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.

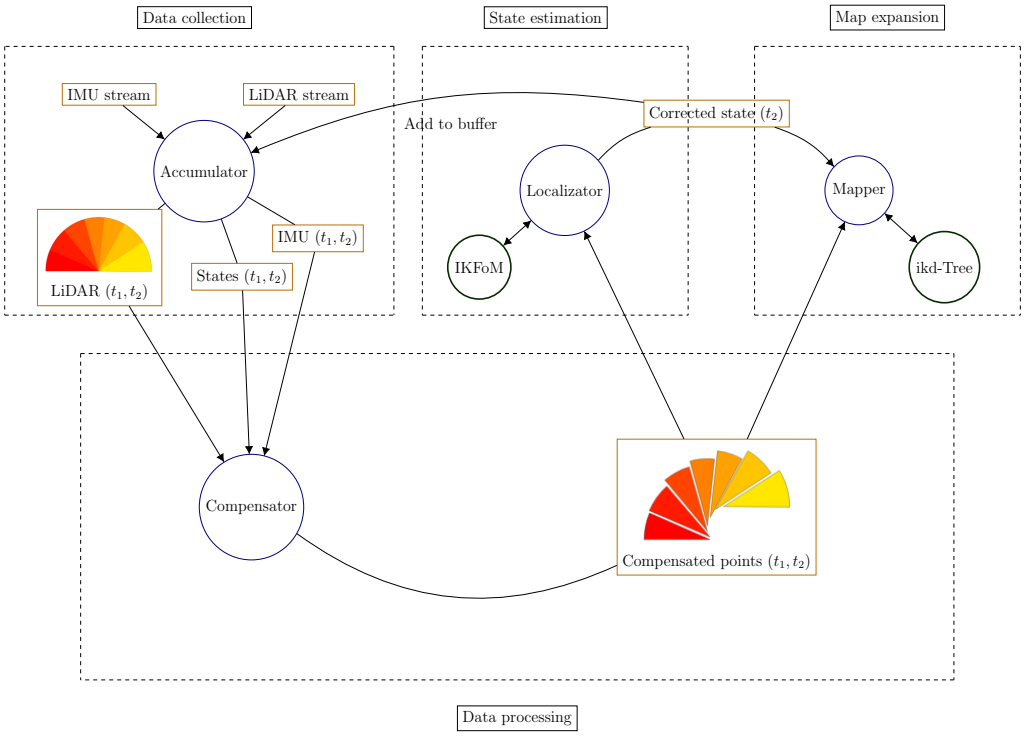


Figure 5.1: LIMO-Velo's pipeline.

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 $\mathbf{-1.09}\%$).

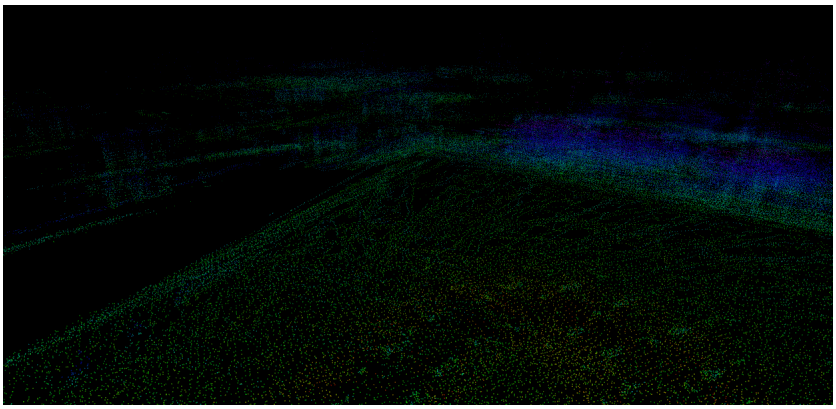
Error is calculated by getting the difference of position at time $t+1$: $p_{t+1} - p_t$ and the orientation rotation matrix at time t: Q_t of the algorithm output and the ground truth data. Then, the error is the sum of:

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

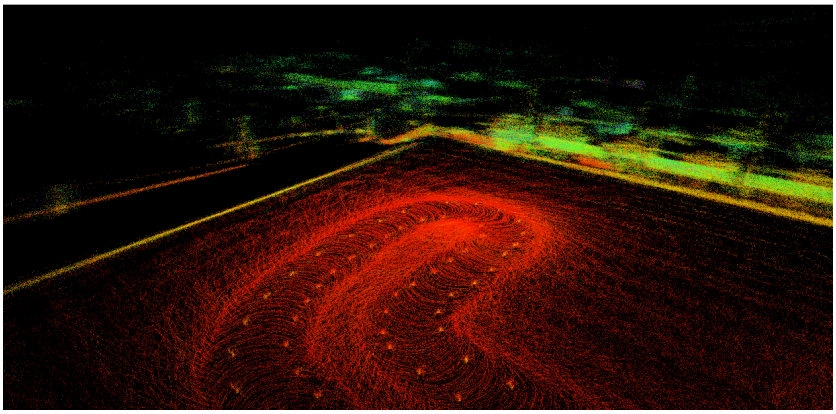
divided by the total sum of the total length of ground truth data: $dp_t^g := \|p_t^g - p_{t-1}^g\|$ and multiplied by 100. The idea around this is that we are comparing locally the position increments at each time t allowing us to detect drift at every incremental dt instead of comparing the accumulated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

