Fachhochschule Dortmund
University of Applied Sciences and Arts
Embedded Systems Engineering

Extending LiDAR Point Clouds with Radial Speed based on Radar Data.

Project Thesis

Author: A. K. Alhalabi, Ali
Matriculation Number: 7216390

Supervisor: Prof. Dr. Andreas Becker
Date: July/2024

Abstract

This document describes the Implementation of sensor fusion between radar (Radio Detection and Ranging) detections and LiDAR (Light Detection and Ranging) clustered data in a unified point cloud structures with the goal of extending the LiDAR point clouds, that lack velocity measurement, with radial speed information from radar detection. This is carried out by creating a C++ Library that provides template functionality of interfacing with three automotive-grade LiDARs and one automotive-grade radar provided by Fachhochschule Dortmund, perform data acquisition via UDP and CAN drivers suited specifically for the LiDARs and radar, create point cloud structures for each form of data, perform DBSCAN (Density-Based Spatial Clustering of Applications with Noise) and IoU (Intersection Over Union) clustering operations where needed, present the results in three dimensional rendering and in textual form, and provide abstract time-based synchronization of frames when GPS is available. The functionality of sensor fusion is done in this way to facilitate better abstraction levels whilst coding, provide better debugging practices and code isolation for easier adaptation of other sensors in the future. This framework targets macOS but is designed to be cross-compatible with other POSIX systems where possible. Processing time and memory usage consideration were of utmost importance while producing this work. It was noticed after testing that each set of points representing an object detected by the LiDAR intersects with only one radar detection due to the automatic clustering ability of the radar used. The result of this work is a set of objects each with the points from the LiDAR clustered point clouds and the radial speed from the radar detections.

All processing of data is explained in detail with pseudo-code when needed to avoid the complexity of C/C++ syntax, and with some C++ code snippets on a higher-level as an example of how to use the toolset to create merged objects of LiDAR point clouds with Radar radio-speed information.

Keywords *– LiDAR, Radar, Point Cloud, sensor fusion, Object Detection, Perception ,DBSCAN, IoU*.

Declaration

Ich versichere, dass ich meine Arbeit selbstständig angefertigt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

04.Juli.2024, ali alhalabi

I hereby confirm, that I have written the Project Thesis at hand independently, that I have not used any sources or materials other than those stated, and that I have highlighted any citations properly.

04.July.2024, ali alhalabi

Table of Contents

List of Abbreviations

Abbreviation	Definition
CAN	Controller Area Network
CW	Continuous Wave
CSV	Comma-Separated Values
DBSCAN	Density-Based Spatial Clustering of Applications with Noise
FMCW	Frequency Modulated Continuous Wave
GPS	Global Positioning System
IoU	Intersection Over Union
LiDAR/Lidar	Light Detection and Ranging
MIMO	Multiple Input Multiple Output
PC	Point Cloud
Radar	Radio Detection and Ranging
RCS	Radar Cross Section
UDP	User Datagram Protocol
ms	Milliseconds

List of Figures

List of Tables

1. Introduction

1.1 Background

Perception in the automotive industry represents a paramount importance in today's vehicles, this does not only apply to self-driving vehicles, but also for human-driven vehicles, for instance cruise control and blind spot detection are features of modern vehicles. Adding the option for vehicles to perceive the environment around it is usually done with multiple Radars (Radio Detection and Ranging), Ultrasonic sensors, LiDARs (Light Detection and Ranging), Cameras, alongside other sorts of sensors[1] all working to militate a safe driving experience. Each one of these sensors has a variety of advantages and disadvantages, where each is usually used in a way to mitigate the limitations and provide optimal results in object detection and tracking to facilitate safe and easy navigation of the vehicle. In Autonomous, or self-driving, vehicles, radars, LiDARs, cameras, ultrasonic sensors, and positioning sensors are required to realize the functionality of driving without human intervention [1].

The ability to better equip the full potential of these sensors and overcome some of their individual limitations is of a great deal of importance. It would be beneficial to employ a holistic overview and representation of all the provided data into one view port, representing all the detected objects, or points, from all the given sensors regardless of how they operate or report data to produce a holistic sensory view of the surroundings. One of the major Benefits of using a LiDAR, such as that of Velodyne or HESAI, is the ability to get multiple reflection points, each carrying range information for one object in a structured manner as shown in Figure 1, unlike a radar point cloud which wouldn't show the same number of points at least not in the same structure as a LiDAR would.

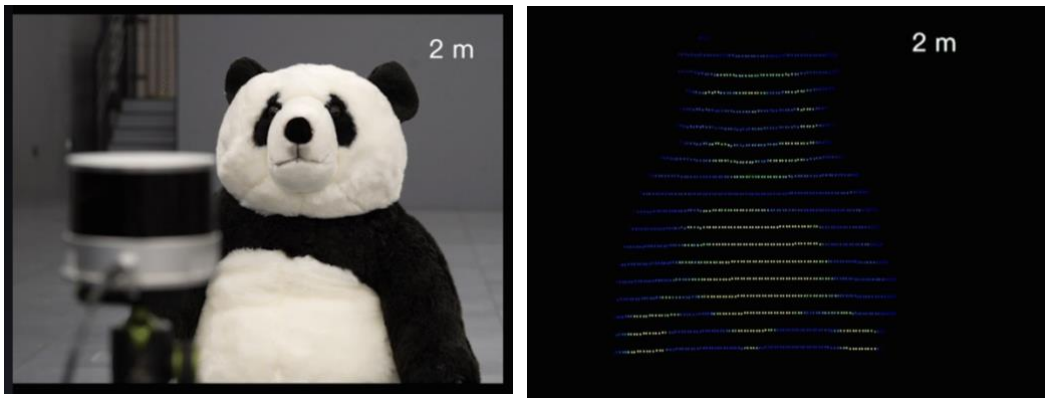

Figure 1: Hesai XT32 LiDAR points of one object (right).[2]

1.2 Problem Statement

A LiDAR provides a consistent point cloud of the scene with every point containing range information as shown in Figure 1 and later discussed in 2.2 Light Detection and Ranging, these points are usually represented in a spherical coordinate system such as the case for [3],[4],[5] where each point is represented by a Radial distance from the LiDAR center $\boldsymbol{R}$, the Azimuth angle $\boldsymbol{\varphi}$ representing the rotor's angle, and an elevation $\boldsymbol{\theta}$ angle related to the number of channels a LiDAR has. In addition to some timing-information. LiDAR sensors usually have no means of measuring speed of an object, only the range. On the other hand, Radars, which are discussed in 2.1 Radio Detection and Ranging characteristically provide, alongside Azimuth angle $\boldsymbol{\varphi}$, elevation $\boldsymbol{\theta}$, and range $\boldsymbol{R}$, radio-speed $\dot{\boldsymbol{R}}$ information of the object. Even though the point-cloud produced is not as straight forward as that of LiDAR sensors. Radars Usually provide similar information of points like the LiDAR but in addition to radial speed $\dot{R}$ information of the point.
The problem statement involves LiDAR specifically, as they provide structured point cloud, they are missing the fourth dimension of speed. This cannot be solved by a 4D Radar as radars typically do not provide a structured point cloud.

1.3 Objectives

The objective of this research is to provide a framework with all the drivers and tools necessary to merge multiple LiDAR sensor data into one point cloud for further study and analysis. This analysis includes clustering of the point cloud, measuring the speed of clustering, and adding Radial speed $\dot{R}$ information to detected clusters based on Radar information. Additional objectives include:

- This framework targets macOS, as modern macOS-based notebooks and computers provide substantial processing power with very low energy consumption based on the use of Apple Silicon processors. However, based on the similarities between POSIX systems, i.e. macOS and Linux-based operating systems, the framework is carefully written in a way that is cross compatible across other POSIX systems (not including Windows) with only minor changes.

- Processing time of this framework revolves around finding the fastest way to process the point clouds within the scope of new data arrival from the LiDARs and Radar sensors.

- Memory managed code with coding practices that limit memory leaks and creation of new memory locations during run-time. It is imperative to limit and manage memory usage of code written in C/C++ to avoid memory leaks which can be unsafe during run-time and heavily impact the overall performance of the experiment.

1.4 Scope and limitations

The current implementation is limited to in-door usage of three multi-channel LiDARs,

- Hesai's Pandar XT32, a 32-channel medium range 360 degrees LiDAR [3],
- Hesai's PandarQT, a 64-channel short-range 360 degrees LiDAR [4],
- Velodyne's VLP-16, a 16 channel 360 degrees LiDAR [5] ,
- and Smart Microwave's UMR11-Type 132 Automotive-grade Radar [6] ,

where Drivers are to be developed to interface with said sensors according to the datasheets and available resources at the time and processing data from that driver as part of the development process. All drivers are written in C/C++ targeting macOS as the platform of development. Without relying on external libraries unless necessary, for instance, access to Ethernet controller (used to interface with the LiDARs mentioned above) is implemented as part of the scope of driver development as this is a function of the operating system (POSIX) and is not hardware dependent. However, developing a driver for a specific USB-to-CAN controller (used by the Radar mentioned above) falls out of the scope of this implementation as multiple different versions of USB-to-CAN hardware exist and cannot be covered by one implementation. This is done to reduce the level of compatibility required to run this code to the bare minimum, unlike the use of Robotics Operating System (ROS) for instance, where despite providing flexible and abstract range of tools, it introduces performance overhead due to said abstraction with additional CPU and Memory usage, lacks real-time characteristics due to the use of message passing architecture with serialization and deserialization of messages, and suffers from the issue of compatibility of packages and dependencies across different operating systems or different versions of the same operating system.

Visualization of results relies on Open Computer Vision Library (OpenCV) as it is an open-source library that is reliably cross-platform that supports multiple operating systems including Linux, macOS, and Windows using multiple programming languages [7] and source codes using OpenCV do not rely on specific version of operating system nor does OpenCV require changes to source codes for different operating systems.

Datatypes used throughout the implementation follow C++17 standard library and all operations on data types conform to C++17 standards and use the standard library definitions and build in functions.

Every driver for each LiDAR and Radar described above will have the ability to interface with the hardware directly (over Ethernet for LiDARs and USB-to-CAN for the Radar) and be able to interface with a logged file, a PCAP file for ethernet data and a coma separated value (.CSV) file for USB-to-CAN data. External log-file processing will depend on external opensource libraries such as PcapPlusPlus [8].

Memory Management and overall instrumentation of executables fall under the scope of development of the drivers only and will not cover any external libraries used for non-essential utility, i.e., OpenCV, and PcapPlusPlus.

2. Literature Review

2.1 Radio Detection and Ranging

The beginning of radio detection and ranging, commonly known as Radar could be traced back to 1886, when the German physicist Heinrich Hertz proved that solid objects can reflect radio waves, followed by the German engineer Christian Hülsmeyer who used radio waves to detect ships in dense fog conditions [9]. And even though this early start was limited to military applications, Radar is now used in a variety of civilian applications including medical applications, navigation, Automated driving, and so on [10].

Casting the light on the Automotive industry applications, Radars are used for both Self-driving cars and human-driven cars, they are used in a variety of applications including park assist, adaptive cruise control, lane change assist, collision detection and avoidance, all of which are integrated into the vehicle's system and are part of the driving experience. Automotive Radars, much like all sorts of other Radars, work by transmitting an electromagnetic wave, usually of 24Ghz or 77Ghz Frequency [10], and waiting for that signal to be reflected off of some object, usual objects of interests are other vehicles on the road, pedestrians, cyclists, etc., to process information about said object. Typical information from a radar contains Range, Radial speed, Radar Cross Section (RCS), Azimuth angle, and in some cases Elevation Angle. The kind of information provided for a detected object depends on the modulation scheme used in the radar hardware.

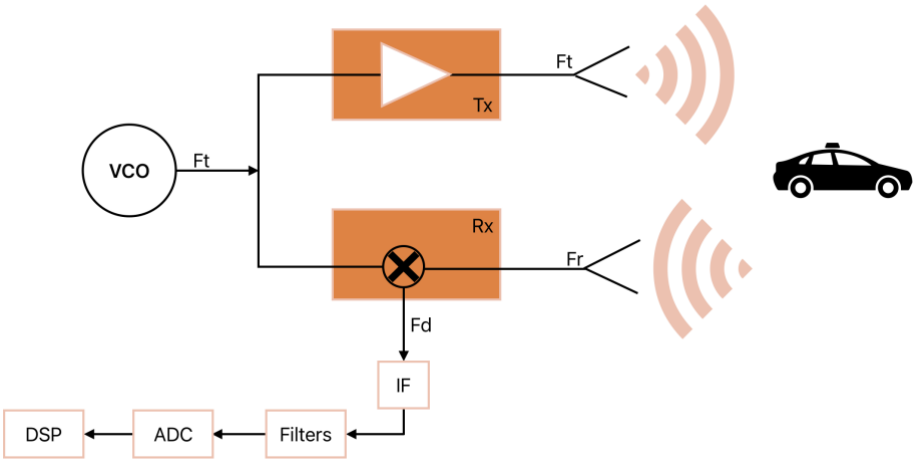

Figure 2: A conceptual block diagram of a CW Radar in a typical scenario. Ft: Frequency Transmitted, Fr: Frequency Received, Tx: Transmitter, Rx: Receiver, vco: Voltage Controlled oscillator, Fd: Doppler Frequency, IF: Intermediate Frequency, ADC: Analog to digital Converter, DSP: Digital Signal processor. adapted from [9].

