SenseBike Dataset

Addressing LiDAR Domain Gaps through the Introduction of a Novel Dataset from a Bicycle's Perspective

Thesis Report

Jan Peter Simons

Delft University of Technology

TU Delft

SenseBike Dataset

Addressing LiDAR Domain Gaps through the Introduction of a Novel Dataset from a Bicycle's Perspective

by

Jan Peter Simons

Student Name	Student Number
J.P. Simons	4368185

Supervisor TU Delft:	H. Caesar
Supervisor Alten:	L. Abdulkadir
Project Duration:	September, 2023 - July, 2024
Studies:	MSc. Robotics
Faculty:	Mechanical Engineering, Delft

Cover:	SenseBike outside of Mechanical Engineering, TU Delft
Style:	TU Delft Report Style

TU Delft

Abstract

LiDAR technology is gaining popularity for use in 3D object detection, necessary for self-driving cars. However, due to class imbalances in state-of-the-art LiDAR datasets, detection algorithms often tend to lack performance in detecting cyclists. To address this issue, we introduce the SenseBike, a LiDAR-equipped bicycle suited for collecting novel data, including more cyclists. We have created the SenseBike dataset, which features distinctive data from the city of Delft, The Netherlands. Recording from a bicycle brings unique challenges, and we explain and evaluate our solutions to these issues. To evaluate the impact of this new dataset on the performance of LiDAR object detection, we adapted an existing pseudo-labeling pipeline. Despite the recommendation, we did not self-train this pipeline, which would have resulted in higher quality pseudo-labels. Nonetheless, when we train CenterPoint, a well-known and fast 3D LiDAR object detector, on these lower-quality pseudo-labels, we still achieve an 85% Average Precision for cyclists, evaluated with maximum center-distance differences of 1m.

Contents

1 Introduction

The arrival of self-driving cars is a significant step into the future of transportation, offering numerous benefits beyond convenience. As Qian et al. [46] envisioned: *"Let the dream be realized, thousands of new employment opportunities shall be created for those physically impaired (Mobility), millions of lives shall be rescued from motor vehicle-related crashes (Safety), and billions of dollars shall be saved from disentangling traffic accidents and treating the wounded (Economics)"*.

To achieve fully autonomous vehicles, a full understanding of the surroundings, or perception, is essential. Perception can be achieved in numerous ways. 2D perception with cameras provides information about the location of objects without depth, while 3D perception includes depth information, typically achieved with 3D sensors like LiDAR, Radar, or stereo cameras [3]. The goal of 3D perception is to understand the vehicle's surroundings, which involves identifying objects, labeling them, outlining their shapes with bounding boxes, determining their distances from the vehicle, and providing a heading angle to indicate orientation [78, 4].

There are several ways to detect this environment. As our recording device is not equipped with fully embedded cameras or radars, we will focus on 3D detection with LiDAR only. LiDAR stands for Light Detection and Ranging. It is commonly used in autonomous driving and its integration on personal vehicles is emerging [75]. A study using one of the earliest public datasets featuring LiDAR [19] demonstrated that LiDAR-based 3D object detection is 68.63% more accurate on vehicles than camera-only methods. This highlights how important LiDAR can be for making object detection more precise and reliable, especially for autonomous vehicles [78]. The standout features of LiDAR include its accuracy, even in low light conditions, and its spatial resolution. Due to its capacity to concentrate laser light and its short wavelength, it becomes feasible to achieve a spatial resolution of approximately 0.1°, and accuracies of $2cm$ at a distance of $100m$ [28]. This is the result of the active sensing principle of LiDAR: It emits a huge amount of infrared light rays and they reflect on surfaces back to the LiDAR. The output of a LiDAR is a pointcloud with reflected points containing x,y and z coordinates and its reflectivity.

Typically, 3D object detection involves integrating Deep Neural Networks (DNN) into detection algorithms. Successful performance of these detection algorithms is attained by training the DNNs on labeled data. Generally, the greater the amount of labeled data, the better the performance [1]. Hence, it is of great importance that multi-modal datasets are released to *"exhibit the full set of challenges associated with building an autonomous driving perception system"* [8].

Most well-known automotive datasets containing LiDAR [8, 55, 33, 19, 65, 24, 6] are recorded in cities with low cyclist presence, leading to an imbalance in the representation of different objects. Specifically, these datasets have fewer cyclists compared to vehicles and pedestrians. Training deep neural networks (DNNs) on such imbalanced data introduces performance biases on other classes [36].

When we want deep neural networks (DNNs) trained on one LiDAR dataset to perform well on another dataset, we encounter the challenge of domain adaptation. Each LiDAR sensor has a distinguishable scanning pattern, leading DNNs to perform poorly when trained on one dataset and tested on another. This performance drop is not solely due to changes in the LiDAR scanning pattern (domain), but also due to geographical shifts. For example, American cities differ fundamentally from Dutch cities,

with variations in the density of nearby cars, the distance to cyclists, and unique environmental features, such as the presence of canals. These differences require robust domain adaptation techniques to ensure the generalization of DNNs across diverse datasets [64, 44].

The TU Delft has recently (2023) purchased a new perception vehicle: A sensor bicycle, named as *SenseBike*, displayed on the front page, produced by Boreal Bikes [27]. We modified it, so it now has three LiDARs divided over both the front and the rear. Equipping a bicycle with these sensors and generating a new dataset serves several purposes, including:

1. *Gathering more Data of Cyclists:* Collecting additional data, specifically of cyclists. As will be discussed later, current widely-used multi-modal datasets for AVs have only a small portion of cyclists in them, which results in a tendency of detectors to underperform on cyclists.
2. *Increased Mobility and Sustainability:* Gathering data directly from bicycles provides access to a wider variety of locations, contrary to collecting data from a car. It can easily navigate to areas typically inaccessible to cars, such as forests, narrow alleys, beaches, car-free city centers, and even inside buildings. Moreover, energy consumption is highly reduced per kilometer, compared to car-based recordings.
3. *Investigating Cyclist Behavior on a Bike:* Facilitating studies focused on understanding and analyzing the behavior of cyclists while riding. This could help in predicting cyclist motion or potential car-bike collisions.
4. *Enabling Research on Bicycle Assistance Features:* Investigating and developing features to enhance the safety and assistance for cyclists. This could include features assisting in holding a two-wheeled balance or avoiding collisions.
5. *Advancing Research in the Field of Urban Planning:* Utilizing gathered data to contribute to urban planning by designing effective and safe bike paths, and identifying locations where accidents involving cyclists occur, contributing to the development of safety measures.

We tackle the issue of data bias favoring non-cyclist road users by introducing a novel, yet unlabeled dataset. By extending and applying an existing domain-adaptation method [59], we investigate whether this dataset can improve the detection performance of cyclists using off-the-shelf 3D LiDAR detection algorithms, thereby enhancing overall performance. Our contributions include:

- We developed a new type of data-collection vehicle, specifically a bicycle, capable of gathering more versatile data on vulnerable road users (VRUs) in both indoor and outdoor environments.
- We discuss the methods used to overcome the unique challenges of recording data from a bicycle, including the inability of a single LiDAR to provide a full-surround view, internal frame rotations, and rolling angles while cornering.
- We apply a pseudo-labeling pipeline, and concentrate on refining the labels for cyclists.
- We evaluate state-of-the-art (SOTA) 3D object detection algorithms - trained on these pseudo-labels - on a small labeled validation set, and achieve increased cyclist detection performance.

Above contributions were the result of the following research question and its subquestions:

Research Question:

Can we create a novel dataset, that enhances the 3D object detection performance of state-of-the-art detectors?

Given that this process can be subdivided into distinct steps, we have outlined it into three specific subquestions:

1. How can we produce a novel dataset, that is truly distinguishable from others?
2. What challenges arise when collecting data from a bicycle, and how can they be addressed?
3. Can the novel *SenseBike* dataset positively affect the performance of existing off-the-shelf 3D object detectors and how can this be evaluated?

Structure
This report begins with an overview of related research in Section 2. It then provides a detailed description of the *SenseBike* configuration in Section 3, followed by used methods in Section 4. We explore the validation of the methodology in Section 5 and end with a summative conclusion in Section 6. The Appendix includes a manual with detailed information on the recording device, instructions on software installation, guidance on recording procedures, and additional details of Sections 2 to 5.

2

Related Work

2.1. Automotive datasets with LiDAR

Many innovations in the field of self-driving cars are benchmarked against publicly available datasets, which are used to train supervised deep learning detector algorithms. As [6] states, they fulfill three purposes: (i) providing a basis for measuring progress, since results can be compared with other results on the same set, (ii) uncovering shortcomings of the current state of the art, which leads to better research directions, and (iii) making it possible to develop approaches without the need to first collect data and having to label it. To assess the performance of trained models, it is important to have access to the ground truth (GT) of these datasets. For object detection, the typical way of representing an object is by the use of a 3D bounding box, containing *x,y,z*, *width, length, height*, and *yaw*. *Yaw* represents an object's heading angle.

A fully human-labeled dataset forms the foundation for all supervised deep learning methods, including techniques like object detection or, for instance, any method used for annotation assistance. There are multiple publicly available datasets, this section discusses four of them. The first three are recorded with multiple modalities, such as camera, radar, or LiDAR, and these have objects labeled in all of these modalities for sequential frames. These datasets include scans from all sensors captured at the nearest possible timestamps, a process known as sensor synchronization. This enables research on deep learning techniques for detecting in single frames on a single modality or, for example, tracking objects on cross-modalities over multiple frames. First, 1) *nuScenes* [8] and 2) *Waymo Open Dataset* [55] are briefly discussed. We also explore two other less-famous datasets: 3) the *Salzburg Bicycle Data Set (SBLD) [40]*, which is a dataset recorded with the same bicycle and a sensor setup similar to that used during this research and 4) the *View-of-Delft Dataset*, which is a multi-modal dataset recorded in the same area as the recordings of the novel dataset from this research.

2.1.1. nuScenes

"The nuScenes dataset [8] provides comprehensive multimodal data, including 6 RGB cameras, 5 radars, and 1 rotating 32-beam LiDAR, offering 360-degree coverage. It also incorporates map information, enabling research on scene understanding. When set side by, for instance, SemanticKITTI [6], nuScenes excels in terms of data scale, complexity, and diversity. Figure 2.1 provides a visual representation of a nuScenes recording, complemented by a human-annotated semantic map.
The dataset contains 1000 distinct driving sequences. Among these, 700 sequences are allocated for training, 150 for validation, and another 150 for testing purposes. Each sequence spans 20 seconds, and the LiDAR operates at a frequency of 20 Hz. Calibrated vehicle pose data is available for every LiDAR frame, box annotations are provided only once every ten frames (2Hz). This data set uses a 32-beam LiDAR, generating approximately 30,000 points per frame. Specifically, there are 28k, 6k, and 6k annotated frames designated for training, validation, and testing, respectively. These annotations contain ten distinct classes, with a distribution skewed towards certain classes, such as cars or trucks.

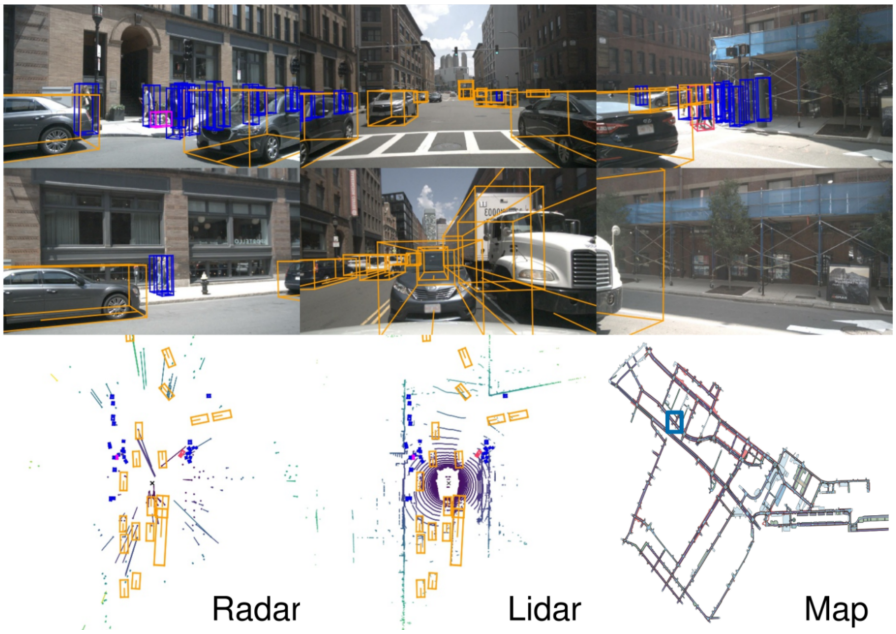


Figure 2.1: Example of the nuScenes dataset. Top: 6 Camera views + Radar + LiDAR. Bottom: Human written descriptions. Taken from [8].

2.1.2. Waymo

The Waymo Open Dataset [55], owned by Alphabet, stands as a robust resource for autonomous driving research, notable for its high quality and extensive coverage. Covering an impressive 76 square kilometers, the dataset includes 230,000 annotated LiDAR frames, each featuring approximately 12 million manually drawn 3D bounding boxes. The dataset is strategically divided into 1,000 scenes for training and validation, and an additional 150 scenes for testing, with each scene lasting for 20 seconds. A distinctive feature is the inclusion of five LiDARs, where they strategically placed one at the rear, one on the top, two at the forward sides, and one in the front of the vehicle. This setup results in an average of 177,000 points per frame, a notable increase compared to other datasets like *nuScenes*. For object tracking research, the Waymo Open Dataset provides annotations for 113,000 unique tracking IDs. This detailed tracking information adds to the dataset's suitability for studying the dynamics and interactions of objects within the driving environment. What makes this dataset stand out is its intentional focus on achieving geographic diversity. Recordings took place in various American cities, ensuring a well-rounded representation of different urban environments.

2.1.3. Salzburg Bicycle LiDAR Data Set (SBLD)

The Salzburg Bicycle LiDAR Dataset (SBLD) [40] was created using five Livox Horizon LiDARs mounted on the Boreal Holoscene X smart bicycle. This configuration, which used a similar bicycle as used in this research, generates five pointclouds that are combined into a single pointcloud and stored as a PCL *.pcd* file. This file is later converted to the SemanticKITTI format. Using the KISS-ICP method [62], a 3D map and relative positioning are established, and the ground is semantically labeled with the SemanticKITTI's *Point Labeler* [6]. During labeling, fuzzy points near the bicycle path are assumed to represent vegetation. The same program is used to label objects close to the bicycle path by comparing 2D images from the camera or Google Maps with the pointclouds. Due to time constraints, about 25% of the points remain unlabeled.

The dataset contains 28 sequences, each having between 23 and 1,020 pointclouds, of which 17 sequences are fully labeled. The labeled points are distributed as: 30% vegetation, 22% terrain, 12% road, 6.4% fence, 2.6% building, and 1.7% other structures. The classes of cars, poles, and moving bicyclists each represent 0.50% of the labeled points.

2.1.4. View-of-Delft Dataset

The View-of-Delft (VoD) dataset [42] presents an innovative collection of automotive data captured in Delft, the Netherlands. This dataset contains over 8600 frames with synchronized and calibrated information from a 64-layer LiDAR, stereo cameras, and 3+1D radar. The radar data includes standard radar outputs such as range, azimuth, and elevation as well as Doppler data. All the data is acquired in complex urban traffic scenarios. Within this dataset, there are more than 123k 3D bounding box annotations for both moving and stationary objects. These annotations cover a wide range, including over 26,500 pedestrians, 10,800 cyclists, and 26,900 cars. It is the only dataset in this chapter featuring the simultaneous use of high-end (64+-layer) LiDAR and any type of radar data.

When comparing the object detection performance between radar and LiDAR pointclouds, both in terms of specific classes and distance, it is observed that 64-layer LiDAR data still exhibits superior performance compared to 3+1D radar data. However, the addition of elevation information and the integration of successive radar scans contribute to narrowing the performance gap between the two modalities. Most remarkably, they show that this addition from the radar can greatly improve the detection of moving cyclists. The dataset stands out in its percentage of annotated cyclists: 29%.

Datasets	Classes	2D bbox	3D bboxes				Segmented LiDAR Points				Track -IDs
			Total	Cars	Bicycles	Peds	Total	Cars	Bicycles	Peds	
nuScenes	32	800k	1.2M	493k (42%)	11.8k **(1.0%)**	222k (19%)	1.2B	38.1M (2.72%)	141k (0.012%)	2.3M (0.19%)	✓
Waymo	23	9.9M	12M	6.1M (51%)	67k **(0.56%)**	2.8M (23%)	-	-	-	-	✓
View-of Delft	13	123k	123k	27k (22%)	36k (29.1%)	26.6k (22%)	-	-	-	-	✓
Salzburg BLD	10	-	-	-	-	-	?	?	?	?	**X**

Table 2.1: Annotation statistics of the discussed automotive datasets. Bboxx = bounding box, peds = pedestrians. '?' = No official statistics published, '-' = None present

2.2. 3D Object Detection

2.2.1. Modalities

In autonomous driving, 3D object detection can identify objects like vehicles, pedestrians, and stationary items to help vehicles respond to their environment. Commonly used modalities include cameras, radars, and LiDARs [46]. This section briefly compares them and their performances, in terms of mean Average Precision (mAP) and nuScenes Detection Score (NDS). Explanations of these scores can be found in Appendix D.2.

Camera-based detection offers a low-cost and interpretable solution. However, the data acquired from camera-based systems is significantly influenced by weather and lighting conditions. Additionally, these systems lack depth information, which requires an estimation, thereby increasing the likelihood of inaccuracies. Models such as Sparse4D-3D [35] have demonstrated performance metrics of 0.63 mAP (mean Average Precision) and 0.694 NDS (NuScenes Detection Score) on the nuScenes dataset.

Radar-based detection uses electromagnetic waves for distance, velocity, and direction. FMCW radars are robust to weather but have limited vertical angles and fewer points. KPConvPillar is the top radar detector on nuScenes with a mAP of 0.049 and an NDS of 0.139 [61].

LiDAR-based detection emits laser pulses to create detailed 3D pointclouds [3]. A big advantage is its unaffectedness by lighting conditions and its high accuracy over long distances. A big drawback is the cost: LiDARs are expensive. *SphereFormer* [30] leads the LiDAR-only *nuScenes* detection challenge with a mAP of 0.685 and an NDS of 0.728.

Multi-modality detection combines sensors to overcome individual limitations. The multi-modal detector *FusionFormer* [25] scores a mAP of 0.766 and an NDS of 0.776.

2.2.2. LiDAR-based 3D detection algorithms

This section examines deep learning detection algorithms (detectors) designed specifically for autonomous driving, which process a single LiDAR pointcloud input to generate a list of predicted 3D detection boxes, each with an associated object category. The detectors are categorized based on their processing approach: those that voxelize a pointcloud (voxel-based), those that process the pointcloud point by point (point-based), and those that integrate both methods (multi-view). Additionally, some detectors incorporate attention mechanisms in their algorithms (transformers). All detectors are either one-stage or two-stage: One-stage 3D object detectors predict objects directly from the input data in a single pass, optimizing for speed, whereas two-stage detectors first generate region proposals and then refine these proposals for more accurate detection, optimizing for precision.

Voxel-based

Voxel-based methods for 3D object detection partition the pointcloud into regular-sized 3D voxels or infinite-height voxels, called pillars. This approach is computationally efficient and good for local feature extraction but loses global pointcloud information, leading to lower accuracy.

VoxelNet [2018] [77] is a one-stage detector that partitions a raw pointcloud into 3D voxels. Each voxel encodes features such as distance to the local mean. The voxel features are processed by a multidimensional convolutional layer, producing a feature map for the Region Proposal Network (RPN), which outputs a probability score map and a regression map. **SECOND [2018]** [67], referring to Sparsely Embedded CONvolutional Detection improves training and inference speed using 3D sparse convolutions, a novel angle loss regression approach, and a new data augmentation method. It uses voxel-wise feature extraction, a convolutional middle layer, and an RPN. The sparse layers reduce computational complexity, and anchors for cars, pedestrians, and cyclists improve bounding box regression. Where [77] and [67] use 3D voxels, **PointPillars [2019]** [31] divides the *x,y*-plane into 2D grids (pillars) with infinite height. Points are augmented with coordinates and features, creating a 9D vector for each point. A simplified *PointNet* extracts features, which a 2D CNN and an encoder then process. A Single Shot Detector (SSD) classifies and detects objects. Pillars are more computationally efficient than 3D voxels. **VoxelRCNN [2021]** [16] is a two-stage detector applying 3D voxel tensors directly. It includes a 3D backbone, a 2D backbone with an RPN, and a detection subnet for box refinement. This approach balances accuracy and efficiency by reducing points processed by the subnet.

Point-based

Point-based methods use raw pointclouds as input, preserving accurate point positions but generally increasing computational complexity.

PointNet [2017] [45] is a pioneering neural network architecture designed for directly processing raw pointcloud data. It applies a novel max-pooling layer to capture global pointcloud features while being invariant to input reordering, ensuring robust object classification and segmentation by learning spatial encoding of points and their local relationships. **PointRCNN [2019]** [50] is a two-stage network. The first stage generates 3D proposals by separating foreground and background points. The second stage pools points in proposal regions, transforms them for spatial feature learning, and refines the bounding boxes and confidence scores using *PointNet* [45]. As [45] and [50] use two stages, **3D-SSD [2020]** [70] is a single-stage detector optimizing inference speed. It introduces a novel point sampling method combining feature and Euclidean distances, a shift for better center prediction, and a modified ground truth classification score to improve average precision. **CenterPoint [2021]** [72] is a two-stage detector estimating object centers first. The second stage refines these estimations using a Multi-Layer Perceptron (MLP) and compares center points across pointclouds to estimate velocity.

Multi-view

Multi-view methods combine point-based and voxel-based feature learning, balancing computational load and feature extraction.

PV-RCNN [2021] [52] integrates voxel-based and *PointNet*-based feature learning. It samples key points, encodes 3D voxels into key points, and uses a Voxel Set Abstraction module to integrate multi-scale voxel features. The Predicted Keypoint Weighting (PKW) module re-weights key points, and RoI pooling refines bounding boxes. **PV-RCNN++ [2022]** [53] improves *PV-RCNN* with a *sectorized*

proposal-centric sampling strategy and *VectorPool* for feature aggregation, enhancing accuracy and efficiency. **SE-SSD [2021]** [76], or Self-Ensembling Single-Stage object Detector, uses a teacher-student framework with novel loss functions and data augmentation, improving precision without additional computational overhead during inference.

Transformers

Transformers use attention mechanisms, introduced by Vaswani et al. [60], to enhance pointcloud performance, particularly in sparse regions. They can still be either point or voxel-based.

FocalFormer3D [2023] [11] uses Hard Instance Probing (HIP) to detect false negatives through multi-stage re-evaluation, enhancing detection scores but requiring high loads of computational power. Where all detectors in this Section use cartesian coordinates, **SphereFormer [2023]** [30] uses radial windows with spherical coordinates, exponential positional encoding, and dynamic feature selection to improve detection in sparse pointclouds. **DSVT [2023]** [63], or Dynamic Sparse Voxel Transformer, uses a novel voxel-based architecture to dynamically allocate computational resources, enabling efficient processing of sparse voxel representations. This approach optimizes accuracy and speed in large-scale 3D pointcloud analysis, achieving a framerate of 27 Hz while maintaining computational efficiency. All of these transformers outperform non-transformer methods on the nuScenes dataset, with *FocalFormer3D*, *SphereFormer*, and *DSVT* scoring mAP & NDS of 0.687 & 0.726, 0.685 & 0.728, and 0.690 & 0.732, respectively. It has to be noted again that these methods are computationally demanding.

2.3. Perception from Bicycles

The field of machine perception from bicycles is relatively unexplored. However, some research has been conducted with bicycles similar to the *SenseBike*, such as the bicycle-lane segmentation work using the Salzburg Bicycle LiDAR Data (SBLD) Set [40], [41]. These studies apply a Semantic-KITTI [6] pre-trained convolutional neural network (CNN) to their dataset and enhance it with self-attention blocks. Additionally, there is a growing field focused on bicycle-vehicle interaction. For instance, [5] explores the potential of Cooperative Intelligent Transport Systems (C-ITS) in improving communication between automated vehicles, which could significantly enhance collision avoidance. [21] investigates the likelihood of collisions between vulnerable road users and vehicles using data from governmental crash reports. As electric bicycles become increasingly popular, the number of collisions has also risen, prompting research on active bicycle safety systems, such as radar-based assistants [22] and 2D LiDAR solutions [22, 66]. [37] estimates cyclist pose using on-rider cameras and gyroscopes. To the best of our knowledge, the SBLD is the only LiDAR dataset recorded from a bicycle that has been used for any type of 3D perception task.

2.4. Unsupervised Domain Adaptation

2.4.1. Knowledge Learning

Unsupervised Domain Adaptation (UDA) is a specialized form of knowledge transfer in which a large deep neural network (DNN) is trained to make predictions, and a smaller, more efficient DNN is trained to emulate the larger one [26]. Hinton et al. [23] describe the larger model as the teacher and the smaller model as the student. The student network can be trained to imitate the teacher network either concurrently with the teacher's training (online) or after the teacher has been trained (offline). The teacher's predictions are referred to as pseudo-labels. Domain adaptation is achieved when the teacher network is specifically designed to address challenges associated with transitioning to a new dataset.

Unsupervised Domain Adaptation (UDA) aims to adapt a deep neural network (DNN) trained on a labeled source domain to perform effectively on a new, unlabeled target domain. UDA methods can be categorized into domain-invariant representation, adversarial techniques, and self-training approaches. In the context of UDA, cross-domain adaptation involves adjusting detectors from a source domain to a different target domain, commonly referred to as single-source to single-target UDA [9, 58, 2].

2.4.2. Pseudo-labelling

The most direct and reliable approach to annotating pointclouds involves human annotators, who process each pointcloud individually, drawing boxes in a user-friendly graphical interface. However, this method is exceedingly time-consuming and thus expensive, encouraging the exploration of various techniques to assist or automate annotation processes [18].

