Camera-Lidar Synchronization in DSP-SLAM

2A-Internship Report

Author: Helou Fadi
Supervisors: Dr. Asmar & Dr. Kourani
Presented for: Dr. Luc Jaulin

August 29, 2025

Contents

Abstract

In this report I explain the work done during my internship on synchronizing camera and LiDAR data in the deep shape priors Simultaneous Localization and Mapping (DSP-SLAM) pipeline. The LiDAR runs at 10 Hz (one sweep every 100 ms) while the camera runs at 25 Hz (one frame every 40 ms), so measurements do not align temporally and must be corrected using ego-motion compensation, interpolation, or other synchronisation strategies, in order to be fed into a deep neural network that reconstruct an accurate dense visualization of the detected object. The report covers: problem statement, motion compensation approach and theory, implementation notes (including a `KittySequence.py` based pipeline), implementation results and metrics with a conclusion.

Introduction

In Ficosa Car system (Ficosa International S.A. is a Spanish multinational company, founded in 1949, headquartered in Barcelona, specializing in the R&D and production of automotive systems and components), the LiDAR scanner produces full 3D point clouds at 10 Hz (one sweep every 100 ms), while the camera captures images at 25 Hz (one frame every 40 ms).

This temporal mismatch means that for each LiDAR sweep there are typically two intermediate camera frames (and occasionally three), so a given LiDAR point cloud and a given camera image do not represent the scene at the same instant.

Because the vehicle is moving, the world changes between the LiDAR timestamp (t_0) and the camera timestamp (t_1).

If we naïvely project the LiDAR points from t_0 onto the image at t_1, static objects will appear misaligned (shifted) in the image and dynamic objects (e.g. other vehicles, pedestrians) will be projected onto "ghost" locations.

.1 Problem Statement

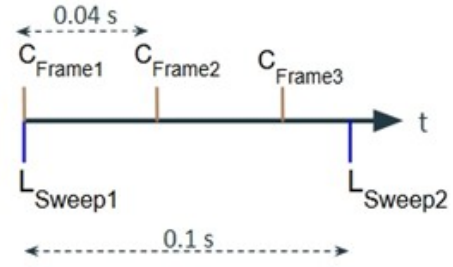


Figure 1: Assuming camera frame 1 and LiDAR sweep 1 are captured simultaneously for simplicity

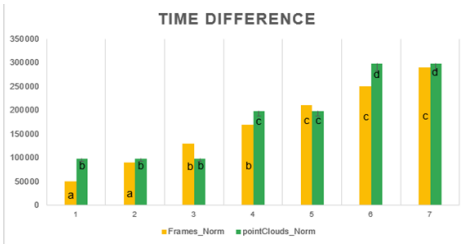


Figure 2: This graph is obtained using real Ficosa data

The graph above (on the right) shows the Timestamps of the camera (in yellow) and the Lidar (in green) in function of camera frame index.

We can easily observe the mismatches and associations (shown in letters on the figure) between each camera frame and the corresponding LiDAR points that are projected onto the frame.

In a SLAM pipeline, especially one that builds a 3D reconstruction (DSP-SLAM) Github Repo here, these misaligned measurements introduce two main problems:

- **Distorted Point Clouds**

  As the LiDAR scans over its 100 ms sweep, the vehicle continues to move. Points collected early in the sweep correspond to a slightly different pose than those collected at the end. Without correcting for this intra-sweep motion, the resulting 3D point cloud is "skewed" or "twisted" degrading map quality.

- **Inaccurate Sensor Fusion**

  SLAM algorithms (including DSP-SLAM with Deep Shape Priors) rely on tightly coupled, synchronous observations from different sensors. If camera images and LiDAR clouds are desynchronized, the algorithm cannot correctly associate 2D image features with 3D points. This breaks geometric consistency, leading to:

- Poor pose estimates (localization drift)
- Incomplete or warped 3D reconstructions
- Object priors (in DSP-SLAM) being projected onto incorrect regions of the point cloud

Addressing this problem ensures that each 3D point and its corresponding image pixel truly represent the same moment and viewpoint. Which yields enhanced accuracy and robustness of 3D reconstruction.

A Investigation Research

A.1 Overview

A practical solution is interpolation or motion compensation: we virtually "warp" or synthesize LiDAR data at the intermediate camera timestamps. In other words, we estimate where the LiDAR points would have been at the image time.

A.2 Interpolation

One approach is to interpolate between LiDAR frames: given two LiDAR sweeps at t0 and t1 and camera frames in between, one can interpolate new point clouds that align with each camera image.

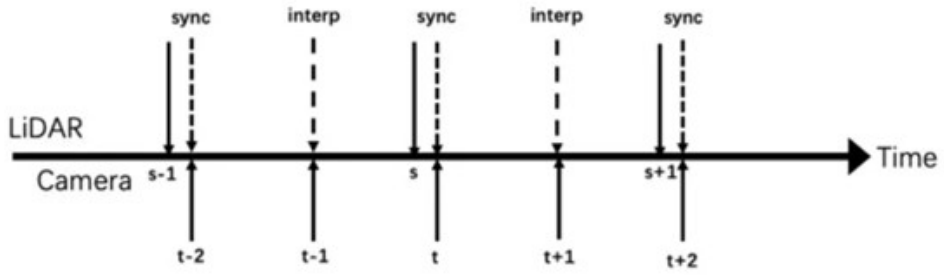


Figure 1. time sync illustration between LiDAR and camera

This effectively doubles the LiDAR frame rate to match the camera, filling in the gaps. Concretely, if the camera is 25 Hz and LiDAR 10 Hz, you might generate "virtual" LiDAR scans at 40 ms intervals from actual scans at 100 ms. These interpolated scans can then be projected onto each image as if captured simultaneously.

The implication is that unsynchronized sensors require careful timing. A naïve overlay of 10 Hz LiDAR on 25 Hz video will misplace objects. Instead, we must either drop camera frames, slow down the video to 10 Hz (use Ros2 synchronization function), or preferably adjust the LiDAR points via motion compensation or interpolation as discussed above so that every camera frame has a corresponding "aligned" point cloud. This is critical in autonomous driving, e.g. a fast-moving car could travel several meters in 40 ms, so uncorrected data fusion would yield significant errors.