2.1.1 Continuous Wave Radars

Figure 2 shows a block diagram of a radar with a continuous wave (CW) modulation scheme, where a monotone frequency (i.e., $A \cdot \cos(2\pi f_t t)$) is transmitted, then it reflects off an object and is picked up by the receive antenna and mixed with the original transmitted signal to get an intermediary frequency at the doppler frequency f_d, which is then processed by the digital signal processor to get the speed, commonly known as doppler speed or Radial speed $\dot{R}$. Which is calculated based on $\dot{R} = \frac{C}{2} \cdot \frac{f_d}{f_t}$, where c is the speed of light [9]. Continuous wave radars lack the ability to detect the distance of the object.

2.1.2 Frequency Modulated Continuous Wave Radars

Other common modulation schemes involve frequency modulated continuous wave (FMCW) where a frequency modulated signal is transmitted. This signal increases from the carrier frequency f_c to $f_c + \Delta f$,where Δf is the bandwidth, over time T_{sw}, known as sweep time or chirp time. This is done multiple times in one frame over Frame time T_{frame} as shown in Figure 3.

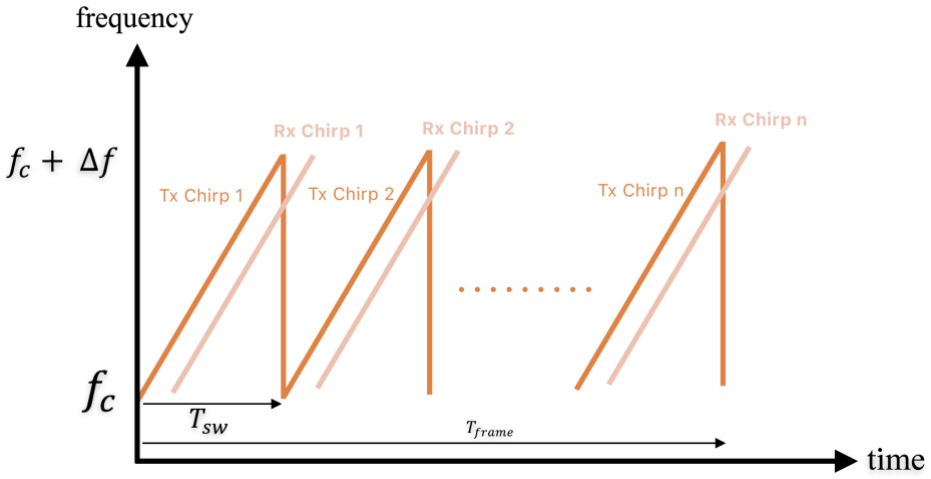

Figure 3: FMCW Radar Modulation Scheme, Adapted from[9]

In an FMCW Radar, the range is calculated via performing an FFT over the time range to get the beat frequency f_b, substituted then in $R = \frac{C \cdot T_{sw}}{2 \cdot \Delta f} \cdot f_b$ then, after calculating the FFT for all the return chirps over T_{frame}, another FFT is calculated over the number of chirps to get the doppler frequencies, which correlate to the radial speed $\dot{R} = \frac{C}{2 \cdot f_c} \cdot f_d$.

Multiple other modulation schemes exist, such as fast chirps, which is similar to FMCW. Pulsed CW, which adds range ability to CW radars, and so on. But the focus of this work would focus particularly on FMCW radars.

2.1.3 Multiple Input Multiple Output Radars (MIMO)

MIMO Radars make use of multiple transmit antennas and multiple receive antennas in order to determine the angle of arrival of each signal (or object detected) [9],[10], this allows the radar to discriminate targets that are moving with the same radial speed $\dot{R}$ or have the same distance from the radar [9].

Multiple Radars are usually installed in different spots with different capabilities, i.e. short range, long range, etc., to detect other vehicles in the surrounding. Such information is then transmitted to the driver and the control systems of the vehicle which allows automated snap decisions, for example collision avoidance. Radars are effective against snow, fog, or rain in addition to their low price point, and with their low price point, they prove effective and common in all sorts of vehicles [1].

2.1.4 Radar used in this work.

The Radar used in this work is Smartmicro's umr-11 type 132, which is a 77GHz radar designed for automotive applications. It features a medium- and a long-range mode with the latter having a narrower field of view. It features measuring range, radial-speed, azimuth, and elevation angles. It also provides separation across range, doppler, and azimuth. The radar also provides filtering and multi-target tracking abilities out of the box [6] with up to 64 objects tracked simultaneously. Table 1 shows the most important features of the UMR-11 Radar alongside a comparison between the Long- and Medium-Range modes.

Table 1: UMR11 Feature set. Adapted from [6]

Parameter		Long-Range mode	Medium-Range Mode
Operating Frequency		76GHz … 77GHz	
Range	Min./Max.	1m … 175m	0.5m …64m
	Separation	≤ 1.8 meter	< 0.66m
	Accuracy	< 0.5m	< 0.25m
Speed	Min./Max.	-400 … +200 km/h	-340…+170 km/h
	Separation	< 0.26 m/s	< 0.26 m/s
	Accuracy	≤ 0.1 m/s	
Azimuth	Field Of View	-16…+16°	-50…+50°
	Separation	4°	
	Accuracy	≤ 0.25°	≤ 0.5°
Elevation	Field Of View	-7.5…+7.5°	
	Separation	None.	
	Accuracy	≤ 0.5°	

Even though UMR-11 Type 132 supports both Ethernet connection and CAN-Bus Connection, the currently available version for this work is set up to use CAN-Bus communication.

2.2 Light Detection and Ranging

Light Detection and Ranging, commonly known as LiDAR, refers to a technology that has appeared as early as 1971, It is thought that the first use of a LiDAR was to map the lunar surface during the Apollo 15 mission [11]. According to [12], LiDAR became the preferred tool for taking measurements in many space-related missions conducted by NASA.
In 2005, LiDAR was used, for the first time, in the automotive scene during the second DARPA challenge [11], [13]. According to [13], during the 3nd DARPA challenge, five out of the six vehicles that finished the race used a Velodyne spinning LiDAR with a 360° field of view and a 100 meters range.
LiDARs can be split into multiple categories based on if it has moving unites, i.e. Scanning LiDAR and non-scanning LiDAR, or based on measuring technique, i.e. Time of Flight (ToF), Phase Ranging, or Triangular LiDARs [11]. According to [11], ToF LiDARs are the most common type of LiDAR in the automotive industry.
Time of Flight (ToF) represents a technology where the travel time of a signal is measured in order to calculate the distance. The signal is emitted, usually from a light emitting diode or from an antenna, reflected by an object, and travels back to a receiver and based on the travel-time of the signal, the distance can be calculated.

2.2.1 LiDARs Used in this Work.

This work supports three different LiDARs from two different companies, Pandar XT32 and PandarQT from Hesai, and VLP-16 from Velodyne. With the ability to use any individually or combined. Table 2 and Table 3 show some relevant information related to the development in this work.

Table 2: Hesai's Pandar XT32 and PandarQT feature set, adapted from[3], [4]

Parameter		Pandar XT32	PandarQT
Scanning method		Mechanical Rotation	
Vertical Channels		32	64
Range	Min/max	0.05 … 120 m	0.1 … 60 m
	Accuracy	± 1 cm	± 3 cm
Horizontal	FoV	360°	360°
	Resolution	0.09°, 0.18°, 0.36°	0.6°
Frame Rate		5Hz, 10Hz, 20Hz	10Hz
Vertical	FoV	31° (-16° to +15°)	104.2° (-52.1° to +52.1°)
	Resolution	1°	Inconsistent, finest at 1.45°
Data Transmission		UDP/IP	
Point-Cloud Density		640,000 points/sec[1]	384,000 points/sec[1]

[1] Based on the current setup, other densities available based on return type, Horizontal Resolution, etc.

Table 3: Velodyne VLD16 feature set, adapted from [5],[14]

Parameter		VLD16
Scanning method		Mechanical Rotation
Vertical Channels		16
Range	Max	Up to 100 m
	Accuracy	± 3 cm
Horizontal	FoV	360°
	Resolution	0.1° – 0.4°
Frame Rate		
Vertical	FoV	30° (-15° to +15°)
	Resolution	2°
Data Transmission		UDP/IP
Point-Cloud Density		300,000 points/sec[1]

2.3 Previous Integration of LiDAR and Radar

Various research is done on the topic of sensor fusion and tracking based on multiple LiDARs, Radars, or combination of both, effort is still done with respect to classical processing and with trend going towards AI and machine learning.

One paper [15] experimented with a tracker implementation that fuses data across 4 single-channel (2D) LiDARs and one Radar. Where data collected from all of the sensors were set-up in a compatible fashion, converted into Cartesian coordinates, and clustered based on "Distance based connectivity, and "Ratio". The multiple 2D LiDARs work together to produce a Point Cloud. Radar Data is then appended to the point cloud. Tracking is performed afterwards where Kalman Filter was used for prediction and Joint Probabilistic Data Association Filter was used for updating. Figure 4 shows red boxes in (a) representing the LiDAR data and violet boxes representing radar data. The results of the tracking in (b) where object size, velocity and direction are assigned with an object ID. Robot Operating System (ROS) was used for this implementation under Linux operating system, it is mentioned that different ROS tools were used for visualization and performance checking. [15] claims that the process took less than 10 milliseconds (ms) under normal traffic conditions.

In [16] a framework called "InterFusion" is presented in which 16-channel LiDAR data is fused with a 4D Radar. The describe their "Self-attention mechanism" that selects key features from both the LiDAR and the Radar point clouds to produce and enhance a "future expression" in which cross-model features are aggregated. In [16], a Detection algorithm is built based on PointPillars, then a 2D Convolutional neural network (CNN) and a RPN head were used to limit the detection space. Then they compared the results with the ground truth by using a 3D intersection over Union (IoU). Figure 5 shows the results of the development descried in [16]. It is claimed in [16] that their method outperformed other single-modularity methods. Further metrics and comparisons are described in [16].