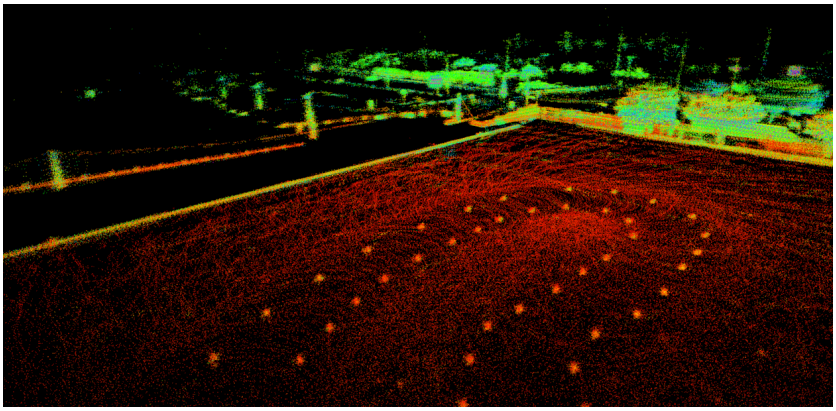
Table 6.1: LIMO-Velo (LV) vs. Fast-LIO (FL) error comparison on the KITTI dataset. LV does better in 21/30 runs. Starred runs (*) are runs with a jump in improvement.



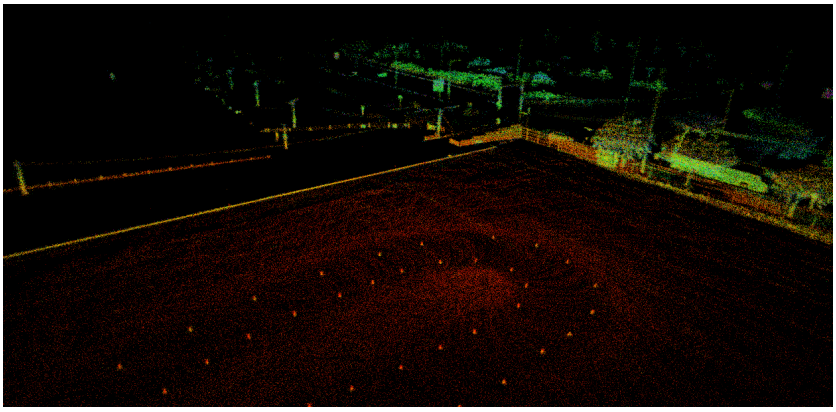
(a) ALOAM - Viewpoint A



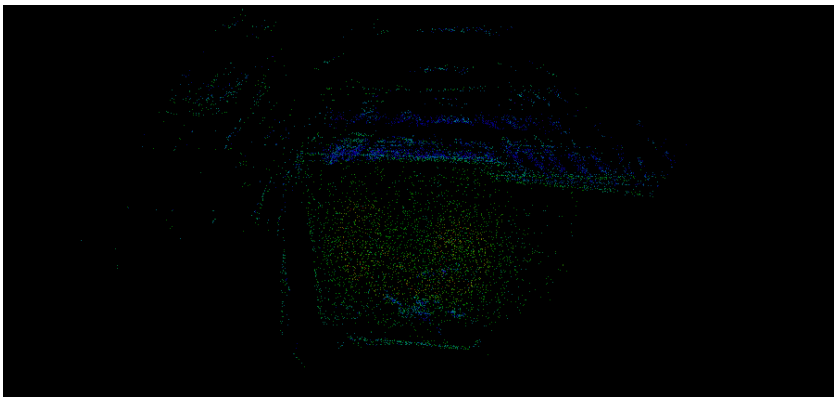
(b) Fast-LIO - Viewpoint A



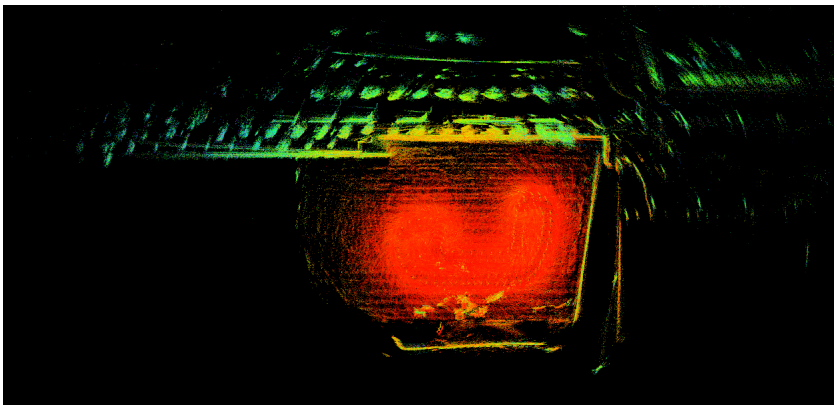
(c) LIO-SAM - Viewpoint A (with $\times 0.5$ player speed)



