) ALOAM - Viewpoint D



(b) Fast-LIO - Viewpoint D



(c) LIO-SAM - Viewpoint D (with ×0.5 player speed)



(d) LIMO-Velo - Viewpoint D

6.3.2 Xaloc's map comparison

Note 6.3.1. LIO-SAM [21] had to be fed data at $\times 0.5$ speed, because it failed otherwise. Fast-LIO2 [27], A-LOAM [29] and LIMO-Velo were performing at real-time.

6.3.3 Xaloc's odometry comparison

The recorded run are two laps in the same circuit for 60 seconds so loop closures can be compared. It is a fast but smooth drive, we give its velocity profile below.

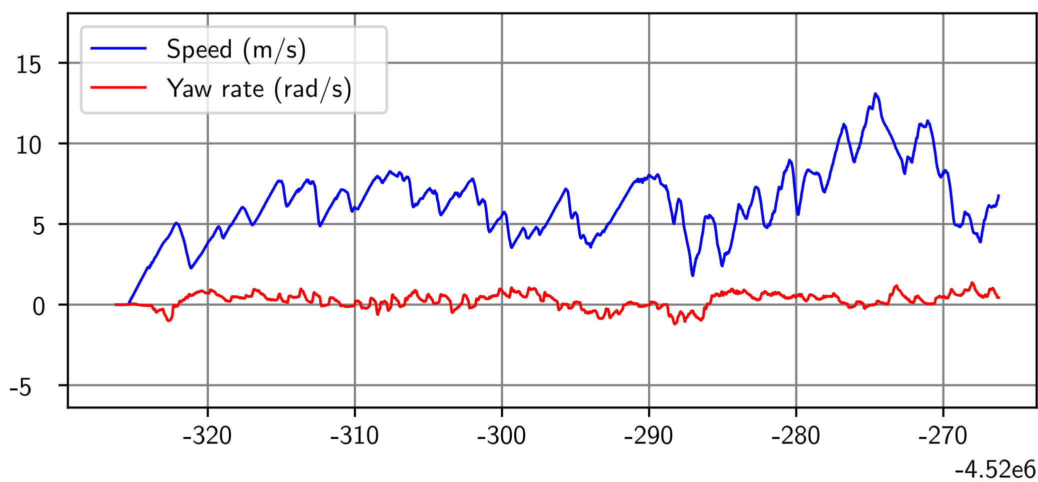


Figure 6.5: Peak speed: 13m/s. Peak turn speed: 80deg/s.

- **A-LOAM** cannot close the loop.
- **LIO-SAM** shows erratic movements.
- **Fast-LIO** shows smooth and accurate loop closures.
- **LIMO-Velo** shows smooth and accurate loop closures.

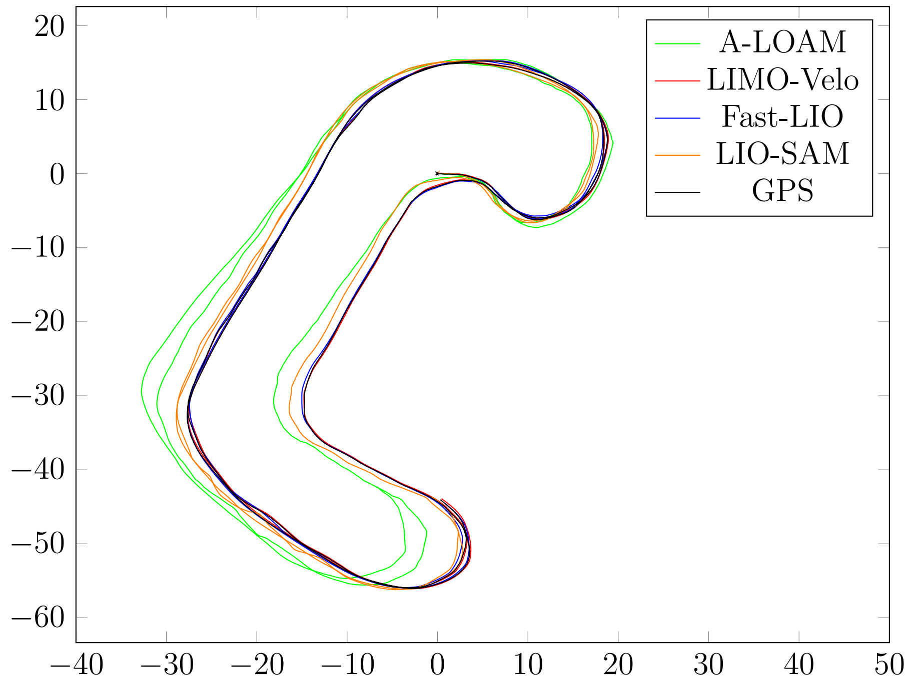


Figure 6.6: All algorithms aggregated.