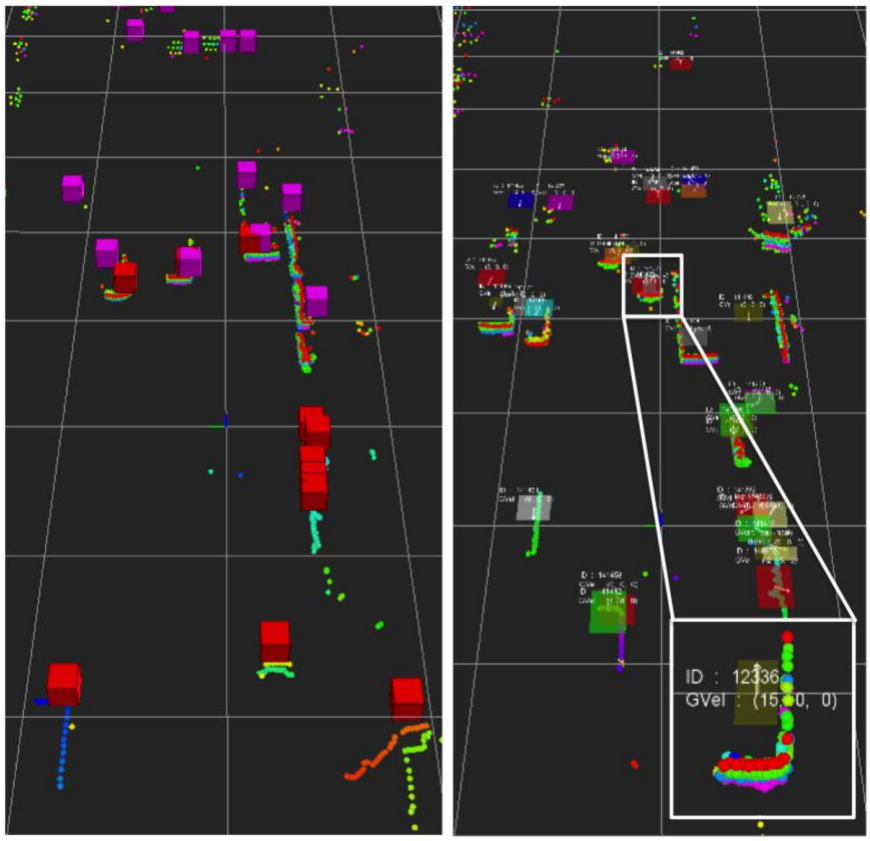

Figure 4: Tracking Results of LiDAR and Radar fusion.[15]

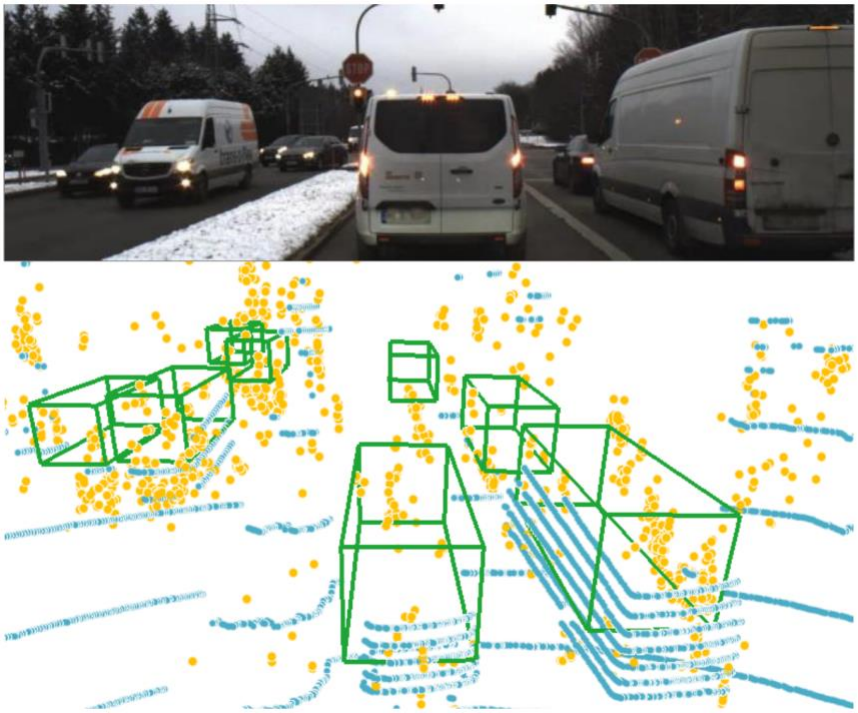

Figure 5: Upper image: camera front view; lower image: 3D detection result in 4D Radar (yellow points) and 16-line LiDAR (blue points) point clouds.[16]

3. Methodology

3.1 Implementation Design

In order to combine any sort of data, it is necessary to ensure that the data sources are compatible with each other within the framework. The hardware used within this implementation should be modeled in a way that guarantees data compatibility. It is helpful to view the software as an assembly of models that interact with each other. Using Unified Modeling Language (UML) [17], the process of designing the system becomes more structured and defining communication between different drivers, and by extension hardware, becomes more consistent throughout the development process as UML provides system designers and engineers a method to model, analyze and test systems at early stages of development [17].

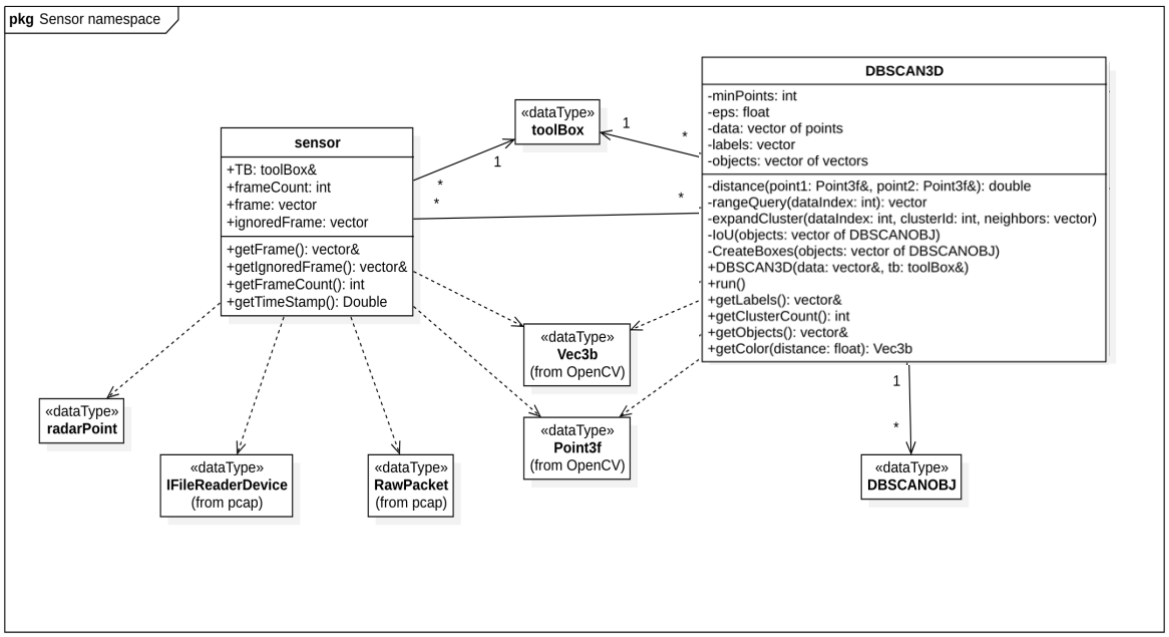

Figure 6: UML Class Diagram of the system.

Figure 6 shows the UML Class diagram describing the basic components of the system, ***sensor*** class represents the software driver for any of the LiDARs or Radar used in this work. ***sensor*** can be thought of as a parent class performing the basic functionality needed to grab point-cloud data from either the LiDARs or the Radar. ***sensor*** consists of multiple basic attributes and operations, Attributes such as Frame counter, which is an integer counting how many frames have been processed so far by the driver, vector of three-dimensional points representing the LiDAR frame (***Point3f***), or four-dimensional points representing the radar point clouds (***radarPoint***), and a reference to a toolbox that includes some user-controlled variables that affect the whole system. ***sensor*** includes some basic attributes such as grabbing a frame from the any of the sensors mentioned above, getting frame count and time stamp. ***sensor*** class relies on some important datatypes, including a 3D-point type from OpenCV library such as Point3f, which is a structure that stores X,Y, and Z values of a certain points as float values, Vec3b, which is usually used to store red, green, and blue values of colors, and network capture file type and reader type from PcapPlusPlus library

for parsing log files. All of which necessitate the functionality of the drivers implementing the ***sensor*** class. Any class implementing the ***sensor*** class represents an abstract hardware driver for a sensor which includes some of the attributes described in ***sensor*** in addition to sensor-specific attributes for any driver implementing ***sensor***. Each driver implementing ***sensor*** can interface with only one ***toolbox*** which sets some restrictions and important details for the driver when obtaining data, such as bounding box dimensions, as in setting restrictions of point relevance to a specific box of operation, clustering properties, and some visualization options. However, one ***toolbox*** can interface with multiple ***sensor*** instances or ***DBSCAN3D*** instances. ***DBSCAN3D*** is a class that performs DBSCAN clustering operation on the point cloud obtained by any instance of ***sensor.*** DBSCAN, or Density Based Spatial Clustering of Applications with Noise [18] is a clustering algorithm designed to discover clusters based on spatial relationships between points based on density, which is defined by minimum number of points in a cluster and the distance between points (Epsilon) [18]. Minimum number of points and Epsilon values, in the design of this implementation, are both controlled by ***toolbox*** instance. ***DBSCAN3D*** follows the general implementation described in [18] with some compromises to speed-up the operation. The current ***DBSCAN3D*** implementation takes in a reference to the vector of 3D points that represents the point cloud and allows the developer to get a vector of labels that matches the size of the vector of 3D points, where each label is a number that corresponds to a cluster. Or alternatively get a vector of ***DBSCANOBJ***, or a DBSCAN Object, in which a vector of points corresponding to all points contained in a cluster and the center of that cluster is stored. It is critical during design to decide which variables are created only once and passed by reference and which variables are created on demand. And vector used with any ***sensor*** instance or ***DBSCAN3D*** instance is passed by reference, this means no copies of any vector is created. This is shown in UML class diagram by appending the ”&” symbol after the datatype name, which is borrowed from the C++ language and means the same, that the variable is passed by reference not by creating a copy. For example, a ***sensor***'s ***getFrame()*** method returns a reference to a vector stored inside the implementation of ***sensor*** and is available for the entire lifetime of a ***sensor*** object. This is due to the large size of the point cloud, where copying each vector representing a point cloud would consume more time and is resource-intensive when it comes to memory consumption. Every sensor included in 1.4 Scope and limitations is going to be implemented as a derived class that publicly inherits from ***sensor*** parent class. This means every derived class implements the basic functionality of ***sensor*** in addition to some sensor-specific functionality and the basic datatypes for each sensor in 1.4 Scope and limitations.

Figure 7 shows all the possible classes that instantiate from ***sensor***, this includes two classes for Velodyne VLD16 LiDAR, one communicates directly with the VLD16 LiDAR over ethernet, and the other parses a network capture file. Similarly, for Hesai PandarXT32 LiDAR, and Hesai PandarQT LiDAR. This is distinguishable with the distinction in class names that start with ***ip*** to indicate this class uses Ethernet for real-time interface with the LiDAR and requires a LiDAR connected on the same network, or ***pcap*** to indicate that this class parses a network capture from previous sessions. SmartMicro's UMR11 Radar follows a similar naming scheme, where ***CAN*** appended in the beginning of class name to indicate that this class interfaces with a UMR11 Radar over CAN interface or ***Offline*** to indicate that

this class reads a text log file for its operation. SmartMicro's UMR11 driver will not be discussed in depth in this document as it was developed for a different class [19] and is simply used as part of the development process for the work described in this document.

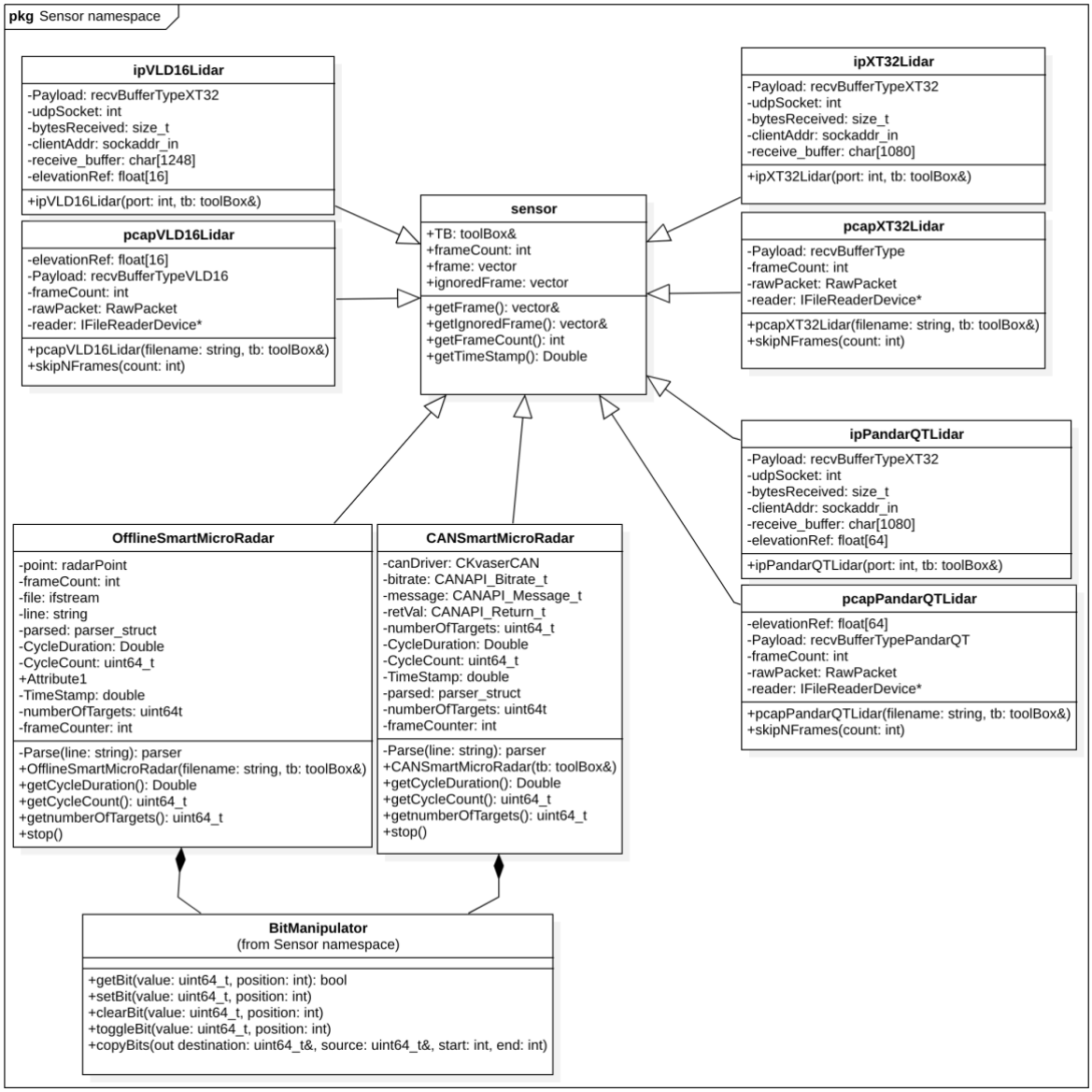

Figure 7: UML Class diagram of drivers implementing the sensor parent class.