A pseudo-label is a high-confident prediction from a framework that uses the knowledge of existing detectors and applies extra confidence-increasing methods to it. Multiple frameworks for pseudo-labeling LiDAR pointclouds with a certain domain shift have been proposed. *ST3D* [68] is the first to present a fully unsupervised self-training framework that generates pseudo-labels that can be used for training on the target pointcloud. More frameworks based on *ST3D* have been proposed, with the addition of multi-object trackers, random object sampling techniques, repeated traversals, constrastive learning, forms of test-time augmentations, and thresholding techniques [74, 73, 69, 71, 38, 12, 13]. We will use and adapt a novel proposed method, *MS3D++* [59], for our research. Its abbreviation stands for Multi-Source 3D. An overview of this adapted method is displayed in Figure 4.7.

MS3D++

MS3D integrates detectors trained on different datasets and with various backbones to generate a large number of predictions, termed Multi-Source detections. The combination of all these detectors forms the detector ensemble (DE). This ensemble is applied to single pointclouds and to different numbers of concatenated pointclouds, a process known as varied multi-frame inference. Additionally, it is used on pointclouds with or without test-time augmentation. The multi-source detections result in numerous detections for a single object, which are filtered and combined using the proposed Kernelized Box Fusion (KBF) method that utilizes Kernel Density Estimation (KDE). Instead of relying solely on predictions with the highest confidence scores, KBF reshapes detections based on the combined confidence of all multi-source detections. These filtered and combined detections are named fused detections. High-confidence predictions from this fused detection set are used as pseudo-labels, while those with lower confidence scores are tracked using a multi-object tracker (MOT). If a track contains a sufficient number of high-confidence detections, all predictions in that track are used as pseudo-labels. Additional refinement is applied to tracks of static vehicles, dynamic vehicles, and pedestrians, a process referred to as Temporal Refinement. MS3D++ achieves high-precision pseudo-labels, especially after several rounds of self-training the entire framework.

3

SenseBike Configuration

This section focuses on the hardware and software setup of the *SenseBike*, the data-collection vehicle for the *SenseBike* dataset. An additional manual including all the details of the hardware and the software, as well as a step-by-step guide on how to record data with it, is found in Appendix A. This section discusses all the hardware choices, describes the software architecture, and ends with the *SenseBike* limitations.

3.1. Hardware

Figure 3.1 shows a sideview of the *SenseBike*. The bicycle itself is a $48V$ $17.5Ah$ electric bicycle, with the battery attached in the frame, just above the *BoReal* logo. It has five different levels of pedal assist, 7 gears, 2 disc brakes, and most other components you would expect on an electric bicycle. All the hardware that transforms it into a data-collection vehicle is placed on either the rear luggage carrier or the front carrier.

Figure 3.1: SenseBike Sideview, taken at the TU Delft

3.1.1. Overview

This section gives a brief overview of all data-collection hardware components. Upon delivery, the bicycle was equipped with front items 2 and 4, and rear items 2, 3, 5 - 12 & 13A. We introduced front items 1 & 3, and rear items 1, 4, 13 & 13B.

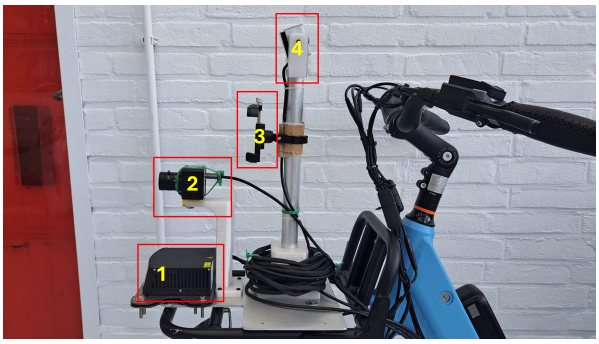


Figure 3.2. SenseBike Front Overview

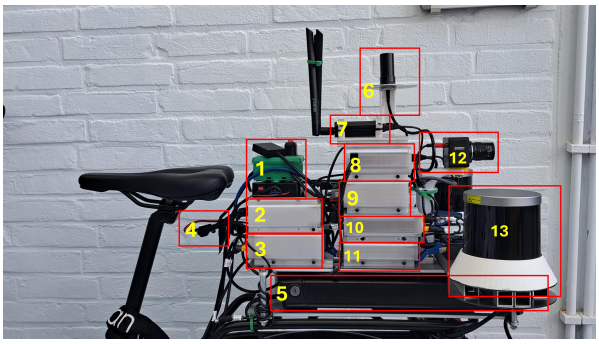


Figure 3.3. SenseBike Rear Overview

No.	Front Component
1	Robosense M1P Solid-State LiDAR
2	Front Camera
3	Phone Holder
4	Gaze Camera

Table 3.1: Front Components Sensebike

No.	Rear Component
1	M1P LiDAR interface + LiDAR electronics
2	Ethernet / Network Switch
3	Voltage Regulators
4	Amp fuse
5	Battery Pack
6	GNSS / RTK Antenna
7	Router
8	USB-C Hub
9	NVIDIA Jetson Nano Orin NX (16GB)
10	Spacer / Voltage regulator
11	GNSS / RTK module
12	Rear Camera
13	LiDAR + IMU Mount
13A	Internal Measurement Unit (IMU) + casing
13B	2x Robosense Helios 32 Rotating LiDAR

Table 3.2: Rear Components Sensebike

3.1.2. LiDARs

The SenseBike is equipped with three LiDARs: two rotating ones positioned at the rear and a solid-state one at the front. This section explains the process that led to this specific setup, and an overview of the specifications of both types of LiDARs.

Positioning

The choice for the triple LiDAR setup and the positioning is based on two principles:

1. We aim for a **full 360-degree LiDAR view** on the X, Y, and Z axes. While a single 3D rotating LiDAR mounted centrally at the top of a car is typically sufficient, a bicycle lacks the possibility for such placement. Given the geometry of the *SenseBike* and the occlusions caused by the rider, we are constrained to use at least three LiDARs.
2. We expect the presence of a high number of nearby vulnerable road users (VRUs) and aim to ensure **full LiDAR coverage of VRUs**. To achieve this, we aim to capture the nearest LiDAR points on the ground plane within a 1-meter radius.

Given that we already possessed a *Robosense M1 Plus* LiDAR, we decided to acquire additional LiDAR units from the same manufacturer. We conducted simulations of various configurations in a 3D *Solidworks* [15] environment, utilizing both the *Robosense Helios* and the *Robosense Bpearl* models. These LiDARs both fully cover a horizontal 360°, but have a different vertical scanning pattern. The *Helios* covers 70° and the *Bpearl* 90°. The *Helios* is produced for short-to-medium range operations, whereas the *Bpearl* is manufactured for nearby blind-spot coverage. The results of the simulations are presented in Figure 3.4. It is evident that the occluded area, depicted in yellow, is larger with two *Bpearls* (top right) compared to two *Helios* LiDARs.

The isometric view of the two *Bpearls* (top left) reveals that this LiDAR emits a substantial number of laser beams in the positive z-direction, where we expect fewer VRUs. Additionally, as shown in the top right, the scan pattern of the *Bpearls* differs from that of the two Helios units (bottom right). For adapting existing deep learning methods to our LiDAR configuration, it is assumed that a scan pattern more similar to those in popular automotive datasets, such as nuScenes or Waymo, may facilitate more effective research. We have therefore chosen a two `Helios` plus one `M1P` LiDAR combination. A side view representation of the occluded area is presented in Figure 3.5.

To achieve a scan pattern more similar to those found in larger automotive datasets, we decide not to rotate the two Helios LiDARs relative to each other. Increasing the relative distance between the units reduces the occlusion area directly in front of the rider, so we opted to maximize this distance. Consequently, the mounting system was widened to match the steering bar's width for practical considerations. An exploded view is illustrated in Figure 3.6.

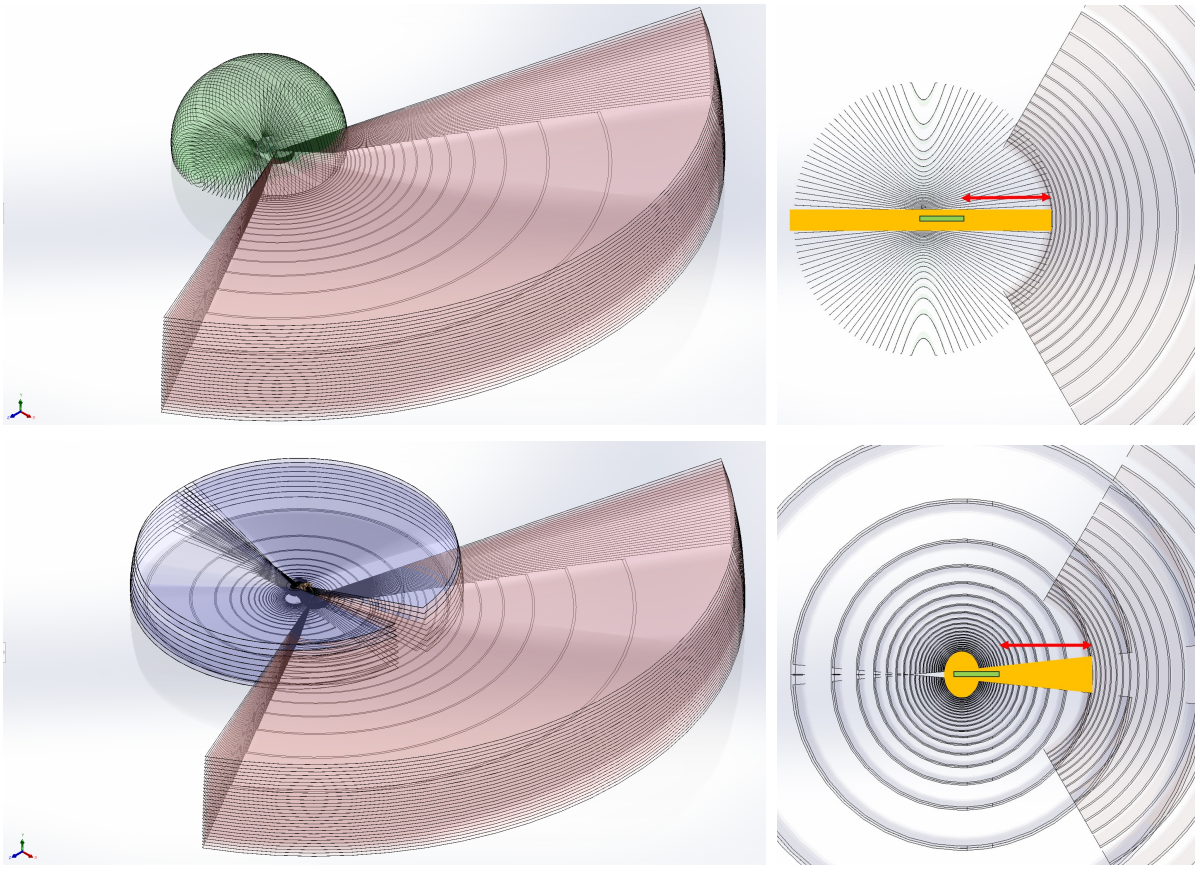


Figure 3.4: 3D simulations of potential LiDAR setups.
a) Top left: Isometric view of two Bpearls (green) and one M1P (red) scan pattern. b) Top right: Top view of scan pattern two Bpearls and one M1P. c) Bottom left: Isometric view of two Helios 32 (blue) and one M1P (red) scan pattern. d) Bottom right: Top view of scan pattern two Helios and one M1P.
For the figures on the right: Yellow area: the occluded region on the ground plane, the red arrow indicates a range of 4 meters, and the green area depicts a 2D projection of the bicycle frame, without the handlebar.
All are produced in [15].

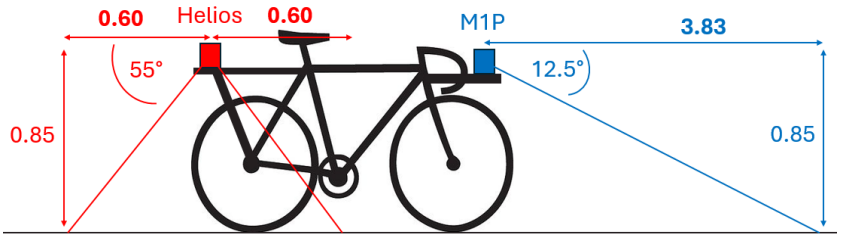


Figure 3.5: Schematic simplified side view of LiDAR FOVs on the Sensebike: Unscaled and units in meters.

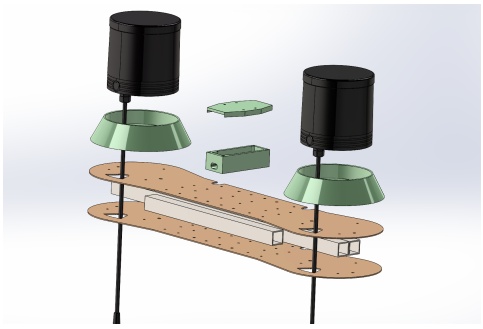


Figure 3.6: Helios LiDAR mount, exploded view, produced in Solidworks [15].

Specifications

Table 3.3 provides a detailed overview of the specifications of the equipped LiDARs. The M1 Plus (M1P) LiDAR captures more points within a smaller Field of View (FoV), resulting in higher detail. Being a solid-state device, it has no moving parts; the only movement involves the order of the emitted laser beams, emitted sequentially from the bottom upward. The Helios 32 LiDAR, in contrast, covers a full 360° view by rotating while scanning. It operates at frequencies of 5, 10, or 20 Hz. As rotating LiDARs require a glass casing around the entire device, the Helios is somewhat larger and heavier than the M1P. Due to its 70° vertical FoV configuration, it provides a substantial number of points close by. However, the sparsity of the pointcloud increases rapidly at greater distances. A visualization of the scanning patterns and field of view of the three LiDARs can be found in Figure 3.7.

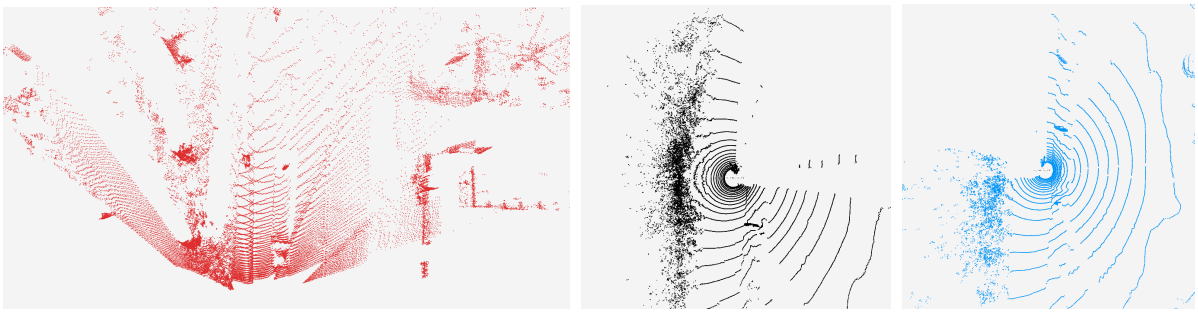
LIDAR	Rotating	Min-Max Range (m)	Horizontal Resolution	Field of View (H x V)	# of Scan Lines	Frame Rate (Hz)	Accuracy @100 *m*
Robosense M1 Plus	✗	0.5 - 200: 150 @ 10% NIST	0.2°	120° x 25°	125	10	±2.5 *cm*
Robosense Helios 32	✓	0.5 - 150: 110 @ 10% NIST	0.1° - 0.4°	360° x 70°	32	5, 10, 20	±2 *cm*

	Max Points / s	Weight (kg)	Dimensions (mm)	Power (W)	*Time sync: NTP / PTP*	*Phase Lock option*
Robosense M1 Plus	750k	0.75	45 x 110 x 108 (H x W x D)	18	✗ / ✓	✓
Robosense Helios 32	576k	1.0	100 x 100 (H x ∅)	12	✗ / ✓	✓

Table 3.3: Specifications of Robosense M1P and Helios 32 LiDARs

3.1.3. Cameras

The *SenseBike* is equipped with three cameras: two for capturing the surroundings and one for filming the rider. The two cameras dedicated to the surroundings are ArduCam IMX477 12MP models, which offer high-resolution imaging specifically designed for scientific research. Currently, the rear camera features an adjustable wide-angle lens, while the front camera is fitted with a fixed-focus lens. Both ArduCams are connected solely via HDMI cables. The gaze camera, which records the rider, connects through USB-C and captures 6MP images, facilitating research on human responses during cycling under various conditions.



(a) (b) (c)

Figure 3.7: Scan patterns of: (a) M1P, (b) Helios L, (c) Helios R, when mounted on the *SenseBike*.

3.1.4. GNSS / RTK Module

The ArduSimple simpleRTK2B is a high-precision GNSS receiver designed for advanced positioning applications. At its core, the U-blox ZED-F9P module provides multi-band GNSS support (including GPS), enabling centimeter-level accuracy in real-time kinematics (RTK) navigation. The module is connected to the GNSS antenna (Figure 3.3: *6*). It can provide a relative location to the Magnetic North pole with a frequency of 7 *Hz*.

3.1.5. IMU

Between the two Helios LiDARs in the rear, the Internal Measurement Unit (IMU) is housed in a specially designed 3D-printed box. This box also contains a Teensy 4.0 microcontroller, which the IMU is directly connected to. The IMU, a SparkFun 9DoF ICM-20948 (Qwiic), measures linear acceleration *(x, y, z)* in m/s^2, angular velocity *(x, y, z)* in rad/s, and magnetic field *(x, y, z)* in mT, providing nine degrees of freedom. The Teensy microcontroller is connected to the onboard computer through USB-C. It is capable of transmitting its measurements at a frequency of 100 Hz.

3.2. Software

The installed software used for recording and storing inputs from the sensors is done by launching docker containers with ROS2, which can be launched through a Web User Interface (WebUI). This section discusses the used architecture and explains the contents of all containers.

3.2.1. Architecture

The bicycle is equipped with multiple sensors, each requiring a specific driver that is only compatible with certain Linux distributions. This situation leads to dependency conflicts when trying to run all drivers on a single system. Consequently, the original manufacturers (BoReal) of the *SenseBike* have decided to containerize the workspace for each sensor. This approach ensures that each container has its own Linux distribution, Robot Operating System (ROS) distribution, Python/C++ version, or any other essential software package needed for the sensor to operate correctly. Detailed discussions on each container are provided in subsection A.2.2 and subsection A.2.3.

To make recording accessible, the onboard computer can be controlled through a Python Flask web user interface (WebUI), accessed when connected to the WiFi network provided by the bicycle's router. Users can start the desired Docker containers through this WebUI, as shown in Figure 3.8. For each container that starts running, we execute a specific script that builds, launches, and runs the necessary components. For example, when the IMU filter container starts, it runs a script that: 1) Builds the required ROS2 environment, 2) Starts the ROS2 IMU-filter package, which opens the IMU-filter node, and 3) Closes the node when the container is stopped.

One of the containers hosts a ROS2 environment equipped with the ROS2-bridge package [48], which allows users to monitor through the WebUI which drivers are publishing data to the ROS system. Another container runs the Precision Time Protocol (PTP) package, named `linuxptp`. When the *Start Recording* button is pressed in the WebUI, a separate container is launched to run the ROS2 bag record package, saving all the desired ROS2 data.

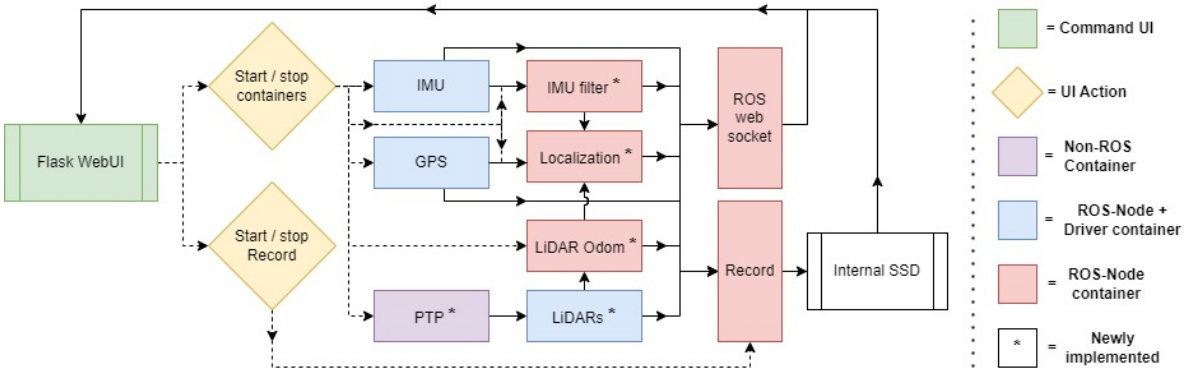


Figure 3.8: Overview of SenseBike software architecture.
A dashed line indicates software start-ups, while a solid line represents data flow. Boxes with a * were not originally present on the bicycle, and are proposed in this work.

3.3. SenseBike Limitations

As the SenseBIke is still in its preliminary stages, there are still some limitations, in both hardware and software. These will be addressed in future updates.

Hardware

As described in Section 4, we experience relative rotations between the front and rear LiDARs during recording. This issue is primarily caused by the luggage carrier not being fully rigid, even though it supports a heavy battery pack, LiDAR mount, and two LiDARs. To address this, we are experimenting with relocating the battery pack to a more central position and reducing the weight of the LiDAR mount.

Software

On the software front, we are developing an update to incorporate the cameras into the ROS environment. This enhancement will enable the recording of ROSbags with time-synchronized camera messages and their corresponding *.jpg* images. At present, we can only provide 25 FPS *.mp4* videos of entire recordings.

Additionally, the localization package and its associated ROS nodes are not yet operational in real-time. While the localization package is accurate when provided with at least one type of odometry, the current method using odometry calculated by KISS-ICP [62] from the LiDAR points is computationally intensive and cannot be performed in real-time. The forthcoming update will include odometry derived from the cadence or speed sensor, which measures pedal or wheel rotational speed, respectively.

Lastly, there are two potential improvements for the IMU data. First, its data in the ROS environment is slightly delayed, likely due to the limited computation speeds of the Teensy Module. Second, we are observing excessively high z-acceleration values when riding over small bumps or dents. Although it is reasonable to expect higher values due to the nature of a bicycle compared to a car, we are recording values exceeding 20 m/s^2. These issues will be addressed in future updates.

4

Methodology

4.1. SenseBike Dataset

As no other dataset from a bicycle's perspective is present, we collect our own, naming it the *SenseBike Dataset*. This dataset will continue to grow in size, quality, annotations, modalities, and applicability in future research. This section explains how we collected this data and processed it into one unified dataset.

4.1.1. Data collection

Data collection was conducted entirely in and around Delft, the Netherlands, as shown in Figure 4.1. When possible, the SenseBike operated on designated bicycle paths or lanes. In areas such as the Delft city center, where there are no dedicated bicycle paths, the bike's position varied depending on other road users or obstacles.

All data was recorded in clear weather conditions without any precipitation, between 8:00 AM and 6:00 PM, over three separate days. Rather than pre-selecting specific routes, we covered various parts of the town to capture a diverse range of road and urban designs, as well as variations in the density of Vulnerable Road Users (VRUs). We aimed to avoid visiting the same locations repeatedly, except for some with a higher likelihood of cyclist presence. One example is the bicycle lane in front of the Mechanical Engineering faculty at TU Delft.

The sequences vary in duration, ranging from approximately 1.5 to 15 minutes. Currently, the dataset contains 24 sequences, with a total duration of 2 hours and 25 minutes. Each sequence includes LiDAR, Internal Measurement Unit (IMU), and GNSS data. In total, the dataset contains 77,474 combined LiDAR point clouds, with a maximum density of 1.94 million points per second. A more detailed statistical overview of the recorded data can be found in Table B.2. All specifications of the recording vehicle are stated in Chapter 3.

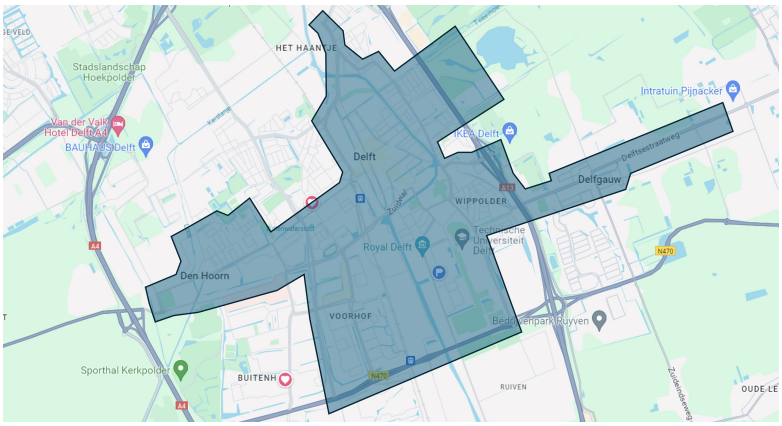


Figure 4.1: SenseBike Dataset recording Area, background taken from [20].

4.1.2. Data processing

We process the recorded data according to the framework in Figure 4.2. The *SenseBike* has three LiDARs, two rotating ones in the back, and one solid-state in the front, providing three separate pointclouds. The reasoning for this specific setup is explained in Chapter 3. Due to the availability of only one GPU on the bicycle, we aim to merge these three separate pointclouds into a single one, enabling real-time object detection or segmentation. This requires calibration, time synchronization, and ego-motion compensation. The latter will be referred to as de-skewing. All parts of the framework are discussed in more detail below.