A.3 Motion Compensation

Car displacement is the change in the vehicle's position and orientation between sensor timestamps. To synchronize LiDAR with a later camera frame, we need the car's odometry (position + orientation over time). This comes from sensors like IMU/GPS. Using this information, we can motion-correct the LiDAR points.

Intuitively, imagine a car moving forward. A LiDAR point measured on a parked car ahead will be recorded at a certain (x, y) in the LiDAR frame. If the vehicle moves forward by 1 m before the camera snapshot, then from the camera's later perspective, that parked car is closer. To align them, we would effectively subtract the car's forward motion (or equivalently, translate the point cloud forward). In 2D terms: if the vehicle moved by $\Delta x = 1$ in its forward axis, we add Δx to the x-coordinates of the LiDAR points (in vehicle frame) before projecting.

More formally, if $T_{\text{car}}(t_0)$ and $T_{\text{car}}(t_1)$ are the vehicle poses in some global frame, the relative transform $T_{\text{motion}} = T_{\text{car}}(t_0) - T_{\text{car}}(t_1)$ gives the car's movement from t_0 to t_1. We then apply T_{motion} to all LiDAR points (in homogeneous coordinates) to move them. This is exactly the motion-compensation step. In practice, we often decompose T_{motion} into translation and rotation and apply them.

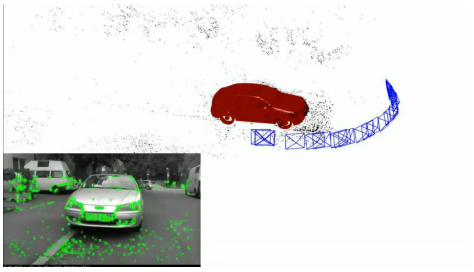
B Implementation Methodology

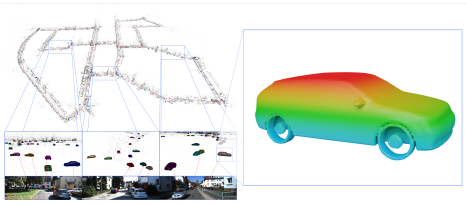
We chose the motion compensation method (known as ego-motion) because it seemed more convenient for us, as we already had the odometry data.

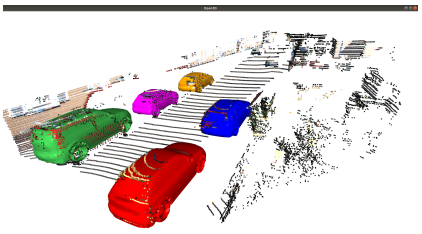
Initially, I downloaded the open-source DSP-SLAM code [1] (see References), which serves as the foundation for my internship project. This codebase was then modified and extended with specific implementations to meet the requirements of the Ficosa project. Then reviewed its overall structure to understand how the various components interact and focused on the specific section where I will apply the ego-motion tailored to our requirements.

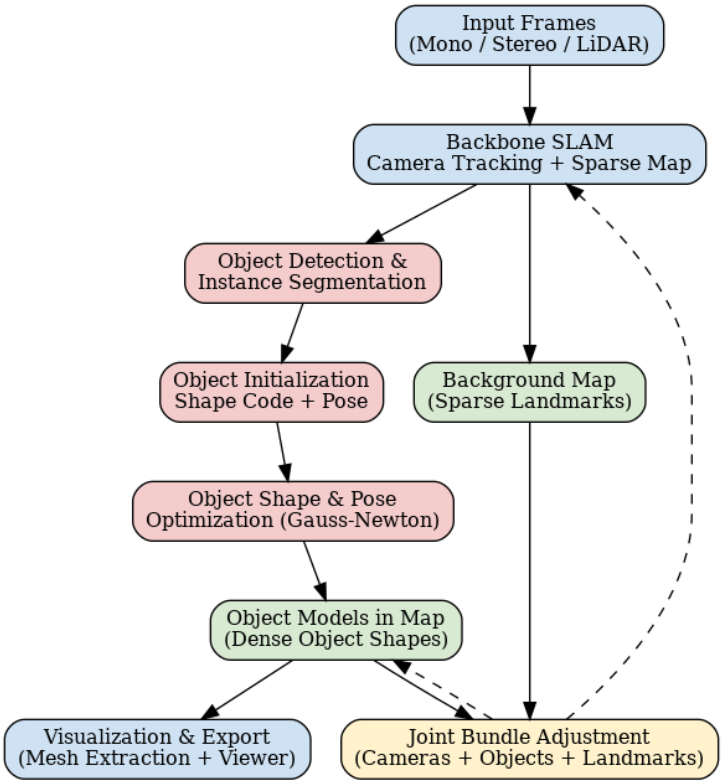
B.1 DSP-SLAM

DSP-SLAM is an object-oriented SLAM system that builds a rich and accurate joint map of dense 3D models for foreground objects, and sparse landmark points to represent the background. DSP-SLAM takes as input the 3D point cloud reconstructed by a feature-based SLAM system and equips it with the ability to enhance its sparse map with dense reconstructions of detected objects. Objects are detected via semantic instance segmentation, and their shape and pose are estimated using category-specific deep shape embeddings as priors, via a novel second-order optimization. Our object-aware bundle adjustment builds a pose-graph to jointly optimize camera poses, object locations, and feature points. DSP-SLAM can operate at 10 frames per second on three different input modalities: monocular, stereo, or stereo+LiDAR.

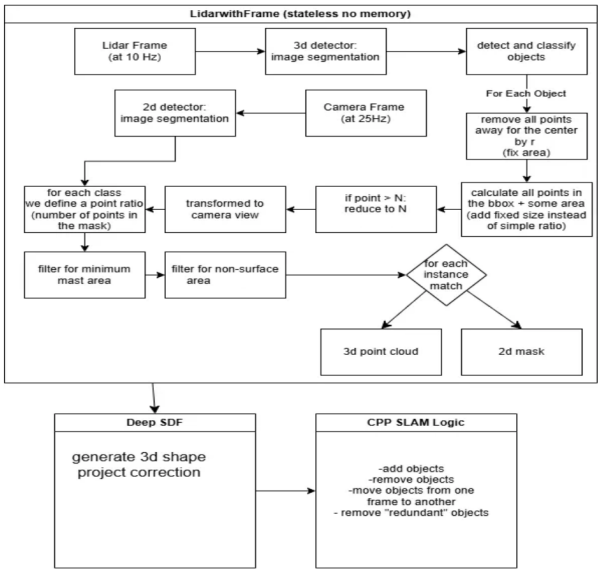








How DSP works?