For Velodyne VLP16 LiDAR's Ethernet driver implementation called ***ipVLD16Lidar***, additional attributes are added including an ethernet receive buffer and a data construct named recvBufferTypeVLP16 that matches the size of data sent by the VLP16 and as described in [5]. Additionally, [5] describes the vertical reference angles which is then mapped as an attribute in the classes. Other than the driver's constructor, the VLP16 LiDAR's Ethernet driver implementation does not implement other function, it still does implement all the functions described in ***sensor*** parent class. For the network capture file implementation, called ***pcapVLD16Lidar***, similar data structures are used as ***ipVLD16Lidar*** without network-specific datatypes, and with some important data structures that are relevant to parsing the pcap network capture file, and additionally implements a function to skip a

number of frames at a time for debug purposes. Which wouldn't be suitable for a real-time implementation. Similarly for PandarQT LiDAR, ***ipPandarQTLidar*** describes a driver that interfaces with Hesai Pandar QT 64-channel LiDAR over ethernet, alongside sockets and network datatypes, it includes a datatype ***recvBufferType*** specifically written to match the data transmitted by the LiDAR as described in [4], it also has an angle reference for the 64 vertical channels. The same applies for all the other LiDAR-based classes including ***pcapPandarQTLidar***, ***ipXT32Lidar***, and ***pcapXT32Lidar***. It is, however, a little different with the UMR11 Radar, described by the classes ***OfflineSmartMicroRadar*** and ***CANSmartMicroRadar***, the latter being responsible for communicating with an CAN-connected UMR11 radar and the first for reading a log file in the form of comma-separated values (csv). Both UMR11 drivers provide a set of targets instead of a generic point cloud, which leads to requiring more specific functionality such as getting number of targets and the vector of targets. ***BitManipulator*** Class is used with the UMR11 Radar to decode messages from the CAN-interface as will be shown in sub-chapter 3.3 Data Processing.

3.2 Data Collection

Data used in this work is directly collected from the hardware described above by using two different software. For recording LiDAR Data, Wireshark network protocol analyzer [20] software was used, while running, it captures network packets sent on a specific interface, i.e. Wi-Fi, Ethernet, etc., and allows one to filter these packets based on Internet Protocol (IP), Port Number, packet type, etc., and store the filtered data. Using Wireshark, data packets sent by the LiDAR can be stored as pcap files into a packet capture file type (pcap) which can then be read by a network capture file parser such as PcapPlusPlus [8] or similar software. Using the stored network capture files allowed reproducing the same environment of operation for gathering LiDAR data epically with capture file's ability to store all the transmitted details including timing characteristics. However, for the CAN data from the radar described above, a small program was written to store the CAN ID and the associated 8-byte message into a comma-separated value file where it can then be read by any software. It is not necessary to use network capture (PCAP) or comma-separated value (CSV) files to be able to run the code as it can interface directly with the hardware over ethernet and CAN. The data was collected indoors, in a room of approximately 20 meters by 9 meters with objects all around. Data collection across sensors is done by starting the different logging applications at the same time with the use of a script. The sensors are started well before-hand to allow warp-up sequences to take place accordingly. This is similar to taking a snapshot of the scene via two different sensors. Additionally, time stamps exist as part of each data packet provided by each sensor which are then used to better synchronize the frames due to the frame-rate differences of each sensor.

3.3 Data Processing

The first step in processing any point cloud, is to get the points that represent the point cloud. In the of any LiDAR or Radar implemented in this work, this is done by calling the method ***sensor::getFrame()***, this method performs all the low-level processes of grabbing the data from the source, i.e. LiDARs, Radar, network capture file, or coma-separated value file. The operation inside this method depends heavily on the kind of sensor is used and how the data is provided, additionally the time spent inside this method depends on the data-source and amount of processing needed, however for the LiDARs the process is quite similar across the models used and if it is a UDP LiDAR or a network capture file of a LiDAR. In order to use ***getFrame()*** on a LiDAR, one must first create an instance of a LiDAR, and then the function ***getFrame()*** can be called. Figure 8 shows an example of how one can initialize a LiDAR, first an instance of toolbox must be created as shown in line 2, and then one can create an instance of a LiDAR that works with a network capture file, as shown in line 3, where a path the LiDAR's pcap file is used as a string, in addition to passing the toolbox instance to the constructor. Or one can directly interface with a networked LiDAR as shown in line 5, where port number of the LiDAR UDP socket and toolbox instance are passed to the constructor.

```
1  int main() {
2      Sensor::toolBox tb;


3      Sensor::pcapXT32Lidar lidar1("path-to-pcap-file", tb);
4      std::vector<cv::Point3f> data1 = lidar1.getFrame();


5      Sensor::ipXT32Lidar   lidar2(PortNumber, tb);
```

Figure 8: Initializing a LiDAR and getting a point-cloud.

After creating an instance of a LiDAR, calling the function ***getFrame()*** will return a point cloud of the current frame, this will be returned as a vector of points each represented in the cartesian system. To better understand what happens inside the ***getFrame()*** function, Figure 9 shows a sequence diagram of the operations for a UDP LiDAR, when user calls the function (step 1), a UDP listener is created (step 2) to wait for the LiDAR data (step 3), when data is available from the LiDAR (step 4), the UDP listener reads the data into a UDP buffer (step 5), which is just an array of bytes of certain size that matches the LiDAR data. The waiting of UDP data is a blocking-operation, as in the code halts execution and only continues when the data is available. The UDP buffer is then converted into a LiDAR Payload construct by memory copy operations (step 6). The Payload construct is coded to match the description provided in the datasheet of each LiDAR, such as described in [3]–[5] and shown in APPENDIX A: Payload structure for Lidars In short, each single payload construct contains the distance values of 8 consecutive azimuth angles each associated with 32 elevation angles for Hesai XT32 LiDAR, consecutive azimuth angles each associated with 64 elevation angles for Hesai PandarQT LiDAR, and the distance values of 24

consecutive azimuth angles each associated with 16 elevation angles for Velodyne VLP16 LiDAR. The payload is processed inside a loop going over the amount of azimuth angles available in each payload (step 7). Loop-time for each LiDAR depends on the data structure shown in APPENDIX A: Payload structure for Lidars, which can be defined as a constant number of 500 Payloads for Hesai XT32 LiDAR, 150 Payloads for Hesai PandarQT, and 75 Payloads for Velodyne VLP16. UDP packets are handled on the system level, any missing UDP packets would result in missing points from the point cloud but would still provide a functioning point cloud.

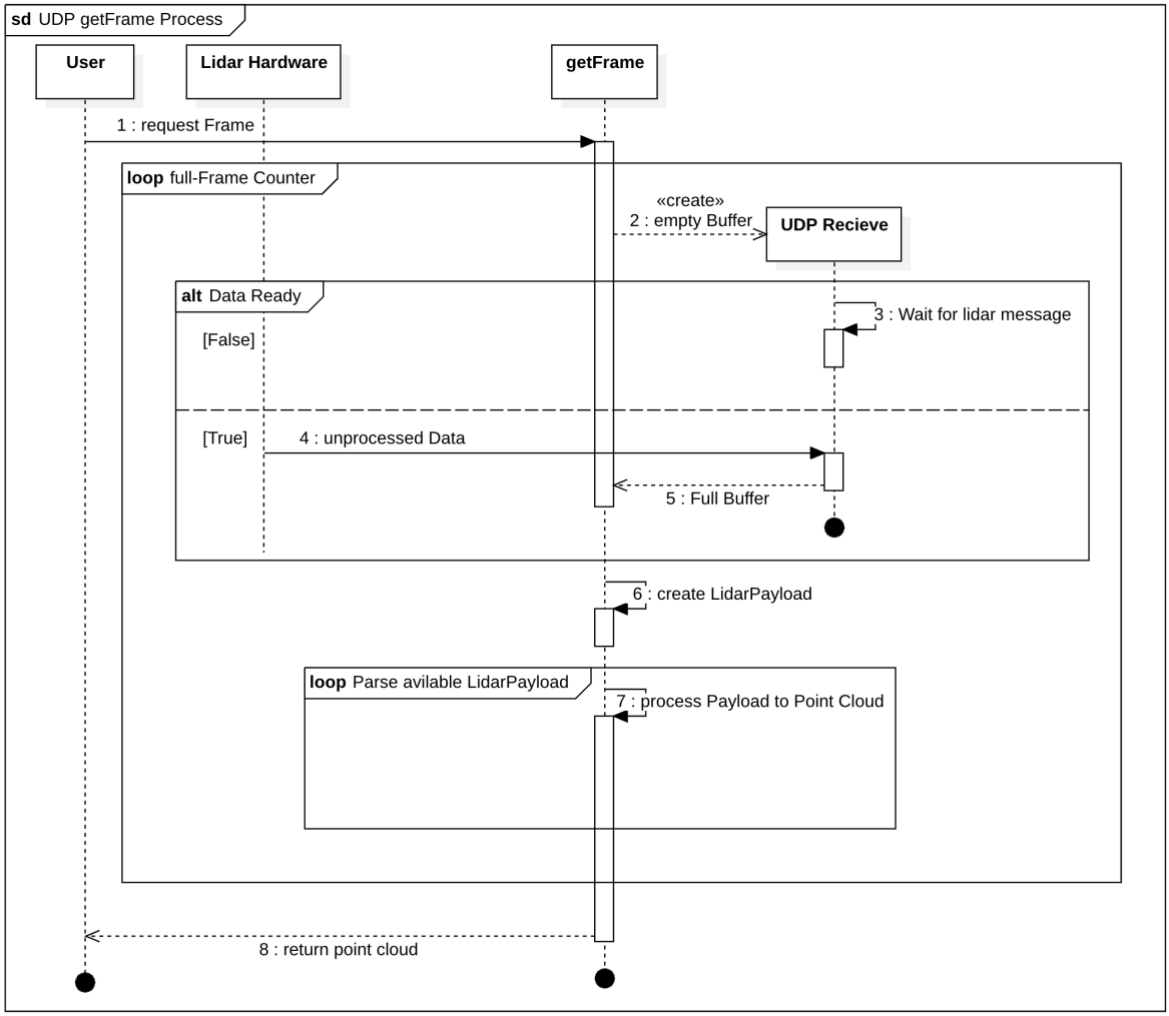

Figure 9: getFrame() operation for UDP LiDARs

During step 7, each distance value along with its associated azimuth and elevation angle gets converted from spherical coordination system into cartesian coordination system by a set of conversion formulas shown in Equation 1,

$$x = d \cdot cos(\varphi) \cdot cos(\theta)$$
$$y = d \cdot \sin(\varphi) \cdot \cos(\theta)$$
$$z = d \cdot \sin(\theta)$$

Equation 1: Spherical to Cartesian conversion

Where $\{x, y, z\}$ represents a point in cartesian 3-dimensional space based on radial distance d, azimuth angle φ, and elevation angle θ. $\{x, y, z\}$ is represented by a ***cv::Point3f*** structure in code that gets pushed into a vector during step 7 and after the full-frame counter loop is finished, a reference to the vector is returned to the user.
Similarly for a network capture LiDAR, most of the operations stay the same except for the UDP receiving process and the existence of a LiDAR, instead, the LiDAR hardware is replaced by a pcap file, and the UDP Receiver is replaced by a pcap reader from the PcapPlusPlus library. This eliminates any waiting and blocking activities in the implementation of ***getFrame()*** function. This is shown in Figure 10 where step 3 is replaced from waiting for UDP data to requesting data from the pcap file, the pcap file will produce what would have been the same data as a UDP read function. And then is processed the same way afterwards.