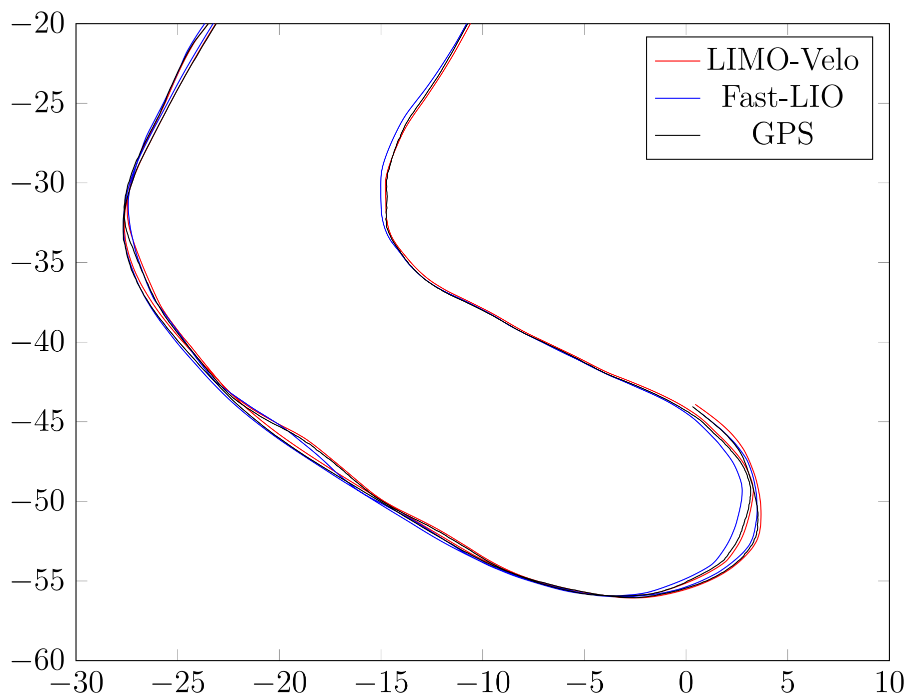


Figure 6.7: Two best performers compared - Down part.