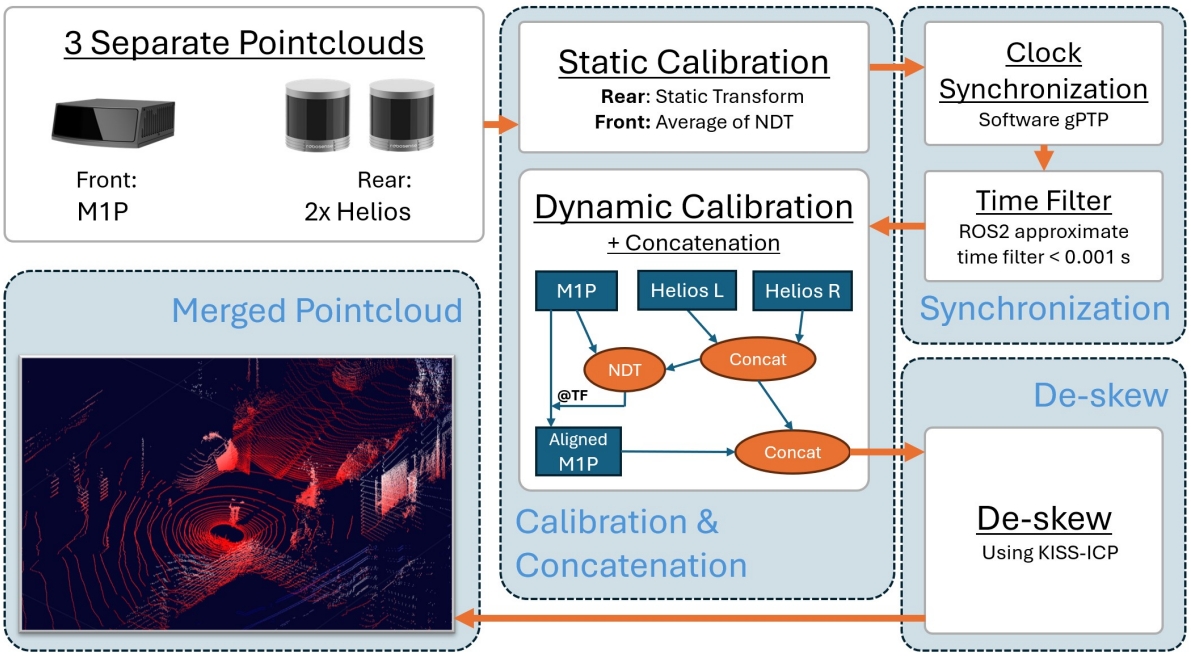


Figure 4.2: Overview of our data-processing framework.

Static Calibration

We calibrate the LiDARs to ensure they are accurately aligned during recording. Calibration involves calculating the transformation matrices between all LiDARs and transforming all points into one coordinate system attached to the *SenseBike*. The used coordinate system is shown in Appendix B.1. Similar to the multiple-LiDAR calibration often used on car-based data collection, we start with an initial estimation of the transformation matrix and run an alignment algorithm. In our case, we run a Normal Distribution Transform (NDT) [7] algorithm 50 times and apply the average of the 50 outcomes. NDT iterates until a certain threshold in mean squared point distances is reached. More details are provided in Appendix B.4.

Synchronization

To merge point clouds containing dynamic objects or recorded from a dynamic point of view, precise timestamps for each point are essential. This necessitates the synchronization of the internal clocks of all LiDAR sensors. We achieve synchronization using the generalized Precision Time Protocol (gPTP) implemented by `linuxptp`. The on-board Linux computer functions as the grandmaster clock, enabling the LiDARs to synchronize with its system clock. Each LiDAR measures its latency relative to the system clock, with latencies observed up to only ±0.01 ms. This latency can potentially introduce merging inaccuracies corresponding to the distance traveled by the ego-vehicle during this time, which is approximately 0.04 mm and considered acceptable, as it is significantly lower than the measurement inaccuracies of the LiDARs itself (Table 3.3). When all clocks are synchronized, precise timestamps for every point collected by the LiDARs can be determined.

Despite the synchronization of the internal clocks of the LiDARs, the exact moments at which each LiDAR begins its scan can still differ. Both the M1P and the Helios 32 models feature a phase-lock

option that controls the phase angle across multiple LiDARs, aiming to synchronize their scan start times. However, this phase-lock synchronization sometimes fails or takes time to initialize. This delay can lead to mistimed recordings and the misalignment of moving objects within the field of view (FoV) of two or more LiDARs, even after applying de-skewing techniques. An illustration of such a scenario is shown in Figure 4.3.

To address this problem, we remove any instances where such delay occurs and remove point clouds from all LiDARs if the timestamp difference exceeds 1 ms. Furthermore, if this discarding process happens more than three times consecutively, the entire recording is discarded. Notably, a 1 ms deviation translates to an accuracy deviation of roughly 0.4 cm when traveling at a speed of 15 km/h. While static objects are not affected as we eventually de-skew based on timestamps, this can cause unwanted distortions in dynamic objects.

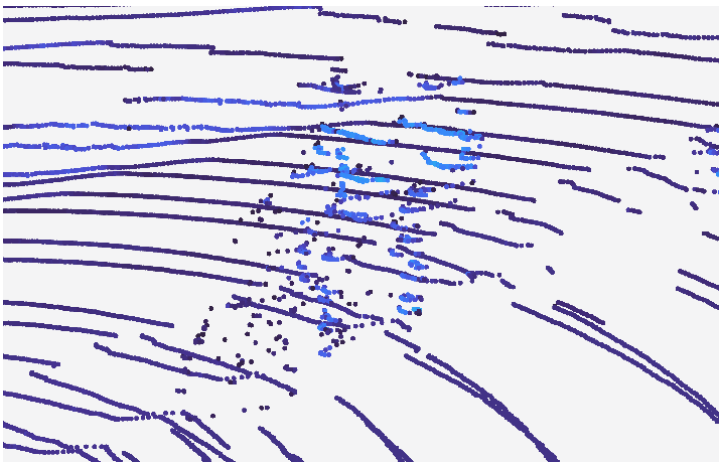
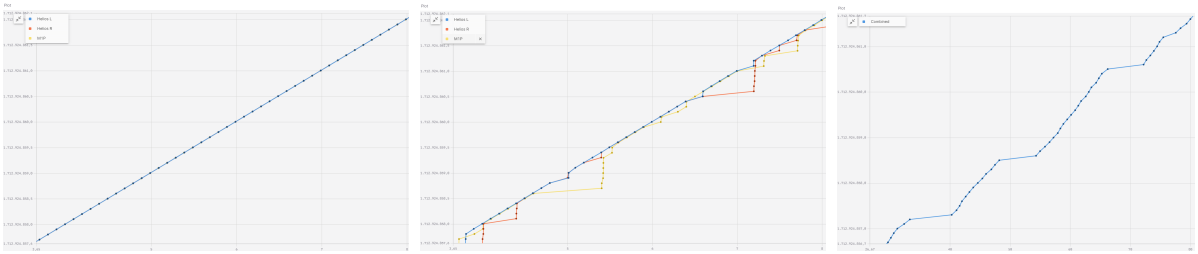


Figure 4.3: Misalignment of a single cyclist in the FoV of the two rear LiDARs, due to mistimed scan starts of these two.

As Figure 4.4 shows, the used LiDARs do not transfer their recordings all at the same time (b), even though the actual scans do have a similar timestamp (a). We, therefore, implement a ROS2 Approximate time sensor, which stores the single-LiDAR pointclouds, until all the other corresponding LiDAR pointclouds have arrived. Its effect is shown in (c).



(a) (b) (c)

Figure 4.4: a) Message timestamps of the three LiDARs, after PTP. b) Actual Receiving times of the three LiDARs. c) Receiving timestamps of published combined pointcloud messages after the use of a ROS message filter.

Dynamic Calibration + Concatenation

As expected, recording from a bicycle, in contrast to recording from a car, presents new challenges. One challenge is the difference in stiffness between a bicycle frame and a car frame. While visualizing some test rounds, we observed that static calibration alone was insufficient: the front LiDAR rotates relative to the rear LiDARs during turns, caused by the bicycle frame's lack of rigidity. Specifically, the luggage carrier tends to roll slightly relative to the stiffer frame due to yaw or roll accelerations.

We address this issue through a method we name dynamic calibration. We apply the NDT algorithm [7], similar to that used for static calibration. Since the two rear LiDARs remain stable relative to each other during motion, we only align the front LiDAR to a concatenation of the two rear LiDARs. The

package used is a modified version of [49], which is an optimized variant of the NDT algorithm from the PCL_ROS package [47]. We aim for a maximum Euclidean Distance fitness of 0.1, consistently achieved after five iterations. This fitness does not mean that we have inaccuracies of 10cm, as one could suggest. Due to the different nature of scanning patterns, this fitness is established by visually inspecting and it leads to inaccuracies similar to the inaccuracies of the measurements of the LiDARs itself. Five iterations of the NDT algorithm are implementable in real-time on a laptop CPU (11th Gen Intel(R) Core(TM) i7-11370H @ 3.30 GHz). To ensure extra certainty and maximize hardware utilization, we run seven iterations of the NDT algorithm per incoming scan from the three LiDARs.

There are two different methods to execute the concatenation and de-skewing blocks: either by de-skewing each point cloud separately before concatenation or by concatenating all point clouds first and then de-skewing the resulting single point cloud. Both methods were evaluated, and no significant visual differences were observed. Consequently, we selected the method with the lowest computational demands. This method, which involves concatenating first and then de-skewing, is illustrated in Figure 4.2. Considering that the de-skewing process is computationally intensive, regardless of point cloud size, it is more efficient to perform it once rather than three times.

De-skewing

We now have a single merged point cloud free from alignment errors caused by inertial frame rotations. However, an additional challenge remains: ego-motion during scanning, also referred to as skewing. A single scan of each LiDAR takes approximately 0.1 seconds. During this interval, at a speed of around 15 *km/h*, we skew approximately half a meter. Consequently, it is necessary to either translate the first point scanned by one of the LiDARs by about half a meter in the direction of travel or adjust the position of the last point by half a meter in the opposite direction. This procedure, referred to as de-skewing, is essential. To determine our location change and perform de-skewing, we utilize KISS-ICP [62]. It takes two sequential point clouds as input, estimates the odometry between them, and relocates the scanned points accordingly. Following this step, we obtain a single, concatenated, and de-skewed point cloud, as visualized in Figure 4.5, which will be referred to as the *merged* point cloud.

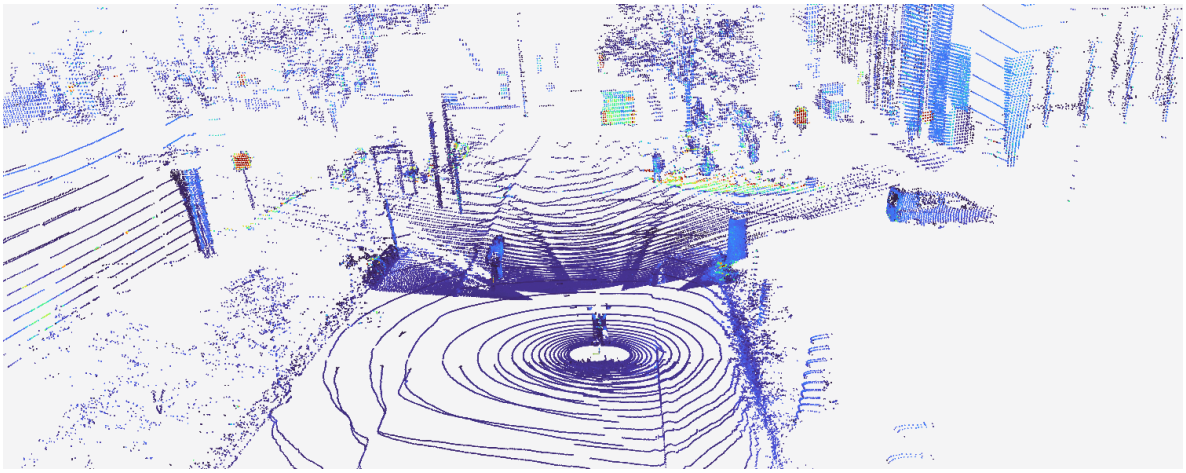


Figure 4.5: Isometric view of a merged pointcloud, visualized in Foxglove.

4.1.3. Validation set

For evaluation purposes, we provide a small validation set of 600 annotated combined LiDAR frames, including 2 sequences, both 30 seconds long. All vehicles, pedestrians and cyclists within a range of [x_{min}, y_{min}, z_{min}, x_{max}, x_{max}, x_{max}] = [*-20 m, -20 m, -2 m, 60 m, 20 m, 4 m*] are labeled with 3D bounding boxes, containing [x_{center}, y_{center}, z_{center}, l, w, h, θ], where *l* = length, *w* = width, *h* = height and θ = yaw or heading angle.

We utilized the lidarLabeler tool from MATLAB [57] to create the 3D labels. However, this process took an unexpectedly long time. We spent nearly 70 hours labeling just 60 seconds of data due to two main issues:

1. **Ineffective ground-plane removal:** The MATLAB lidarLabeler, like most 3D labeling software, is designed to streamline the labeling process by removing the ground plane. This allows users to select a cluster of points, and the software automatically draws a fitting box around the selected cluster. However, ground plane removal algorithms typically assume that the ego-vehicle remains level without any roll. Since our bicycle experiences considerable roll while in motion, this made it challenging to accurately select clusters. One of the algorithms that can assist with labeling is a cluster-tracker; which was therefore useless for our dataset.
2. **High yaw variations:** Another commonly used algorithm that assists in labeling is an interpolator. It works by annotating two non-sequential frames, and it interpolates the boxes in between these two frames. The interpolator is effective only when the relative yaw angles between the ego-vehicle and the object being labeled are small. On a bicycle, we experience greater variations in yaw compared to a car, making this interpolating function ineffective.

Both issues resulted in having to label each frame individually, without the assistance of any automation tools. We annotated 600 frames, resulting in 599 vehicle labels, 453 pedestrian labels, and 2036 cyclist labels. An insight into the process of labeling a pedestrian is illustrated in Figure 4.6.

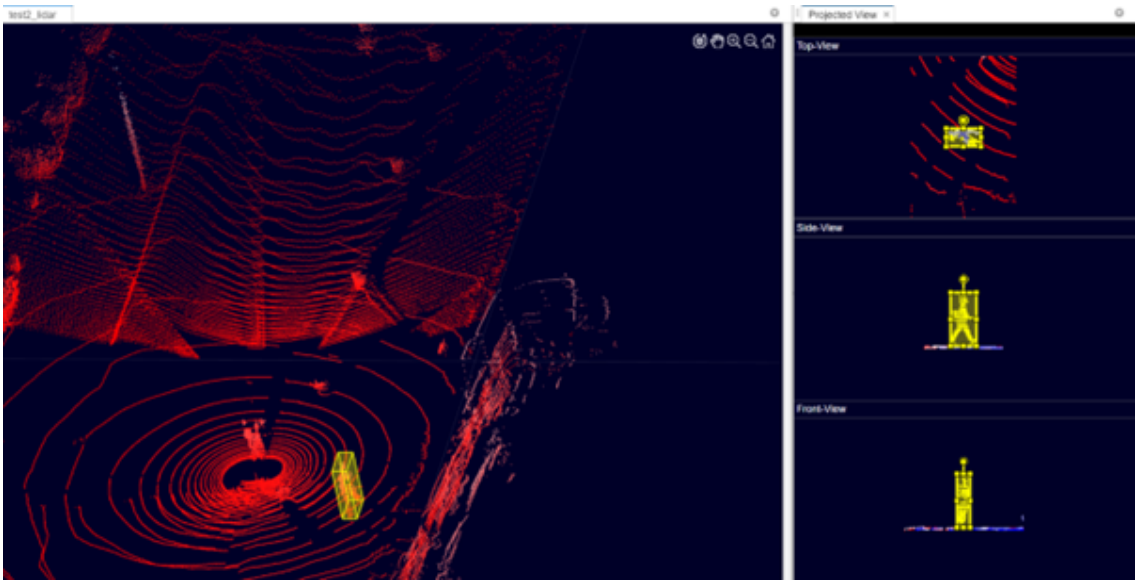


Figure 4.6: Insight in the labeling process of a pedestrian in a single frame. Screenshot of the lidarLabeler from Matlab.

4.2. Domain Adaptation

Our main research question is to study whether our new dataset can contribute to an increase in object detection performance. To evaluate this, we apply an offline unsupervised domain adaptation (UDA)pipeline. This contains a pseudo-labelling method (teacher) and the training of a detector (student) on these pseudo-labels. We then evaluate this student's performance on our validation set, discussed in Section 4.1.3.

As our pointcloud contains two different types of LiDARs - a rotating and a solid-state one - we decide to use and adapt the MS3D++ framework from Tsai et al. [59] for creating pseudo-labels. The original method is discussed in more detail in Section 2.4.2. MS3D++ focuses on fusing knowledge from multiple datasets and detectors into these pseudo-labels. An overview of the existing method can be found in Figure 4.7. This framework initially focuses solely on pedestrians and vehicles; thus, we introduce pseudo-labeling for cyclists, which we describe in this section.

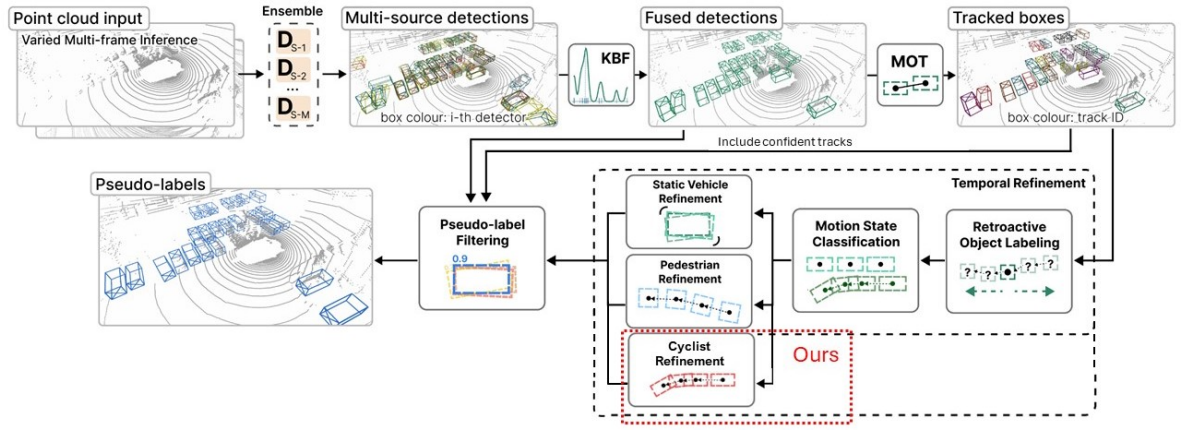


Figure 4.7: Adapted MS3D++ framework for creating Pseudo-Labels, copied and adapted from [59].

4.2.1. Pseudo-labeling of cyclists

The process of implementing cyclists in the original method is an iterative one. The thresholds are based on trial and error, by comparing pseudo-label sets with different thresholds. To generate them, we use the multi-source detections and apply the same Kernel Density Estimation (KDE) Box Fusion (KBF) as in the original method. From there, we apply the decision tree scheme from Figure 4.8.

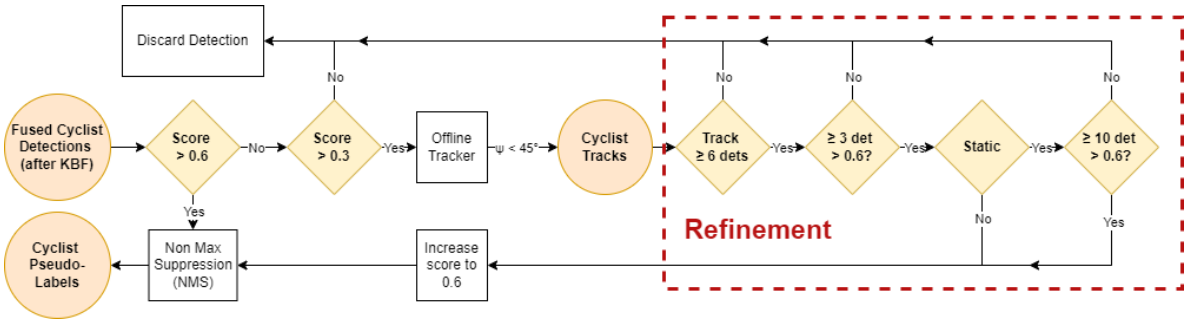


Figure 4.8: Cyclist Pseudo-labeling method

All cyclist detections with scores higher than 0.6 are used in the final pseudo-label filtering with Non-Max suppression. All cyclist detections within the confidence score range of [0.3,0.6] still contain a lot of True Positives. The challenge is to separate the accurate boxes from the *"myriad of false positives within this range"* [59]. To separate these, we run the original Kalman Filter-based multi-object tracker (MOT) [43]. We discard a track if it ranges shorter than 6 frames. Then, if a track contains less than 3 detections

above the confidence score threshold of 0.6, we discard the entire track. The remaining tracks are then classified into static or dynamic, based on a maximum difference in begin-to-end distance, in our case $1m$. The reason for this is explained later in this section. If a track is classified as static, we discard it if less than 10 detections have a confidence score below 0.6. For all the remaining tracks, we increase the confidence of each detection to 0.6 and pass them to the Pseudo-label Filtering block, which performs Non-Max Suppression (NMS), eliminating redundant or overlapping boxes based on their confidence scores. In the final step, we gather all predictions that have now confidence score higher than 0.5 and use these as a pseudo-label set.

Issues & Refinement:
When visually inspecting the first generation of cyclist pseudo-labels, we identified three issues that consistently occurred to labels within the confidence range of 0.3 to 0.6:

1. **Cyclist as Cyclist + Pedestrian:** As Figure 4.9 shows, cyclists often tend to be pseudo-labeled as both a cyclist and a pedestrian. This only happens to actual cyclists, not vice-versa to actual pedestrians.
2. **Bicycle as Cyclist:** We often noticed that parked bicycles, which we do not classify as cyclists (Appendix D), were classified as such. An example is shown in Figure 4.10
3. **Incorrect Heading Angle:** It regularly happens that a cyclist, when only seeing its back, gets a pseudo-label with a matching centerpoint, but incorrect heading angle, as illustrated in Figure 4.11.

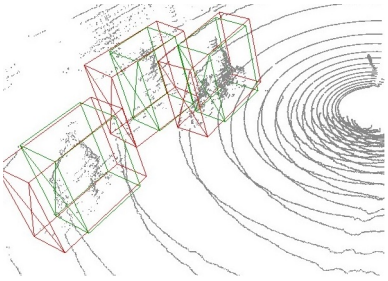


Figure 4.9: Cyclists pseudo-labelled as Cyclist (red) + Pedestrian (green)

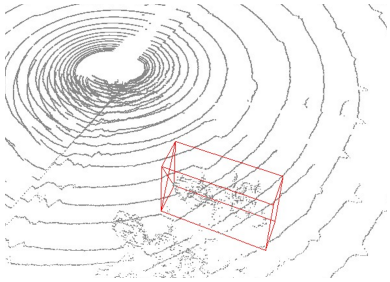


Figure 4.10: Bicycle as Cyclist, which we do not classify as such

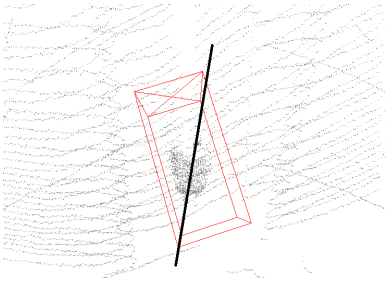


Figure 4.11: Incorrect Heading angle; Black line is actual heading angle

To tackle these issues, we implement the following methods:

1. As this issue only happens to actual cyclists, and not to actual pedestrians, we solve this solution by removing overlapping pedestrian boxes with a 2D Intersection of Union (Appendix D.2) higher than 0.5.
2. Using the tracker, we determine if a cyclist is static or dynamic based on the distance they travel. As illustrated in Figure 4.8, we apply stricter criteria to static cyclists. Since a cyclist can remain stationary, such as when waiting at a traffic light, we do not want to exclude them entirely.
3. We apply a maximum heading angle difference of 20° within two sequential cyclist detections, tracked by the MOT.

4.2.2. Pseudo-label sets

To fully examine the impact of our cyclist refinement, we generate four sets of pseudo-labels. The exact details on the detector ensembles are listed in Appendix C.1. The names of these pseudo-label sets, which are used in the experiments described in Chapter 5, include:

- **STND**: Off-the-shelf Standard (STND) MS3D++ [59], with a detector ensemble (DE) fully provided in their *GitHub*. This DE includes a *Voxel-RCNN* backbone [16] with either an Anchor or Center detection head, trained on *Lyft* [32], *nuScenes* [8], or *Waymo* [55], resulting in six different detector models. Each detector is used six times: three times with and three times without test-time augmentations (TTA) and with multi-frame inferences of 1, 2, or 4. Altogether, this results in 30 sets of detections. Unlike the original framework, which discards all cyclist detections made by the detector ensemble, we include them here and use them in the *Pseudo-label Filtering*.

- **STND+**: This pseudo-label set is similar to *STND*, but uses a larger detector ensemble. We include detections from other 3D detection models, such as *CenterPoint* [72], *PV-RCNN++* [53], and *Part-A*^{2} *net* [51], also trained on *KITTI* [33], and an *MS3D++*-self-trained *Voxel-RCNN* model, trained on a 128-beam LiDAR dataset. These additions lead to 49 different sets of detections.
- **Ours**: Our pseudo-label set is produced similarly to STND+, but it now incorporates the proposed cyclist refinement discussed in Section 4.2.1. Also the detector ensemble remains similar to that used in *STND+*.

4.2.3. Transfer Learning

We apply a transfer learning scheme to achieve the domain adaptation, meaning we train off-the-shelf 3D LiDAR object detection algorithms (detectors) on our generated pseudo-labels. We choose three *OpenPCDet* integrated detectors: 1) Voxel-based detector *VoxelRCNN* [16], 2) point-based detector *CenterPoint* and pointvoxel-based detector *PV-RCNN++* [53], as described in section 2.2. We evaluate their performance on our validation set, from Section 4.1.3. An overview of the applied scheme is shown in Figure 4.12. Since we have three different sets of pseudo-labels and three different detectors, we will end up with nine distinct detector models.