B.2 Preliminary Definitions

B.2.1 Camera Intrinsics & Extrinsic

Intrinsics: refer to the internal parameters of a camera that describe how it projects 3D points from the world onto the 2D image plane. These parameters include the focal length, optical center (principal point), and potential lens distortion coefficients. Intrinsic parameters are essential for understanding the geometric properties of the camera and are usually represented in the camera calibration matrix.

Extrinsics: define the position and orientation of the camera in the world coordinate system. They describe the transformation (rotation and translation) between the world coordinates and the camera coordinates. Extrinsic parameters are used to relate the camera's perspective to the surrounding environment

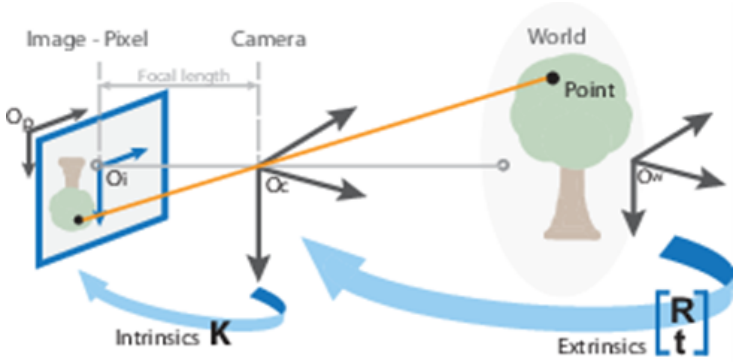


Figure 3: 3D projection into camera frame

B.2.2 LiDAR Intrinsics & Extrinsic

Intrinsics: for a LiDAR sensor refers to its internal characteristics, such as the number of laser beams (channels), vertical and horizontal field of view, angular resolution, and firing sequence. These parameters define how the LiDAR scans its environment and are essential for interpreting the raw point cloud data accurately.

Extrinsics: describe the position and orientation of the LiDAR sensor relative to another coordinate system, such as the robot or world frame. They include the rotation and translation parameters that transform LiDAR points into another frame of reference.

B.2.3 3D Transform (homogeneous form)

In 3D computer vision and robotics, transformations such as rotation, translation, and scaling are fundamental operations for manipulating points and objects in space. These transformations can be represented in different mathematical forms, among which homogeneous coordinates offer a unified and efficient approach.

A point in 3D space using Cartesian coordinates is represented as a vector $p = [x, y, z]^T$. However, in homogeneous coordinates, this point is extended with an additional coordinate, typically written as $p_h = [x, y, z, 1]^T$.

This extension allows both rotation and translation to be expressed as a single 4×4 transformation matrix, enabling the composition of multiple transformations through simple matrix multiplication.

A typical 3D homogeneous transformation matrix is:

$$T = \begin{bmatrix} R_{3\times3} & t_{3\times1} \\ 0_{1\times3} & 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure 4: Transformation matrix

Where:

- R: is a 3×3 rotation matrix,
- t: is a 3×1 translation vector,
- The bottom row [0 0 0 1] ensures compatibility with homogeneous vectors.

Why Use Homogeneous Coordinates Instead of Cartesian Form?
- Unified Representation
- Easier Composition
- Convenient for Computer Graphics and Robotics

B.3 KittySequence.py

Inside the DSP SLAM Library, we can find the kitty sequence code that will be my main target and where I will apply the changes.

The main Pipeline for this code is:

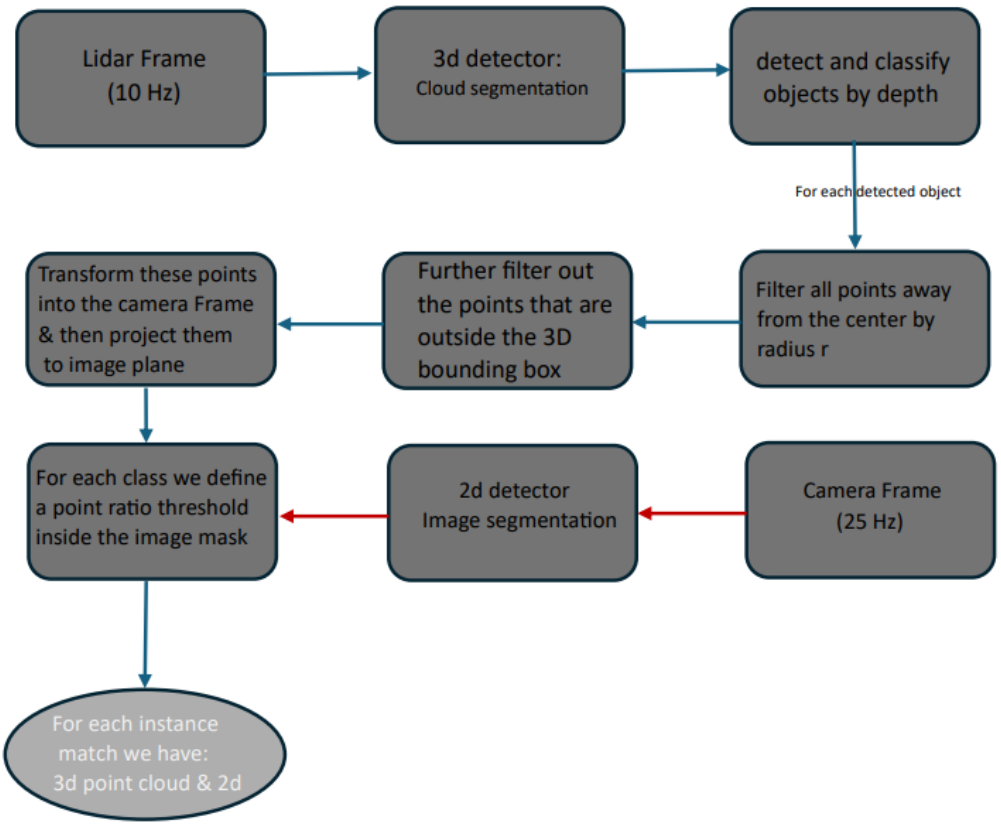


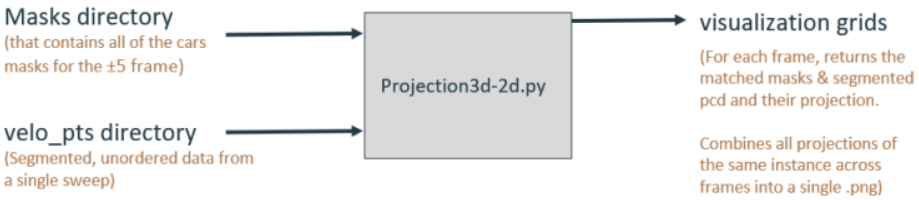
Figure 5: Kittysequence.py

These results will be then fed to the Deep SDF code for generating 3D shape reconstruction.

B.4 Feasibility study

At first, to assess the mismatch, I selected a single LiDAR sweep at a known timestamp and examined multiple camera frames captured within a ± 5 frame window around that timestamp and projected the point clouds onto these consecutive frames to quantify the problem and compare the results before and after the ego motion compensation.

B.4.1 Code Overview



B.4.2 Code Architecture

Input data Processing:

1- Takes 3D point cloud data (.ply files)

2- Takes 2D mask images (.png files)

3- Validates mask images using `is_mask_image()` function to ensure they are binary masks.

4- Defines camera intrinsics matrix (K) for projection

5- Defines extrinsic transformation matrix (T_cam_velo) for Lidar-camera

B.4.3 Main Processing Pipeline

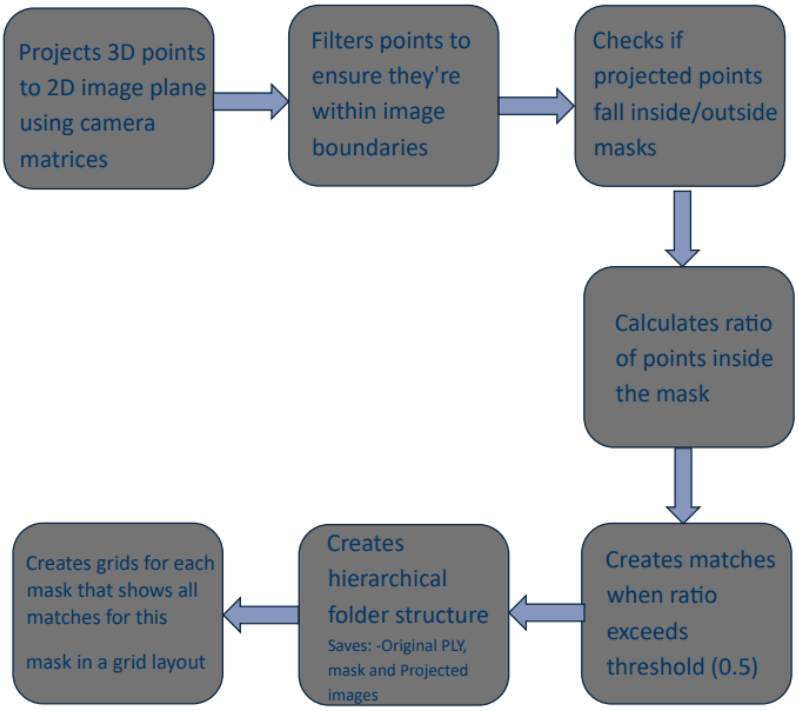
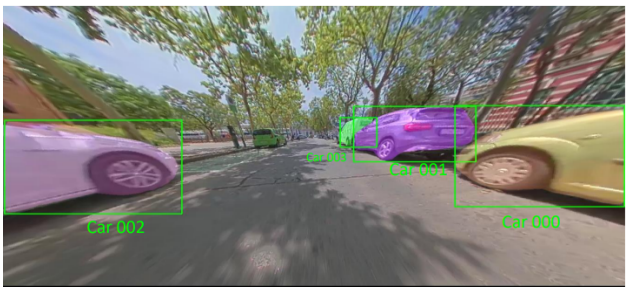


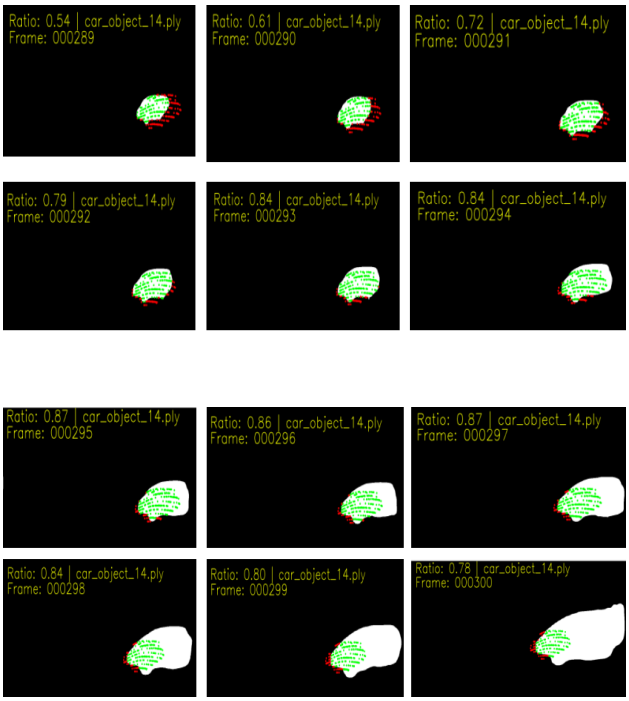
Figure 6: Projection LiDAR points into camera frame

B.4.4 Results

Tested the projection code on various 2D mask datasets, including fisheye, undistorted original, and undistorted masked versions. Among these, the undistorted masked data provided the best alignment results, and it is the one showcased below.