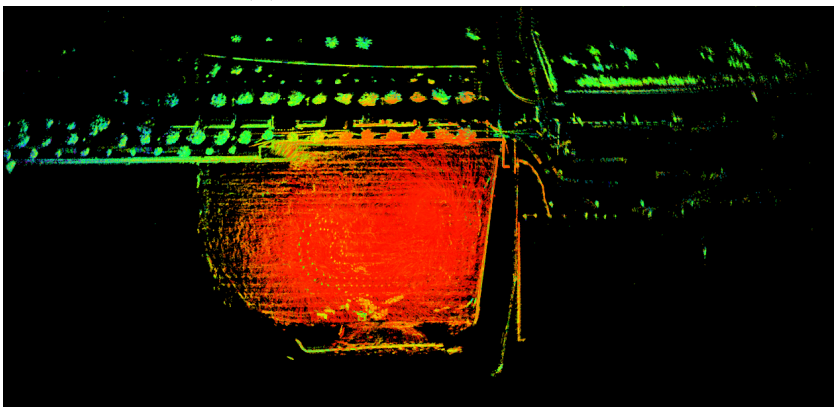
(d) LIMO-Velo - Viewpoint A



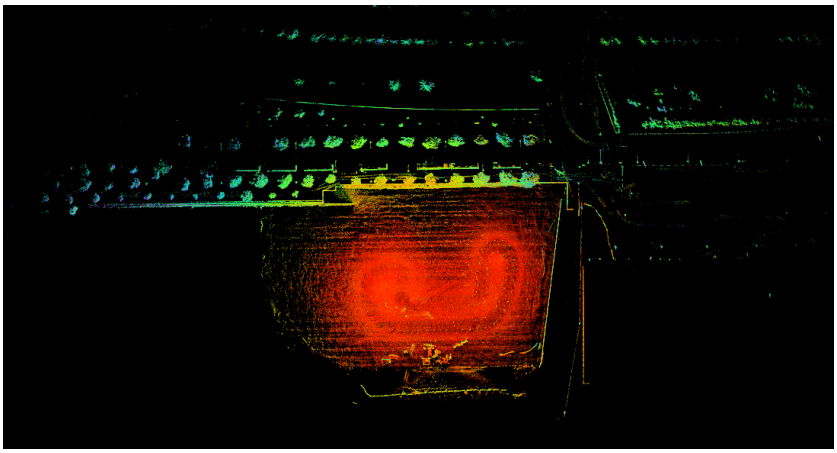
(a) ALOAM - Viewpoint B



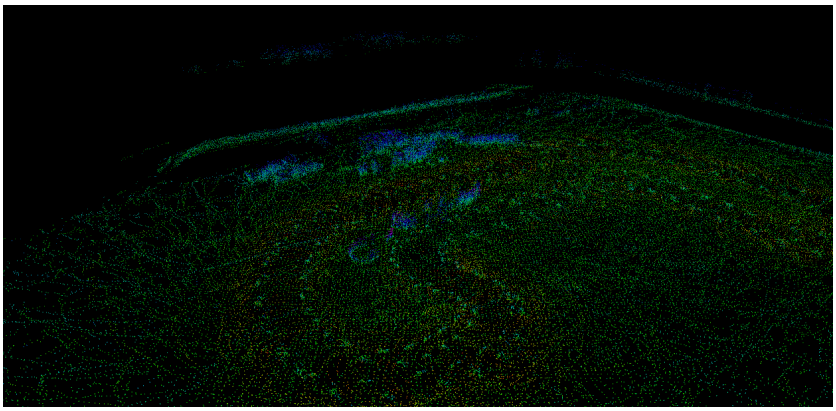
(b) Fast-LIO - Viewpoint B



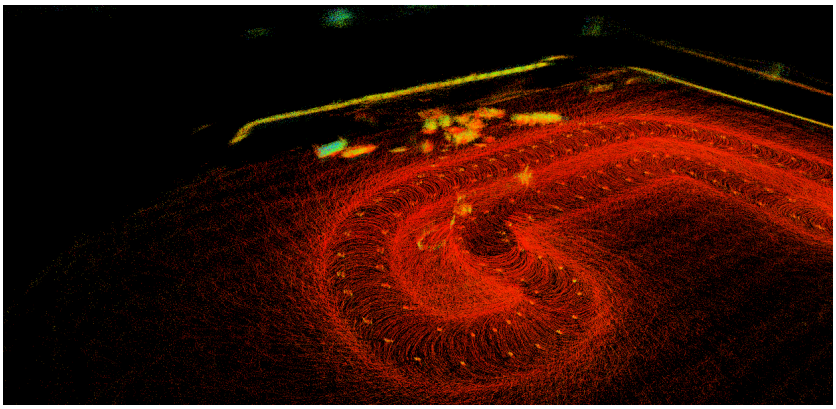
(c) LIO-SAM - Viewpoint B (with $\times 0.5$ player speed)