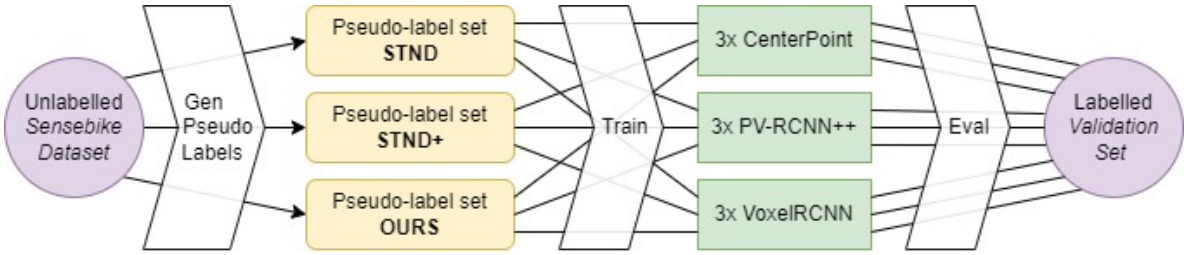


Figure 4.12: Transfer Learning Scheme

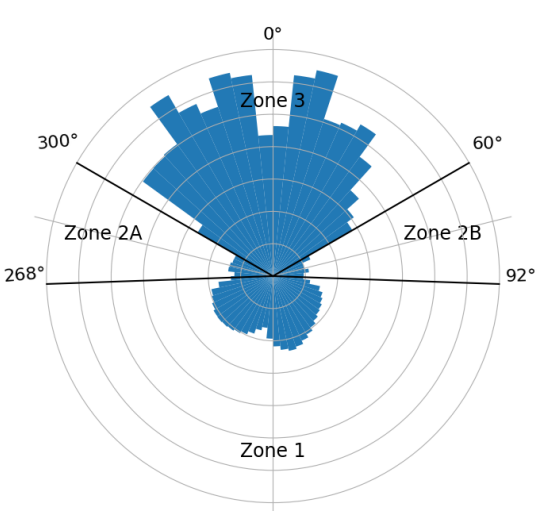
5 Experiments

5.1. SenseBike Dataset

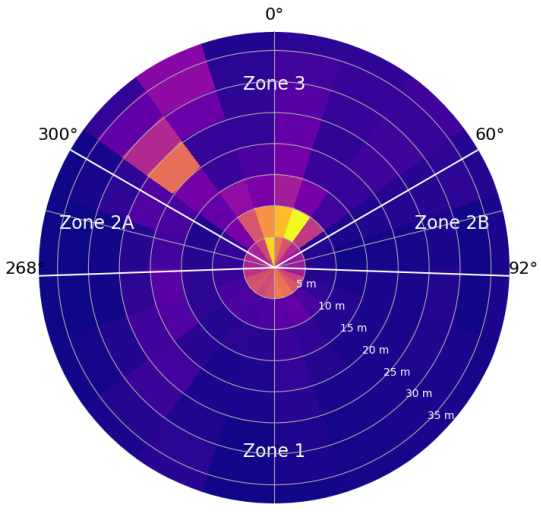
The dataset we work with for the experiments in this section contains 77,434 *.pcd* pointclouds, each with a theoretical maximum of 196*k* points, depending on how many laser beams are reflected. We use recorded sequences from three different days, with a minimum length of 93 seconds to a maximum of 907 seconds. The sizes of these are 4.96 GB and 46.25 GB, respectively. An in-depth overview of the names, lengths, and sizes for all recorded sequences can be found in Table B.2. In this section, we look at the distribution of points and the influence of roll on our data.

5.1.1. Point Distribution

The isometric visualization of a recorded point cloud, as depicted in Figure 4.5, clearly illustrates that the point distribution is not uniform across the entire *xy*-plane, with *z* representing the altitude in the gravitational direction. This variation is due to the three-LiDAR configuration: a dense, small FoV, solid-state LiDAR at the front, and two sparser, rotating LiDARs at the back. Consequently, we analyze the point distribution per azimuth angle, as shown in Figure 5.1a. We select 600 random frames from the dataset and plot a circular histogram of this distribution. The same 600 random frame distribution is depicted in Figure 5.1b, including Euclidean distance up to 35 meters, where most points are reflected.



(a) Point distribution over Azimuth angle, averaged over 600 random frames.



(b) Point distribution over Azimuth angle and Euclidean distance up to 35 meters, averaged over 600 random frames.

Figure 5.1: Point distributions over Azimuth angle and Euclidean distance. The top of the graphs (0°) corresponds with the front (positive X) of the bicycle.

Discussion
It can be seen that a majority of the points are distributed facing the front (0°), in the FoV (-60°- +60°) of the M1P LiDAR. We will refer to this part of the pointcloud as Zone 3. Two clear zones contain the least points, on the sides, referred to as zones 2A and 2B. These two zones include LiDAR points of only one of the Helios LiDARs, due to occlusion of one of the other or the rider. A visual explanation is given in Figure 3.7b and Figure 3.7c. We refer to the zone including points from both the Helios LiDARs as zone 1. The final result contains four zones, with borders at +60°, +92°, +268° (= -92°) and +300° (= -60°). Figure 5.1b indicates that the majority of points are reflections from objects within a 5 to 10-meter range, from the M1P LiDAR (zone 3), with the number of points decreasing as the distance increases. This outcome is expected since reflections from the ground plane are also included. The two rear LiDARs (zone 1) reflect fewer points at greater distances due to their increased sparsity with distance.

5.1.2. Bicycle Dynamics

Our dataset is unique as it is collected from a bicycle, introducing distinct dynamics compared to recordings from a car. When making turns, a bicycle needs to lean to the side, resulting in an ego-vehicle roll angle. The used definition for roll angle is visualized in Figure B.1. We inspect the roll angle in combination with the driven path for two different types of sequences. We examine a sequence that only contains a straight-line track and compare it to a sequence that includes a 90° turn. The annotated test sequences *2024-05-02_16-26-45* and *2024-04-30_15-24-18* fulfill these requirements. The roll angle is calculated by comparing two sequential odometry transformation matrices, which we compute using KISS-ICP [62]. While driving straight in sequence *2024-05-02_16-26-45*, we assume the average roll angle is 0° and level it accordingly. For sequence *2024-04-30_15-24-18*, we assume the average roll after the first 100 frames equals 0° and level it out correspondingly. Specifically, the right-hand turn occurs in the first 50 frames; from there on, the desired track is only in the forward x-direction.

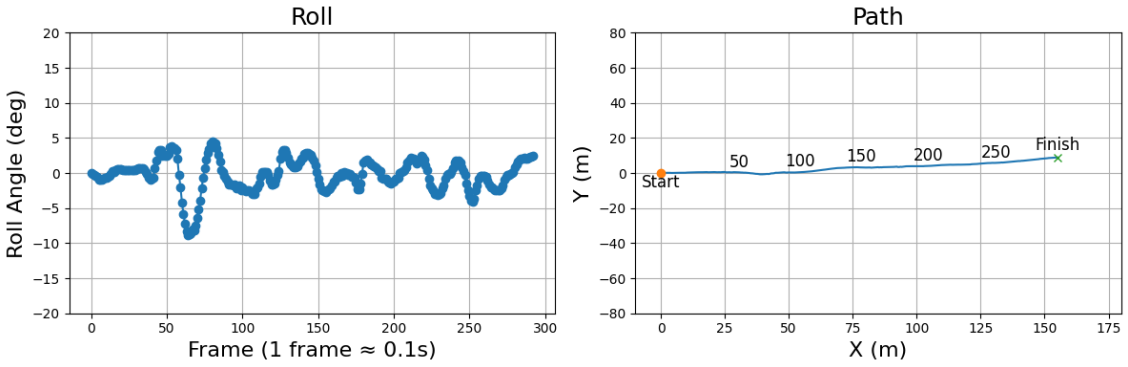


Figure 5.2: Roll & Driver path for a 30-second sequence without any turns or bends. Right plot numbers on the line indicate the frame number. Sequence: 2024-05-02_16-26-45.

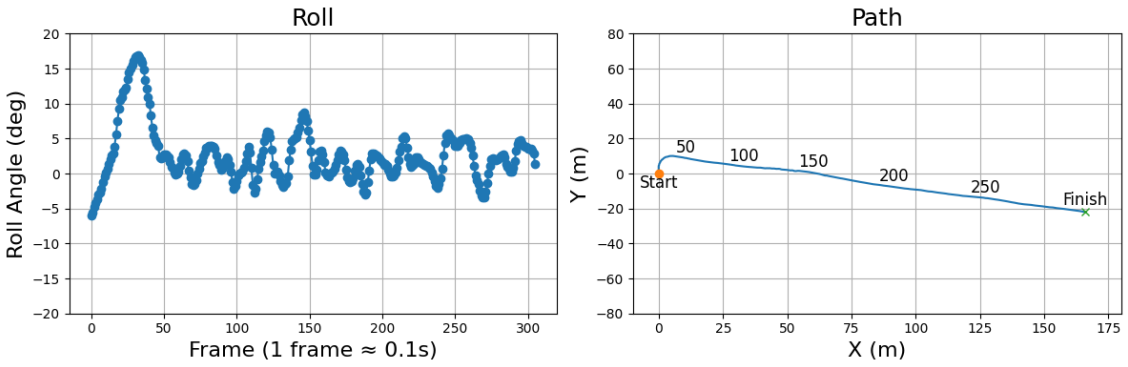
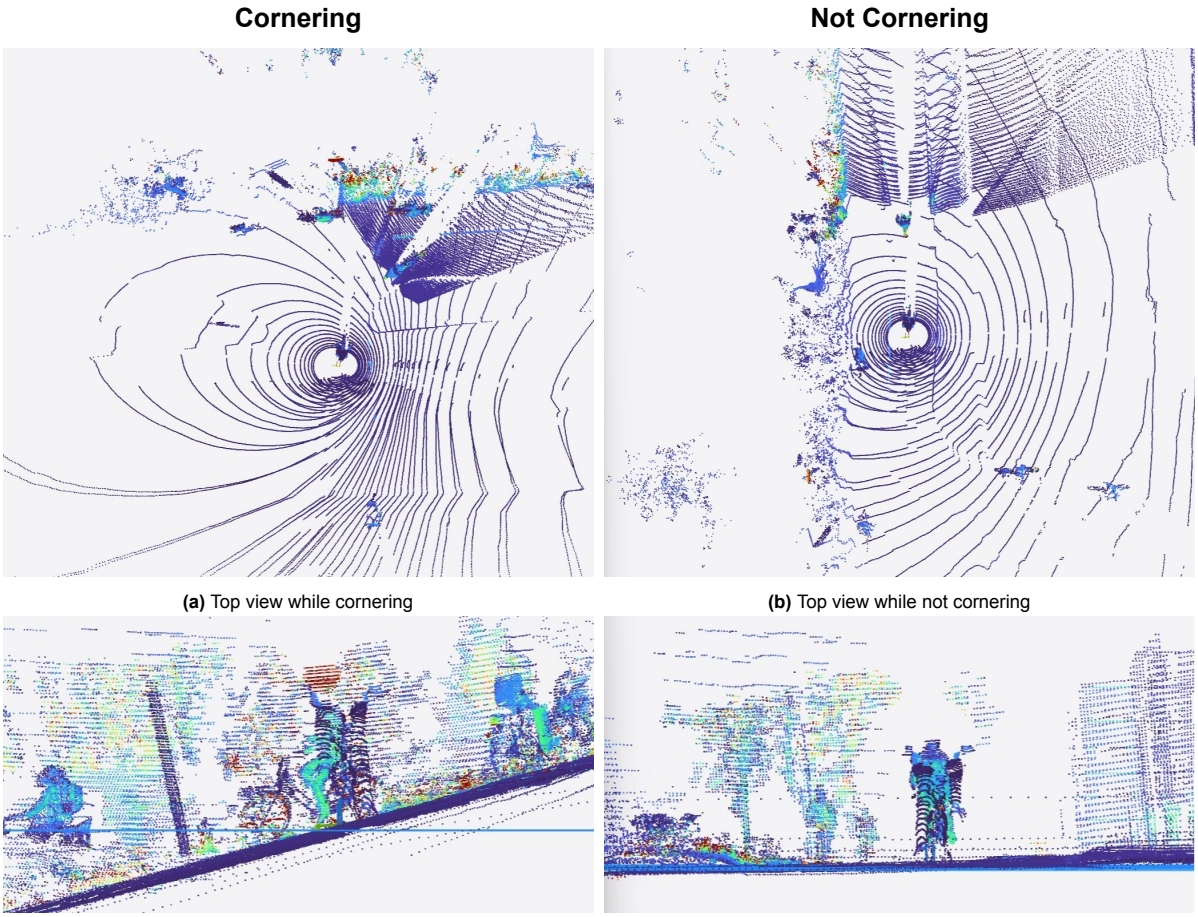


Figure 5.3: Roll & Driven path for a 30-second sequence with a > 90° right-hand turn in the first 50 frames. Right plot numbers on the line indicate the frame number. Sequence: 2024-04-30_15-24-18.

We also examine the change in scan pattern during bicycle roll, which is present while cornering. For that, we compare the top and rear view of the turn made at the first 50 frames of sequence *2024-04-30_15-24-18*, which is plotted in Figure 5.3. These views are displayed in Figure 5.4.



(a) Top view while cornering

(b) Top view while not cornering

(c) Rear view while cornering. The blue line indicates the $z = 0$ plane, meaning the roll angle ≈ 15°.

(d) Rear view when not cornering. The blue line ($z = 0$), is aligned with the points on the ground, meaning the roll angle ≈ 0°.

Figure 5.4: Comparison of scanning patterns during cornering, which involves exerting roll, versus the same views immediately after cornering.

Discussion

Figure 5.2 and Figure 5.3 illustrate the results for the straight-line ride and the right-hand turn ride, respectively. For comparison purposes, the x and y limits of both sequences are kept constant. It is noteworthy that the bicycle experiences roll even on the straight track, with roll angles up to 10°. The roll angle in Figure 5.3 reaches up to 16.5° during the right-hand turn. By contrast, the Waymo dataset [55], recorded from a car, exhibits maximum roll angles of ±1.0°.

The impact of this roll on the scanning pattern is clearly visible, as shown in Figure 5.4. When leaning the bicycle for a turn and thus rolling, the rear LiDARs (Helios) change their point distribution: the side the bicycle leans towards receives a denser distribution of points nearby and fewer points further away. The opposite occurs on the other side. As a result, other road users and objects are likely to appear sparser or denser depending on which side they are on during cornering. The same phenomenon happens with the front LiDAR (M1P). Since this lidar has more beams, the change in distribution is less noticeable.

5.1.3. Validation Set

The validation set contains 599 vehicles, 2036 cyclists, and 453 pedestrian labels. As we will later evaluate the quality of pseudo-labels and the performance of trained detectors per azimuth zone, we also examine the number of annotated labels per zone in the validation set. These are illustrated in Figure 5.5. We also visualize the effect of the annotated bounding boxes when cornering. Figure 5.6 shows the same view as Figure 5.4c, now with the bounding boxes included. There are only 453 pedestrians labeled in the validation set, many of which include two pedestrians visible only from the waist up, as shown in Figure 5.7. Specifically, 124 pedestrians from zone 3 come from these two difficult-to-detect pedestrians.

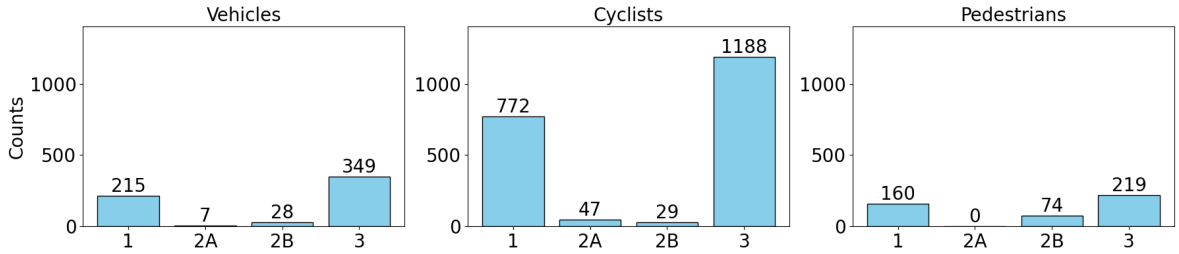


Figure 5.5: Number of annotations in the validation sets per azimuth zone, as described in Figure 5.1.

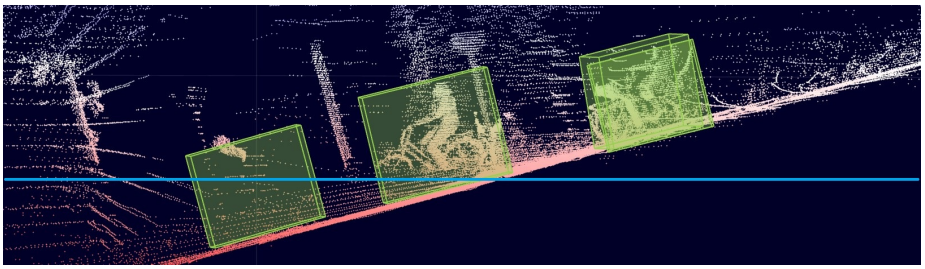


Figure 5.6: A visualization of the relative roll or pitch angle for the drawn bounding boxes on other road users when leaning in for a corner. The blue line indicates the $z = 0$ plane.

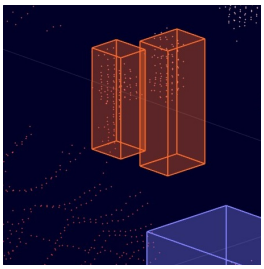


Figure 5.7: Two half-visible pedestrians.

Discussion

Figure 5.5 shows the imbalance of other road users in every azimuth zone. This is not a surprise, as zones 2A and 2B are relatively small compared to zones 1 and 3. Also, they are located on the left and right sides of the ego vehicle, which naturally contains fewer road users. This only happens when passing opposing traffic, or during overtaking. The combination of fewer LiDAR points and the presence of fewer objects in zones 2A and 2B will likely result in reduced detector performance in these areas. The high number of cyclists compared to pedestrians and vehicles is because 50% of the validation set was recorded on a bicycle-dense path.

It is clear that a 9 DOF bounding box, which includes the relative pitch and roll angles of an object, is needed to accurately represent another road user, such as the cyclist shown in Figure 5.6. This is, however, a problem while evaluating the quality of our pseudo-labels and the performance of our detectors in the following sections. Existing off-the-shelf methods only implement predictions and evaluations of 7 DoF bounding boxes, only including yaw and not pitch or roll. This does make a 2D Intersection over Union (Appendix D.2) unreliable, as it assumes that the center point of the bounding box is directly above the middle of the bounding box's ground plane.

5.2. Data processing

In this section, we focus on the impact of the calibration, synchronization, merging, and de-skewing of three separate point clouds into one, as introduced in Section 4.1.2.

5.2.1. Calibration

Static

The estimated static transformation matrices for the three LiDARs relative to the used coordinate system (Appendix B.1), as explained in Section 4.1.2, are:

Rear: Helios L

$$\begin{pmatrix} -0.99 & 0.00 & -0.01 & 0.00 \\ 0.00 & -1.00 & 0.00 & 0.20 \\ 0.01 & 0.00 & -0.99 & 0.92 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Rear: Helios R

$$\begin{pmatrix} -0.99 & 0 & -0.01 & 0 \\ 0 & -1 & 0 & -0.20 \\ 0.01 & 0 & -0.99 & 0.92 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Front: M1P

$$\begin{pmatrix} 0.99 & -0.035 & -0.032 & 1.51 \\ 0.035 & 0.99 & 0 & -0.04 \\ 0.034 & -0.0011 & 0.99 & 0.89 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Dynamic

The dynamic Normal Distributions Transform (NDT) algorithm [7] aligns the front LiDAR with the two rear LiDARs to compensate for internal bicycle-frame rotations, as we explain in Section 4.1.2. We examine the extent of this compensation, by looking at the transformation matrix (TM) the NDT algorithm produces after every input of pointclouds. This TM is the outcome of the 7 NDT iterations.

When cornering, the bike leans, increasing the roll angle. Sudden changes in this angle can lead to increased internal bicycle rotations. To study this, we compare the roll angle with the compensation needed for these rotations. Since compensations can be in 3D, we plot the roll angle against the three degrees of necessary rotation. Essentially, we investigate whether changes in the bike's roll angle, due to leaning during direction changes, result in internal rotations that the NDT algorithm must compensate for, including potential pitch or yaw angle adjustments. The results are shown in Figure 5.8. We use the same sequence for the calculation of roll as in Figure 5.2, but focus only on the part from frame 100 to frame 150, for better visuality.

We also examine the effect of the dynamic calibration with NDT by visualizing the differences of similar frames before and after in Figure 5.9.

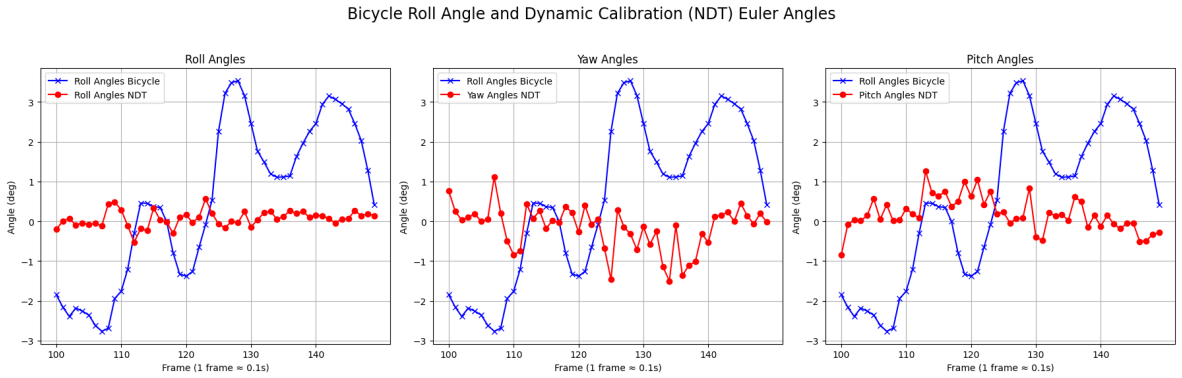
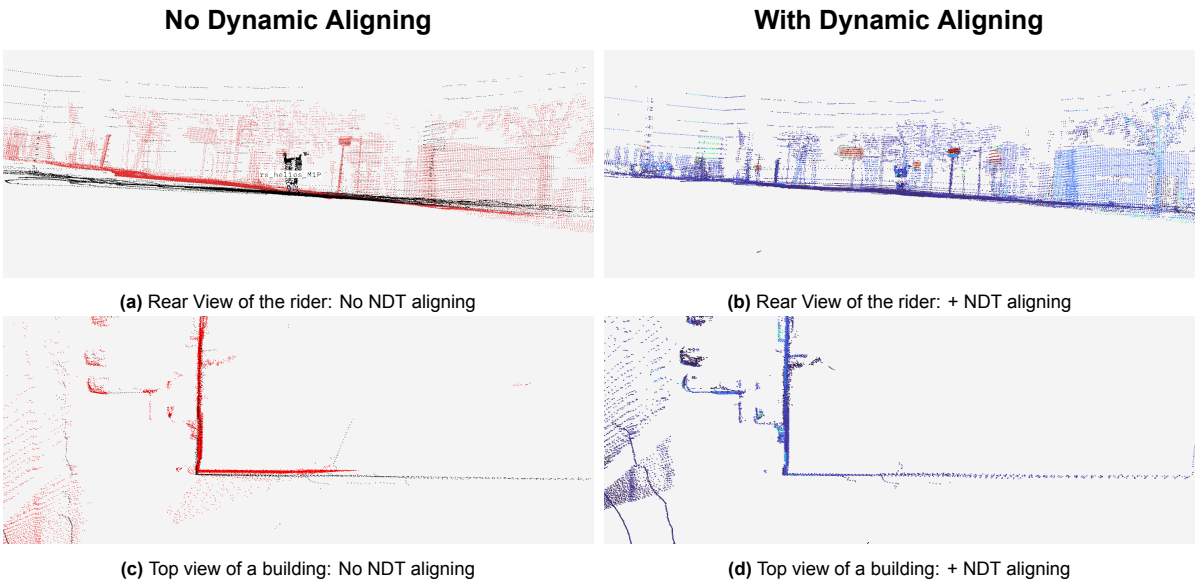


Figure 5.8: Bicycle roll (blue) for the NDT calculated transform in Euler Angles (red), best seen in color. Left: NDT Roll, Middle: NDT Yaw, Right: NDT Pitch. The frames are taken from the same sequence as in Figure 5.2, but we zoom in on frame 100 -> 150 for better visualization.



(a) Rear View of the rider: No NDT aligning

(b) Rear View of the rider: + NDT aligning

(c) Top view of a building: No NDT aligning

(d) Top view of a building: + NDT aligning

Figure 5.9: NDT aligning vs No NDT Aligning from the same recording. Left: No dynamic aligning, with the M1P LiDAR in red, and the two Helios LiDARS in black. Right: The combined pointcloud after NDT Aligning.

Discussion

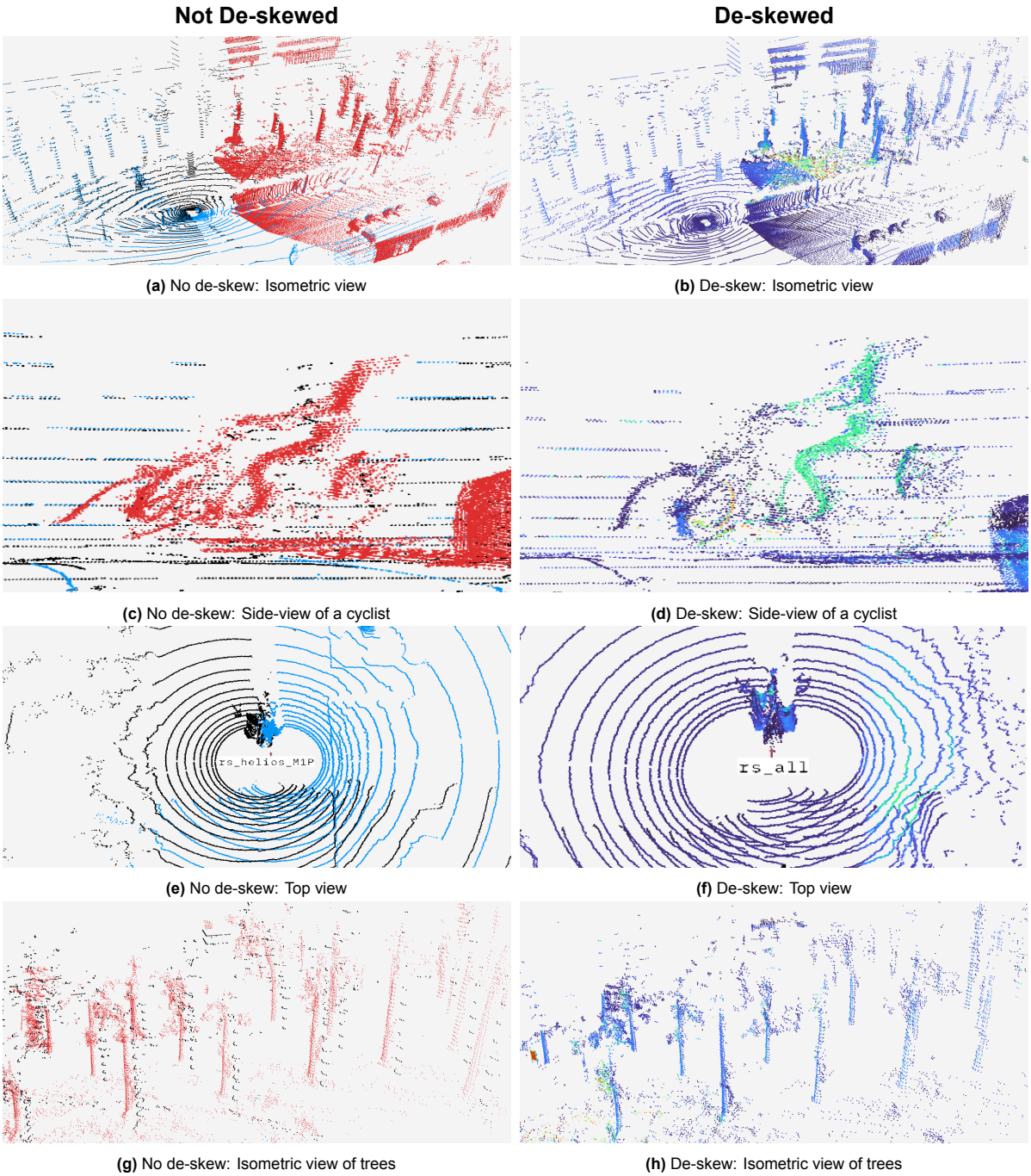
Figure 5.8 demonstrates that the NDT algorithm continuously compensates for the internal rotations of the bicycle frame, including roll, yaw, and pitch angles. The algorithm consistently adjusts for variations between -1.5° to 1.5° across all three angles. However, there is no apparent dependency between bicycle roll changes during cornering and the required compensation by the NDT algorithm. In other words, the relative rotation from the M1P LiDAR to the Rear Helios LiDARs is not influenced by our steering input beyond the roll angle, but rather always present.