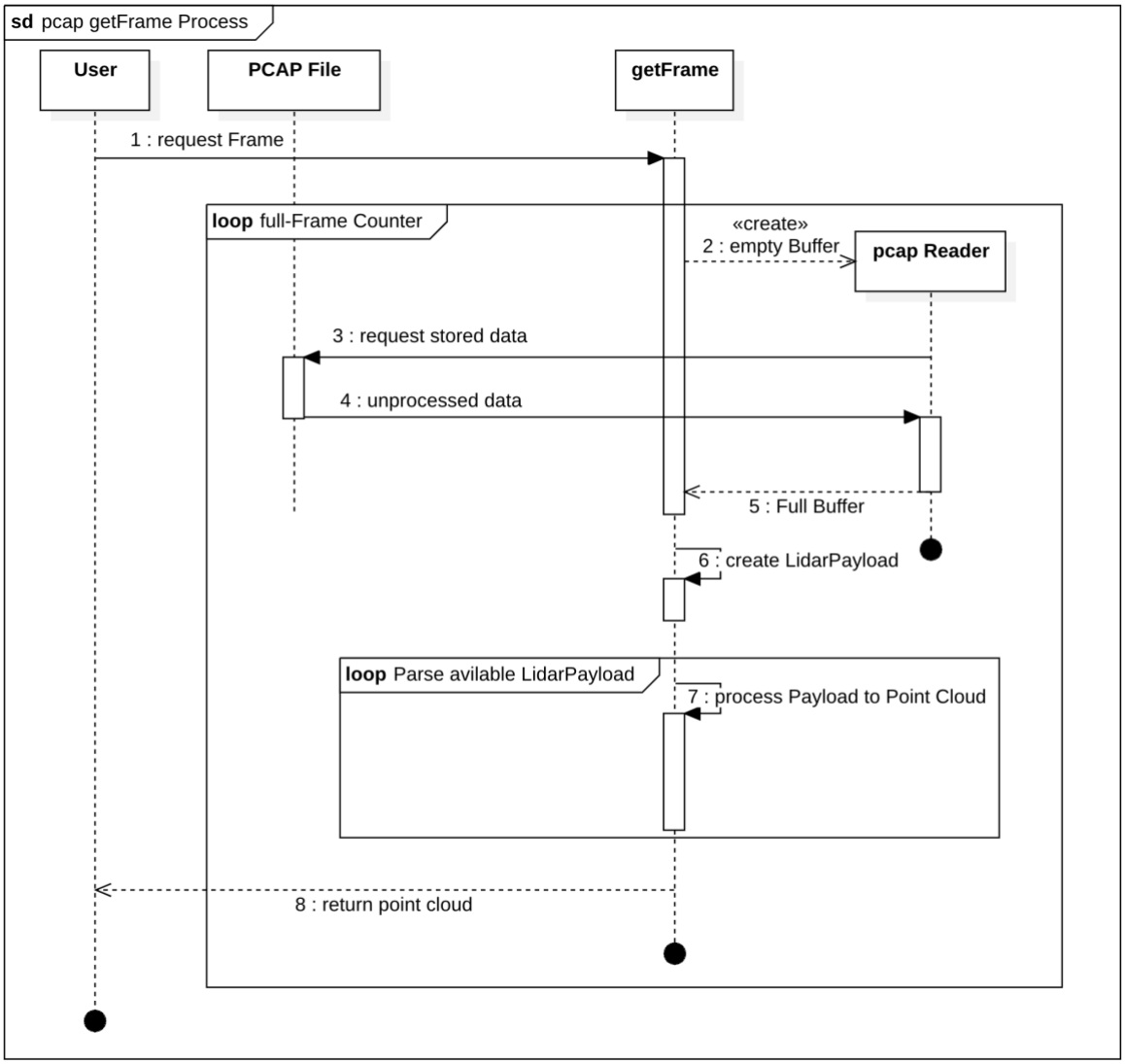

Figure 10: getFrame() operation for PCAP LiDARs

As for the UMR11 Radar data, the ***getFrame()*** function processes the CAN data or the CVS file differently, due to the major differences in protocol of communication and nature of the

sensor. To better understand the functionality of the ***getFrame()*** implementation for UMR11 radar, it is important to understand the messages sent over CAN bus that carry the information from the radar.

Table 4: UMR11 CAN messages description.

Byte 8	Byte 6	Byte 5	Byte 4	Byte 3	Byte 2	Byte 1	Byte 0	ID	Name
7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0		
0 0 AcquisitionSetup	# Targets	Cycle Count				Cycle Duration		0x400	Header 0
0 1				Time Stamp					Header 1
1 0				Time Stamp Fraction					Header 2
	Target Radial Speed			Azimuth			Target Range 0	0x401 ... 0x47F	Frame 0
		Elevation			Noise Level	Signal Level	RCS 1		Frame 1

Table 4 shows the most important messages sent by the UMR11 Radar over CAN bus, each message is 8 bytes long with overlapping CAN Identification number (ID) and contains fragmented information about the targets and detections of the Radar. There are five important CAN messages that can be split into two groups, a group that describes the current detection cycle of the radar and another group that describe the objects detected.
Header 0, ***Header 1***, and ***Header 2*** Messages fall into the first group of describing the current cycle of detection, where all these three messages share the same CAN ID of 0x400 but are differentiated by the last 2 bits of the last byte of each message, where

- the binary value of "0 0" represents ***Header 0*** message which carries the cycle duration of the current detection cycle in seconds as a fraction number, the cycle count, number of targets detected in this scan cycle,
- the binary value of "0 1" represents ***Header 1*** which carries the whole part of the time stamp value in seconds, and
- the binary value of "1 0" represents ***Header 2*** which carries the fraction part of the time stamp value in seconds.

Values belonging to the first group are only sent once every detection cycle and then are followed by the messages belonging to the second group, ***Frame 0*** and ***Frame 1*** which carry information about the detected targets in the detection cycle. Each message ***Frame 0*** and ***Frame 1*** are sent once for each object detected based on number of targets described in ***Header 0***. The first object's ***Frame 0*** and ***Frame 1*** will have the CAN ID 0x401 and increases incrementally up to the number of targets. ***Frame 0*** and ***Frame 1*** can be distinguished apart for each object by the value of the first bit of the first byte, where

- the binary value of "0" represents ***Frame 0*** message of a target, which carries the radial distance as a fractional number in meters, the Azimuth angle of the target in degrees, and the radial speed of the target in meters per second, and
- the binary value of "1" represents ***Frame 1*** message of a target, which carries the Radio cross-section (RCS) or a target, the Signal level in decibels, the noise level in decibels, and the elevation angle of the target in degrees.

The task of ***getFrame()*** function is to read the can messages in order and determine which message was sent, interpret it, and assign the data to the correct object. The process described above is shown through the pseudocode shown in Figure 11. It is important to note that the CAN bus implementation also introduces execution-blocking due to waiting for data to be transmitted where it is usually not present when reading from a CSV log file.

```
1   Function getFrame() -> Vector of radarPoint:

2     While True:
3       If CANDriver.ReadMessage(message) returns NoError:
4         parsed = Parse(message)

5         If MessgaeID == 0x400:

6           If modeBit == 0:
7             numberOfTargets = bit 47 to 54
8             CycleCount = bit 7 to 46
9             CycleDuration = bit 0 to 11* 0.000064

10          Else If modeBit == 1:
11            TimeStamp = bit 0 to 31

12          Else If modeBit == 2:
13            TimeStamp += bit 0 to 31* 0.00000000023283064365386962890625
14            Break

15    While numberOfTargets > 0:
16      If CANDriver.ReadMessage(message) returns NoError:
17        parsed = Parse(message)

18        If MessgaeID > 0x400 and MessgaeID < 0x47F:

19          If modeBit == 0:
20            point.Range = bit 1 to 31 * 0.04
21            point.RadioSpeed = bit 39 to 50 * 0.04
22            point.Azimuth = bit 22 to 31 * 0.16 //degree
23            point.ObjectID = MessageID - 0x400

24          If modeBit == 1:
25            point.Elevation = bit 37 to 46 * 0.04 //degree

26            frame.push_back(point)
27            numberOfTargets -= 1

28    Return frame
```

Figure 11: pseudocode of getFrame() for UMR11 over CAN

The function ***getFrame()*** is the most important function when it comes to data accusation throughout this work, however, additional functions exist that facilitate an optimal use of both the radar and the LiDARs. As shown in Figure 7, ***getFrameCount()*** function can help keeping track of how many frames have been processed so far, ***getTimeStamp()*** function facilitates matching frames from different sensors in the same point in time.

3.4 Implementation Details

With the functionality of interfacing with the LiDARs and radars shown in Figure 7 and described in sub-chapter 3.3 Data Processing we can now start with the actual processing of LiDAR and radar data. So far getting the Point cloud as a vector of points is just the first step of adding radar radial speed information to LiDAR point cloud. To reiterate, calling ***getFrame()*** function on a LiDAR would return a vector of points in cartesian coordination system as shown in Equation 2. Originally, they were encoded by the LiDAR as a spherical set of points but converted internally by the code by using Equation 1. Every element $[x, y, z]$ inside Equation 2 represents only one point, and no relationship between the points are discovered so far.

$$\begin{bmatrix} [x_0, y_0, z_0] \\ [x_1, y_1, z_1] \\ [x_2, y_2, z_2] \\ \vdots \\ [x_n, y_n, z_n] \end{bmatrix}$$

Equation 2: LiDAR Point cloud.

And when calling the same function of a radar would return a vector of data structures as shown in Equation 3. Each element $[ID, R, \dot{R}, \varphi, \theta]$ is assumed by the Radar's hardware and internal algorithms to be an object not just a detection point, thus every element comes with an identification number ID in addition to Radial Range R, Radial Speed $\dot{R}$, Azimuth Angle φ and Elevation Angle θ.

$$\begin{bmatrix} [ID_0, R_0, \dot{R}_0, \varphi_0, \theta_0] \\ [ID_1, R_1, \dot{R}_1, \varphi_1, \theta_1] \\ [ID_2, R_2, \dot{R}_2, \varphi_2, \theta_2] \\ \vdots \\ [ID_m, R_m, \dot{R}_m, \varphi_m, \theta_m] \end{bmatrix}$$

Equation 3: Radar Point Cloud

The first step of operation is to cluster the LiDAR point cloud shown in Equation 2 into a set of objects. This is performed by clustering operations, Which groups point of data into meaningful classes, in this case, physical objects inside a room. This reduces the work area from a large set of points with counts that reach to tens of thousands, into a set of objects with clear points and centers. As in one does not need to understand, predict, and track the behavior of each point in the point cloud, but focus on the interaction of a whole object where the points mostly have the same properties. In [21], a taxonomy of clustering algorithms is shown, they divide it into Single-Machine and Multi-Machine clustering techniques. Focusing on Single machine techniques, 2 major categories branch out, Datamining and Dimension reduction techniques. The first, Datamining, being categorized as an unsupervised classification tool with timing and resource needs depending on the amount of data, where more data means more time. And the second, Dimension reduction Techniques, where the dimensionality of data is reduced to a relevant subset of information. [21]

The method used in this work is DBSCAN, Density Based Spatial Clustering of Applications with Noise [18], where it falls under Density based clustering category of Data Mining techniques of single-machine clustering algorithms. Density based algorithms depend on density regions, where points of same location-density get clustered regardless of the direction. In-depth details about DBSCAN can be found at [18], [21].

For this current implementation, an adapted version of DBSCAN is used, with all the functionality needed to cluster available in the ***DBSCAN3D*** class which is part of the ***sensor*** namespace. The ***3D*** appended to DBSCAN is to indicate that it works with three-dimensional point cloud. ***DBSCAN3D*** class exposes a high-level interface to control a cluster the point-cloud, with the constructor of an object of ***DBSCAN3D*** requiring a reference to a point cloud, or vector of three-dimensional points, and a reference to a toolbox and stance the need for a reference for the point cloud is to reduce the amount of memory copy operations and maintain a manageable amount of memory usage. However, the reference for the toolbox is required to any changes done by the user can be instantly transferred to the ***DBSCAN3D*** instance.