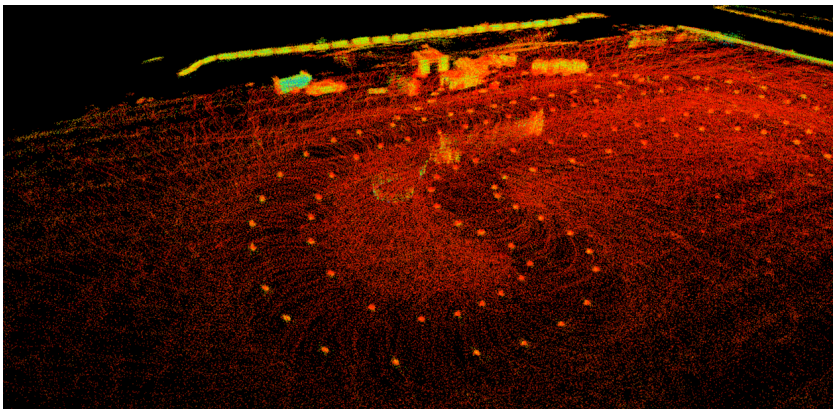
(d) LIMO-Velo - Viewpoint B



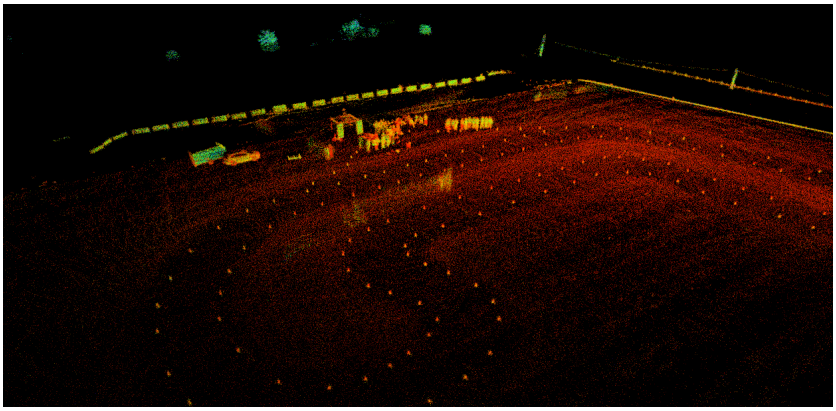
(a) ALOAM - Viewpoint C



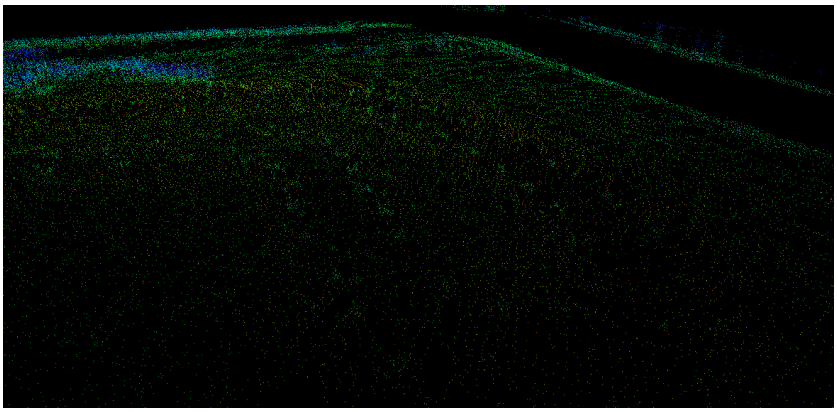
(b) Fast-LIO - Viewpoint C



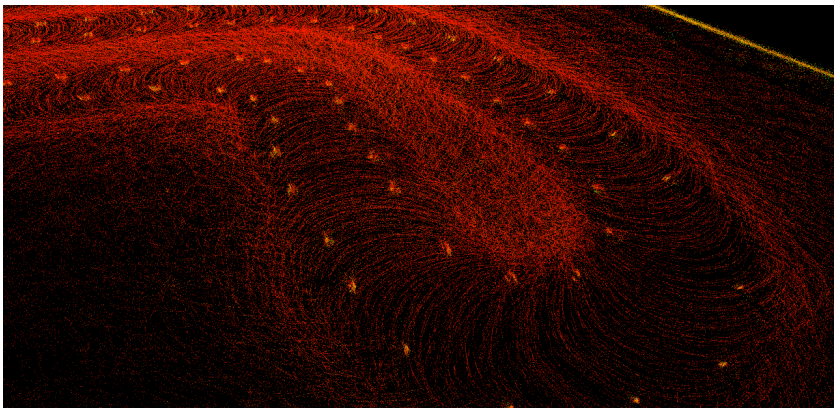
(c) LIO-SAM - Viewpoint C (with $\times 0.5$ player speed)