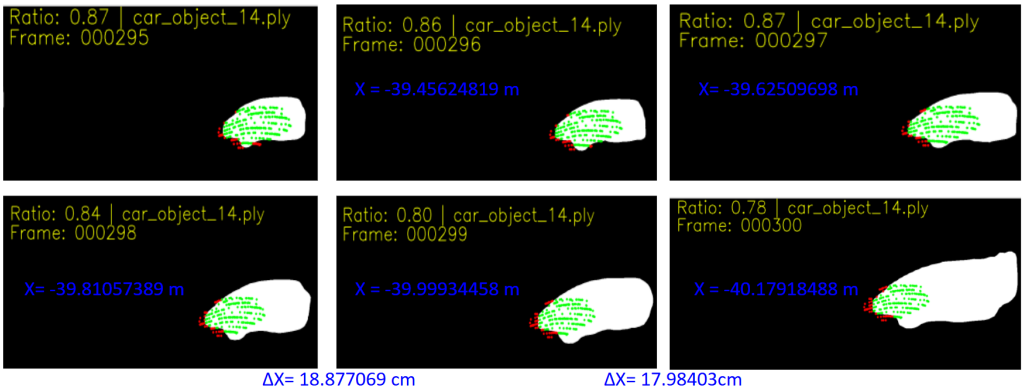
Results: For car instance 001



Error Visualization:

Note that : In these sample taken, the car is moving forward. Only the X position with respect to local frame is shown

Results: For car Instance *001*

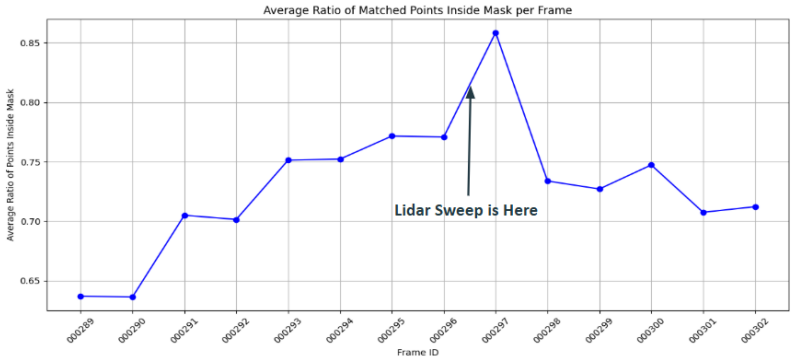


B.4.5 Metrics

Frame ID	frames	Velo_files	Difference Nanoseconds	Difference Microseconds
289	1688725890000611876	1688725890000896954	285078	285.078
290	1688725890000651917	1688725890000896954	245037	245.037
291	1688725890000691826	1688725890000896954	205128	205.128
292	1688725890000731933	1688725890000896954	165021	165.021
293	1688725890000771847	1688725890000896954	125107	125.107
294	1688725890000811869	1688725890000896954	85085	85.085
295	1688725890000851882	1688725890000896954	45072	45.072
296	1688725890000891873	1688725890000896954	5081	5.081
297	1688725890000931934	1688725890000896954	-34980	-34.98
298	1688725890000971942	1688725890000896954	-74988	-74.988
299	1688725891000011917	1688725890000896954	-999114963	-999114.963
300	1688725891000051873	1688725890000896954	-999154919	-999154.919
301	1688725891000091935	1688725890000896954	-999194981	-999194.981
302	1688725891000131932	1688725890000896954	-999234978	-999234.978

Timestamps since epoch for camera and Lidar

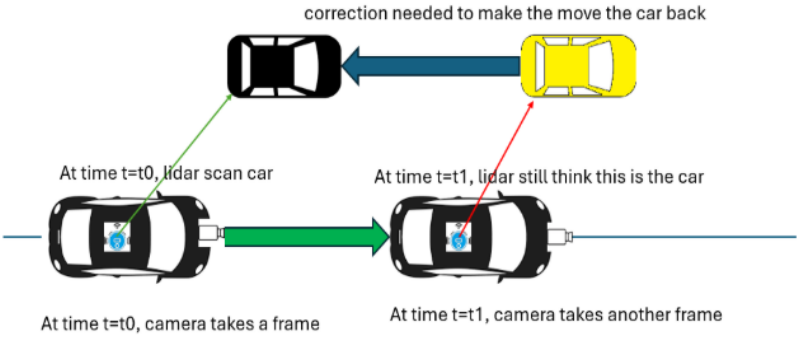
Here's the selected Frame ID's with their corresponding Timestamps. We observe that the LiDAR and Camera data are most closely aligned at frame ID's 295, 296, and 297.



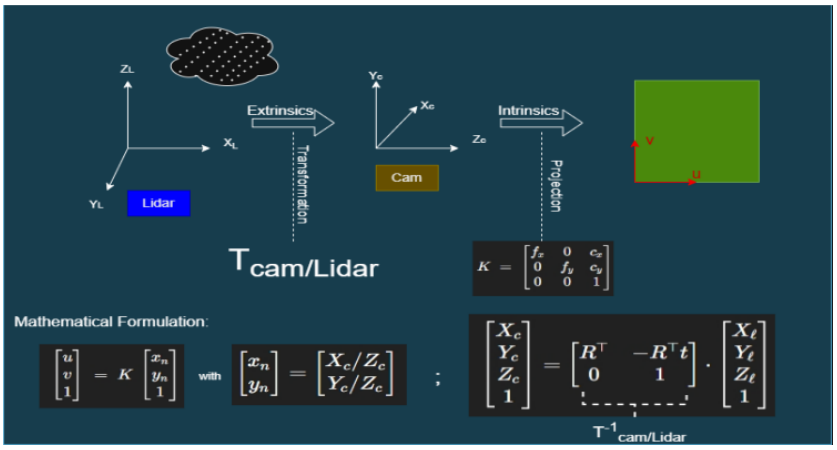
This concludes that the ratio of matched points within the mask across different frames follows a bell-shaped normal distribution, as expected. After motion compensation, I expect the variance to decrease and the ratios to converge toward the mean, which corresponds to frame 297.

So, motion compensation will be applied on these sample values first and results will be analyzed, if they were satisfying, the code will be generalized and adapted for the DSP SLAM to check whether the 3D reconstruction is more efficient.

C Ego-Motion Implementation



C.1 Keyframe Processing



In a dynamic vehicle context, applying the static transformation $T_{\text{cam/Lidar}}$ is insufficient as both the LiDAR and camera are moving. Therefore, **ego-motion compensation** is critically applied *before* the coordinate transformation.