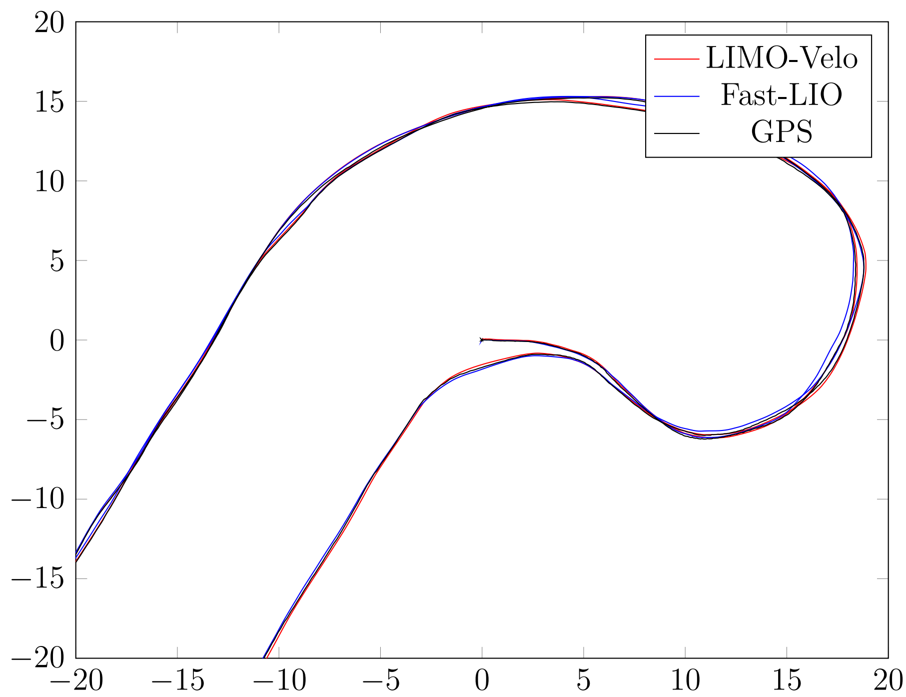


Figure 6.8: Two best performers compared - Up part.