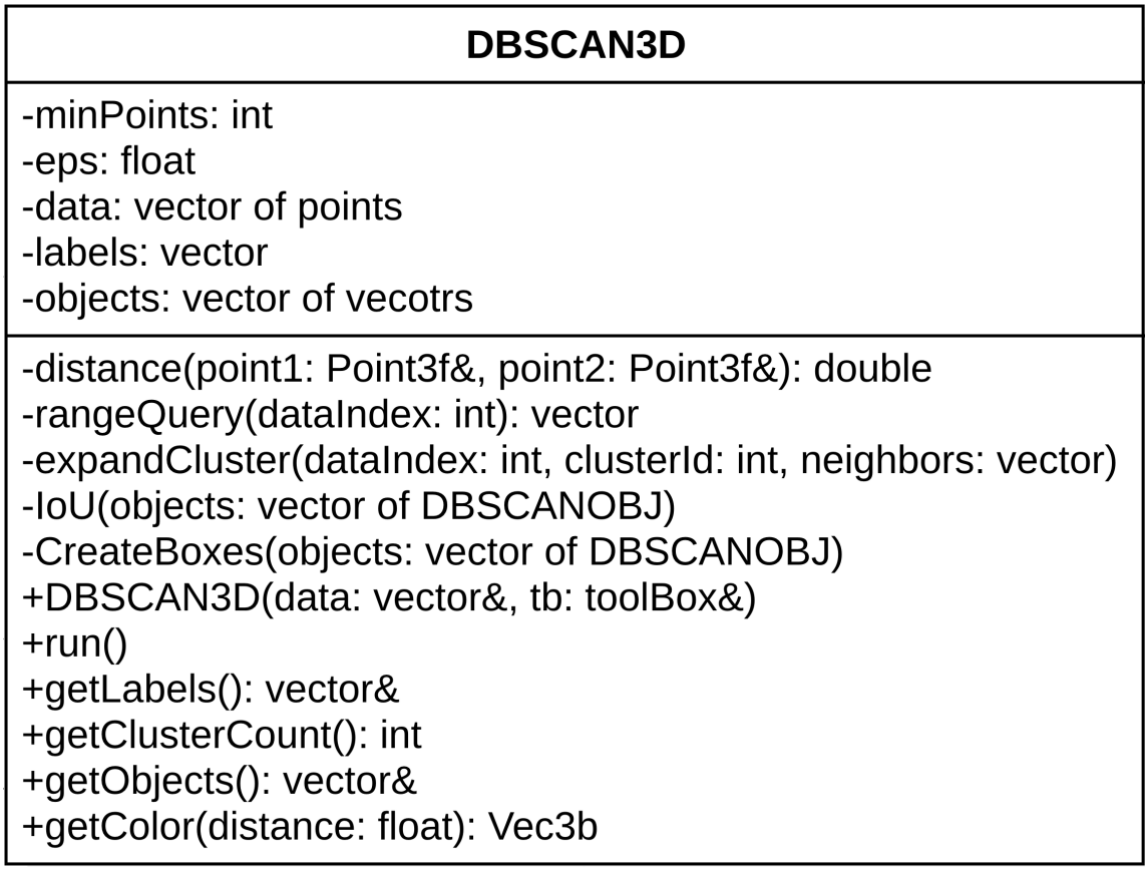

Figure 12: DBSCAN class description.

After creating an instance, the user can call the ***run()*** function Which will perform all the internal operations of plastering according to DBSCAN implementation. ***Run()*** function perform multiple time-consuming operations on the points which will be explained in a later step. After clustering the user is introduced to multiple ways to get access to the class data ***getLabels()*** function for instance returns of victor of equal size to the point cloud with

numbers indicating to which cluster ***Point n*** in the point cloud belongs to. The vector returned from ***getLabels()*** function contains values ranging from ***-1***, which represents noise, to a maximum number represented by ***getClusterCount***() function. For instance, the given point cloud ***PC*** = $[\boldsymbol{P_0}, \boldsymbol{P_1}, \boldsymbol{P_2}, \boldsymbol{P_3}, \boldsymbol{P_4}, \boldsymbol{P_5}, \boldsymbol{P_6}, \boldsymbol{P_7}]$ with ***labels*** = $[\mathbf{2}, \mathbf{2}, -\mathbf{1}, \mathbf{0}, \mathbf{2}, \mathbf{1}, \mathbf{1}, \mathbf{0}]$ show that P_7 and P_5 belong to cluster 0, P_5 and P_6 belong to cluster 1, P_0, P_1, and P_4 belong to cluster 2, and P_2 is noise. Alternatively, the user can use ***getObjects()*** in which a vector of DBSCAN objects ***DBSCANOBJ*** is returned, each object and the vector contains a victor of 3-D points representing all points making up that object, the center of that object, and the minimum and maximum point of the bounding box of that object. For instance, one object from the point cloud ***PC*** = $[\boldsymbol{P_0}, \boldsymbol{P_1}, \boldsymbol{P_2}, \boldsymbol{P_3}, \boldsymbol{P_4}, \boldsymbol{P_5}, \boldsymbol{P_6}, \boldsymbol{P_7}]$ with ***labels*** = $[\mathbf{2}, \mathbf{2}, -\mathbf{1}, \mathbf{0}, \mathbf{2}, \mathbf{1}, \mathbf{1}, \mathbf{0}]$, would be described in a ***DBSCANOBJ*** as shown in Equation 4, where ***getObjects()*** what return of vector of Equation 4.

$$\left[\boldsymbol{c}\ , \begin{bmatrix} \boldsymbol{P_0} \\ \boldsymbol{P_1} \\ \boldsymbol{P_4} \end{bmatrix}, \boldsymbol{P_{min}}, \boldsymbol{P_{max}} \right]$$

Equation 4: DBSCAN object, C: centroid of the object.

getObjects() and ***getLabels()*** only work after calling the ***run()*** function otherwise an old vector of labels or vector of DBSCAN objects would be returned and would only be valid on the point cloud on which the ***run()*** function was called. To call the ***run()*** function that performs the DBSCAN clustering operation, one must initialize a ***DBSCAN3D*** instance.
The ***run()*** function performs the bulk of the clustering operation. This is shown in Figure 13, where the functions iteratively run in a loop over all the data points in the provided point cloud.

```
1   Function run() is:

2     For each point 'i' in data from 0 to size of data - 1:

3       If labels[i] is not 0:  // If the point is already visited or assigned to a cluster
4         Continue to the next iteration

5       Initialize neighbors as the result of rangeQuery(i)

6       If the number of neighbors is less than minPts:
7         Set labels[i] to -1  // Mark the point 'i' as noise

8       Else:
9         Increment clusterId by 1
10        Call expandCluster with parameters 'i', 'clusterId', and 'neighbors'

11  End Function
```

Figure 13: DBSCAN run function algorithm.

The ***labels*** vector is, at the beginning, initiated to zeros, and when calling the ***run()*** function, the code checks if this point is originally part of another cluster by checking the value of that point in the ***labels*** vector. If not then ***rangeQuery()*** is called on that point where the distance between the specific point and every other point in the point cloud represented by the vector

data is measured, as shown in Equation 5, and added to the ***neighbors*** vector if the distance is less than or equal to epsilon. Then if the size of the ***neighbors*** vector is more than the minimum points, create a new cluster for the new parent point and every other point in the ***neighbors*** vector, otherwise mark the parent point as noise by setting its value to -1 in the ***labels*** vector.

$$d = \sqrt{(P0_x - P1_x)^2 + (P0_y - P1_y)^2 + (P0_z - P1_z)^2} \qquad \textit{Equation 5}$$

After calling ***run()*** One is then presented with the option of calling the ***getLables()*** or ***getObjects()*** methods, the ***getLables()*** requires more processing on the programmer's end however, it is still provided for different cases. ***getObjects()*** on the other hand provide direct access to object that have been clustered in addition to one extra step, where if the center distance of that object is less than epsilon of any other object, they get merged into one object. ***getObjects()*** returns the structure shown in Equation 4.
The ***getObjects()*** method extends the DBSCAN operations described in [18] by further processing the labels detected in the ***run()*** method via

1. merging objects with centers closer than epsilon distance used with DBSCAN clustering into one object, this is done by comparing the distance between the center of each object and merging the two vectors if the case applies.

2. Then creating bounding boxes for each object and storing the two furthest points of the boxed in the $\boldsymbol{P_{min}}, \boldsymbol{P_{max}}$ in Equation 4, this is done by initially sitting $\boldsymbol{P_{min}}$=$-\infty$ and $\boldsymbol{P_{max}}$=∞ and then finding the minimum value in x, y, and z and assigning it to $\boldsymbol{P_{min}}$, and the maximum value in x, y, and z and assigning it to $\boldsymbol{P_{max}}$ for each object. This is shown in Figure 14.

```
1   Function createBoxes(objects: List of DBSCANOBJ) is:
2     For each obj in objects:
3       For each point in obj.points:

4         obj.minPoint.x = Minimum(obj.minPoint.x, point.x)
5         obj.minPoint.y = Minimum(obj.minPoint.y, point.y)
6         obj.minPoint.z = Minimum(obj.minPoint.z, point.z)

7         obj.maxPoint.x = Maximum(obj.maxPoint.x, point.x)
8         obj.maxPoint.y = Maximum(obj.maxPoint.y, point.y)
9         obj.maxPoint.z = Maximum(obj.maxPoint.z, point.z)
10      End For
11    End For
12  End Function
```

Figure 14: Object bounding box based on the minimum and maximum points pseudo code.

3. Then finding the intersection over union of each object and merging objects accordingly, this is done bye it's writing over all pairs of objects and finding the corner points of intersection of each object. The minimum point of the intersection is the maximum of the two minimum points of either object, and the maximum point

of intersection is the minimum of the maximum of either object. Then, the volume of intersection is calculated by comparing the maximum between zero and the subtraction of the minimum and maximum points. Then the volume is calculated by adding the volume of both objects. The intersection of our union is calculated by dividing the intersection volume of the union volume. This is shown in Figure 15.

```
1   Function IoU(objects: List of DBSCANOBJ)
2     For i from 0 to objects.size - 1:
3       For j from i+1 to objects.size - 1:
          // Calculate the intersection bounds
4         minPoint = Point3f(
            Maximum(objects[i].minPoint.x, objects[j].minPoint.x),
            Maximum(objects[i].minPoint.y, objects[j].minPoint.y),
            Maximum(objects[i].minPoint.z, objects[j].minPoint.z)
          )
5         maxPoint = Point3f(
            Minimum(objects[i].maxPoint.x, objects[j].maxPoint.x),
            Minimum(objects[i].maxPoint.y, objects[j].maxPoint.y),
            Minimum(objects[i].maxPoint.z, objects[j].maxPoint.z)
          )

          // Calculate the volume of intersection
6         intersection = Maximum(0, maxPoint.x - minPoint.x) *
                         Maximum(0, maxPoint.y - minPoint.y) *
                         Maximum(0, maxPoint.z - minPoint.z)

          // Calculate the volume of union
7         union = (objects[i].maxPoint.x - objects[i].minPoint.x) *
                  (objects[i].maxPoint.y - objects[i].minPoint.y) *
                  (objects[i].maxPoint.z - objects[i].minPoint.z) +
                  (objects[j].maxPoint.x - objects[j].minPoint.x) *
                  (objects[j].maxPoint.y - objects[j].minPoint.y) *
                  (objects[j].maxPoint.z - objects[j].minPoint.z) -
                   intersection

          // Calculate IoU
8         iou = intersection / union

          // Merge objects if IoU is significant
9         If iou > ratio Then
10          Append objects[j].points to objects[i].points
11          Remove objects[j] from objects
12        End If
13      End For
14    End For
15  End Function
```

Figure 15: Intersection Over Union (IoU) pseudo code.

4. And finally updating the new bounding boxes for each object before returning the list of objects as shown previously in step 2.

All operations inside the DBSCAN class are serialized operations that run directly on the CPU. Using multiple-threading introduces major overhead on the system including the issues of synchronization of threads, Deadlocks, and the overhead of allocating and

managing threads with wait-time. Alternatively, a General-Purpose Graphics Processing Unit already exists on the system (Apple Silicon M2 Pro) where Metal Shading Language provides access to use the GPU for general-purpose calculation. However, this would, counter intuitively, slow down the calculation operations done during the dbscan operations compared to a serial CPU calculation execution cycle. As introduced in [22], for one-dimensional arrays smaller than 2^{22} elements performing SAXPY (Single precision A X plus Y) operations, execution time on the GPU would be higher than that of CPU for . The DBSCAN algorithm in this current implementation will typically work with a subset of the point cloud that is much smaller than the point cloud itself (which is the result of removing walls and ground planes by comparing the point position to a boundary cube and excluding any point that falls outside of it) which is typically around 128000 elements for Hesai Pandar XT32 in which, according to [22], would be approximately 25 times slower, as a size of 128000, or approximately 2^{17} single-indexed element needs approximately $10^{5.7}$nano second on the GPU compared to $10^{4.3}$ nano second on the CPU for SAXPY operations as shown in Figure 16. In addition to overhead of memory copy operations between the GPU and the CPU memory spaces of the code.