The top left of Figure 5.9 shows that the ground plane of the M1P (red) does not align with the ground plane from the Helios LiDARs. In contrast, the same frame which includes the dynamic calibration alignment, displays a neat and aligned ground plane. This pattern is also observed in the top view of a random building in the top two pictures. On the left, the black points from the Helios LiDARs do not match those of the M1P LiDAR (red), whereas the same frame on the right shows improved alignment.

5.2.2. De-skewing

Figure 5.10 shows a visualization of similar pointclouds before and after de-skewing with KISS-ICP [62]. There are some apparent improvements:

- **Figure 5.10c & 5.10d:** On the left side, we see a cyclist represented by points recorded from our M1P LiDAR. This LiDAR scans from the bottom to the top. As we move approximately 40 cm during the time it takes to complete one scan (about 0.1 seconds), the points at the top appear to be further away, making all upright objects seem to ”lean” backward. The de-skewed version of the same bicycle clearly shows that this effect is no longer present.
- **Figure 5.10e & 5.10f** show the difference in the combined Helios LiDAR scan pattern on the ground plane. The Helios LiDARs both start their rotation facing backward (towards the bottom of the image) and then rotate clockwise. The right image clearly shows that the points recorded first are shifted approximately 40 cm backward, similar to the bicycle’s movement during one scan (0.1 seconds).
- **Figure 5.10g & 5.10h** illustrate how the Helios LiDARs recorded the trees at a different time compared to the M1P LiDAR. The de-skew function of KISS-ICP [62] effectively resolves this issue, transforming the data into a de-skewed point cloud where the points from all three LiDARs coincide accurately along the same tree stem, as displayed on the right.



(a) No de-skew: Isometric view

(b) De-skew: Isometric view

(c) No de-skew: Side-view of a cyclist

(d) De-skew: Side-view of a cyclist

(e) No de-skew: Top view

(f) De-skew: Top view

(g) No de-skew: Isometric view of trees

(h) De-skew: Isometric view of trees

Figure 5.10: De-skewing vs no de-skewing (ego-motion-compensation). Left column: Pointcloud visualization without any de-skewing. Red = M1P Lidar, Black = Helios LiDAR left, Blue = Helios LiDAR right. Right column: visualization of the same pointcloud WITH de-skewing, with color mapping according to intensity value.

5.3. Pseudo-labels

As described in section 4.2, we work with three different sets of pseudo-labels. Table 5.1 provides the total amounts of pseudo-labels per set for each class. We use only the predictions with a confidence score of TH or higher as a pseudo-label. To examine the quality of these labels, we generate them on our validation set as well and compare these to our self-made annotations, by use of average precision (AP), explained in Appendix D.2. It is important to note again that this validation set is small and includes only seven DoF bounding boxes without roll and pitch angles. Although there are relative roll and pitch angles with other road users due to bicycle dynamics, the set lacks these. Additionally, there are only five tracked pedestrians in the entire set, with two being occluded, leading to uncertainties in the evaluation of pedestrian pseudo-labels, as explained in Section 5.1.3.

Class	TH	STND	STND+	Ours
Vehicle	0.5	189k (35.4%)	253k (41.1%)	234k (37.5%)
Cyclist	0.5	86k (16.1%)	77k (12.5%)	**104k** (16.7%)
Pedestrian	0.5	259k (48.5%)	286k (46.4%)	286k (45.8%)

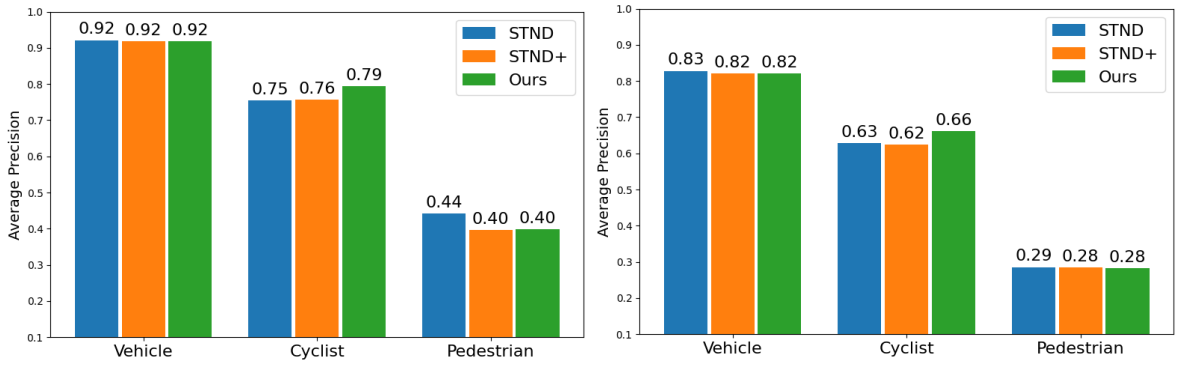
Table 5.1: Numbers of generated pseudo-label sets per class for the complete dataset. TH = Minimum confidence score to be used as pseudo-label

We compare the three generated pseudo-label sets with our annotations from the validation set and classify a True Positive based on a maximum center distance difference d, comparable to the evaluation of nuScenes [8]. In contrast, we use a maximum of d=1m, where they use d=2m. The results are summarized in Table 5.2. Other evaluation scores such as F1-score, ATE, ASE, and AOE as used in [8], are provided in Table C.8. An explanation of what this means is provided in Appendix D.2.

Set	Class	Detections	TP	FP	FN	Precision	Recall	AP
STND	Vehicle	1134	590	**544**	9	**0.52**	0.98	0.92
	Cyclist	1633	1551	82	482	0.95	0.76	0.75
	Pedestrian	1139	362	**777**	86	**0.32**	0.81	**0.44**
STND+	Vehicle	1307	**591**	716	8	0.45	0.99	0.92
	Cyclist	1640	1556	84	477	0.95	0.77	0.76
	Pedestrian	1493	375	1118	73	0.25	0.84	0.40
Ours	Vehicle	1307	**591**	716	8	0.45	0.99	0.92
	Cyclist	1717	**1633**	84	**400**	0.95	**0.80**	**0.79**
	Pedestrian	1412	375	1037	73	0.27	0.84	0.40

Table 5.2: Evaluation scores for the Pseudo-Label Sets and Classes.
TP = True Positives, FP = False Positives, FN = False Negatives, AP = Average Precision.

The average precisions (APs) of Table 5.2 are plotted and colorized in Figure 5.11a. Figure 5.11b provides a the calculated APs for 2D IoU evaluation. We use the same method as [19], with a minimum 2D IoU of 0.5 for pedestrians and cyclists, 0.7 for vehicles.

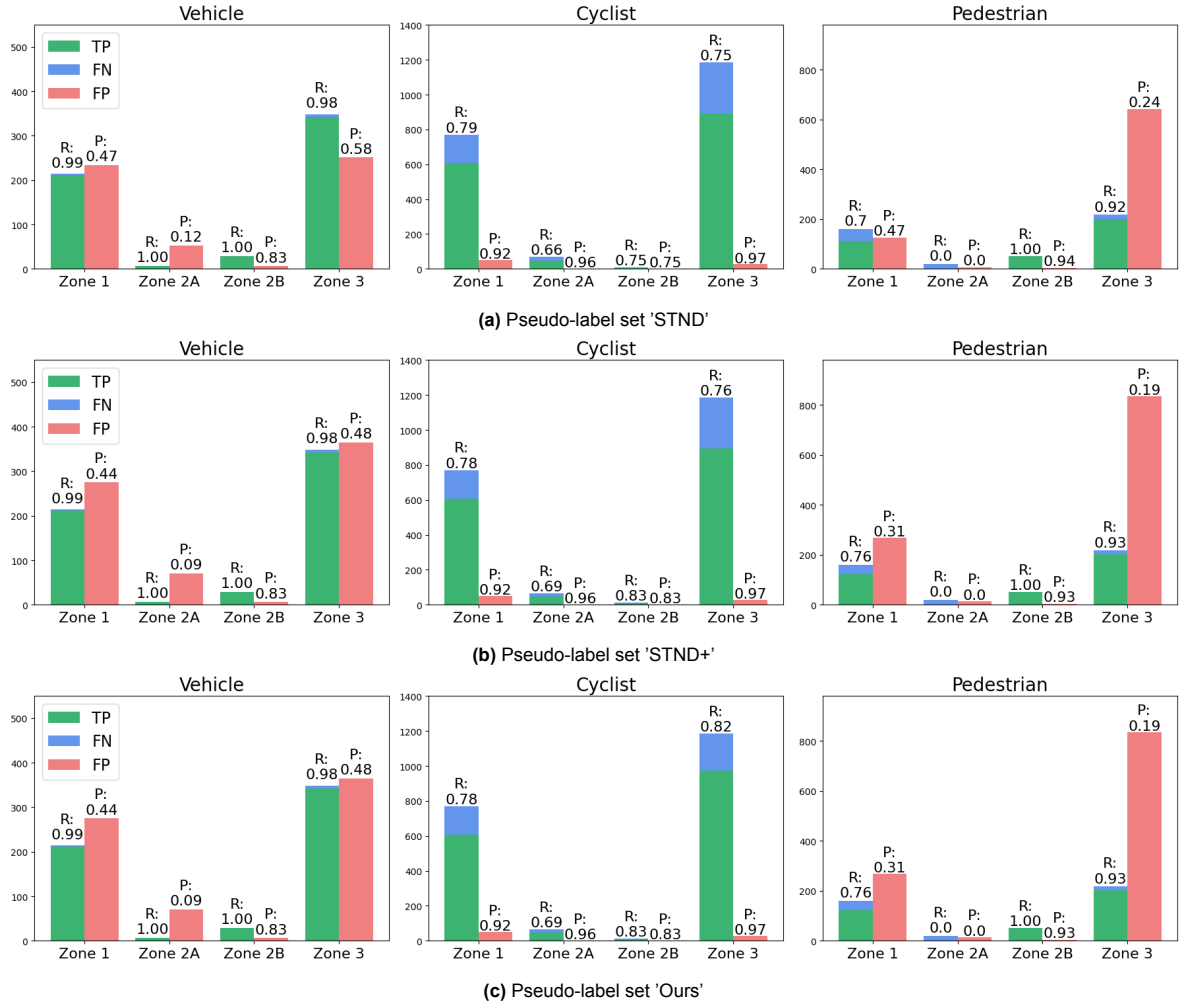


(a) Evaluation: maximum Center Distance (CD) d = 1m.

(b) Evaluation: 2D Intersection Over Union (IoU)

Figure 5.11: Evaluation of pseudo-labeling methods on our validation set, with two different evaluation types.

A more in-depth review of how these APs are formed corresponding to the zones from Figure 5.1a, can be found in Figure 5.12, for all sets of pseudo-labels. These figures provide the distribution of true positives (TP), false positives (FP), and false negatives (FN), as well as the precision (P) and recall (R) scores, for each class per zone. The evaluation type is again a maximum center distance of d = *1m*. Please note that the TP + FN bars correspond to the total number of annotations, as shown in Figure 5.5.



(a) Pseudo-label set 'STND'

(b) Pseudo-label set 'STND+'

(c) Pseudo-label set 'Ours'

Figure 5.12: Evaluations of different pseudo-label sets for each class per zone. Evaluation type: *d=1m*. TP = True Positive, FN = False Negative, FP = False Positive, R = recall, P = precision.

Discussion

Table 5.1 shows the total amount of generated pseudo-labels for the entire *SenseBike* dataset. The vehicle and pedestrian class increase in size with a larger detector ensemble (DE), as to be expected. Remarkable is that the number of cyclist pseudo-labels decreases with a larger DE in the STND+ pseudo-label set. The number in bold proves that the implementation of the cyclist refinement does generate a lot more cyclist pseudo-labels, increasing up to 104*k*.

Figure 5.11a clearly shows how our pseudo-labeling method increases the average precision (AP) for cyclists compared to the standard pseudo-labeling methods (STND and STND+). The AP for cyclists improves from 0.76 to 0.79 without any losses in the vehicle class. However, this improvement comes at the cost of a lower pedestrian AP. We acknowledge that the number of pedestrians in the test set is very low, and they are all located far from the ego-vehicle, leading to inaccurate numbers. Additionally, the original pseudo-labeling pipeline [59] does not use static pedestrian tracks for generating extra labels in the first round due to frequent misclassification of pole-like objects as pedestrians. Our validation set contains 6 pedestrians, three of whom are static. We assume this number would significantly increase

if we were to self-train our entire pseudo-labeling method, including static pedestrian tracks at a certain stage. Since we are solely assessing the dataset's potential for SOTA performance, we omit that step.

It is remarkable to see that the AP of any of the classes does not increase much more by the addition of more different detectors in the detector ensemble, as is the difference between *sTBD* and *STND+*, while Table 5.1 proves that a larger DE does generate more pseudo-labels in the vehicle and pedestrian class on the whole dataset.

In Figure 5.11b we see the same results as in Figure 5.11a, with the difference being the evaluation method. We set the 2D Intersection Over Union (IOU) thresholds to 0.4 for pedestrians, 0.5 for cyclists, and 0.7 for cars. We still see an increase in AP for cyclists on our pseudo-label set compared to the original method, without really losing anywhere else, making the cyclist refinement method successful.

Figure 5.12 shows in what zone of the pointcloud the pseudo-labels achieve the greatest precision or recall, for the *STND*, *STND+* and *Ours* pseudo-label set, respectively. Several important points require attention, discussed per object class:

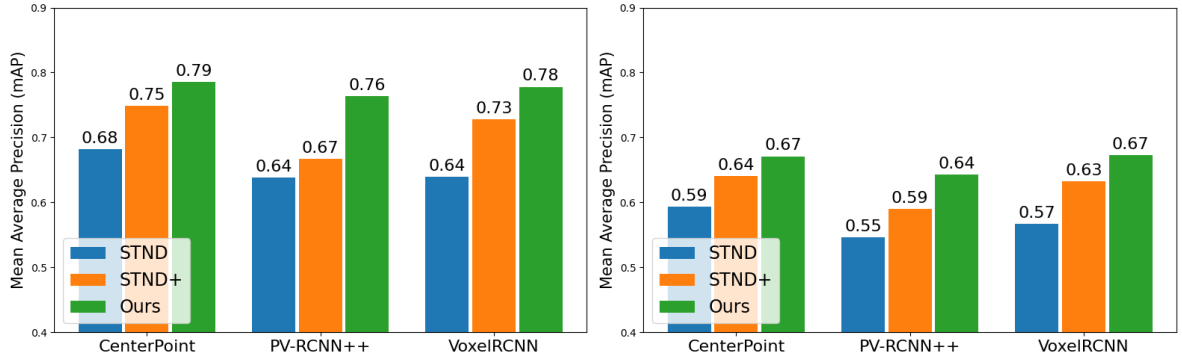
Vehicles: Table 5.2 has already shown that the recall of our pseudo-labels is relatively high for the vehicle class. This indicates that we rarely miss a vehicle, resulting in very few false negatives (FNs). However, there is a significant issue with the high number of false negatives, particularly in Zones 1 and 3. In the next round of multi-stage pseudo-labeling, it will be necessary to increase the thresholds for vehicles. Another noteworthy observation is that the precision levels for the vehicle class decrease with larger detection ensembles (DE), as seen in the sets *STND+* and *Ours*. This decline is likely due to the increase in fused detections (Figure 4.7), which raises the likelihood of false positives. This issue can also be addressed by raising the threshold for the vehicle class, either in the initial round of self-training or in subsequent rounds. Additionally, the number of vehicles present in Zones 2A and 2B is remarkably low. Fortunately, all the vehicles in these zones are detected in all pseudo-label sets, resulting in a 100% recall. Once again, increasing the confidence score threshold for pseudo-labeling will help mitigate the issues in these zones.

Cyclists: In contrast to the vehicle class, the cyclist class exhibits lower recall numbers than precision numbers across all zones and pseudo-label sets. In future stages of self-training, or in the initial round next time, a lower confidence threshold for cyclist pseudo-labels could be implemented. It is noteworthy that the influence of cyclist refinement in pseudo-label set *Ours* is primarily noticeable in Zone 3, as indicated by an increased recall score. This refinement does not affect any of the other zones compared to the pseudo-label set *STND+*.

Pedestrians: The first notable observation in the scores of pedestrian pseudo-labels evaluated on our self-made annotations is the extremely low precision in Zone 3. Many false predictions are generated as pseudo-labels (FN), even though there were no actual pedestrians present. However, we do achieve a high recall across all sets in Zone 3, suggesting that the confidence score threshold for a prediction to become a pseudo-label could be increased. In Zones 2A and 2B, there are almost no pedestrians. Nonetheless, the pseudo-labels for those in Zone 2B are very accurate, with a recall of 1.0 and a precision of 0.93. In Zone 1, both recall and precision are low. The pedestrian refinement process includes removing all static pedestrian tracks in the initial rounds of pseudo-labeling due to the tendency to misclassify pole-like objects as pedestrians. Conducting multiple rounds of self-training and eventually using the static pedestrian tracks for refinement could help address this issue. The increase in scores when using a larger detection ensemble (DE) is not noteworthy for the pedestrian class.

5.4. Domain Adaptation

This section focuses on the transfer learning - used to achieve domain adaptation - of three off-the-shelf detectors, as explained in Section 4.2.3. We evaluate the detection performances of *CenterPoint*, [72], *PV-RCNN++* [53] and *Voxel-RCNN* [16], which are trained on our pseudo-label sets from Section 5.3, on our labeled test set. The exact configurations of these models can be found in Appendix C.2. We train these backbones for 15 epochs on the three sets of pseudo-labels and evaluate them on our validation set, with a confidence score threshold of 0.6. We first consider a prediction True Positive (TP), when its center distance d is within $d = 1m$ from the ground truth (GT) label and calculate the mAP as introduced by KITTI [19]. The results can be found in Figure 5.13.



(a) Evaluation: maximum Center Distance (CD) $d = 1m$.

(b) Evaluation: 2D Intersection Over Union (IoU)

Figure 5.13: Evaluation of transfer learned detectors on our validation set.

To determine whether our dataset is sufficiently large, we evaluate the impact of starting the training phase with pre-trained weights on the eventual performance. Specifically, we utilize nuScenes pre-trained weights [8] for training *CenterPoint*, *PV-RCNN++*, and *VoxelRCNN* on both the *STND+* and *Ours* pseudo-label sets. We then compare the resulting mean Average Precision (mAP) with those obtained from detectors trained without pre-trained weights. The outcomes are depicted in Figure 5.14.

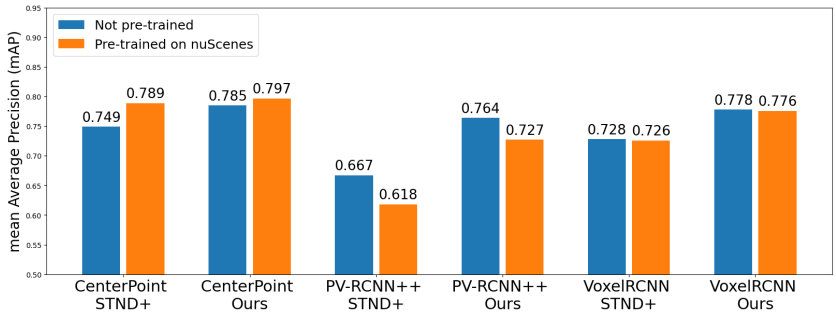


Figure 5.14: Effect of starting a training with pre-trained weights of the models on the nuScenes [8] dataset. TP Evaluation: $d = 1m$

To experiment with the effect of domain adaptation, we compare the scores of detectors trained on Waymo versus those trained on our pseudo-label sets, evaluating them on the validation set. The mAP and AP per class of this evaluation are provided in Table 5.3.

Detector	Training Set	mean AP	AP Vehicle	AP Cyclist	AP Pedestrian
VoxelRCNN	Waymo	0.54	0.88	0.41	0.34
VoxelRCNN	Ours	**0.78**	**0.9**	**0.81**	**0.62**
PV-RCNN++	Waymo	0.50	**0.91**	0.35	0.25
PV-RCNN++	Ours	**0.76**	0.87	**0.82**	**0.60**

Table 5.3: Evaluation of Waymo pre-trained versus pseudo-label set *Ours* pre-trained detectors on our validation set. Evaluation type: *d=1m*.

To evaluate whether the *SenseBike* dataset can enhance state-of-the-art (SOTA) 3D object detection using LiDAR, we must compare our results with those achieved on established SOTA datasets. Therefore, we compare our domain adaptation results with the detection benchmark scores from the Waymo dataset [55]. It is important to recognize that this comparison involves datasets and evaluation rules with inherent differences, requiring a careful interpretation of the results. The main differences are first provided in Table 5.4, and the actual evaluation scores are provided in Figure 5.15. Numbers of the performance on Waymo originate from [16, 53, 72, 55, 56].

Differences	**Waymo Dataset**	**SenseBike Dataset**
LiDAR Scan Pattern	1x rotating 64 beam 4x rotating 64 beam, inclined	1x solid-state 125 beam 2x rotating 32 beam
Grid Range	x: [*-75m, 75m*], y: [*-75m, 75m*]	x: [*-20m, 60m*], y: [*-20m, 20m*]
Detection Difficulties	Two levels: LEVEL_1 for non-occluded objects; LEVEL_2 for all labels within the grid range	No distinction in object detection difficulty
Ego-Vehicle	Recorded from a car with roll angles up to ±2° when cornering	Recorded from a bicycle with roll angles up to ±20° when cornering

Table 5.4: Differences in evaluation Waymo vs SenseBike dataset to consider when interpreting Figure 5.15.

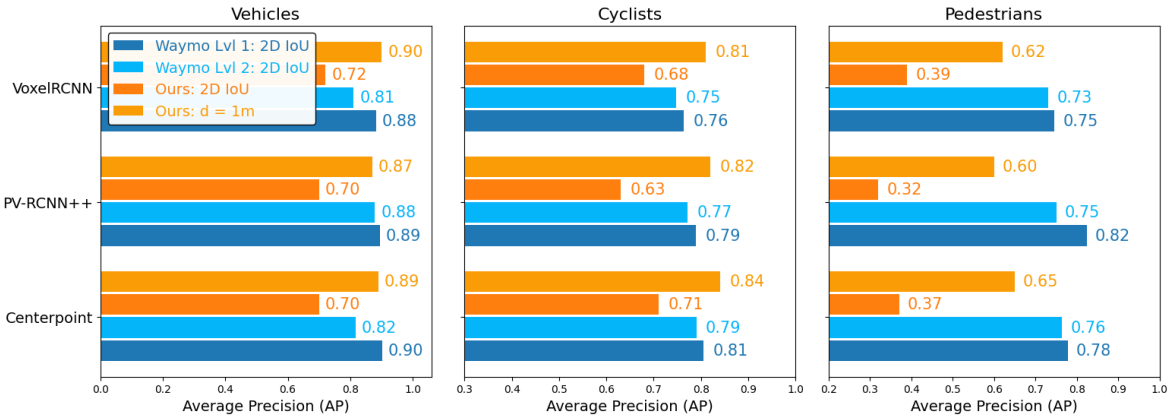


Figure 5.15: Average Precisions per class, compared to scores of detectors on Waymo set. Please read Table 5.4, to check the differences.

Similar to the comparison of pseudo-labels on our validation set in Figure 5.12, we evaluate the predictions from the trained detectors on pseudo-label set *Ours*, on our validation set. The results are provided in Figure 5.16, 5.17 and 5.18. For a convenient comparison, we also include the quality of the pseudo-label set '*Ours*' that these detectors are trained on below in Figure 5.19.