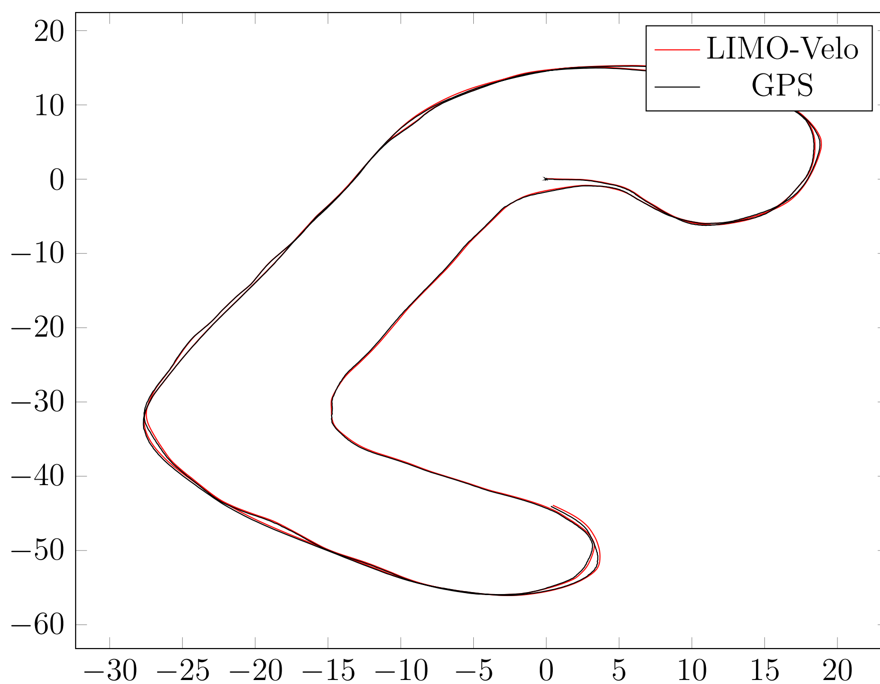


Figure 6.9: LIMO-Velo odometry

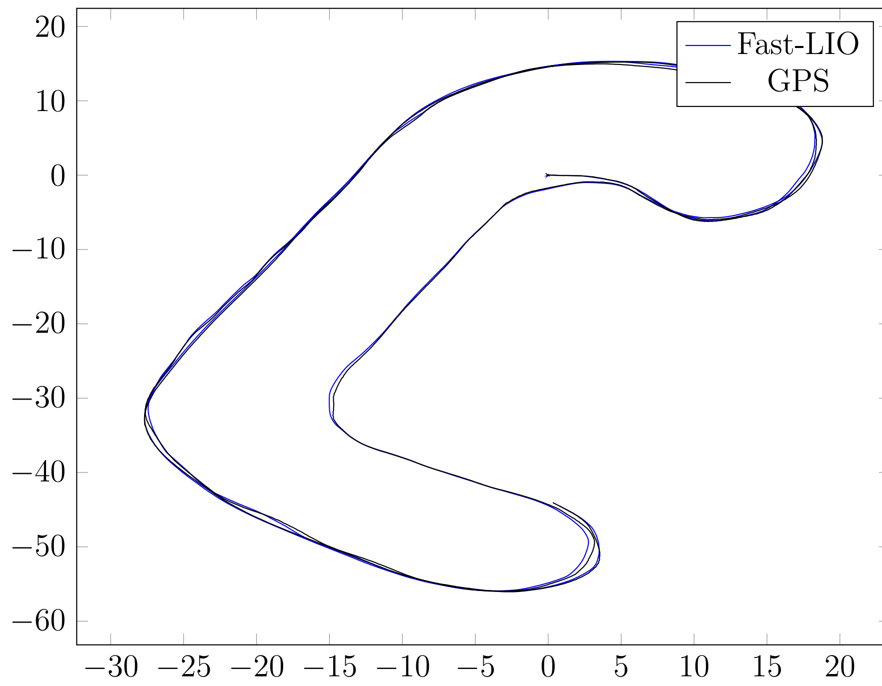


Figure 6.10: Fast-LIO odometry

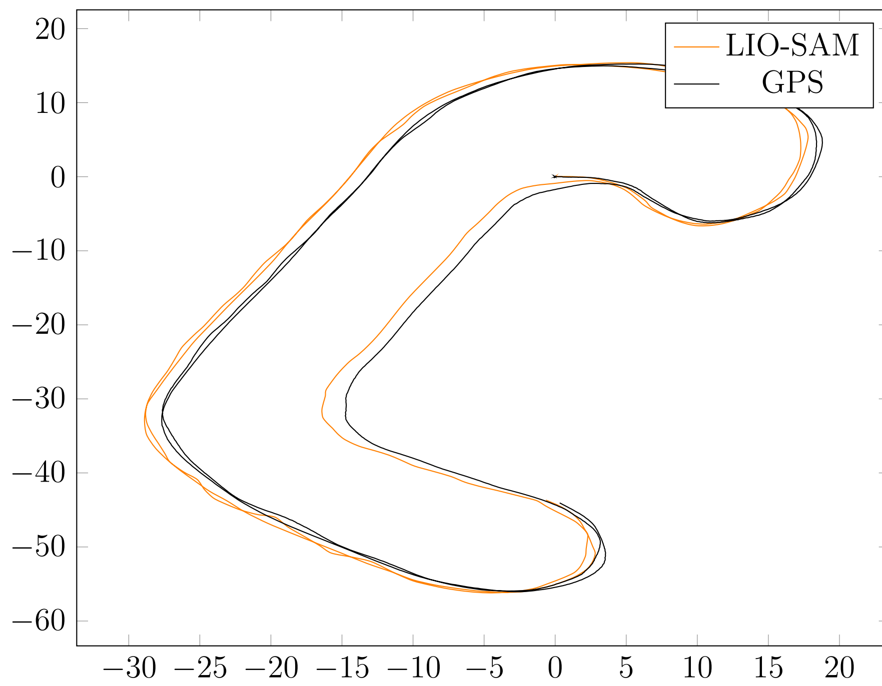


Figure 6.11: LIO-SAM odometry

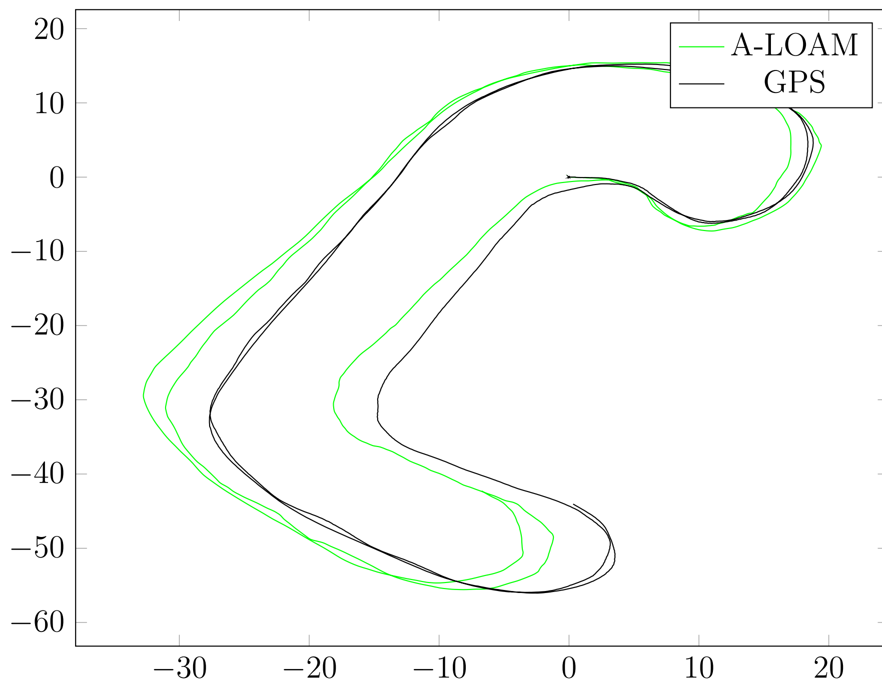


Figure 6.12: A-LOAM odometry

Chapter 7

Conclusions

7.1 Main key contributions

7.1.1 Localize Intensively

We propose that in order to keep centimeter-level accuracy under high velocities, the algorithm needs more frequent localization results.

7.1.2 Map Offline

We propose, when working with spinning LiDARs, to accumulate all the points from a full rotation and then map them after it ends. Contrary to mapping them at the localization's rate.

7.2 Derived improvements

7.2.1 Improvement over the SOTA

Odometry improvement

We show a 20.04% improvement over Fast-LIO on the KITTI dataset [6] and show that LIMO-Velo works better than Fast-LIO2 [27], LIO-SAM [21] and A-LOAM [29] on our data.

Map quality improvement

We show a qualitative improvement of map quality over other SOTA algorithms. We show a $\sim$ 2cm error over the known specifications of the cone.

7.2.2 Improvement in infrastructure

Pointclouds as a stream of points

Instead of treating pointclouds as rigid data structures, we treat them as streams of stamped points. This conceptual decision helps achieve higher flexibility in the code and allows for easier motion compenstaion, easier localization updates and easier mapping updates.

Code designed to be improved

The code has been designed in a modular way meant to be improved. Its internal data structures are independent and can be changed and new sensors can be added smoothly.

7.3 Achieved objectives

LIMO-Velo achieves the objectives outlined in the first chapter of what we wanted from a SLAM algorithm.

A - Hard requirements

1. It successfully detects if we are crossing or not with the track limit with centimeter-level accuracy.

2. It outputs information at the frequency we desire, with options to go up to 1000Hz.

3. Successfully handles fast and aggressive motions of 20m/s and 500deg/s.

B - Soft requirements

1. It creates a long-range, high-density, centimeter-level map with enough detail to identify cones in it.

2. The Kalman Filter updates the sensor calibration parameters in real-time.