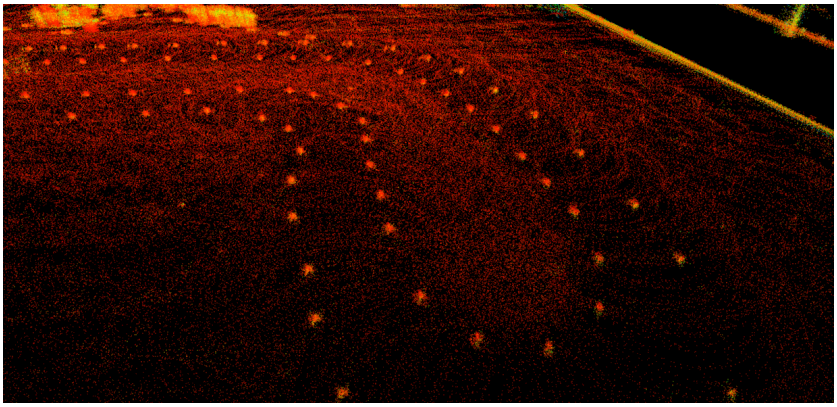
(d) LIMO-Velo - Viewpoint C



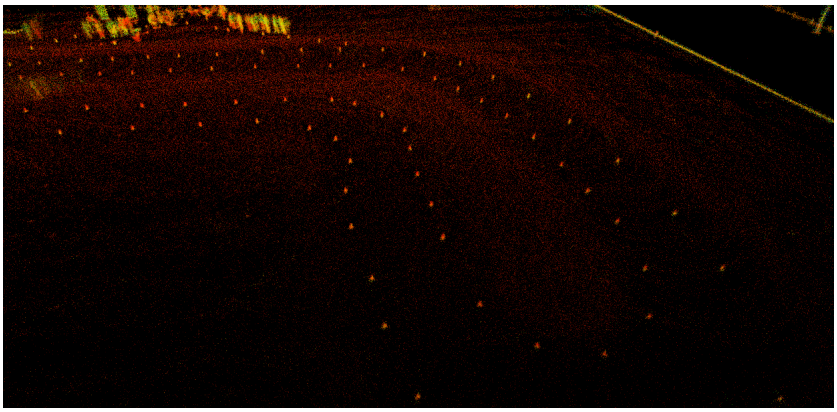
(a) ALOAM - Viewpoint D



(b) Fast-LIO - Viewpoint D



(c) LIO-SAM - Viewpoint D (with $\times 0.5$ 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.

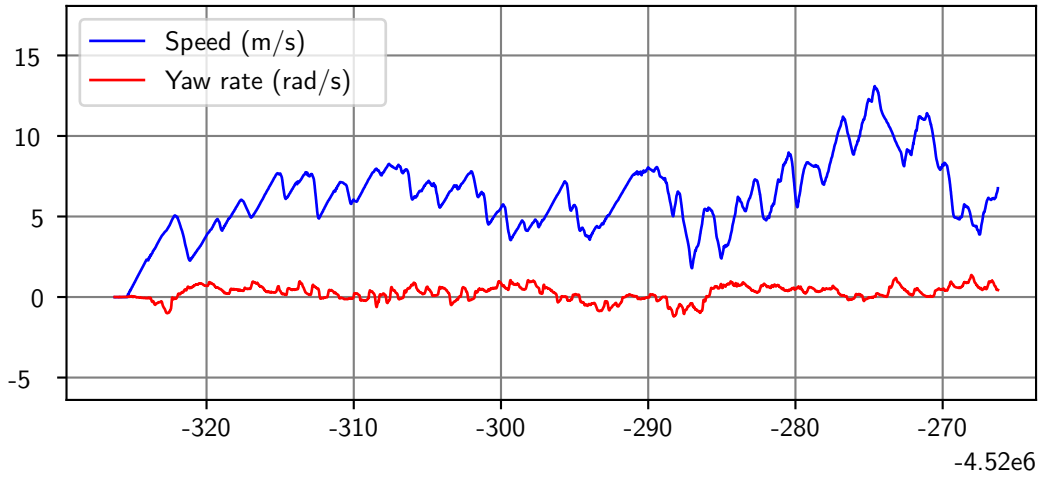


Figure 6.5: Peak speed: 13m/s. Peak turn speed: 80deg/s.

- **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.