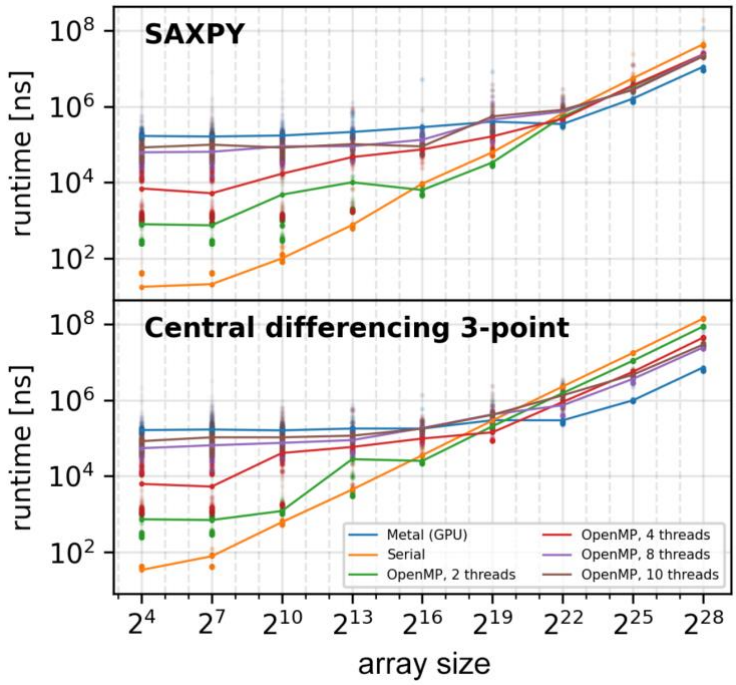

Figure 16: Runtime of one-dimensional operations on data of various sizes [22].

Multi-threading operation is reserved, in this current implementation, for capturing the UDP and CAN data from the sensors as a method of avoiding blocking the code for each sensor which would otherwise introduce loss of sensor data. A typical use case is shown in Figure 17 Where in line 3 and 4 the LiDAR and the radar are set up for operation over UDP and CAN, then two different point clouds are created for each sensor. Then the ***getFrame()*** method of each sensor is called inside a different thread, where the UDP and CAN blocking operations happen outside the main code and the used can perform other operations while the data is ready. When performing data on the point clouds, the user can call the ***join()***

method on either thread, which will block execution until the thread is done processing, which, if timed correctly introduces no or minimum blocking time, and the user can process the point cloud data of each sensor separately.

```
1   int main() {
2     Sensor::toolBox tb;
3     Sensor::ipXT32lidar lidar(2370, tb);
4     Sensor::CANSmartMicroRadar radar(tb);

5     std::vector<cv::Point3f> lidarPC;
6     std::vector<Sensor::radarPoint> radarPC;

7     std::thread t1([&](){lidarPC = lidar.getFrame();});
8     std::thread t2([&](){radarPC = radar.getFrame();});

      //Other Operations
9     t1.join();
      //Other Operations
...   t2.join();
      //Other Operations
```

Figure 17: Multi-threaded Data acquisition.

Thread ***t1*** is responsible for grabbing the data that will be clustered with DBSCAN, thus after waiting calling ***join()*** method on thread ***t1,*** DBSCAN operations can be started without needing to wait for thread ***t2*** with the radar data. This is shown in Figure 18which is a continuation of the code snippet shown in Figure 17.

```
      //Other Operations
9     t1.join();

10    Sensor::DBSCAN3D dbscan(lidarPC,tb);
11    dbscan.run();
12    std::vector<Sensor::DBSCANOBJ> objects = dbscan.getObjects();

…     t2.join();
      //Other Operations
```

Figure 18: DBSCAN code Operations.

In Figure 18, it is necessary to wait for the LiDAR data before clustering them, that's why calling ***join()*** on thread ***t1*** is important before the clustering operation. Then the user can create a new DBSCAN object with the acquired LiDAR point cloud, call the ***run()*** method, and then call the ***getObjects()*** method. During this processing time, the thread ***t2*** responsible for grabbing and decoding the radar data should have finished, the user can call the ***join()*** method on ***t2*** to make sure execution is done, if not, extra wait (block) time on the ***join()*** method will occur.

Now that we are done processing the LiDAR frame and finished waiting for the radar frame, we can proceed with processing the radar data. There is no specific class or specific chosen

method to process the radar points as they are not raw data or reflections, as those of the LiDAR or other kind of radars, but a clustered data on-board the radar. Some basic functions are available to set the orientation of the point cloud in reference to the position of other sensors or middle points. Figure 19 shows the processing done for the radar point cloud called. We first rotate the point cloud by a certain degree on the Z-axis as shown in line 14 this corresponds to the physical mounting of the radar with respect to the center of the room. Then we iterate over each point in the point cloud to calculate its Cartesian coordinates according to Equation 1. Where we are interested in the Radial distance R, Azimuth angle and Elevation Angle.

```
13    t2.join();
14    Sensor::rotateVector(radarPC, degree , 'z');
15    for (auto& point : radarPC)
      {
16    auto X3d = point.R*cos(toRad(point.Azimuth))*cos(toRad(point.Elevation));
17    auto Y3d = point.R*sin(toRad(point.Azimuth))*cos(toRad(point.Elevation));
18    auto Z3d = point.R * sin(toRad(point.Elevation));
19    auto Radius = 2*(point.R)*sin(toRad(0.5)/2);
...     //Other Operations
```

Figure 19: Radar Point could processing.

Each point in a radar point cloud can be thought of as the center of a detection. However, due to the inaccuracy of the azimuth and elevation angle of 0.5 degrees each, every point can be thought of as a sphere of error where the actual point of detection would be anywhere inside that sphere. Now the process involves checking if each sphere intersects with any object detected by the LiDAR, if that was the case, both the detection from the radar and the object created by the LiDAR clustering gets pushed into one ***combinedObject*** structure which is a structure that holds one ***radarPoint*** and one ***DBSCANOBJ.***

```
19  auto Radius = 2*(point.R)*sin(toRad(0.5)/2);
20  std::vector<Sensor::combinedObject> combinedObjects;
21  for (auto& object : objects) {
22   if (X3d + Radius < object.minPoint.x || X3d - Radius > object.maxPoint.x
    || Y3d + Radius < object.minPoint.y || Y3d - Radius > object.maxPoint.y ||
    Z3d + Radius < object.minPoint.z || Z3d - Radius > object.maxPoint.z)
    {
23   continue; // No intersection check next object
     }
24   combinedObjects.push_back(Sensor::combinedObject{point, object});
    }
```

Figure 20: intersection between LiDAR objects and radar detection.

Figure 20 shows this operation of intersection. Combined objects are only logged during this process as separate objects but visualized no differently than any other objects as will be shown in 4.1 Presentation of Implemented Results. The full code presented from Figure 17 to Figure 20 can be summarized in one piece of code as shown in Figure 21. This code uses the ***Sensor*** namespace with all its provided tools and drivers to interface with a CAN Radar and an Ethernet LiDAR, cluster and fuse all in one source file.

```
int main() {
  Sensor::toolBox tb;
  Sensor::ipXT32Lidar lidar(2370, tb);
  Sensor::CANSmartMicroRadar radar(tb);

  std::vector<cv::Point3f> lidarPC;
  std::vector<Sensor::radarPoint> radarPC;

  while(Condition) {

    std::thread t1([&](){lidarPC = lidar.getFrame();});
    std::thread t2([&](){radarPC = radar.getFrame();});

    t1.join();
    Sensor::DBSCAN3D dbscan(lidarPC,tb);
    dbscan.run();
    std::vector<Sensor::DBSCANOBJ> objects = dbscan.getObjects();

    t2.join();
    Sensor::rotateVector(radarPC, degree , 'z');
    for (auto& point : radarPC){
      auto X3d =point.R*cos(toRad(point.Azimuth))*cos(toRad(point.Elevation));
      auto Y3d = point.R*sin(toRad(point.Azimuth))*cos(toRad(point.Elevation));
      auto Z3d = point.R*sin(toRad(point.Elevation));
      auto Radius = 2*(point.R)*sin(toRad(0.5)/2);
      std::vector<Sensor::combinedObject> combinedObjects;
      for (auto& object : objects) {
         if (X3d + Radius < object.minPoint.x || X3d - Radius > object.maxPoint.x ||
         Y3d + Radius < object.minPoint.y || Y3d - Radius > object.maxPoint.y || Z3d
         + Radius < object.minPoint.z || Z3d - Radius > object.maxPoint.z) {
           continue; // No intersection check next object
           }
        combinedObjects.push_back(Sensor::combinedObject{point, object});
      }
    }
  }
```

Figure 21: Full code using the sensor Library.

4. Results

4.1 Presentation of Implemented Results

The ***Sensor*** Toolset provides multiple levels of processing. None of which particularly requires visualization in a certain way, however, since we are working in 3D space, it seems suitable to visualize the results accordingly. The tool used is Open Computer Vision (OpenCV) C++ Library. The ***cv::viz*** namespace provides all the tools needed to draw 3D points and shapes to present the result in an attractive way with a lot of desirable features such as platform independency. The main rendered types that ***cv::viz*** provides and are used in code:

1. ***cv::viz::Wcloud,*** which can render the vector of 3D points,
2. ***cv::viz::Wcube,*** which renders a 3D cube,
3. ***cv::viz::WText3D*** which renders text in 3D, and
4. ***cv::viz::WcoordinateSystem,*** which shows a gimble of the axis.

All of these types are contained in an instance of ***cv::viz::Viz3d window*** which represents a rendering window where all 3D elements are rendered on using the function ***cv::viz::Viz3d::showWidget(…)***.
With this functionality, we can now start visualizing the data on 3D space, the most basic operation is to grab a LiDAR frame and render the points, this is shown in Figure 22, in which only 3 elements show, A point cloud object with all the points from the LiDAR, a boundary box wireframe, a coordinate gimble showing x-axis in red, y-axis in green and z-axis in blue pointing towards the ground.

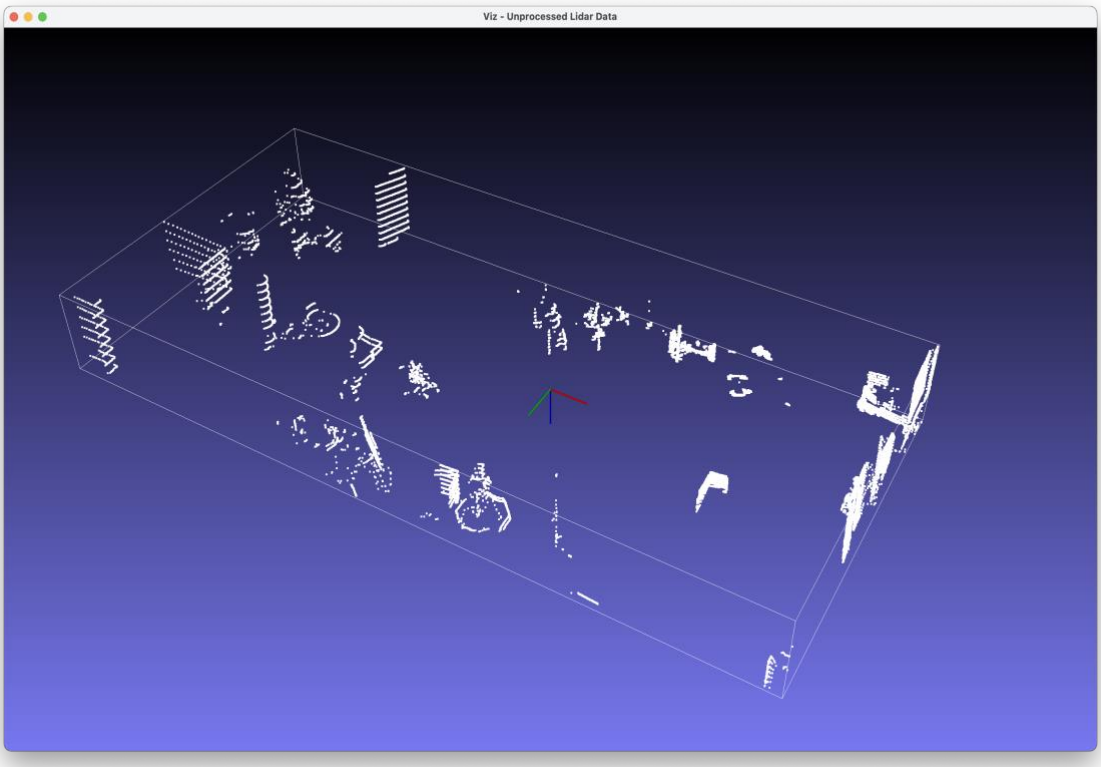

Figure 22: unclustered LiDAR Point cloud.