3. It is stable to occasional sensor failure and degeneracy.

C - Design requirements

1. Does not depend on GPS signal nor lightning conditions.
2. It has an easy and modular design made to be easily maintained.
3. CPU cost is low enough to not intercept with other parts of the pipeline.

Chapter 8

Future work

Note 8.0.1. This part is unusually long and detailed as its purpose is to clearly outline what can be improved of the algorithm in the next 2 to 3 seasons by the new generations of BCN eMotorsport.

8.1 Natural step: Dynamic objects' removal

In Chapter 5 it is already discussed that "Mapping Offline" allows the processing of the pointcloud that we will register on the map. An extra package for dynamic object detection has to be added to the code but a part from that, the infrastructure is already there. Here's the steps that need to be taken in order to have the dynamic object removal functionality:

1. Add as dependency a package for identifying moving objects from a pointcloud such as LMNet (LiDAR-MOS) by Chen et. al, 2021 [3].

2. Create a new object: the Masker. With the four following public methods:

   Identify. Given a collection of points, return the same collection of points marked with an extra variable determining if they belong to a moving object or not.

   Set. Given a collection of marked points, return a voxelgrid with 0s and 1s depending if each grid is corrupted or not.

   Apply. Given a point, return if that point is on a corrupted area or not.

3. While waiting to map, identify dynamic objects and set the mask.

4. Before mapping or localizing, apply the mask to the points and only map/localize with the non-corrupted points.

8.2 Other possible lines of work

8.2.1 Estimation of future actions of moving objects

Additional work can be done on tracking the removed moving objects as actual objects inside the framework. Possibly using Kalman Filters to estimate their state and predict where will they move to and create better masks. Work to be done is:

1. Have the "dynamic objects' removal" module up and running.

2. Define a new Agent object that consists of a list of attributes conveying information of the moving object and a Kalman filter estimating pose, orientation and velocity vector.