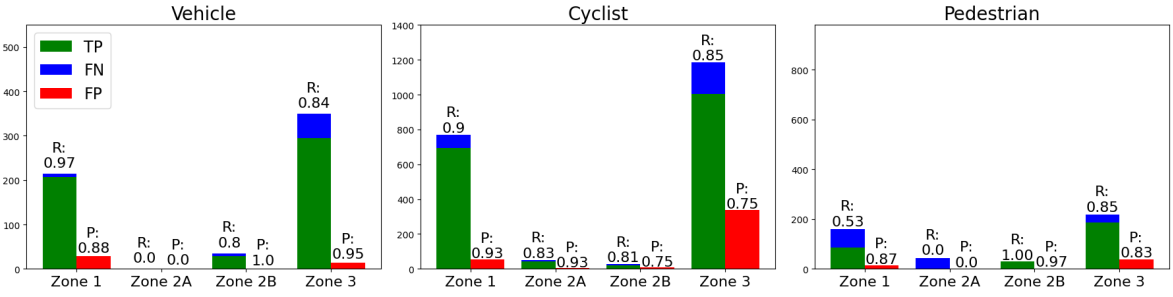


Figure 5.16: Detector *CenterPoint*

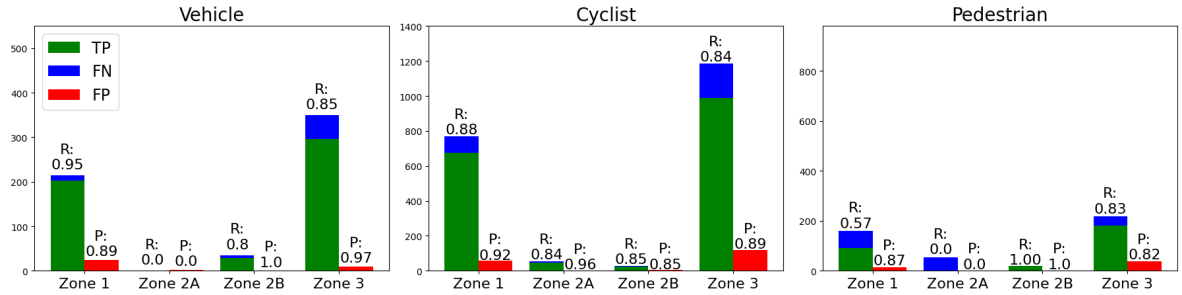


Figure 5.17: Detector *PV-RCNN++*

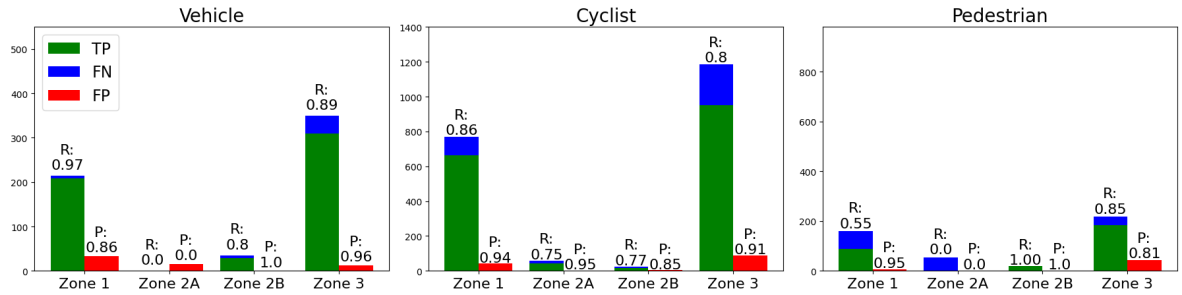


Figure 5.18: Detector *VoxelRCNN*.
Evaluations of three detectors on the validation set, trained on pseudo-label set *Ours*.
TP = True Positive, FN = False Negative, FP = False Positive, R = recall, P = precision.

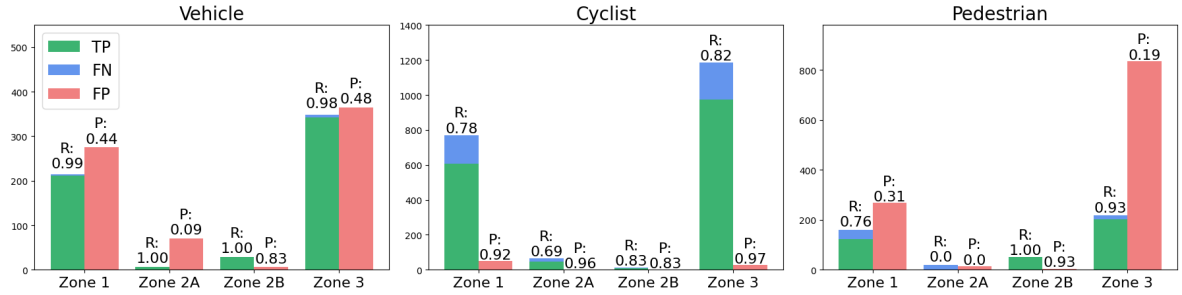


Figure 5.19: The quality of pseudo-label set *Ours*, when generated and compared on our validation set. Copy of Figure 5.12c.

Discussion

Figure 5.13 clearly shows the impact of the cyclist refinement, included in the pseudo-label set '*Ours*', in comparison to the pseudo-label sets *STND* and *STND+*, which did not implement this refinement. The mean average precisions (mAP) achieved by the transfer-learned detectors increase for each set. Additionally, despite Figure 5.11 indicating that the pseudo-label set's quality did not improve with a larger detector ensemble (DE), the final performance of detectors trained on a pseudo-label set generated with a larger DE does improve. Thus, combined with the numbers from Table 5.1, it can be concluded that more pseudo-labels lead to better domain-adapted detector performance.

In Figure 5.14, we examine the mAP of similar detectors trained on similar pseudo-label sets, either using pre-trained weights or starting from scratch. There is no clear improvement or reduction in performance, even among different detectors.

Table 5.3 demonstrates the effect of domain adaptation, showing that two detectors trained on our pseudo-labels perform better on our validation set than the same detectors trained on Waymo. The AP for vehicles for *PV-RCNN++* does decrease slightly.

Figure 5.15 compares the performance of detectors trained on our pseudo-labels with their performance on our validation set, evaluated with a true positive maximum center distance difference $d = 1m$. This achieves comparable performance to the same detectors trained on Waymo, validated with a 2D IoU evaluation. However, when evaluating with 2D IoU, our performance scores significantly lower, reducing APs by up to 20% for vehicles, 10% for cyclists, and up to 39% for pedestrians. As noted in Table 5.4, our recording vehicle experiences roll angles when cornering, meaning bounding boxes around other road users should include a roll angle, as explained in Section 5.1.3. Currently, *OpenPCDet* [56], used to generate all results, only supports 7DoF bounding boxes without roll or pitch angles. This makes achieving a 2D IoU evaluation for a true positive more challenging with data that includes roll and pitch angles. Therefore, we believe a $d=1m$ evaluation in our dataset fairly compares to a 2D IoU evaluation in a non-rolling dataset like Waymo. Differences in heading angle and bounding box size are not yet considered, which should be conducted in future updates.

As illustrated in Figure 5.16, Figure 5.17, Figure 5.18, the three trained detectors do not show large performance differences in any of the zones. The main distinction is the cyclist class performance, where *CenterPoint* achieves slightly higher recall than *PV-RCNN++* or *VoxelRCNN*, at the cost of lower precision across all zones.

Given that zones 2A and 2B contain fewer LiDAR points compared to zones 1 and 3, higher recalls and precisions are expected in these zones. However, for all classes and detectors, this is not necessarily the case. Despite the small validation set and the limited number of labeled objects in these areas, making the scores less reliable, we conclude that these areas contain enough points to enable full 360° detection performance, meaning the current LiDAR setup suffices.

For all three detectors, recall for cyclists in zones 1 and 3 is relatively low compared to cars, where precisions are around 90%. This suggests that in future works, we should consider lowering the prediction confidence thresholds for this class slightly.

Interestingly, the low precision for pedestrians in zone 3 of the pseudo-labels (Figure 5.19), compared to our labels in the validation set, does not lead to low precision in zone 3 for detectors trained on these pseudo-labels. Comparing this pseudo-label quality with *VoxelRCNN* performance in Figure 5.18, we observe an 81% precision for pedestrians in zone 3. The major issue with the pedestrian prediction for all three detectors is the low recall in Zone 1. Increasing the number of pseudo-labels for pedestrians, by applying multiple rounds of self-training in the pseudo-label pipeline, would likely improve pedestrian detection scores for all detectors. This concept is explained in more detail in the discussion of Section 5.3.

6

Conclusion

High-quality datasets are essential for all perception tasks in intelligent vehicles. However, a limitation of most available datasets is their tendency to biases based on the geographical domains in which they are recorded, leading to biases in detecting performance once trained on them. For instance, biases can arise in the presence numbers of certain Vulnerable Road Users (VRUs), the typical distance from these VRUs, or the average sizes of cars. To address these biases, a diverse array of datasets is necessary. To this end, we introduce the *SenseBike* dataset, recorded from a bicycle in the Dutch city of Delft. This dataset includes LiDAR scans from pedestrian-dense city centers, bicycle-dense bicycle paths, and vehicle-dense areas, such as parking lots. We explore whether this dataset can enhance state-of-the-art LiDAR object detection performance, particularly by addressing the low cyclist presence observed in other large automotive datasets, such as *Waymo* [55] and *nuScenes* [8]. To achieve this, we established a pseudo-labeling pipeline designed to address LiDAR domain differences and implemented pseudo-labeling specifically for cyclists. The research process was divided into three subquestions, which are addressed below. After that, we answer the main research question and conclude with recommendations for both the *SenseBike* project and the domain adaptation pipeline.

1) How can we produce a novel dataset, that is truly distinguishable from others?
The *SenseBike* dataset is characterized by its unique LiDAR pointclouds, resulting from the fusion of two rotating 32-beam LiDARs and one solid-state 125-beam LiDAR. This distinct scan pattern sets it apart from many other datasets, as well as the fact that it is recorded in a Dutch city, including narrow city centers. However, what truly distinguishes the *SenseBike* dataset is that it is recorded from a bicycle. Recording from a bicycle path imparts unique characteristics to the data, such as a higher presence of nearby cyclists. Our generated pseudo-labels for the entire dataset include 104*k* cyclists, making up 16.7% of all pseudo-labels, which is substantially higher than the 0.56% of labeled 3D bounding boxes for cyclists in the Waymo dataset. Due to the non-uniform distribution of points per azimuth angle in our pointcloud, we divided the pointcloud into four zones for a more detailed evaluation of object detection performance in each zone. Our findings indicate that each zone within the pointcloud achieves comparable performance levels, demonstrating that every part of the recorded pointclouds can be valuable for detection tasks.

2) What challenges arise in collecting data from a bicycle, and how can they be addressed?
We experienced two main challenges while collecting data from a bicycle. The first is that, in contrast to a car, the frame from rear to front LiDARs is not fully rigid. This meant that we needed some dynamic aligning of the pointclouds from the front and rear LiDARs. We referred to this as dynamic calibration, in which we constantly run a Normal Distribution Transform aligning algorithm, which iteratively aligns the front to the rear LiDAR. As we could not find a clear pattern between steering input and the internal rotations in the bicycle, we constantly had to run the dynamic calibration.

The second challenge inherent to recording from a bicycle is that a bicycle leans sidewards when cornering, inducing a change in roll angle. This also means that when we corner, all the data induces a roll angle. This means that all other road users also rotate with a roll or pitch angle relative to us. One problem is that the state-of-the-art pointcloud detection tool *OpenPCDet* [56] does not have an

implementation of roll or pitch angles as outputs of predicted bounding boxes from 3D object detectors, meaning a 2D Intersection over Union evaluation does not suffice anymore. We therefore evaluate our eventual performance of object detectors with a maximum center distance *d* difference.

3) Can the novel SenseBike dataset positively affect the performance of existing off-the-shelf 3D object detectors and how do we evaluate that?
To examine this subquestion, we set up a domain-adaptation pipeline, which we use for transfer learning off-the-shelf detectors. In such a way, we can achieve domain adaptation of these off-the-shelf detectors, making it possible to evaluate their performance on the *SenseBike* dataset. We only run one stage of pseudo-labeling, whereas the original method recommends using multiple. This would improve the quality of the pseudo-labels significantly, as the original method concludes, especially for the pedestrian class. As our dataset will incorporate human-annotated labels in future updates, we only examine if we can already achieve worthy performance without any self-training stages.

To examine the quality of the pseudo-labels, we generate them on the same data as our labeled validation set, and compare them with these labels. After, we train three detectors on these pseudo-labels and evaluate their performance on our validation set. We classify a true positive prediction of one of these detectors as True Positive when it is within a center distance *d* of maximum *1m*. The results on a domain adapted *CenterPoint* are an average precision (AP) of 0.89, 0.84, and 0.65 on vehicles, cyclists, and pedestrians, respectively. Especially the 84% AP on cyclists is notable, exceeding the 2D IoU Level 1 performance of the same detector on Waymo, which is 81%. This is mainly due to the extensive presence of cyclists in the dataset.

Main: Can we create a novel dataset, that enhances the 3D object detection performance of state-of-the-art detectors?
The proposed domain adaptation pipeline, which integrates pseudo-labeling followed by transfer learning with these labels, demonstrates promising results despite the absence of self-training to enhance pseudo-label quality. Utilizing *CenterPoint*, we achieved an average precision (AP) of 89% for vehicles and 84% for cyclists, showing only a marginal deviation compared to other datasets. This performance is the result of the cyclist-dense data within the *SenseBike* dataset. Consequently, we believe that adding human annotations to this dataset could greatly enhance state-of-the-art 3D object detection in automotive applications. The *SenseBike* dataset addresses the common bias towards vehicles and pedestrians in many existing datasets. While our pseudo-labels are not flawless, the performance of detectors trained on them remains noteworthy.

It is important to note that comparisons with similar detectors on other datasets are not entirely balanced due to variations in evaluation criteria, which could be rectified with a larger and more comprehensive validation set. Furthermore, to ensure robustness, modern intelligent vehicles should not rely solely on a single modality. Accordingly, we plan to incorporate camera images into the dataset to enhance its utility and redundancy.

Recommendations
The domain adaptation pipeline uses multi-frame detections to generate multi-source detections and employs a tracker to assess low-confidence predictions. Both of these depend heavily on accurate localization. Currently, we compute odometry using KISS-ICP [62], which assumes a constant velocity model. This method can fail when our bicycle undergoes sudden rotation, such as during cornering. The localization package that integrates IMU, GPS, and odometry data could resolve this issue, but it currently faces computational challenges. It requires a source of odometry, and currently we are using LiDAR odometry calculated by *KISS-ICP* again. Introducing another odometry source, like wheel or cadence odometry, could address this. Nevertheless, the IMU still produces unreliable z-acceleration values, which also needs to be resolved. Once this localization package operates in real-time, the quality of pseudo-labels is expected to improve.

The dynamic calibration used for overcoming internal bicycle frame rotations is mainly necessary due to the tendency of the luggage carrier to vibrate. This luggage carrier carries the weight of many hardware components, including two LiDARs and a battery. A practical solution to this issue could be to reduce the weight carried by the luggage carrier. At the moment of writing, we are analyzing different setups for this part of the *SenseBike*.

Integrating synchronized cameras or radars onto the *SenseBike* and into its ROS environment would enhance research on object detection with different modalities and could assist in the human labeling process.

Throughout the design of the domain-adaptation pipeline, we made several assumptions and decisions that requires further analysis. Firstly, contrary to the recommendations of the original proposed method, we did not re-train our pseudo-labeling pipeline. The original method suggests that re-training would significantly improve the quality of pseudo-labels, particularly for the pedestrian class. Since cyclists were not included in the initial implementation, this might also apply to cyclists. We also recommend a thorough analysis of all parameters in the refinement of cyclist pseudo-labels, rather than setting them based solely on visual inspections.

Finally, to confidently assess the value of the *SenseBike* dataset for state-of-the-art 3D object detection, we recommend expanding the validation set. We only labeled two sequences of 30 seconds each, which does not capture the full variability and contents of the data. A larger validation set would provide more accurate numbers on the quality of pseudo-labels and the performance of domain-adapted detectors.

References

[1] Amina Adadi. “A survey on data□efficient algorithms in big data era”. In: *Journal of Big Data* 8.1 (Jan. 26, 2021), p. 24. DOI: 10.1186/s40537-021-00419-9.

[2] Sungsoo Ahn et al. *Variational Information Distillation for Knowledge Transfer*. Apr. 11, 2019. arXiv: 1904.05835[cs].

[3] Simegnew Alaba, Ali Gurbuz, and John Ball. *A Comprehensive Survey of Deep Learning Multisensor Fusion-based 3D Object Detection for Autonomous Driving: Methods, Challenges, Open Issues, and Future Directions*. Aug. 23, 2022. DOI: 10.36227/techrxiv.20443107.v2.

[4] Eduardo Arnold et al. “A Survey on 3D Object Detection Methods for Autonomous Driving Applications”. In: *IEEE Transactions on Intelligent Transportation Systems* 20.10 (Oct. 2019), pp. 3782–3795. DOI: 10.1109/TITS.2019.2892405.

[5] A. Author, B. Author, and C. Author. “Point Cloud Semantic Segmentation for Detecting Bicycle Paths Using Bicycle-Mounted LiDARs”. In: *Journal of Advanced Transportation Research* 45.2 (2023), pp. 123–134.

[6] Jens Behley et al. *SemanticKITTI: A Dataset for Semantic Scene Understanding of LiDAR Sequences*. Aug. 16, 2019. arXiv: 1904.01416[cs].

[7] P. Biber and W. Strasser. “The normal distributions transform: a new approach to laser scan matching”. In: *Proceedings 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2003) (Cat. No.03CH37453)*. 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2003) (Cat. No.03CH37453). Vol. 3. Las Vegas, Nevada, USA: IEEE, 2003, pp. 2743–2748. DOI: 10.1109/IROS.2003.1249285.

[8] Holger Caesar et al. *nuScenes: A multimodal dataset for autonomous driving*. version: 5. May 5, 2020. arXiv: 1903.11027[cs,stat].

[9] Benjamin Caine et al. *Pseudo-labeling for Scalable 3D Object Detection*. Mar. 2, 2021. arXiv: 2103.02093[cs].

[10] P. Cangley et al. “A model for performance enhancement in competitive cycling”. In: *Movement & Sport Sciences* n° 75.1 (2012), p. 59. DOI: 10.3917/sm.075.0059.

[11] Yilun Chen et al. *FocalFormer3D : Focusing on Hard Instance for 3D Object Detection*. Aug. 8, 2023. arXiv: 2308.04556[cs].

[12] Zhuoxiao Chen et al. “Revisiting Domain-Adaptive 3D Object Detection by Reliable, Diverse and Class-balanced Pseudo-Labeling”. In: *2023 IEEE/CVF International Conference on Computer Vision (ICCV)*. 2023 IEEE/CVF International Conference on Computer Vision (ICCV). Paris, France: IEEE, Oct. 1, 2023, pp. 3691–3703. DOI: 10.1109/ICCV51070.2023.00344.

[13] Zhuoxiao Chen et al. *Revisiting Domain-Adaptive 3D Object Detection by Reliable, Diverse and Class-balanced Pseudo-Labeling*. Aug. 16, 2023. arXiv: 2307.07944[cs].

[14] cra-ros-pkg. *Robot Localization*. https://github.com/cra-ros-pkg/robot_localization. 2023.

[15] Dassault Systèmes. *SOLIDWORKS 2023*. https://www.solidworks.com. Version 2023. 2023.

[16] Jiajun Deng et al. *Voxel R-CNN: Towards High Performance Voxel-based 3D Object Detection*. Feb. 5, 2021. arXiv: 2012.15712[cs].

[17] SparkFun Electronics. *Qwiic_Ublox_Gps_Py*. https://github.com/sparkfun/Qwiic_Ublox_Gps_Py. Accessed: 2024-05-29. 2024.

[18] Awet Haileslassie Gebrehiwot et al. “Teachers in concordance for pseudo-labeling of 3D sequential data”. In: *IEEE Robotics and Automation Letters* 8.2 (Feb. 2023), pp. 536–543. DOI: 10.1109/LRA.2022.3226029. arXiv: 2207.06079[cs].

[19] A Geiger et al. “Vision meets robotics: The KITTI dataset”. In: *The International Journal of Robotics Research* 32.11 (Sept. 2013), pp. 1231–1237. DOI: 10.1177/0278364913491297.

[20] Google. *Google Maps overview of Delft, the Netherlands*. Retrieved May 23, 2024, from https://goo.gl/maps/.

[21] Simon Gröchenig and Karl Rehrl. “Towards C-ITS-based communication between bicycles and automated vehicles”. In: *Proceedings of the International Conference on Smart Cities and Smart Grid*. 2021, pp. 118–120.

[22] Manfred Hagelen et al. “Safety and Comfort Enhancement with Radar for a Bicycle Assistance System”. In: *2019 20th International Radar Symposium (IRS)*. 2019 20th International Radar Symposium (IRS). Ulm, Germany: IEEE, June 2019, pp. 1–7. DOI: 10.23919/IRS.2019.8768109.

[23] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. *Distilling the Knowledge in a Neural Network*. Mar. 9, 2015. arXiv: 1503.02531[cs,stat].

[24] John Houston et al. *One Thousand and One Hours: Self-driving Motion Prediction Dataset*. Nov. 16, 2020. arXiv: 2006.14480[cs].

[25] Chunyong Hu et al. *FusionFormer: A Multi-sensory Fusion in Bird’s-Eye-View and Temporal Consistent Transformer for 3D Object Detection*. Oct. 8, 2023. arXiv: 2309.05257[cs].

[26] Haotian Hu et al. *EA-LSS: Edge-aware Lift-splat-shot Framework for 3D BEV Object Detection*. Aug. 29, 2023. arXiv: 2303.17895[cs].

[27] Louis P Huard. *Boréal Active Mobility - V2X C-V2X E-BIKES*. Boreal Bikes - Connected Micromobility - AI ADAS - V2X. URL: https://www.borealbikes.com/ (visited on 01/25/2024).

[28] Motaz Khader and Samir Cherian. “An Introduction to Automotive Lidar”. In: (2023).

[29] CCNY Robotics Lab. *imu_tools: ROS tools for IMU devices*. https://github.com/CCNYRoboticsLab/imu_tools. Accessed: 2024-05-29. 2024.

[30] Xin Lai et al. *Spherical Transformer for LiDAR-based 3D Recognition*. Mar. 22, 2023. arXiv: 2303.12766[cs].

[31] Alex H. Lang et al. *PointPillars: Fast Encoders for Object Detection from Point Clouds*. May 6, 2019. arXiv: 1812.05784[cs,stat].

[32] Guopeng Li et al. *Large Car-following Data Based on Lyft level-5 Open Dataset: Following Autonomous Vehicles vs. Human-driven Vehicles*. Nov. 21, 2023. arXiv: 2305.18921[cs,eess].

[33] Yiyi Liao, Jun Xie, and Andreas Geiger. “KITTI-360: A Novel Dataset and Benchmarks for Urban Scene Understanding in 2D and 3D”. In: (2021). Publisher: arXiv Version Number: 2. DOI: 10.48550/ARXIV.2109.13410.

[34] RoboSense LiDAR. *rslidar_sdk: RoboSense LiDAR SDK for ROS & ROS2*. https://github.com/RoboSense-LiDAR/rslidar_sdk. Accessed: 2024-05-23. 2024.

[35] Xuewu Lin et al. *Sparse4D: Multi-view 3D Object Detection with Sparse Spatial-Temporal Fusion*. Feb. 10, 2023. arXiv: 2211.10581[cs].

[36] Mingyu Liu et al. *A Survey on Autonomous Driving Datasets: Statistics, Annotation Quality, and a Future Outlook*. Apr. 23, 2024. arXiv: 2401.01454[cs].

[37] Xiang Lu et al. “Whole-Body Pose Estimation in Physical Rider–Bicycle Interactions With a Monocular Camera and Wearable Gyroscopes”. In: *Journal of Dynamic Systems, Measurement, and Control* 139.7 (July 1, 2017), p. 071005. DOI: 10.1115/1.4035760.

[38] Zhipeng Luo et al. *Unsupervised Domain Adaptive 3D Detection with Multi-Level Consistency*. Aug. 17, 2021. arXiv: 2107.11355[cs].

[39] Sebastian O H Madgwick. “An e□cient orientation filter for inertial and inertial/magnetic sensor arrays”. In: (Mar. 30, 2010).

[40] Armin Niedermüller. “Salzburg Bicycle LiDAR Data Set”. Thesis. Salzburg University of Applied Sciences, Feb. 15, 2023.

[41] Armin Niedermüller and Moritz Beeking. “Transformer based 3D semantic segmentation of urban bicycle infrastructure”. In: *Journal of Location Based Services* (Jan. 25, 2024), pp. 1–23. DOI: 10.1080/17489725.2024.2307969.

[42] Andras Palffy et al. “Multi-Class Road User Detection With 3+1D Radar in the View-of-Delft Dataset”. In: *IEEE Robotics and Automation Letters* 7.2 (Apr. 2022), pp. 4961–4968. DOI: 10.1109/LRA.2022.3147324.

[43] Ziqi Pang, Zhichao Li, and Naiyan Wang. *SimpleTrack: Understanding and Rethinking 3D Multi-object Tracking*. Nov. 18, 2021. arXiv: 2111.09621[cs].

[44] Charles R Qi et al. “Deep Hough voting for 3D object detection in point clouds”. In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2019, pp. 9277–9286.

[45] Charles R. Qi et al. *PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation*. Apr. 10, 2017. arXiv: 1612.00593[cs].

[46] Rui Qian, Xin Lai, and Xirong Li. “3D Object Detection for Autonomous Driving: A Survey”. In: *Pattern Recognition* 130 (Oct. 2022), p. 108796. DOI: 10.1016/j.patcog.2022.108796. arXiv: 2106.10823[cs].

[47] ROS-Perception. *pcl_ros: PCL (Point Cloud Library) ROS interface stack*. https://github.com/ros-perception/perception_pcl. Accessed: 2024-06-09. 2024.

[48] ROS2 Developers. *ROS2 Bridge*. https://github.com/ros2/ros1_bridge. Accessed: May 28, 2024. 2024.

[49] Ryohei Sasaki. *ROS2 porting of ndt_omp*. https://github.com/rsasaki0109/ndt_omp_ros2. Accessed: 2024-06-09. 2024.

[50] Shaoshuai Shi, Xiaogang Wang, and Hongsheng Li. “PointRCNN: 3D Object Proposal Generation and Detection From Point Cloud”. In: *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR). Long Beach, CA, USA: IEEE, June 2019, pp. 770–779. DOI: 10.1109/CVPR.2019.00086.

[51] Shaoshuai Shi et al. *From Points to Parts: 3D Object Detection from Point Cloud with Part-aware and Part-aggregation Network*. Mar. 16, 2020. arXiv: 1907.03670[cs].

[52] Shaoshuai Shi et al. *PV-RCNN: Point-Voxel Feature Set Abstraction for 3D Object Detection*. Apr. 9, 2021. arXiv: 1912.13192[cs,eess].

[53] Shaoshuai Shi et al. *PV-RCNN++: Point-Voxel Feature Set Abstraction With Local Vector Representation for 3D Object Detection*. Nov. 7, 2022. arXiv: 2102.00463[cs].

[54] Xiuying Shi et al. “The Iterative Closest Point Registration Algorithm Based on the Normal Distribution Transformation”. In: *Procedia Computer Science* 147 (2019), pp. 181–190. DOI: 10.1016/j.procs.2019.01.219.

[55] Pei Sun et al. *Scalability in Perception for Autonomous Driving: Waymo Open Dataset*. May 12, 2020. arXiv: 1912.04838[cs,stat].