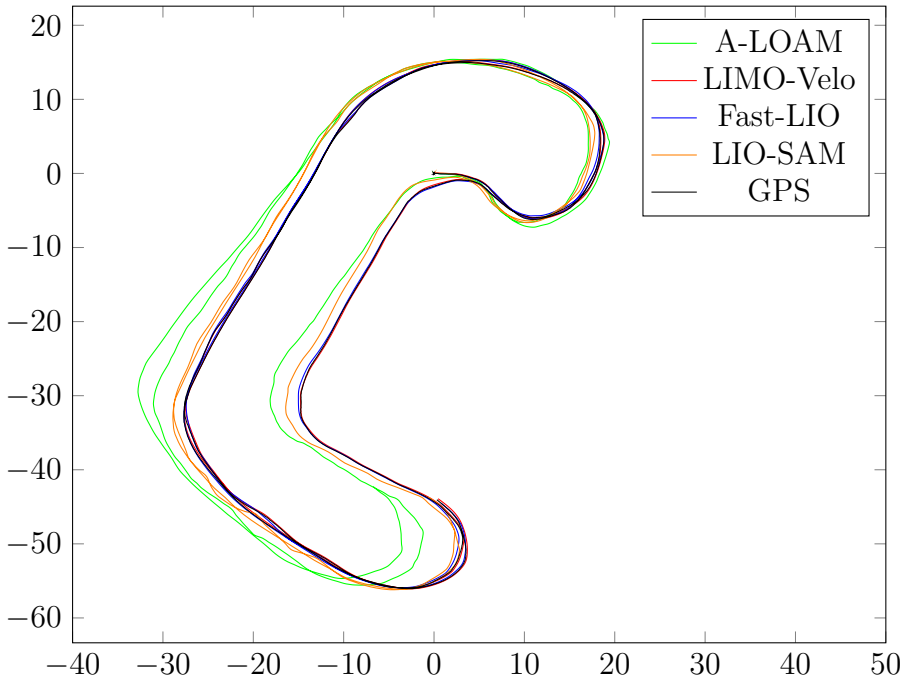


Figure 6.6: All algorithms aggregated.

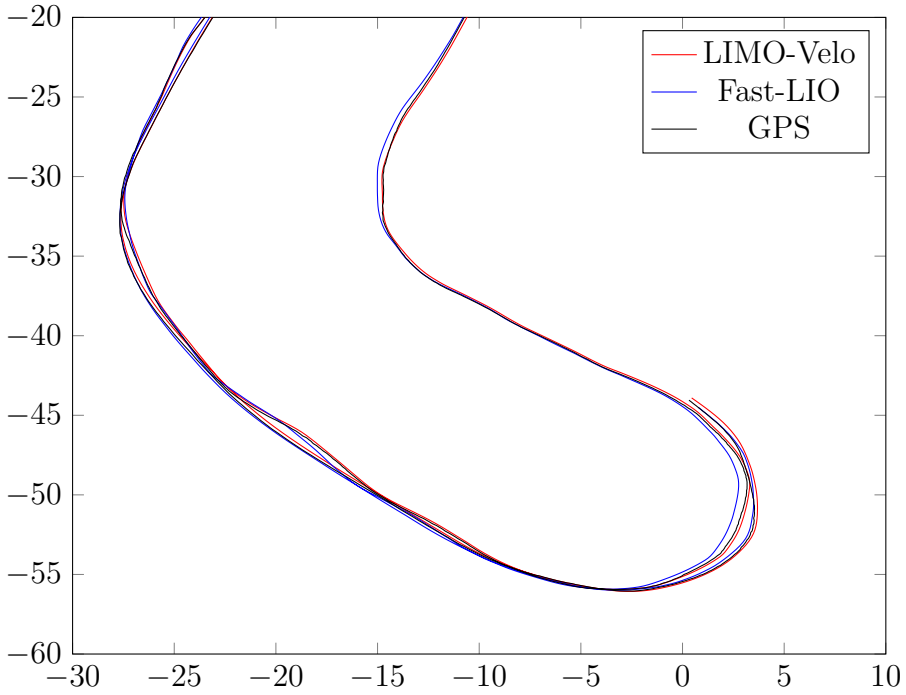


Figure 6.7: Two best performers compared - Down part.