3. Given the pointcloud of corrupted points, cluster them into different entities.

4. For each entity, check if it already exists an Agent object which its predicted position's likelihood is high enough given this new observation and create an association. If there's not a close Agent, create a new one.

5. When setting the mask, consider masking the 95% percentile of each Agent's predicted position covariance matrix.

8.2.2 Loop closure detection

If we aim to build consistent maps of long routes, loop closures are a requirement. They help reduce accumulated drift in repeatable scenarios, such as a race with multiple laps. To perform loop closures efficiently, however, we would need yet another new tree structure: a temporal incremental KD-tree.

Modifying the ikd-Tree by timestamp

The algorithm now maps once and the points get lost on the sea of points in the tree, we do not have an efficient way to recover points from a same scan and modify them all at once. With loop closures, what happens is that we find a match between two states' positions at times t_k and t_l. When this happens, we add the constraint $p_{t_k} = p_{t_l}$ and therefore we need to smoothly correct all previous map addings until t_l to fit it.

We would need to efficiently access all points added in the time span $[t_l, t_k]$ and correct their location with the new estimates $x^*_{l:k}$. That would mean a

major change: being able to modify previous map addings by timestamp. To do that efficiently, a new data structure - a temporal incremental KD-Tree - should be created having the collection of all past corrected states and the mapped points associated to them, just like in GraphSLAM. Said states and the points mapped through them should be accessed and modifiable efficiently by timestamp.

8.2.3 Multiple LiDARs and IMUs

Another natural step is to ask how could we incorporate more LiDAR and IMU sensors. If we wanted to do that we would need to change a couple of things:

1. The Accumulator should consider more than one source for sensor.
2. The state used by the Kalman Filter should consider more than one group of calibration parameters.

The easy way to accept multiple LiDAR sensors would be to do corrections one LiDAR at the time. With multiple IMU sensors, the predicted state would have to take into account all the IMU predictions (considering their extrinsic and intrinsic calibrations) and weight them into a single prediction.

8.2.4 Removing long term drift with GNSS

With long term travels, a part from loop closure, GNSS is a feature-agnostic sensor that even though it has a high accuracy error it is the most stable through long travels. To add GNSS to our framework, we would have to make the following changes:

1. The Accumulator should consider a new data source.
2. We should add the GNSS bias on the Kalman Filter's state since it changes a lot.
3. We should define a covariance matrix for the GNSS sensor on the configuration files.
4. The Localizator should take as correction the GNSS measures before correcting accurately with the LiDAR.
5. The GPS heading should be aligned with the IMU's heading.

8.3 The next step: LiDAR-Visual-Inertial Odometry and Mapping

A next step that naturally comes up is to add cameras into the mix for their good complementary benefits with LiDAR sensors. Cameras offer greater detail and can still see features on geometrically degenerate scenarios. LiDAR, on the other hand, brings precise depth estimation that can be complemented with the intensity information of the cameras.

8.3.1 Adding cameras without correcting pose

We will assume the LiDAR-camera online calibration will be done parallelly with a separate program. In order to add cameras to LIMO-Velo, we would have to do the following steps:

1. The Accumulator should consider a new data source.
2. The Point object should be extended to have RGB attributes.
3. Before mapping, we should project the camera(s) color values to the points we are about to register to the map.

8.3.2 Adding cameras for correcting pose: R^2LIVE and R^3LIVE

The Fast-LIO [26] [27] authors also have thought about adding cameras to improve Fast-LIO's accuracy and they published two solutions: R^2LIVE (Jin et. al, 2021) [15] and R^3LIVE (Jin et. al, 2021) [14] that do exactly that. The first one, R^2LIVE (2021), mixes Fast-LIO (2020) with VINS-Mono (2018) [19] a widely used Visual-Inertial solution by Qin et. al from the HKUST. R^3LIVE (2021) mixes Fast-LIO2 (2021) with their own (unpublished) direct visual-inertial solution. The idea is to use a shared state and shared global map and estimate them using both camera intensities and LiDAR points. Therefore, have two tightly-coupled LIO and VIO SLAMs parallelly where:

1. LIO: contributes to the geometry structure of the map and serves as the VIO's depth estimation.
2. VIO: contributes to the map's texture.