[56] OpenPCDet Development Team. *OpenPCDet: An Open-source Toolbox for 3D Object Detection from Point Clouds*. https://github.com/open-mmlab/OpenPCDet. 2020.

[57] The MathWorks, Inc. *MATLAB*. Version R2023a. The MathWorks, Inc. Natick, Massachusetts, 2023.

[58] Darren Tsai et al. *MS3D: Leveraging Multiple Detectors for Unsupervised Domain Adaptation in 3D Object Detection*. May 8, 2022. arXiv: 2304.02431[cs].

[59] Darren Tsai et al. *MS3D++: Ensemble of Experts for Multi-Source Unsupervised Domain Adaptation in 3D Object Detection*. Sept. 4, 2023. arXiv: 2308.05988[cs].

[60] Ashish Vaswani et al. *Attention Is All You Need*. Aug. 1, 2023. arXiv: 1706.03762[cs].

[61] Marta Martinez Vazquez. *Radar For Automotive: Why Do We Need Radar?* Semiconductor Engineering. Mar. 24, 2022. URL: https://www.renesas.com/us/en/blogs/radar-transceivers-key-component-adas-autonomous-driving-blog-1-why-do-we-need-radar (visited on 10/23/2023).

[62] Ignacio Vizzo et al. “KISS-ICP: In Defense of Point-to-Point ICP – Simple, Accurate, and Robust Registration If Done the Right Way”. In: *IEEE Robotics and Automation Letters* 8.2 (Feb. 2023), pp. 1029–1036. DOI: 10.1109/LRA.2023.3236571. arXiv: 2209.15397[cs].

[63] Haiyang Wang et al. *DSVT: Dynamic Sparse Voxel Transformer with Rotated Sets*. Mar. 20, 2023. arXiv: 2301.06051[cs].

[64] Tai Wang et al. “Train in Germany, test in The USA: Making 3D object detectors generalize”. In: *arXiv preprint arXiv:2005.08139* (2020).

[65] Pengchuan Xiao et al. *PandaSet: Advanced Sensor Suite Dataset for Autonomous Driving*. Dec. 23, 2021. arXiv: 2112.12610[cs].

[66] Zhenming Xie and Rajesh Rajamani. “On-Bicycle Vehicle Tracking at Traffic Intersections Using Inexpensive Low-Density Lidar”. In: *2019 American Control Conference (ACC)*. 2019 American Control Conference (ACC). Philadelphia, PA, USA: IEEE, July 2019, pp. 593–598. DOI: 10.23919/ACC.2019.8814442.

[67] Yan Yan, Yuxing Mao, and Bo Li. “SECOND: Sparsely Embedded Convolutional Detection”. In: *Sensors* 18.10 (Oct. 6, 2018), p. 3337. DOI: 10.3390/s18103337.

[68] Jihan Yang et al. *ST3D: Self-training for Unsupervised Domain Adaptation on 3D Object Detection*. Mar. 27, 2021. arXiv: 2103.05346[cs].

[69] Jihan Yang et al. *ST3D++: Denoised Self-training for Unsupervised Domain Adaptation on 3D Object Detection*. Aug. 15, 2021. arXiv: 2108.06682[cs].

[70] Zetong Yang et al. “3DSSD: Point-Based 3D Single Stage Object Detector”. In: *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR). Seattle, WA, USA: IEEE, June 2020, pp. 11037–11045. DOI: 10.1109/CVPR42600.2020.01105.

[71] Zeng Yihan et al. “Learning Transferable Features for Point Cloud Detection via 3D Contrastive Co-training”. In: *Advances in Neural Information Processing Systems*. Vol. 34. Curran Associates, Inc., 2021, pp. 21493–21504.

[72] Tianwei Yin, Xingyi Zhou, and Philipp Krähenbühl. *CenterPoint: Center-based 3D Object Detection and Tracking*. Jan. 6, 2021. arXiv: 2006.11275[cs].

[73] Yurong You et al. “Learning to Detect Mobile Objects from LiDAR Scans Without Labels”. In: *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR). New Orleans, LA, USA: IEEE, June 2022, pp. 1120–1130. DOI: 10.1109/CVPR52688.2022.00120.

[74] Yurong You et al. *Unsupervised Adaptation from Repeated Traversals for Autonomous Driving*. Mar. 27, 2023. arXiv: 2303.15286[cs].

[75] Philipp Zell et al. “LiDAR Technology for Autonomous Driving and its Integration into Personal Vehicles”. In: *Sensors* 20.15 (2020), p. 4140. DOI: 10.3390/s20154140.

[76] Wu Zheng et al. *SE-SSD: Self-Ensembling Single-Stage Object Detector From Point Cloud*. Apr. 20, 2021. arXiv: 2104.09804[cs].

[77] Yin Zhou and Oncel Tuzel. *VoxelNet: End-to-End Learning for Point Cloud Based 3D Object Detection*. Nov. 16, 2017. arXiv: 1711.06396[cs].

[78] Walter Zimmer et al. *A Survey of Robust 3D Object Detection Methods in Point Clouds*. version: 1. Mar. 31, 2022. arXiv: 2204.00106[cs].

A

SenseBike Manual

This manual discusses the usage of the *SenseBike*, a specialized bicycle owned by the TU Delft Robotics Department. As this manual serves as an appendix to the thesis report, it sometimes references parts of Chapter 3 to avoid redundancy. In the standalone version of this manual, these chapters are combined into one for clarity and coherence.

The manual begins with an extensive overview of the hardware, followed by an in-depth look at the software and the process of installing new updates. The manual ends with a straightforward, step-by-step guide on starting a recording, requiring no prior knowledge of the software or its structure.

A.1. Hardware

This section focuses on all the hardware present on the bicycle which makes the bicycle a data-collection vehicle. We do not take into consideration the standard electricity bicycle hardware. A brief overview and visualization of this hardware can be found in Section 3.1. Table A.1 gives a more in-depth explanation of all the hardware components and describes how they are connected.

A.1.1. Hardware Modules

Table A.1: Overview of hardware modules and how they are connected

	Component	Explanation
1.	M1P LiDAR interface + LiDAR electronics	The green box contains cables and cable dividers, ensuring the LiDARs are powered and can be switched on or off. The black box is the interface module of the front LiDAR.
2.	Ethernet / Network Switch	This is a device that turns one ethernet port into multiple ports, in this case, five.
3.	Voltage Regulators	These are transformers that convert the incoming 48V from the battery pack (5) into multiple 12V outputs. Both these inputs and outputs use an XT60 connector.
4.	Amp fuse	This is an ampere fuse located between a 12V output and the LiDAR circuit, preventing potential amp overload on the LiDARs. It contains a 5A fuse.
5.	Battery Pack	A 48V 17.5Ah battery pack, with a XT60 connection.
6.	GNSS / RTK Antenna	This antenna is connected to the GNSS / RTK module (11). Does not need any active power.

Continued on next page

Table A.1 – *Continued from previous page*

	Component	Explanation
7.	Router	The Teltonika Router is configured as an Access Point (AP). This means it both receives internet and can be connected to via WiFi. The SSID is SenseBike. Its settings can be changed in its WebUI (192.168.1.102)
8.	USB-C Hub	This module effectively turns one USB-C port into multiple ports, in this case, five. The GNSS / RTK module (6) and IMU (13c) are connected and powered through this hub.
9.	NVIDIA Jetson Nano Orin NX (16GB RAM)	This is the core of the bicycle sensors. It is a small Linux (Ubuntu 22.04) PC with significant GPU power. To display it, connect to the DisplayPort connection.
10.	Spacer / Voltage regulator	This component partly serves as space for cables and partly as a 48V to 24V transformer.
11.	GNSS / RTK module	The ArduSimple simpleRTK2B GNSS/RTK module is connected to the Jetson (9) through the USB-C hub (8). GNSS is the name for all GPS-like systems on Earth.
12.	Rear Camera	This is an Arducam 12MP IMX477. It is both powered and connected through the HDMI cable.
13.	LIDAR Mount	This mount contains LiDARs (13A) and an IMU (13B).
13A	2x Rotating LiDAR + 2x interface box	These are two Robosense Helios 32 70° FOV LiDARs. Each LiDAR is a 32-beam LiDAR rotating at 10Hz. Both are connected to a separate LiDAR Interface Box. These are powered by the LiDAR electronics box (1) and connected to the Jetson via an ethernet cable in the ethernet switch (2).
13B	IMU	The SparkFun 9DoF ICM-20948 (Qwiic) Internal Measurement Unit (IMU) measures linear acceleration, angular velocity, and magnetic field in x, y, and z directions, hence (3x3=) 9 DoF. It is connected to a Teensy board (Arduino-like), which is connected to the Jetson through the USB-C hub (8).

A.1.2. Power consumption

When adding new hardware to the *SenseBike*, ensure it does not exceed the transformer's power capacity. The transformer converts $48V$ from the battery to a usable voltage and can supply up to $10A$, providing a maximum of $120W$. An overview of what it does is provided in Figure A.1. Most devices need more power at start-up than during regular operation. Currently, the Nano Jetson shuts down when the LiDAR circuit is started because the LiDARs require a significant portion of the $120W$. Future updates should consider using a different transformer.

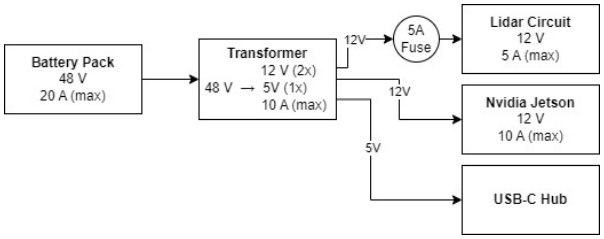


Figure A.1: The circuit around the transformer on the Sensebike

A.2. Software

An overview of the deployed software architecture is provided in Figure 3.8. This software is all stored in the *sensebike_delft*-directory in the *home*-directory in the Ubuntu 20.04 UI on the Nvidia Jetson Orin Nano. This section starts with an explanation of the folder structure within this *sensebike_delft*-directory. It then discusses the contents of the currently existing containers. We then discuss the ins-and-outs of the Robot Operating Software (ROS) on the bicycle and end with a step-by-step guide on how to install a new container or change the WebUI.

A.2.1. Docker containerization

As mentioned, the bicycle runs every single ROS-node in a separate Docker container. A Docker container is a lightweight, isolated environment that packages software and its dependencies. This avoids dependency issues across multiple ROS-packages. The big advantage is that we can install specific versions of Ubuntu, ROS, and Python in every individual container. It is also fairly easy to start a Docker container through the WebUI, which we can open on a separate device as a mobile phone. A drawback is that we have to install these in every single container, consuming a lot of storage space. To run a container, we first need to build its blueprint, also referred to as its image. We can then run the software installed on this image, opening up a container.

If you want to create your own docker image, please follow the official Docker Guides (Link). This section briefly explains the main principles of containerization on the bicycle, for a better understanding of the other sections.

Dockerfile

After installing Docker on the main Ubuntu system, we can build Docker images using a Dockerfile, or by pulling them from an internetsource, such as DockerHub. We can also combine both by pulling a docker image and adding our specific packages to it. The latter method is mainly used on the bicycle. An example of this method is provided below. These lines of code are commonly saved as a *.Dockerfile* and to build a Docker image from it, we can execute *docker build -f path/to/.Dockerfile*. When built, the lines in this *.Dockerfile* do the following:

- Line 1: Pull a docker with the ROS Galactic software in it, downloaded from DockerHub.
- Line 2 & 3: Add a specific key to *sources.list.d*. This step is similar to what you would do when installing ROS regularly.
- Line 4 & 5: Install the ROS-galactic software.

```
1 FROM dustynv/ros:galactic-ros-base-l4t-r35.1.0
2 RUN wget -O - https://apt.kitware.com/keys/kitware-archive-latest.asc 2>/dev/null
      | gpg --dearmor - | tee /usr/share/keyrings/kitware-archive-keyring.gpg >/dev/
      null \
3     && echo 'deb [signed-by=/usr/share/keyrings/kitware-archive-keyring.gpg] https
          ://apt.kitware.com/ubuntu/ focal main' | tee /etc/apt/sources.list.d/
          kitware.list >/dev/null
4 RUN apt-get update && \
5     apt-get install -y --no-install-recommends ros-galactic-rosbag2-storage-mcap
```

We now have a Docker image, which is the blueprint for a container. To name this container, we use the *-t* argument while building. Let's say we name the above example as *ros-galactic-image*, by using *docker build -f path/to/.Dockerfile -t ros-galactic-image*.

On the bicycle, we typically organize all the code for one Docker image in a single directory, such as in *sensebike_delft/combo-bag/rosbag*. To avoid repeatedly typing the *docker build* command with all its arguments, we place this command in an executable shell script called *install.sh*. This allows you to build the image simply by running *./install.sh*. Note that any changes made to the *Dockerfile* will only be reflected in the Docker image after running *docker build* again.

Starting a container

To run an image and start a container, we use the *docker run* command. For example, to run the image *ros-galactic-image*, we would use *docker run ros-galactic-image*. However, this starts a container

without allowing input or output interactions, and it cannot access any files on our main Ubuntu (host) system. Additionally, the container is given a random name by default. We can use arguments to modify these behaviors. The most commonly used arguments on the bicycle include:

- **-d, –detach**: Run the container in detached mode, running in the background and printing the container ID.
- **–rm**: Automatically remove the container when it exits, useful for temporary containers.
- **-v, –volume**: Bind mount a volume, mapping a host path to a container path. Syntax: *-v /host/-path:/container/path*. This argument ensures that we can open files from the host system within the container, or access files inside the container from our host system.
- **–name**: Assign a name to the container. If not specified, Docker generates a random name.
- **–network**: Connect the container to a specific network, allowing communication with other containers on the same network. Syntax: *–network network-name*.
- **–privileged**: Give extended privileges to the container, allowing it to perform actions typically restricted to the host system.
- **-i and -t or -it**: Allocate a pseudo-TTY (*-t*) and keep STDIN open even if not attached (*-i*). Combined as *-it*, they enable an interactive terminal session inside the container.

An example of the initialization of the record container is provided below. It starts a container from the *ros-galactic-image* image, which it names *rosbag_container*, removes it when it is closed, and uses the network *host* to communicate with the host (main installation of Ubuntu on Nvidia Jetson). It creates some volumes, meaning it can access host directories. When starting up, it immediately executes the */workspace/entry.sh* file, which we will discuss below. Note that only *ros-galactic-image* and */workspace/entry.sh* should be placed at the bottom, the rest of the order is irrelevant.

```
docker run \
    --name=rosbag_container \
    --rm \
    --network=host \
    -v /etc/localtime:/etc/localtime:ro \
    -v ${recording_path}:/output \
    --env BOREAL_ROSBAG_NAME=rosbag \
    -v ${CONFIG.get("REPO_ROOT")}/combo-bag/rosbag:/workspace \
    ros-galactic-image \
    /workspace/entry.sh
```

The *entry.sh* script is executed right after start-up of the docker container. This option is used for almost every container on the *SenseBike*. An example of the *entry.sh* script from the example container is shown below. We first source the ROS environment, then go to the correct directory and define the topics. These are read from a separate *.txt*-file. Lastly, it starts recording these topics, until the container is closed.

```
#!/bin/bash
source /opt/ros/galactic/setup.bash

cd /output
topics=$(cat /workspace/topics.txt)

ros2 bag record \
    -o /output/$BOREAL_ROSBAG_NAME \
    $topics \
    -d 60
```

A.2.2. Filesystem

A tree of the main folders within the *sensebike_delft* directory is shown below. This section provides a brief overview of the contents of each folder. When we mention that a folder contains the files for a certain container, we refer to the files as explained in subsection A.2.1.

```
sensebike_delft
├── combo-bag
│   ├── rosbag
│   └── video-recording
├── configuration
├── gpsd
├── linuxptp
├── ros_nodes
│   ├── gps_ws
│   ├── teensy
│   ├── imu_tools
│   ├── rslidar_ws
│   ├── kiss_icp
│   ├── robot_localization
│   └── tf_ws
├── services
└── websocket
    ├── file-explorer
    ├── flask_web
    └── ros_websocket
```

combo-bag

In the combo-bag directory, we find three subdirectories: 1) *fusion*, 2) *rosbag* and 3) *video-recording*. The *rosbag*-directory contains the files for the recording (ROS) container, and the video-recording contains files that read the input from the cameras and turn it into a *.mp4*-video. Inside *fusion*, we find a python script that can combine a recorded rosbag and video-recording into one merged *zip*-file.

configuration

The configuration directory contains a file named *config*, which defines all the Docker environment variables. Currently, these include:

- DNS_NAME = *boreal.control*
- REPO_ROOT = */home/boreal/sensebike_delft*
- LIDAR_HOST = *192.168.1.200*
- ADHOC_HOST = *192.168.1.102*
- ADHOC_PORT = *5000*
- FILE_EXPLORER_PORT = *5002*
- ROS_WEBSOCKET_PORT = *5003*
- RECORDING_ROOT = */home/boreal/captured_data*
- GPS_FREQ = *7*

These variables are loaded in the executables of other Docker containers. To update them, you can add or modify any of the above variables. After making changes, execute the *run.sh* script with *sudo ./run.sh*.

gpsd

This directory holds the files for a GPSD container. It is a Docker container designed to run *GPSD* (GPS daemon) with specific configurations. The container is automatically launched at startup using a systemd service unit named boreal-gpsd.service. *GPSD* must be running for the GNSS module to function correctly. Please note, that this directory does not contain the files for the GNSS ROS-node.

linuxptp

Inside this directory, we find the files for the PTP container, which ensures time synchronization between the LiDARs and the host computer. It also contains the file necessary to open the UI of the M1P LiDAR.

When the Precision Time Protocol (PTP) container is started, it runs the following command: `ptp4l -E -2 -S -i eth0 -m`. The *ptp4l* command is used to start the PTP daemon from the `linuxptp` package. The line arguments are defined as follows:

- **-E**: Using the delay request-response mechanism, instead of a peer delay mechanism.
- **-2**: Selecting IEEE 802.3 network transport, instead of IPv4 or IPv6.
- **-S**: Software timestamping instead of hardware.
- **-m**: Printing messages to the terminal.
- **-i eth0**: Selecting the ethernet0 port as PTP port, which is the ethernet port to which all LiDARs are connected.

ros_nodes

Inside this directory, we find all the Robot Operating System (ROS) packages, except for the one that rosbags when recording. Every subdirectory is a ROS workspace, containing both the container files and all the necessary ROS packages source code. When the container runs, it creates a volume (argument *-v*) with this same workspace, enabling access from within the container to the host workspace directory and vice-versa. All the ROS nodes are discussed in more detail in subsection A.2.3. A list of the subdirectories here includes:

- gps_ws
- teensy
- imu_tools
- rslidar_ws
- kiss_icp
- robot_localization
- tf_ws

services

This directory contains the scripts and configuration files needed to set up various services on the system. It includes files to ensure the *gpsd* container is always running, as well as services for configuring the Ubuntu environments in the Docker containers. If you want to add something to the ROS environment, you likely will not need this directory.

websocket

The websocket directory contains three subdirectories: 1) *file-explorer*, 2) *flask_web*, and 3) *ros_websocket*.

1) The *file-explorer* contains the code for WiFile, which enables exploration of the host computer's complete directory and the downloading of any file using a Flask Server. All the recordings from the bicycle are stored as a *.zip*-file, after which they can be downloaded to a device connected to the bicycle's WiFi network.

2) The WebUI runs on a Python Flask server, of which the source code can be found in the *flask_web*-directory. This Flask Server is mainly used to start or stop docker containers.

3) This subdirectory contains the source code and container files for the ROS2 Bridge, which includes the ROS2-websocket. This ROS2 websocket server, even though it runs inside a container, can communicate through the host with all the other ROS nodes. This enables visual feedback in the WebUI about whether a ROS node is working properly.

A.2.3. ROS Environment

This section only focuses on the ROS nodes that all run in their separate container but communicate through each other on the host network. That is made possible by running every separate container with the *–network=host* argument. From now on, we will refer to them as ROS nodes. We discuss the same ones as listed in Appendix A.2.2. A visualization with RQT-graph of all the communication of the ROS nodes is shown in Figure A.2.

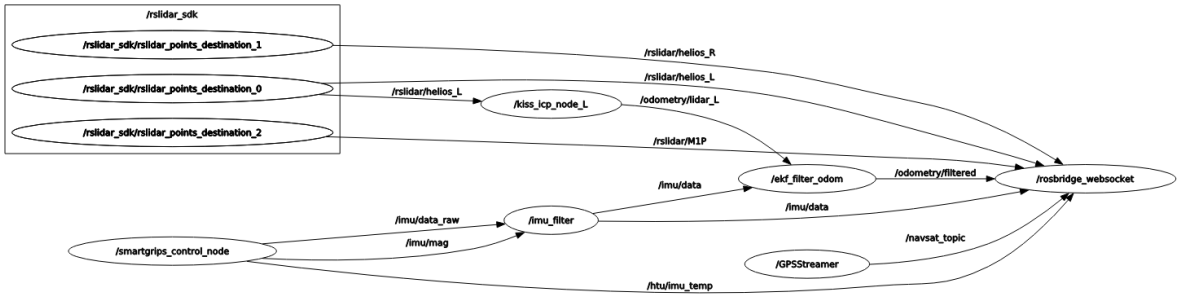


Figure A.2: RQT_graph of the ROS nodes and their subscriptions + publications on every topic.

GPS_WS

The GPS ROS-node includes, besides the ROS-node-code, a forked version of the Ublox driver [17], a block that transforms all incoming raw GNSS data to useful units. We use a `sensor_msgs/NavSatFix` message and publish the GPS data onto the `navsat_topic` topic with a frequency of 7 *Hz*.

Teensy / IMU

The IMU ROS node incorporates both the Teensy Telemetry package and the IMU ROS node code. The Teensy Telemetry package includes a *.ino* file, similar to those used with Arduino, which processes input from the Sparkfun IMU. It converts the raw voltage inputs into physical units such as m/s^2, rad/s, and mT, and publishes these data through the ROS2 node to the ROS2 core. Specifically, it publishes a `sensor_msgs/Imu` message on the `imu/data_raw` topic. This message type can contain data on 3D orientation, 3D angular velocity, and 3D linear acceleration, although the orientation field is not populated in this case. Additionally, the ROS node publishes a `sensor_msgs/MagneticField` message to the `\imu\mag` topic. Both of these topics are recorded and used by the IMU filter node.

IMU_tools

The IMU filter Ros node starts the IMU filter from [29]. We use the Madgwick filter which fuses angular velocities, accelerations, and (optionally) magnetic readings from a generic IMU device into an estimated orientation, based on [39]. As input we take the output from the IMU node, containing the `sensor_msgs/Imu` and `sensor_msgs/msg/MagneticField`, transform this into an orientation, appending this to the IMU message and publish it back on topic `imu/data`. We use the magnetic field to calculate the initial orientation at the start of each recording. However, as the Teensy microcontroller is located right on top of the IMU, this magnetic field may experience some magnetic distortion from the electronics of the Teensy board, leading to some undesired initial orientation errors.

RS-LiDAR_WS

Inside the LiDAR container, we find the software development kit (SDK) provided by *Robosense* [34]. It includes both the necessary drivers and the ROS2 node. As depicted in Figure 3.8, the LiDAR container only initiates if the PTP container is running, which is discussed hereafter. The driver can handle multiple LiDARs, so the node is only run once, publishing three `sensor_msgs/PointCloud2` messages on topics `/rslidar/helios_L`, `/rslidar/helios_R`, and `/rslidar/M1P` at a frequency of 10 Hz. Both Helios LiDARs are configured to avoid emitting points towards each other. Meaning, within the range of plus or minus 89° to 91°, preventing the generation of incorrect outlier points.

KISS_ICP

As the Localization node inside the localization container works much better with at least one type of odometry as input, we estimate this by applying KISS-ICP [62] to one of the Helios pointclouds,

currently the one on the left. The KISS-ICP (Keep It Small and Simple - Iterative Closest Point) algorithm effectively estimates the transformation between two point clouds in an iterative way. As input, it takes one LiDAR pointcloud message and publishes a `sensor_msgs/Odometry` to the `/odometry/lidar` topic. We currently use the pointcloud from the left Helios LiDAR.

Robot_Localization

The localization container runs the Robot Localization package from *cra-ros-pkg* [14]. The Robot Localization package is a set of ROS (Robot Operating System) nodes and an Extended Kalman Filter (EKF) algorithm designed to improve localization accuracy for mobile robots. It can combine inputs from GPS, IMU, and odometry and transform this into a combined and more robust odometry message. The incoming odometry can come from various sensors, such as cadence sensors, wheel sensors, or liDAR odometry. As it uses multiple sensors as input, the chances of noise are reduced. The whole package estimates a 15-dimensional state of the *SenseBike*:

$$(X, Y, Z, roll, pitch, yaw, X^{\cdot}, Y^{\cdot}, Z^{\cdot}, roll^{\cdot}, pitch^{\cdot}, yaw^{\cdot}, X^{\cdot\cdot}, Y^{\cdot\cdot}, Z^{\cdot\cdot}).$$

We implement the recommended configuration as provided by [14], illustrated in Figure A.3. However, we deviate by employing LiDAR odometry instead of wheel odometry. The KISS-ICP node within the LiDAR odometry container computes the LiDAR odometry. Notably, the system comprises three active nodes: an Extended Kalman Filter (EKF) local node, an EKF global node, and the navsat_transform node. The EKF nodes utilize an omnidirectional motion model to project the state forward in time and refine this projection based on observed sensor data. Conversely, the navsat_transform node generates an odometry message in a coordinate system consistent with the robot's global frame, facilitating direct integration into the state estimate. As input, we take the messages from the IMU filter, the GPS node, and the LiDAR Odom node and publish the results as a `sensor_msgs/Odometry` to the `/odometry/filtered/global` topic. The package automatically calculates the transforms from `odom` -> `base` and from `map` -> `odom` and publishes them to the `/tf` topic.

Due to the substantial computational needs of the LiDAR odometry node, it currently operates at lower frequencies, limiting the frequency of updates to the Robot Localization package. Future updates will incorporate odometry data from the cadence sensor, ensuring that the Localization package can achieve a publishing frequency of 30 Hz. This enhancement will enable more frequent and precise *Sensebike's* localization.