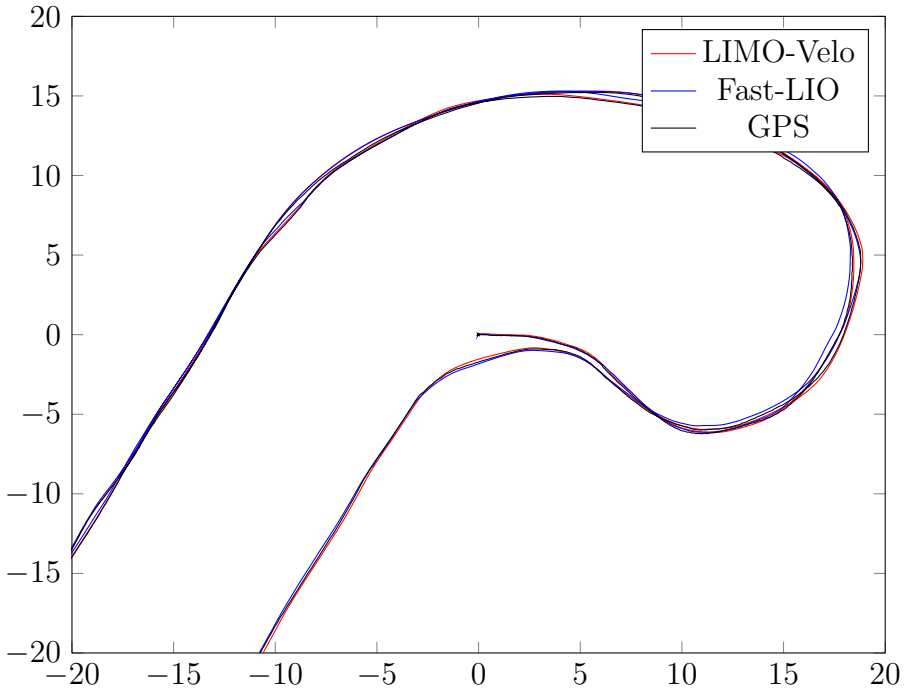


Figure 6.8: Two best performers compared - Up part.