R^2LIVE's main structure is a factor graph and optimizes LiDAR and camera observations. R^3LIVE, on the other hand, relies on the fact that LiDAR

gives geometric structure and camera gives texture information. Therefore, it can use LiDAR to build a geometric foundation that serves as depth to the camera and the camera acts as a refined odometry by using all the details vision offers.

8.4 The goal: Visual-Inertial SLAM

R^3LIVE [14] already proves that cameras offer greater feature-rich environments and can achieve more refined results than LiDAR solutions. Therefore, it's only natural to think of ways to bypass the LiDAR [17] altogether and overcome its liabilities.

8.4.1 LiDAR liabilities

As discussed in Chapter 1, LiDARs are prohibitively expensive, hard to maintain and offer little details in geometrically degraded environments. In the contrary, cameras are cheap we can put lots of them everywhere and are easy to replace and maintain for their wide usage.

8.4.2 Difficulties and requirements

As R^3LIVE [14] shows, having a great depth estimation goes a long way and major efforts are taking place on depth estimation, specially from the deep learning community, to solve this problem.

Deep learning approaches still have a long journey to match the robustness and accuracy of current state-of-the-art SLAM methods. The community effort to push forward the field to achieve a truly "complete" SLAM is very present on the number of SLAM papers released yearly, and my bet is that it will come from non-end-to-end methods first.

There's still room for lots of improvement and it's exciting to see that Formula Student can be a key contributor to the autonomous revolution.

Bibliography

[1] Jon Louis Bentley. "Multidimensional Binary Search Trees Used for Associative Searching". In: *Commun. ACM* 18.9 (1975), 509–517. ISSN: 0001-0782. DOI: 10.1145/361002.361007. URL: https://doi.org/10.1145/361002.361007.

[2] Yixi Cai, Wei Xu, and Fu Zhang. "ikd-Tree: An Incremental K-D Tree for Robotic Applications". In: *CoRR* abs/2102.10808 (2021). arXiv: 2102.10808. URL: https://arxiv.org/abs/2102.10808.

[3] X. Chen et al. "Moving Object Segmentation in 3D LiDAR Data: A Learning-based Approach Exploiting Sequential Data". In: *IEEE Robotics and Automation Letters (RA-L)* (2021). ISSN: 2377-3766. DOI: 10.1109/LRA.2021.3093567.

[4] H. Durrant-Whyte and T. Bailey. *Simultaneous localization and mapping: part I*. 2006. DOI: 10.1109/mra.2006.1638022. URL: http://dx.doi.org/10.1109/MRA.2006.1638022.

[5] *Formula Student Germany Website*. URL: https://www.formulastudent.de/fsg/.

[6] A Geiger et al. *Vision meets robotics: The KITTI dataset*. 2013. DOI: 10.1177/0278364913491297. URL: http://dx.doi.org/10.1177/0278364913491297.

[7] Formula Student Germany. *FSG Rules 2022*. https://www.formulastudent.de/fileadmin/user_upload/all/2022/rules/FS-Rules_2022_v1.0.pdf.

[8] Dongjiao He, Wei Xu, and Fu Zhang. *Kalman Filters on Differentiable Manifolds*. 2021. arXiv: 2102.03804 [cs.RO].

[9] Jocelyn Quaintance Jean Gallier. *Linear Algebra and Optimization with Applications to Machine Learning: Volume I: Linear Algebra for Computer Vision, Robotics, and Machine Learning*.

[10] Rudolf E. Kálmán. "A new approach to linear filtering and prediction problems" transaction of the asme journal of basic". In: 1960.

[11] Kistler. *Correvit SFII (GSS sensor) Data Sheet.* https://www.kistler.com/files/document/000-812e.pdf.

[12] Kailai Li, Meng Li, and Uwe D. Hanebeck. "Towards High-Performance Solid-State-LiDAR-Inertial Odometry and Mapping". In: *IEEE Robotics and Automation Letters* 6.3 (2021), pp. 5167–5174. DOI: 10.1109/LRA.2021.3070251.

[13] Jiarong Lin and Fu Zhang. *Loam livox: A fast, robust, high-precision LiDAR odometry and mapping package for LiDARs of small FoV.* 2020. DOI: 10.1109/icra40945.2020.9197440. URL: http://dx.doi.org/10.1109/ICRA40945.2020.9197440.