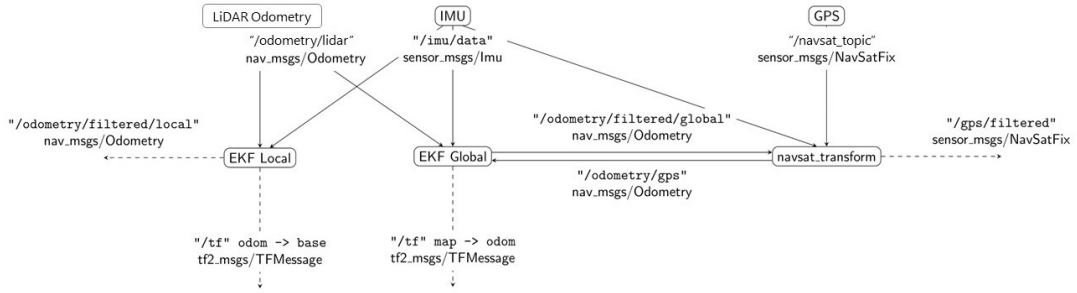


Figure A.3: Setup for the robot localization package. We run three ROS2 nodes: 1) EKF Local, 2) EKF Global and 3) navsat_transform. Taken and adapted from [14].

Transform container

For visualization and conventional purposes, we initiate a Docker container that launches multiple ROS2 Static Transform nodes, specifying that the transformation between all LiDARs, the IMU, and the GPS module is zero. This convention primarily facilitates compatibility with the ROS framework and enables sensor visualization in RVIZ, rather than serving as a means for actual data collection, and thus, is not shown in Figure 3.8.

A.2.4. Changing the WebUI

The WebUI is fully defined in the directory */sensebike_delft/websocket/flask_web*. This directory contains both the back-end and front-end code, as well as supporting *HTML* and *JavaScript* files. To modify the WebUI, first change the front-end, and then update the actions that occur when a specific button or tab in the front-end is selected.

Front-end

Most of the front end is defined in the *./templates/index.html* file. To open it, simply drag it into a browser. If you want to add a button, tab, or grid item, just copy and paste an existing element. Be sure to give it a unique ID, as this is what the back-end reads. What you define as the title will appear in a small grey information box when you hover your cursor over the item. In file *./static/css/style.css*, all the sizes and colors of every element are defined.

Back-end

The back-end contains two different sections:

1. A *javascript* part, containing most web-utilities. These web utilities include coloring the hardware statuses, functions for restarting the system, or definitions of what happens when a certain button is pressed.
2. A flask Python and server, which mainly starts reading the web UI and can start certain processes on the host system

Here is an example of how these two work together:

Suppose the web UI is accessed by navigating to *ADHOC_HOST_WEBSOCKET_PORT* in a browser, which is *192.168.1.102:5000*. When we press the ”Start ROS Nodes” button in the web UI, it triggers the following JavaScript function. This function sets the value of *192.168.1.102:5000/start_ros_nodes* to True. The *start_ros_nodes* is the *id* of the front-end definition of the button. This function is defined in *./static/js/ros_websocket.js*.

```
const xhr = new XMLHttpRequest();
xhr.open(
  "GET",
  "http://" + ADHOC_HOST + ":" + ADHOC_PORT + "/start_ros_nodes",
  true
);
xhr.onload = function () {
  if (this.status === 200) {
  }
};
xhr.send();
```

To test this, open the page *192.168.1.102:5000/start_ros_nodes* on the *SenseBike* and press the *Start Ros Nodes* button somewhere else on the Web UI. When reloading, it changes state.

The Flask server is constantly reading the *192.168.1.102:5000/start_ros_nodes* webpage and once it turns its state to True, it triggers the following python function from *./backend.py*. This is just part of the whole function, as it opens multiple Docker containers.

```
@cross_origin()
@backend_blueprint.route('/start_ros_nodes', methods=['GET'])
def start_ros_nodes():

    # OPEN UBLOX GPS -> ROS2 DOCKER
    subprocess.Popen(['docker', 'run', '-d', '--rm',
            '--device=/dev/boreal-ublox-gps:/dev/boreal-ublox-gps',
            '-v', '/etc/localtime:/etc/localtime:ro',
            f"--volume={CONFIG.get('REPO_ROOT')}/ros_nodes/gps_ws:/workspace",
            '--name=ublox_container','--network=host',
            'dustynv/boreal-ublox-gps:latest',
            '/bin/bash', '-c', '"/workspace/entry.sh"'])
```

A.3. How to collect data

This section contains a step-by-step guide on how to record data from the *SenseBike* and how to collect it. The steps include:

1. Booting up the SenseBike
2. (Optional) Rosbag settings
3. Connecting to the SenseBike WiFi
4. Opening the WebUI
5. Recording
6. Transferring data

Booting up the SenseBike

Ensure the SenseBike is properly charged or connected to a power source. The bike has two battery packs: one in the frame that powers the electric motor, and another that powers the entire data collection system. To switch on the data collection hardware, press the red button located at the rear on the bottom. Wait a couple of minutes for the onboard computer and router to start up.

[Optional] Rosbag settings

If you want to record only specific topics or add any other *ros2bag*-record settings such as maximum duration or storage type, connect the onboard computer to a display, go to the file */sensebike_delft/combo_bag/rosbag/entry.sh* and add or remove *ros2bag* arguments there.

Connecting to the SenseBike WiFi

Open the WiFi settings on your device (laptop, smartphone, tablet). Search for available networks and select the *SenseBike* WiFi network. Enter the required password.

Opening the WebUI

Open a web browser on your device connected to the SenseBike WiFi. Enter the IP address of the WebUI (e.g., `192.168.1.102:5000`) in the address bar. The WebUI should load, showing a TU Delft logo at the top. A screenshot of the WebUI on an Android phone is illustrated in Figure A.4.

Recording

Navigate to the recording section on the WebUI. Press the ”Start Recording” button to begin capturing data. To stop recording, press the ”Stop Recording” button. While recording, you should see the values behind *Recording Now* go up. With three LiDARs on, it typically goes up to 50 MiB / second. If this is not the case, either press ’End Recording’ and try again. It can also help to press ’Stop Sensors’, wait for a minute, press ’Start Sensors’, wait another minute, and ’Start Recording’ again. If that does not work, reboot the whole system by going to the ’System’ tab and pressing reboot.

Transferring data

Connect a USB storage device or external SSD to the SenseBike. Navigate to the data transfer section on the WebUI, located at */captured_data/*. Select the data files you wish to transfer and choose the USB storage device as the destination. You can use any USB-A or USB-C port on the onboard computer. The USB-C ports on the back of the SenseBike also work but are relatively slower compared to the ones on the computer itself.

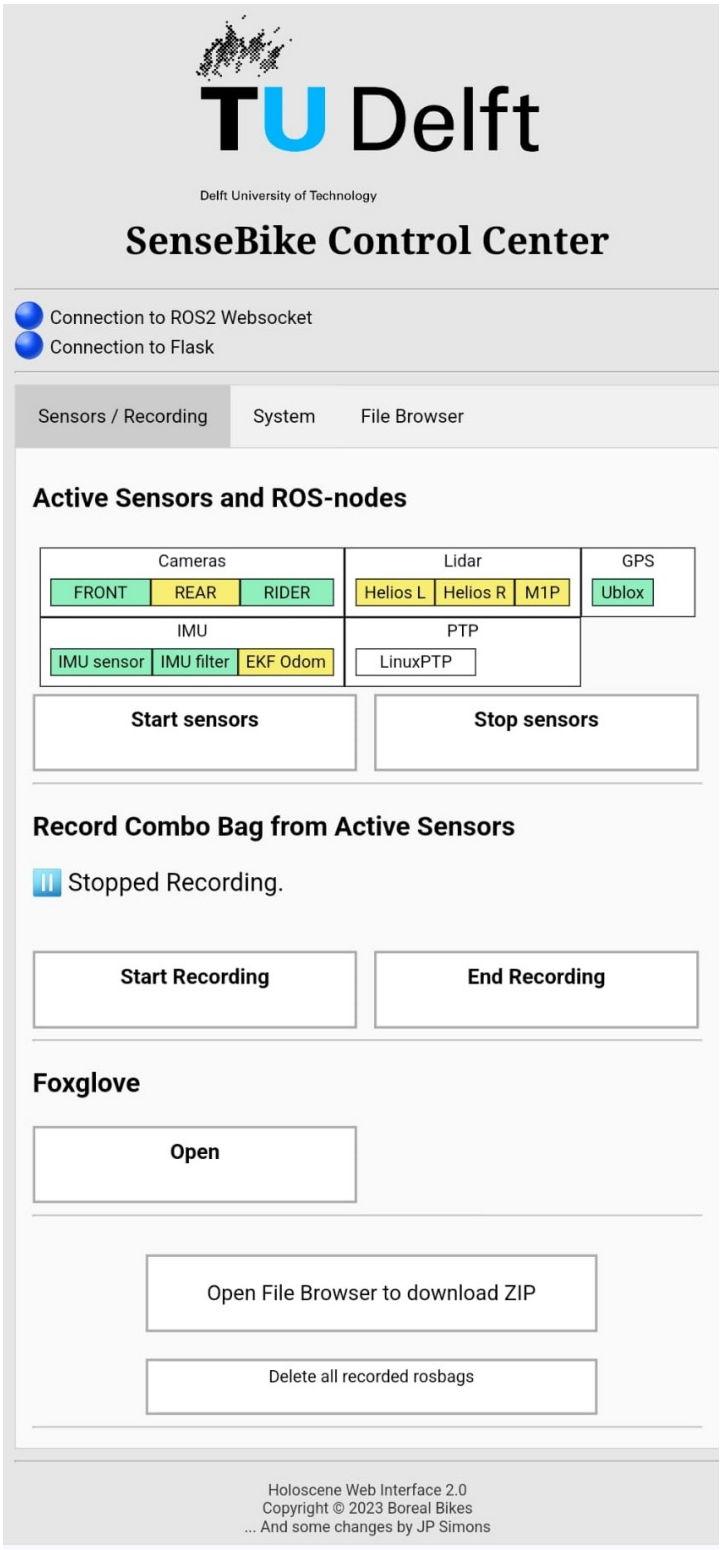


Figure A.4: Screenshot from an Android phone, of the WebUI used to record data. A green or yellow box indicates whether a sensor or ROS-node is functioning properly.

B

Sensebike Dataset

B.1. Coordinate System

The coordinate system used throughout the report is a right-handed orthogonal one, consisting of a longitudinal *x*-axis, a lateral *y*-axis, and a vertical *z*-axis. Figure B.1 presents a visualization of the model and its Degrees of Freedom. When viewing the system from the rear, the positive axis orientations are as follows: *x* points forward, *y* points left, and *z* points up. The *z*-axis is located exactly between the two Helios LiDARs, meaning the Euclidean distance from the centers of the two Helios LiDARs to the *z*-axis is equal. The *x*-axis passes through the two points where the rear and front tires make contact with the road surface.

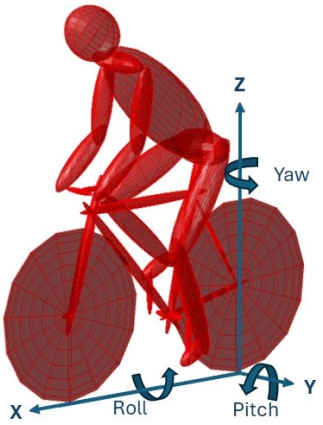


Figure B.1: 3D View of the used coordinate system. Taken and adapted from [10].

B.2. Class Mappings

Table B.1 provides the remapping of classes used in other datasets to our three categories: Vehicle, Cyclist, and Pedestrian. Any class not included in this list is considered as not an object.

Original Class KITTI / nuScenes / Lyft / Waymo	**Mapped Class**
car	Vehicle
Car	Vehicle
truck	Vehicle
bus	Vehicle
construction_vehicle	Vehicle
trailer	Vehicle
Vehicle	Vehicle
motorcycle	Cyclist
bicycle	Cyclist
Cyclist	Cyclist
pedestrian	Pedestrian
Pedestrian	Pedestrian

Table B.1: Class Mapping to the three *SenseBike* Classes

B.3. Recorded sequences

The exact amount of *.pcd*-files, durations and memory sizes of all the used dataset sequences can be found in Table B.2.

Sequence	***.pcd* files**	**Length (s)**	**Total Size (GB)**
2024-04-03_12-50-36	1092	115	5.97
2024-04-03_12-52-33	1123	117	6.13
2024-04-03_12-53-42	1037	110	5.67
2024-04-03_12-56-36	8636	907	46.25
2024-04-03_12-56-42	1045	111	5.69
2024-04-03_15-17-39	1697	180	9.59
2024-04-03_15-18-03	885	93	4.96
2024-04-03_15-21-01	2029	215	11.40
2024-04-03_15-22-09	1337	140	7.49
2024-04-03_16-47-18	1316	139	7.30
2024-04-03_18-02-45	2384	250	13.28
2024-04-12_11-24-18	2748	289	15.29
2024-04-12_11-28-59	2444	258	13.45
2024-04-12_12-30-34	3749	395	20.98
2024-04-12_12-37-04	2379	249	13.16
2024-04-12_12-41-56	1363	145	7.21
2024-04-12_12-47-01	6822	716	37.85
2024-04-12_13-34-16	6302	664	34.68
2024-04-12_13-45-23	2932	309	15.85
2024-04-12_13-50-45	4727	500	26.04
2024-04-12_13-59-35	1243	130	6.91
2024-04-12_14-04-04	4755	501	26.66
2024-04-23_11-34-15	6945	733	38.16
2024-04-23_11-40-08	8444	883	46.24
Total	**77474**	**8724**	**457.21**

Table B.2: PCD File Counts, Lengths, and Directory Sizes

B.4. Algorithms

B.4.1. Normal Distributions Transform (NDT)

The NDT algorithm [7] divides 3D space into a grid of voxels, fitting a Gaussian distribution to the points in each voxel. To align two point clouds, it estimates the transformation by maximizing the likelihood that points from one cloud fit the distributions of the other, refining the alignment iteratively for accurate registration. This continues until reaching a specific Euclidean fitness score, which is the sum of squared distances from the output cloud to the closest point in the target cloud. NDT is faster than the traditional Iterative Closest Point (ICP) algorithm, especially for larger point clouds, but it requires a good initial guess to avoid diverging from an optimized solution [54].

B.4.2. SimpleTrack Kalman Filter

The Kalman filter in SimpleTrack [43], which is used as Multi-Object Tracker in *MS3D++* [59], functions as a key component for tracking objects by estimating their state over time. It operates by predicting the future state of an object based on its previous state and accounting for uncertainties in the model and measurements. The filter uses a two-step process: prediction and update. In the prediction step, it projects the current state and uncertainty into the next time step. During the update step, it incorporates new measurements to refine the state estimate and reduce uncertainty. This iterative process allows SimpleTrack to maintain accurate and smooth object trajectories even in the presence of noise and occlusions.

C

Domain Adaptation

C.1. Detector Ensemble configurations

Table C.1 gives the configurations of the Standard (STD) detector ensemble, as explained in section 4.2. The extra used models and their configurations for the full detector ensemble (DE) are shown in Table C.2. Note that the full DE contains both the models from the standard DE as the extra DE. All models use a 2D Base BEV Backbone and a HeightCompression Map to Bird's Eye View (BEV). Table C.3 provides an overview of the used anchors for the models using an anchor Dense Head.

Model Name	Training Data	Dense Head	VFE	Backbone 3D	Point-Head	ROI-Head
VoxelRCNN	Lyft	Anchor	DynMeanVFE	VoxelBackBone8x	-	VoxelRCNNHead
VoxelRCNN	Lyft	Center	DynMeanVFE	VoxelBackBone8x	-	VoxelRCNNHead
VoxelRCNN	nuScenes	Anchor	DynMeanVFE	VoxelBackBone8x	-	VoxelRCNNHead
VoxelRCNN	nuScenes	Center	DynMeanVFE	VoxelBackBone8x	-	VoxelRCNNHead
VoxelRCNN	Waymo	Anchor	DynMeanVFE	VoxelBackBone8x	-	VoxelRCNNHead
VoxelRCNN	Waymo	Center	DynMeanVFE	VoxelBackBone8x	-	VoxelRCNNHead

Table C.1: Configurations for STND Detector Ensemble

Model Name	Training Data	Dense Head	VFE	Backbone 3D	Point-Head	ROI-Head
PV-RCNN++	Lyft	Anchor	MeanVFE	VoxelResBackBone8x	PointHeadSimple	PVRCNNHead
PV-RCNN++	nuScenes	Anchor	MeanVFE	VoxelResBackBone8x	PointHeadSimple	PVRCNNHead
PV-RCNN++	nuScenes	Center	MeanVFE	VoxelResBackBone8x	PointHeadSimple	PVRCNNHead
CenterPoint	nuScenes	Center	MeanVFE	VoxelResBackBone8x	-	-
VoxelRCNN	Sydney	Center	DynMeanVFE	VoxelResBackBone8x	-	VoxelRCNNHead
PointRCNN	KITTI	-	MeanVFE	UNetV2	-	PartA2FCHead

Table C.2: Configurations for STND+ & Ours Detector Ensemble

	Vehicle	Cyclist	Pedestrian
Anchor Sizes	[4.7, 2.1, 1.7]	[1.78, 0.84, 1.78]	[0.91, 0.86, 1.73]
Anchor Rotations	[0, 1.57]	[0, 1.57]	[0, 1.57]
Anchor Bottom Heights	[0]	[0]	[0]
Align Center	False	False	False
Feature Map Stride	8	8	8
Matched Threshold	0.55	0.5	0.5
Unmatched Threshold	0.4	0.35	0.35

Table C.3: Anchor Configurations

C.2. Detector model configurations

Table C.4 shows the configurations of the detectors that are trained on the pseudo-labels.

Model Name	Training Data	Dense Head	VFE	Backbone 3D	Point-Head	ROI-Head
CenterPoint	-	Center	DynMeanVFE	VoxelBackBone8x	-	-
PV-RCNN++	-	Center	DynMeanVFE	VoxelResBackBone8x	PointHeadSimple	PVRCNNHead
VoxelRCNN	-	Center	DynMeanVFE	VoxelBackBone8x	-	VoxelRCNNHead

Table C.4: Configurations for three detectors

C.3. Resource usage

This section gives three tables on the GPU usage in hours for three different steps in the domain-adaptation pipeline. Note that we tried two different sets of threshold configurations for pseudo-label set *Ours*, referring to *TH1* and *TH2*. All GPUs were randomly selected from Nvidia V100, A100, or A40 models. The three steps include:

1. Table C.5: generating the multi-source detections
2. Table C.6: combining these detections into one pseudo-label set
3. Table C.7: training detectors on these pseudo-labels.

Detector Name	Original Training Data	GPUs	Duration
VoxelRCNN - center	Lyft	4	17 h
VoxelRCNN - anchor	Lyft	4	19.5 h
VoxelRCNN - center	nuScenes	4	18.5 h
VoxelRCNN - anchor	nuScenes	4	16 h
VoxelRCNN - center	Waymo	4	16.5 h
VoxelRCNN - anchor	Waymo	4	21 h
PV-RCNN++ - anchor	Lyft	4	19 h
PV-RCNN++ - anchor	nuScenes	4	14 h
PV-RCNN++ - center	nuScenes	4	20.5 h
CenterPoint - center	nuScenes	4	10.5 h
VoxelRCNN - center	Sydney	4	15.5 h
PointRCNN	KITTI	4	23 h

Table C.5: GPU usage for generating multi-source detections, rounded to the nearest 0.5 hours.

	GPUs	Duration
STD	4	11.5 h
STD++	4	10 h
Ours - TH1	4	11 h
Ours - TH2	4	12.5

Table C.6: GPU usage and duration for generating different sets of pseudo-labels

	Pretrained	Pseudo-Label Set	GPUs	Duration
CenterPoint	**X**	STD	2	16 h
CenterPoint	✓	STD	2	16 h
CenterPoint	**X**	STD++	2	17.5 h
CenterPoint	✓	STD++	2	17.5 h
CenterPoint	**X**	Ours - TH1	2	28.5 h
CenterPoint	✓	Ours - TH1	2	17 h
CenterPoint	**X**	Ours - TH2	2	16.5 h
CenterPoint	✓	Ours - TH2	2	16.5 h
PV-RCNN++	**X**	STD	2	21.5 h
PV-RCNN++	✓	STD	2	20 h
PV-RCNN++	**X**	STD++	2	22 h
PV-RCNN++	✓	STD++	2	28.5 h
PV-RCNN++	**X**	Ours - TH1	2	22 h
PV-RCNN++	✓	Ours - TH1	2	22.5 h
PV-RCNN++	**X**	Ours - TH2	2	23 h
PV-RCNN++	✓	Ours - TH2	2	31.5 h
VoxelRCNN	**X**	STD	2	32 h
VoxelRCNN	✓	STD	2	32 h
VoxelRCNN	**X**	STD++	2	35.5 h
VoxelRCNN	✓	STD++	2	35 h
VoxelRCNN	**X**	Ours - TH1	2	33 h
VoxelRCNN	✓	Ours - TH1	2	33.5 h
VoxelRCNN	**X**	Ours - TH2	2	34.5 h
VoxelRCNN	✓	Ours - TH2	2	36 h

Table C.7: GPU usage for training of three detectors

C.4. Pseudo-labels

Table C.8 provides more evaluation scores for the different sets of pseudo-labels, as an addition to Table 5.2.

Set	Class	F1-score	ATE 2D	ATE 3D	ASE	AOE
STND	Vehicle	0.8538	0.2739	0.8935	0.2562	0.0504
	Cyclist	0.7580	0.1743	0.8104	0.3345	0.2398
	Pedestrian	0.4145	0.0818	0.8692	0.4660	0.3289
STND+	Vehicle	0.8644	0.2737	0.8889	0.2572	0.0448
	Cyclist	0.7572	0.1726	0.8088	0.3356	0.2363
	Pedestrian	0.3955	0.0836	0.8628	0.4880	0.3235
Ours	Vehicle	0.8538	0.2739	0.8935	0.2562	0.0504
	Cyclist	0.7821	0.1726	0.8081	0.3400	0.2184
	Pedestrian	0.4148	0.0818	0.8692	0.4660	0.3289

Table C.8: Additional Evaluation Metrics for Different Pseudo-Label Sets and Classes. ATE = Average Translation Error, AOE = Average Orientation Error.

D

Definitions

D.1. Cyclist

We use the same definition of cyclist as the labeling rules for a cyclist in the Waymo dataset [55], including:

- Bicycles that are parked or riderless are not a cyclist.
- A pedestrian is a pedestrian until they are about to mount the bicycle; they are defined as a cyclist once they assume the riding position. Similarly, a cyclist is defined as cyclist until they start dismounting, at which point they are defined as a pedestrian.
- Scooters are included in our automotive dataset as cyclists due to their similarity in size, speed, and behavior to bicycles, making them relevant for understanding and predicting interactions in urban traffic environments. A scooter can be a moped, light moped, or motorcycle.
- Special cases also defined as cyclist:
  - children riding bicycles, tricycles, or wheeled toys
  - unicycles, tricycles, and recumbent bicycles
  - large, multi-seat bicycles

D.2. Evaluation Metrices

We evaluate the quality of the pseudo-labels and the predictions of the detectors trained on the pseudo-labels according to the following metrics:

1. **Precision:** Precision is the ratio of true positive (TP) predictions to the total number of positive predictions made by the model, defined as:

$$\text{Precision} = \frac{TP}{TP + FP}$$

   Recall (or sensitivity) is the ratio of true positive predictions to the total number of actual positives, defined as:

$$\text{Recall} = \frac{TP}{TP + FN}$$

   The **precision-recall curve** is a graphical representation that shows the trade-off between precision and recall for different threshold values, helping to evaluate the performance of a classification model by plotting recall (y-axis) against precision (x-axis).
2. **Average Precision (AP)** combines the accuracy of identifying objects in 3D space. It considers both precision (accuracy of positive predictions) and recall (completeness of predicted objects). When both are high, the AP is high as well. To be more precise, it is the area under the precision-recall (PR) curve. The range for AP is between 0 and 1.

3. **Mean Average Precision (mAP)** is the average of AP values calculated across multiple object classes or categories, considering their varying sizes and shapes. It provides insight into a model's overall performance in object detection or recognition tasks.
4. **2D Intersection over Union (IoU)** is a metric used to measure the accuracy of an object detector on a particular dataset. It is calculated as the area of overlap between the predicted bounding box and the ground truth bounding box divided by the area of their union:

$$\text{IoU} = \frac{\text{Area of Overlap}}{\text{Area of Union}}$$

   The IoU value ranges from 0 to 1, where 0 indicates no overlap and 1 indicates a perfect overlap.
5. **3D Intersection over Union (IoU)** extends the concept of 2D IoU to three dimensions, commonly used in 3D object detection tasks. It is calculated as the volume of overlap between the predicted 3D bounding box and the ground truth 3D bounding box divided by the volume of their union:

$$\text{3D IoU} = \frac{\text{Volume of Overlap}}{\text{Volume of Union}}$$

   Similar to 2D IoU, the 3D IoU ranges from 0 to 1, with higher values indicating better accuracy of the predicted 3D bounding boxes.
6. **F1 Score** is a metric that combines precision and recall to provide a single measure of a model's performance. It is the harmonic mean of precision and recall, and it is particularly useful when the class distribution is imbalanced:

$$\text{F1 Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

   The F1 Score ranges from 0 to 1, with 1 indicating perfect precision and recall, and it helps to balance the trade-off between these two metrics.
7. **Average Translation Error (ATE)** measures the average error in the estimated positions of objects. It is the Euclidean distance between the predicted and ground truth object centers. Lower ATE values indicate more accurate position estimations.
8. **Average Scale Error (ASE)** measures the average error in the estimated sizes (scale) of objects. It is calculated as the difference between the predicted and ground truth object dimensions. Lower ASE values indicate more accurate size estimations.
9. **Average Orientation Error (AOE)** measures the average error in the estimated orientations of objects. It is calculated as the difference in orientation angles between the predicted and ground truth objects. Lower AOE values indicate more accurate orientation estimations.
10. **Waymo Level 1 and 2** are evaluation levels defined by the Waymo Open Dataset to assess object detection models. Level 1 includes easy and moderate difficulty objects, while Level 2 includes hard difficulty objects. These levels help to evaluate model performance across varying levels of detection challenge.
11. The **nuScenes Detection Score (NDS)** is a metric used to evaluate the performance of object detection models on the nuScenes dataset. It combines several individual metrics, including mean Average Precision (mAP) and metrics assessing translation, scale, orientation, velocity, and attribute errors, to provide a holistic measure of a model's detection accuracy and robustness.