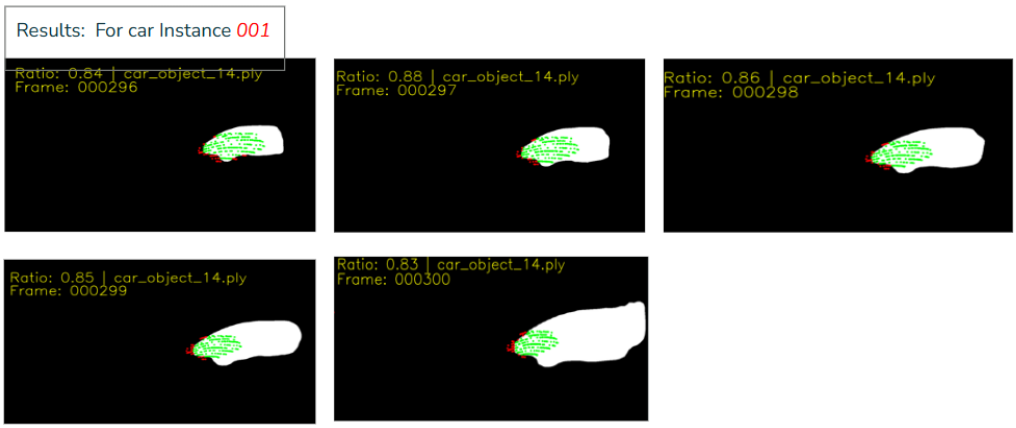
For a LiDAR point cloud captured at time t, we first compensate for the vehicle's own motion between a reference time (e.g., the keyframe time t_0) and t. This is done by applying the vehicle's ego-motion transform, $T_{\text{ego}}^{t_0 \to t}$, to project all LiDAR points into a common, stationary reference frame (typically the vehicle frame at t_0).

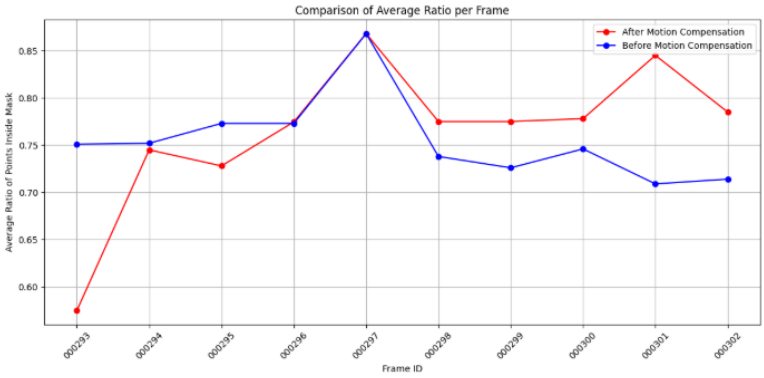
$$P_{t_0} = T_{\text{ego}}^{t_0 \to t} \cdot P_t \tag{1}$$

Where P_t is a point from the LiDAR sweep at time t, and P_{t_0} is its motion-compensated position in the t_0 frame.

To ensure computational efficiency for tasks like 3D object reconstruction, this pipeline is **not applied to every camera frame**. Instead, it is triggered only for selected **keyframes**. Keyframes are typically frames with significant new visual information from which objects (e.g., cars, pedestrians) have been successfully detected and extracted. The motion-compensated LiDAR points are then transformed into the camera frame of these keyframes using $T_{\text{cam/Lidar}}$ to provide dense, accurate 3D data for reconstruction.

C.1.1 Results after compensation

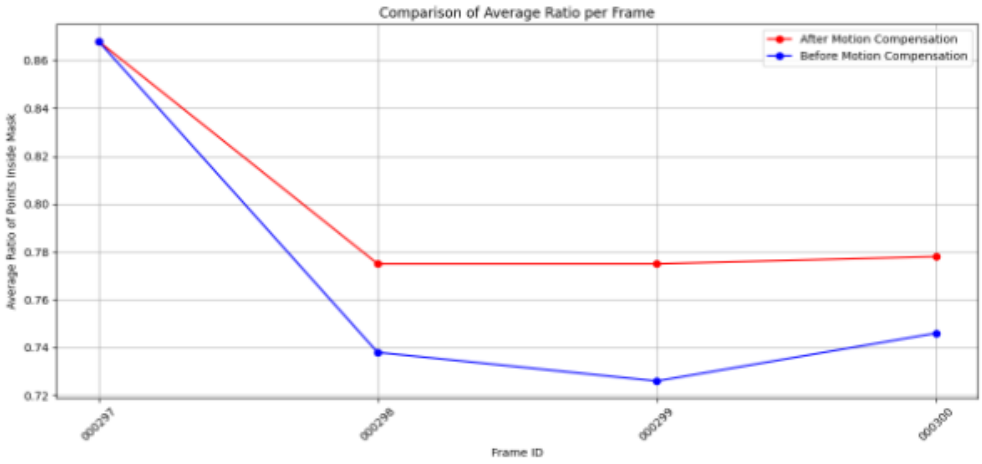




As shown in the results, the motion compensation successfully improves the average ratio of 3D points inside the object masks across successive frames.

However, it is important to note that not all these frames are relevant for our final evaluation. Due to the differing frequencies of the LiDAR (10 Hz) and camera (25 Hz) sensors, there are typically only 2 or 3 camera frames captured between two consecutive LiDAR sweeps.

Therefore, our analysis focuses specifically on the most informative frames namely, frames 297, 298, and 299. We know that a LiDAR sweep occurs between frames 296 and 297 so these selected frames immediately follow a new LiDAR scan. Consequently, the performance metrics will be calculated and concluded based on this critical subset of frames.



The improvement is about 5%

D Integration with the main DSP algorithm

D.1 Overview

This module bridges the C++ SLAM keyframe logic with a Python-based LiDAR processing pipeline (`ficosa_sequence.py`) to obtain object detections and motion-compensated LiDAR points synchronized with camera keyframes. For each camera KeyFrame, the system identifies the corresponding LiDAR scan via timestamp matching, computes the camera pose at the LiDAR timestamp through interpolation, and exchanges data with the Python module to retrieve object detections that are subsequently integrated into the SLAM map structures.