[14] Jiarong Lin and Fu Zhang. *R3LIVE: A Robust, Real-time, RGB-colored, LiDAR-Inertial-Visual tightly-coupled state Estimation and mapping package.* 2021. arXiv: 2109.07982 [cs.RO].

[15] Jiarong Lin et al. *R2LIVE: A Robust, Real-time, LiDAR-Inertial-Visual tightly-coupled state Estimator and mapping.* 2021. arXiv: 2102.12400 [cs.RO].

[16] Michael Montemerlo et al. "FastSLAM: A Factored Solution to the Simultaneous Localization and Mapping Problem". In: Nov. 2002.

[17] Raul Mur-Artal and Juan D. Tardos. "ORB-SLAM2: An Open-Source SLAM System for Monocular, Stereo, and RGB-D Cameras". In: *IEEE Transactions on Robotics* 33.5 (2017), 1255–1262. ISSN: 1941-0468. DOI: 10.1109/tro.2017.2705103. URL: http://dx.doi.org/10.1109/TRO.2017.2705103.

[18] Chao Qin et al. *LINS: A Lidar-Inertial State Estimator for Robust and Efficient Navigation.* 2020. arXiv: 1907.02233 [cs.RO].

[19] Tong Qin, Peiliang Li, and Shaojie Shen. "VINS-Mono: A Robust and Versatile Monocular Visual-Inertial State Estimator". In: *IEEE Transactions on Robotics* 34.4 (2018), 1004–1020. ISSN: 1941-0468. DOI: 10.1109/tro.2018.2853729. URL: http://dx.doi.org/10.1109/TRO.2018.2853729.

[20] Tixiao Shan and Brendan Englot. *LeGO-LOAM: Lightweight and Ground-Optimized Lidar Odometry and Mapping on Variable Terrain.* 2018. DOI: 10.1109/iros.2018.8594299. URL: http://dx.doi.org/10.1109/IROS.2018.8594299.

[21] Tixiao Shan et al. *LIO-SAM: Tightly-coupled Lidar Inertial Odometry via Smoothing and Mapping.* 2020. DOI: 10.1109/iros45743.2020.9341176. URL: http://dx.doi.org/10.1109/IROS45743.2020.9341176.

[22] Joan Solà. "Quaternion kinematics for the error-state Kalman filter". In: *CoRR* abs/1711.02508 (2017). arXiv: 1711.02508. URL: http://arxiv.org/abs/1711.02508.

[23] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic Robotics (Intelligent Robotics and Autonomous Agents)*. The MIT Press, 2005. ISBN: 0262201623.

[24] Sebastian Thrun and Michael Montemerlo. *The Graph SLAM Algorithm with Applications to Large-Scale Mapping of Urban Structures*. 2006. DOI: 10.1177/0278364906065387. URL: http://dx.doi.org/10.1177/0278364906065387.

[25] Sebastian Thrun et al. *Stanley: The Robot That Won the DARPA Grand Challenge*. 2007. DOI: 10.1007/978-3-540-73429-1_1. URL: http://dx.doi.org/10.1007/978-3-540-73429-1_1.

[26] Wei Xu and Fu Zhang. *FAST-LIO: A Fast, Robust LiDAR-Inertial Odometry Package by Tightly-Coupled Iterated Kalman Filter*. 2021. DOI: 10.1109/lra.2021.3064227. URL: http://dx.doi.org/10.1109/LRA.2021.3064227.

[27] Wei Xu et al. *FAST-LIO2: Fast Direct LiDAR-Inertial Odometry*. 2022. DOI: 10.1109/tro.2022.3141876. URL: http://dx.doi.org/10.1109/TRO.2022.3141876.

[28] Ji Zhang, Michael Kaess, and Sanjiv Singh. *On degeneracy of optimization-based state estimation problems*. 2016. DOI: 10.1109/icra.2016.7487211. URL: http://dx.doi.org/10.1109/ICRA.2016.7487211.

[29] Ji Zhang and Sanjiv Singh. *LOAM: Lidar Odometry and Mapping in Real-time*. 2014. DOI: 10.15607/rss.2014.x.007. URL: http://dx.doi.org/10.15607/RSS.2014.X.007.