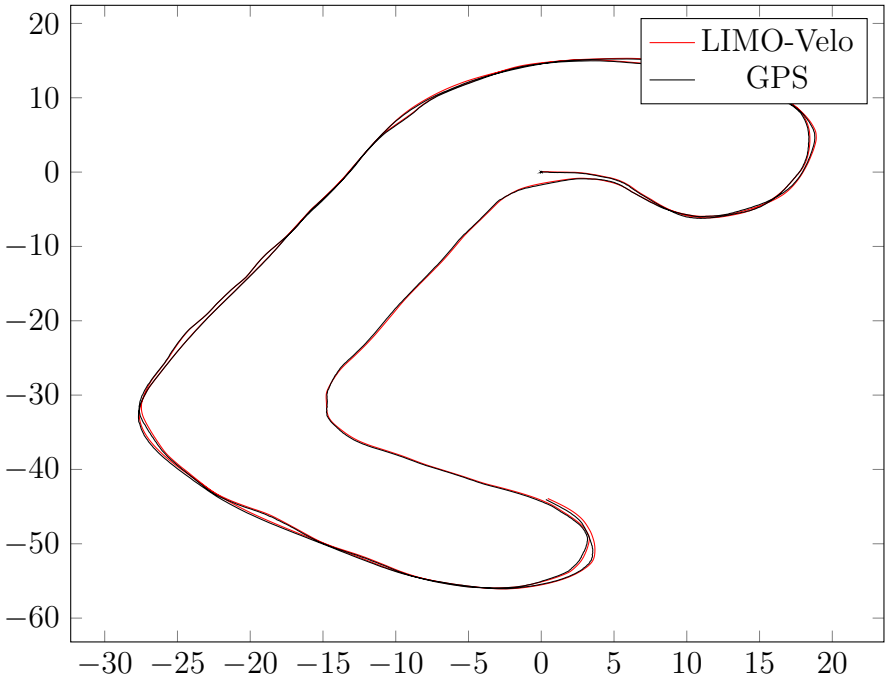


Figure 6.9: LIMO-Velo odometry

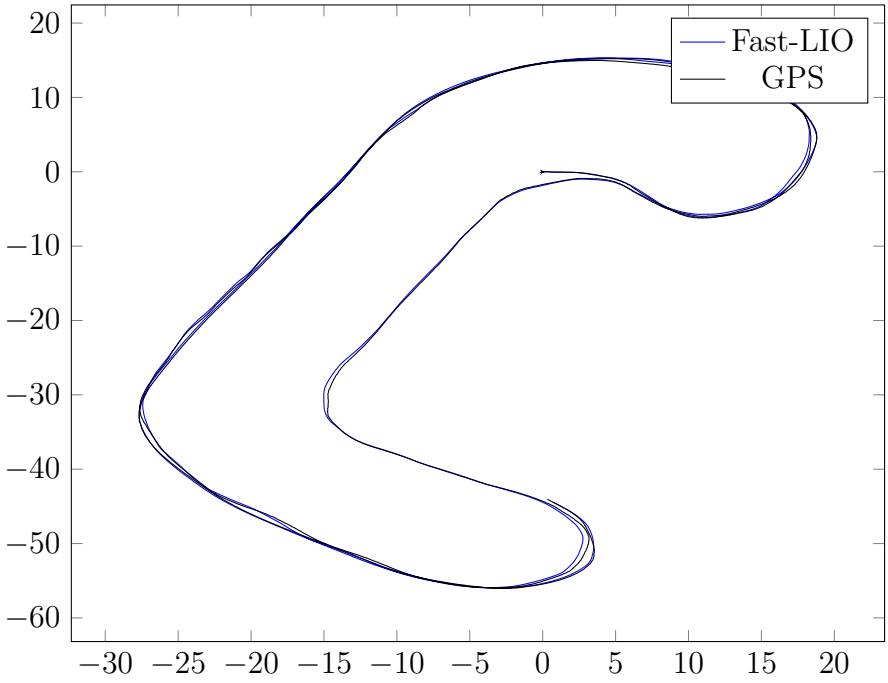


Figure 6.10: Fast-LIO odometry

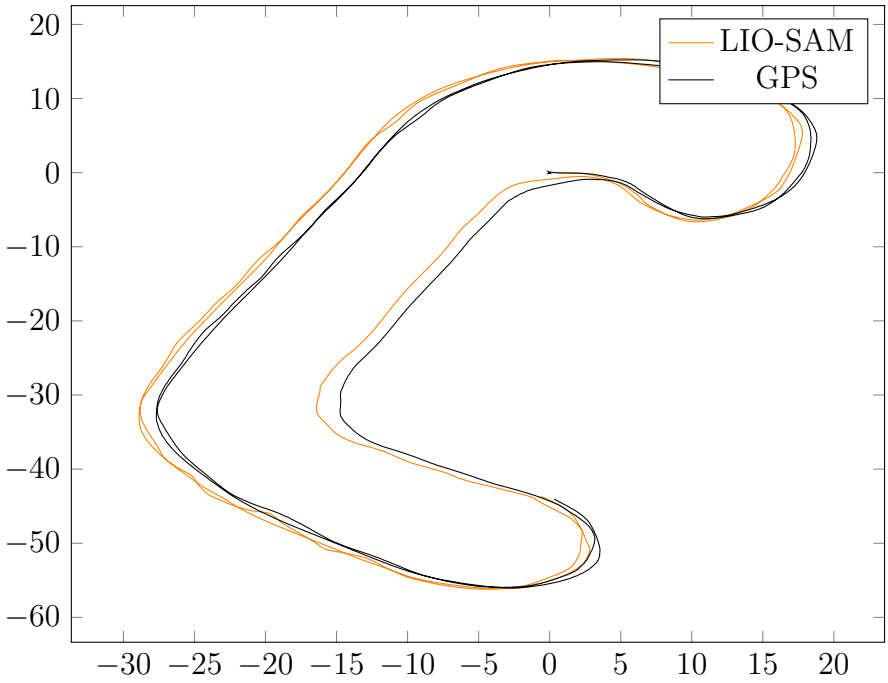


Figure 6.11: LIO-SAM odometry

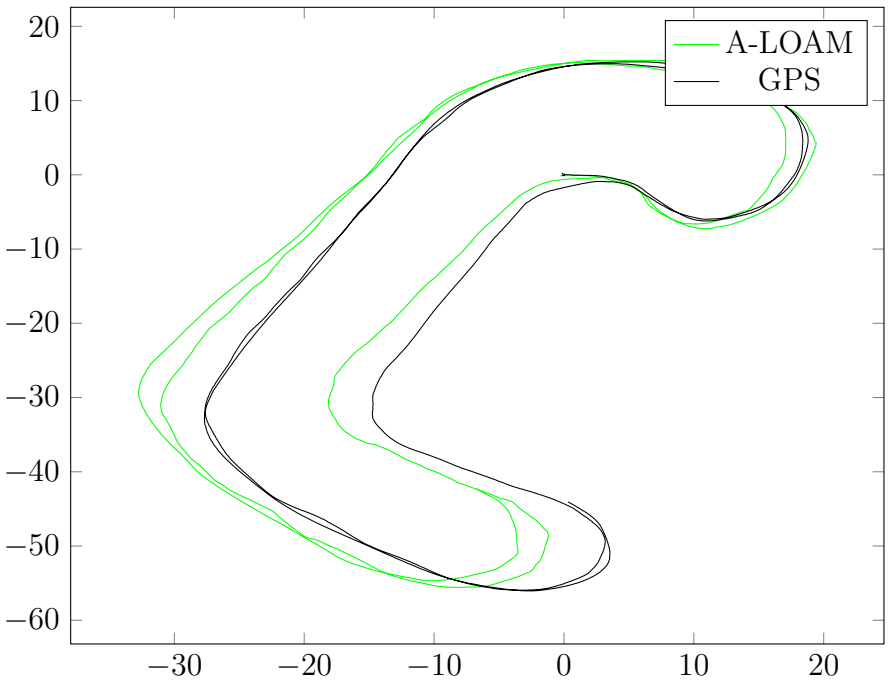


Figure 6.12: A-LOAM odometry

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.