D.2 Motivation

The fusion of camera and LiDAR data necessitates temporal synchronization between sensors operating at different frequencies (e.g., camera at 25 Hz, LiDAR at 10 Hz). To ensure consistent fusion, the system must identify the LiDAR scan that best matches each camera keyframe temporally and compute the corresponding camera pose at the LiDAR timestamp for accurate ego-motion compensation.

D.3 Pseudo codes

```
void Tracking::GetObjectDetectionsLiDAR(KeyFrame *pKF) {

    // Get the pose as a 4x4 transformation matrix
    Eigen::Matrix4f T_keyframe_camera = pKF->GetPose().
       matrix();
    double timestamp_keyframe = pKF->mTimeStamp;
    Eigen::Matrix4f T_current_camera = mCurrentFrame.
       GetPose().matrix();
    double timestamp_current = mCurrentFrame.mTimeStamp;
    Eigen::Matrix4f T_last_camera = mLastFrame.GetPose()
       .matrix();
    double timestamp_last = mLastFrame.mTimeStamp;
    Eigen::Matrix4f T_lastlast_camera = mLastLastFrame.
       GetPose().matrix();  // Get the last-last frame
       pose
    double timestamp_lastlast = mLastLastFrame.
       mTimeStamp;
```

```
12
13      // Get timestamps for each pose
14
15      // Get LiDAR timestamp by matching timestamps from
            CSV
16      auto [lidar_timestamp, lidar_id] = MatchTimestamps(
            pKF->mnFrameId);
17
18      // Debug output
19      if (lidar_id != -1) {
20          std::cout <<  Using LiDAR ID:   << lidar_id <<
                 with timestamp:   << lidar_timestamp << std
                ::endl;
21      } else {
22          std::cout <<  Warning: Using default LiDAR
                timestamp (0.0) for frame   << pKF->mnFrameId
                 << std::endl;
23      }
24
25      // Interpolate/extrapolate camera pose for LiDAR
            timestamp
26      Sophus::SE3f interpolated_camera_pose =
            InterpolatePoseForLiDAR(
27          lidar_timestamp,
28          mCurrentFrame.GetPose(), timestamp_current,
29          mLastFrame.GetPose(), timestamp_last,
30          mLastLastFrame.GetPose(), timestamp_lastlast
31      );
32
33      // Convert to 4x4 matrix
34      Eigen::Matrix4f T_lidar_pose =
            interpolated_camera_pose.matrix();
35
36      PyThreadStateLock PyThreadLock;
37
38      py::list detections = mpSystem->pySequence.attr(
            get_frame_by_id )(pKF->mnFrameId,lidar_id,
            T_keyframe_camera, T_lidar_pose);
39      for (auto det : detections) {
40          auto pts = det.attr( surface_points ).cast<Eigen
                ::MatrixXf>();
41          auto Sim3Tco = det.attr( T_cam_obj ).cast<Eigen
                ::Matrix4f>();
```

```
42          auto rays = det.attr( rays );
43          // int class = det.attr( class );
44          Eigen::MatrixXf rays_mat;
45          Eigen::VectorXf depth;
46
47          if (rays.is_none()) {
48              // std::cout <<  No 2D masks associated!  <<
                    std::endl;
49              rays_mat = Eigen::Matrix<float, 0, 0>::Zero
                    ();
50              depth = Eigen::VectorXf(0);
51          } else {
52              rays_mat = rays.cast<Eigen::MatrixXf>();
53              depth = det.attr( depth ).cast<Eigen::
                    VectorXf>();
54          }
55          // Create C++ detection instance
56          auto o = new ObjectDetection(Sim3Tco, pts,
                rays_mat, depth); // ,class);
57          pKF->mvpDetectedObjects.push_back(o);
58      }
59      pKF->nObj = pKF->mvpDetectedObjects.size();
60      pKF->mvpMapObjects = vector<MapObject *>(pKF->nObj,
            static_cast<MapObject *>(NULL));
61  }
```

Listing 1: C++ Implementation: GetObjectDetectionsLiDAR Function

```
1   std::pair<double, int> Tracking::MatchTimestamps(int
        camera_frame_id)
2   {
3       // TODO: Replace with actual CSV file path
4       std::string csv_file_path =  path_to_your_timestamps
            .csv ;
5
6       std::ifstream file(csv_file_path);
7       if (!file.is_open()) {
8           std::cerr <<  Warning: Could not open timestamps
                CSV file:   << csv_file_path << std::endl;
9           return {0.0, -1}; // Return default values
10      }
11
```

```
    std::string line;
    double camera_timestamp = -1.0;
    std::vector<std::pair<double, int>> lidar_data; // {timestamp, id}

    // Skip header line if it exists
    std::getline(file, line);

    // Read CSV file
    while (std::getline(file, line)) {
        std::stringstream ss(line);
        std::string token;

        // Parse CSV columns: Camera_ID, Camera_Timestamp, LiDAR_ID, LiDAR_Timestamp
        std::vector<std::string> tokens;
        while (std::getline(ss, token, ',')) {
            tokens.push_back(token);
        }

        if (tokens.size() >= 4) {
            int csv_camera_id = std::stoi(tokens[0]);
            double csv_camera_timestamp = std::stod(tokens[1]);
            int csv_lidar_id = std::stoi(tokens[2]);
            double csv_lidar_timestamp = std::stod(tokens[3]);

            // If we found our camera frame ID, store its timestamp
            if (csv_camera_id == camera_frame_id) {
                camera_timestamp = csv_camera_timestamp;
            }

            // Store all LiDAR data for later search
            lidar_data.push_back({csv_lidar_timestamp, csv_lidar_id});
        }
    }

    file.close();

    // Check if camera frame ID was found
```

```
49      if (camera_timestamp == -1.0) {
50          std::cerr <<  Warning: Camera frame ID   <<
               camera_frame_id <<   not found in CSV file!
               << std::endl;
51          return {0.0, -1}; // Return default values
52      }
53
54      // Find the nearest lower LiDAR timestamp
55      double best_lidar_timestamp = -1.0;
56      int best_lidar_id = -1;
57      double min_difference = std::numeric_limits<double
           >::max();
58
59      for (const auto& lidar : lidar_data) {
60          double lidar_timestamp = lidar.first;
61          int lidar_id = lidar.second;
62
63          // Condition: LiDAR timestamp must be less than
               camera timestamp
64          if (lidar_timestamp < camera_timestamp) {
65              double difference = camera_timestamp -
                   lidar_timestamp;
66              if (difference < min_difference) {
67                  min_difference = difference;
68                  best_lidar_timestamp = lidar_timestamp;
69                  best_lidar_id = lidar_id;
70              }
71          }
72      }
73
74      // Check if we found a valid LiDAR timestamp
75      if (best_lidar_timestamp == -1.0) {
76          std::cerr <<  Warning: No LiDAR timestamp found
               that is less than camera timestamp   <<
               camera_timestamp << std::endl;
77          return {0.0, -1}; // Return default values
78      }
79
80      std::cout <<  Matched: Camera Frame   <<
           camera_frame_id
81                <<   (timestamp:   << camera_timestamp
82                <<  ) -> LiDAR ID   << best_lidar_id
83                <<   (timestamp:   << best_lidar_timestamp
```

```
                    <<  )  << std::endl;

    return {best_lidar_timestamp, best_lidar_id};
}

} //namespace ORB_SLAM
```

Listing 2: C++ Implementation: MatchTimestamps Function

```
# Ego motion compensation
T_Cam_inv = np.linalg.inv(self.keyFame_camera_pose)
T_relative = T_Cam_inv @ self.current_Lidar_pose
R_relative = T_relative[:3, :3]
t_relative = T_relative[:3, 3]
pts_surface_velo = (pts_surface_velo[:, None, :3] *
   R_relative).sum(-1) + t_relative

# Transform the points to the camera frame
pts_surface_cam = (pts_surface_velo[:, None, :3] * self.
   T_cam_velo[:3, :3]).sum(-1) + self.T_cam_velo[:3, 3]
T_cam_obj = self.T_cam_velo @ T_velo_obj
```

Listing 3: Python Implementation: Ego-Motion Compensation

D.4 Architecture and Data Flow

D.4.1 Timestamp Matching (`MatchTimestamps`)

- **Input**: Camera frame ID (integer)
- **Process**: Queries a CSV file containing mappings between camera frame IDs/timestamps and LiDAR IDs/timestamps
- **Selection Logic**: Identifies the LiDAR scan with the nearest timestamp strictly preceding the camera timestamp
- **Output**: Pair (LiDAR timestamp, LiDAR ID) or default values (0.0, -1) if no match is found

D.4.2 Pose Interpolation (`InterpolatePoseForLiDAR`)

- **Input**: Current, previous, and second-previous camera poses with timestamps
- **Process**: Interpolates or extrapolates the camera SE(3) pose to the target LiDAR timestamp
- **Output**: Camera pose aligned with the LiDAR scan timestamp

D.4.3 Python Processing (`ficosa_sequence.py`)

- **Inputs**: Frame ID, LiDAR ID, keyframe camera pose (4×4), LiDAR pose (4×4)
- **Processing Steps**:
  - Loads corresponding RGB image and LiDAR point cloud (.bin file)
  - Performs 2D image detection and 3D LiDAR object detection
  - Associates LiDAR surface points with 2D masks
  - Applies ego-motion compensation using provided poses
- **Output**: List of detections containing surface points (N×3), camera-to-object transforms (4×4), and optional rays/depth information

D.4.4 C++ Integration (`GetObjectDetectionsLiDAR`)

- **Input**: KeyFrame with frame ID, pose matrix, and temporal context from recent frames
- **Process**:
  - Calls `MatchTimestamps` to find corresponding LiDAR scan
  - Computes interpolated camera pose via `InterpolatePoseForLiDAR`
  - Invokes Python processing with GIL protection
  - Converts Python detections to C++ `ObjectDetection` instances
- **Output**: Vector of `ObjectDetection*` stored in the KeyFrame

D.5 Key Design Considerations

D.5.1 Pose Utilization in Python Processing

The Python module utilizes two critical poses:

- **Keyframe Camera Pose**: Enables transformation between camera and LiDAR coordinate frames for sampling and ego-motion compensation
- **LiDAR-time Camera Pose**: Facilitates computation of relative pose between camera and LiDAR for motion compensation of LiDAR points into the camera reference frame

D.6 Edge Cases and Limitations

- **CSV Dependency**: System relies on precomputed timestamp mappings; missing entries result in fallback behavior
- **Timestamp Selection**: Current implementation prefers the latest LiDAR scan before camera capture, which may not be optimal for all scenarios
- **Interpolation Accuracy**: Pose interpolation accuracy degrades with larger temporal gaps or high-speed motion
- **Data Transfer Overhead**: Substantial matrix and point cloud data crosses Python-C++ boundary, incurring copying costs
- **Thread Safety**: GIL protection ensures basic thread safety, but concurrent map access requires careful management

D.7 Summary

The `GetObjectDetectionsLiDAR` function enables robust sensor fusion by synchronizing camera keyframes with LiDAR scans through timestamp matching and pose interpolation. The system delegates LiDAR processing to Python for object detection and motion compensation, then integrates the results back into the C++ SLAM framework for subsequent data association and mapping operations.

References

[1] Wang, J., et al. (2020). DSP-SLAM: Object Oriented SLAM with Deep Shape Priors. Retrieved from `https://github.com/JingwenWang95/DSP-SLAM`

[2] DSP-SLAM GitHub repository. (2020). `https://github.com/JingwenWang95/DSP-SLAM`
bibitemSensorFusionBlog LiDAR and Camera Sensor Fusion in Self-Driving Cars - Think Autonomous

[3] A Survey on LiDAR Motion Compensation Methods - Wiley Online Library

[4] Motion Compensation in LiDAR Point Cloud - MATLAB Documentation

[5] Self-Driving Car Dataset CH2 - Udacity GitHub

[6] PointPainting: Sequential Fusion for 3D Object Detection - arXiv

[7] Livox Cloud Undistortion - GitHub Repository

[8] A Survey on LiDAR Motion Compensation Methods - Wiley Online Library (Duplicate)

[9] Deep Continuous Fusion for Multi-Sensor 3D Object Detection - ECCV 2018

[10] SLAM Toolbox - ROS 2 Humble