The next step is to perform DBSCAN on the point cloud, the results of DBSCAN are the vector of labels and can be presented in multiple ways, in Figure 23, random color is assigned for each label. It is clear that the DBSCAN method that is used is not enough on its own to produce reliable results but is tuned for speed over accuracy.

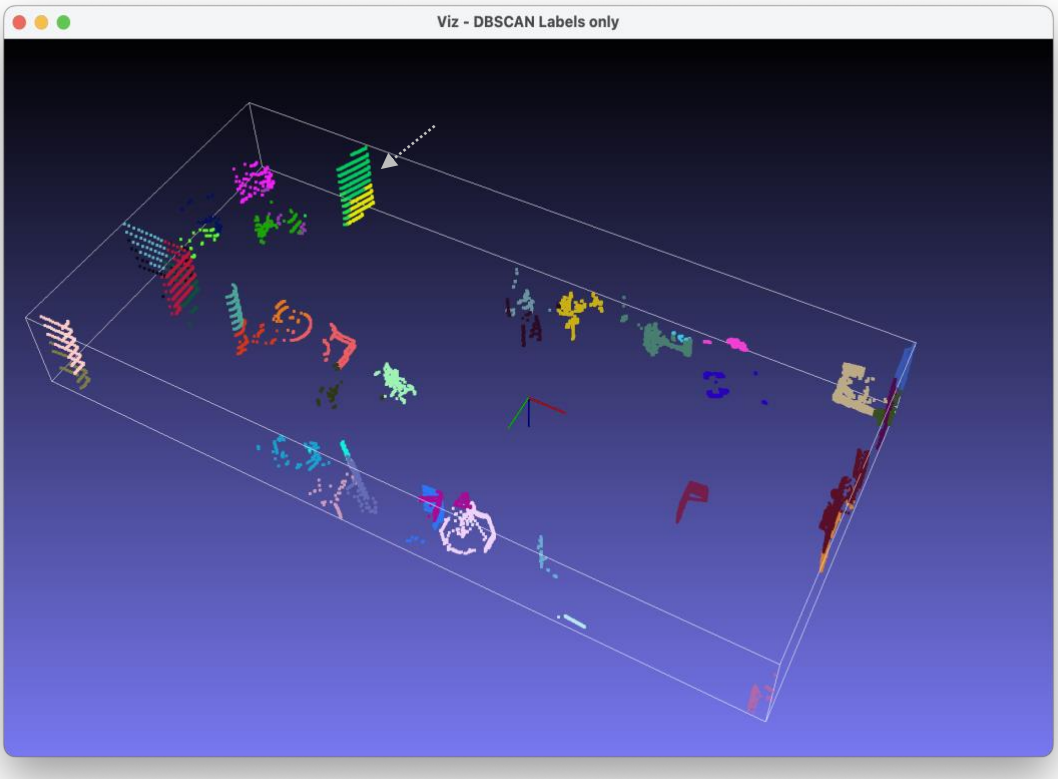

Figure 23: DBSCAN on LiDAR Point cloud.

The first step is to combine objects with center-to-center distance less than epsilon together. This is done in-code as part of the DBSCAN library and is discussed in 3.4 Implementation Details. The result is after merging centers can be seen in Figure 24 and it already shows better results, for instance the green and yellow sections in the top left part of Figure 23, pointed at by the grey arrow, represent an opened door as two sections of the same door, where in Figure 24, represented in red, shows as one object. At this point, where we have defined objects, it would be more consistent to give colors more meaning, we can use colors to represent how far an object is in addition to drawing boxes around objects, where green means a closer object to the center and blue/violet means a further object. This is shown in Figure 25, additionally, the results of performing intersection over union object merging is shown and distance between the center of the room (represented by the 3D axis gimble) to the center of each object (in centimeters) is added.

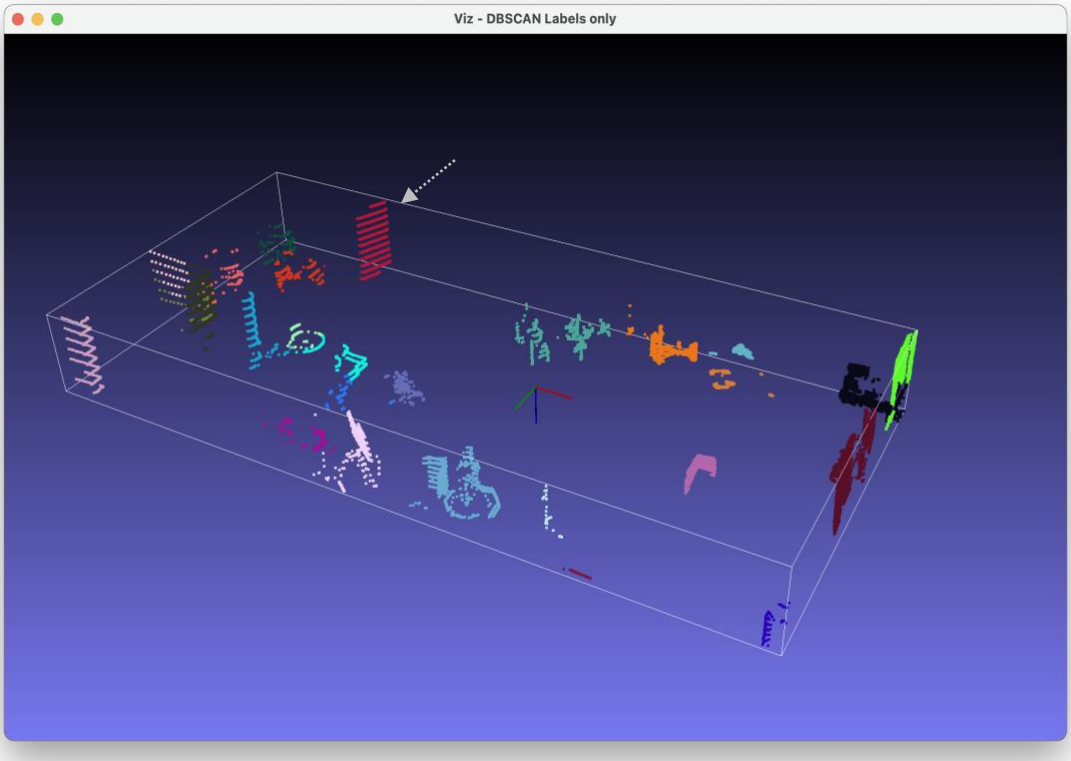

Figure 24: DBSCAN with close centers merged.

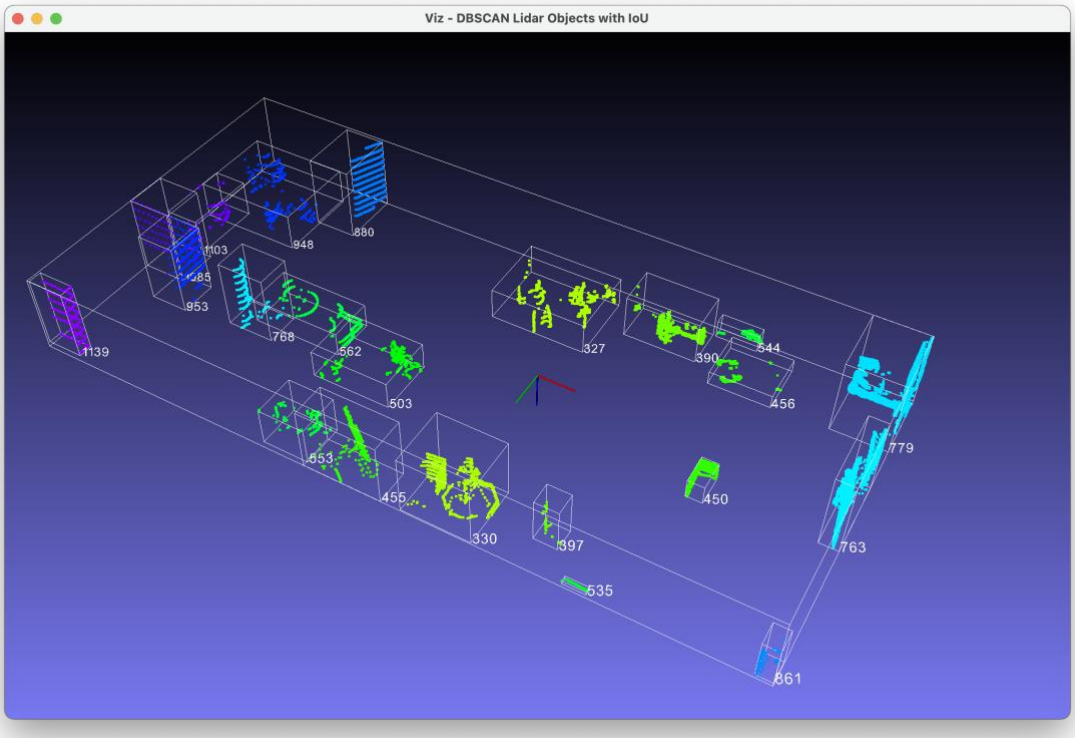

Figure 25: DBSCAN objects with IoU, Bounding boxes, and Distance from the center (in centimeters)

We can now proceed with showing the Radar detection on the same 3D space as the LiDAR objects. This is shown as spheres in Figure 26 at this point no further visualization is needed but a new kind of C++ class representing an object is created and stores the instances of the LiDAR object after clustering and the Radar instance.

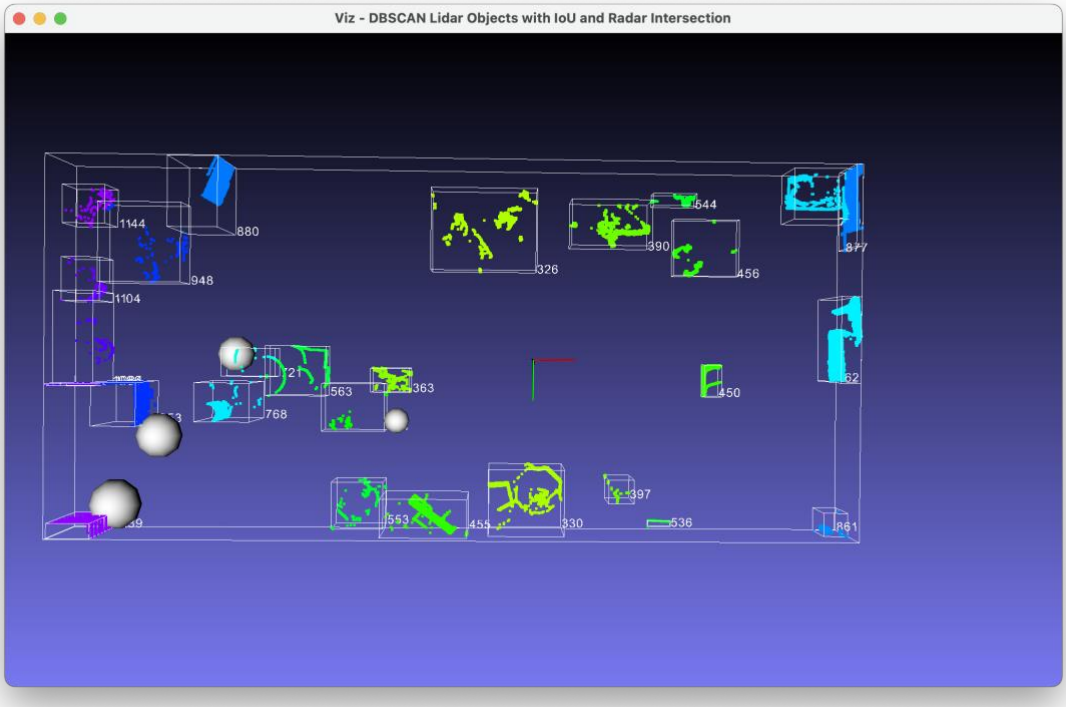

Figure 26: LiDAR objects with Radar Detections

An example of the merger of objects can be showed in terminal dump of the vector of combined objects as shown in Figure 27 where 4 objects are combined with radar points and each object has a reference to the Radar Point (RP) and LiDAR Object(LO).

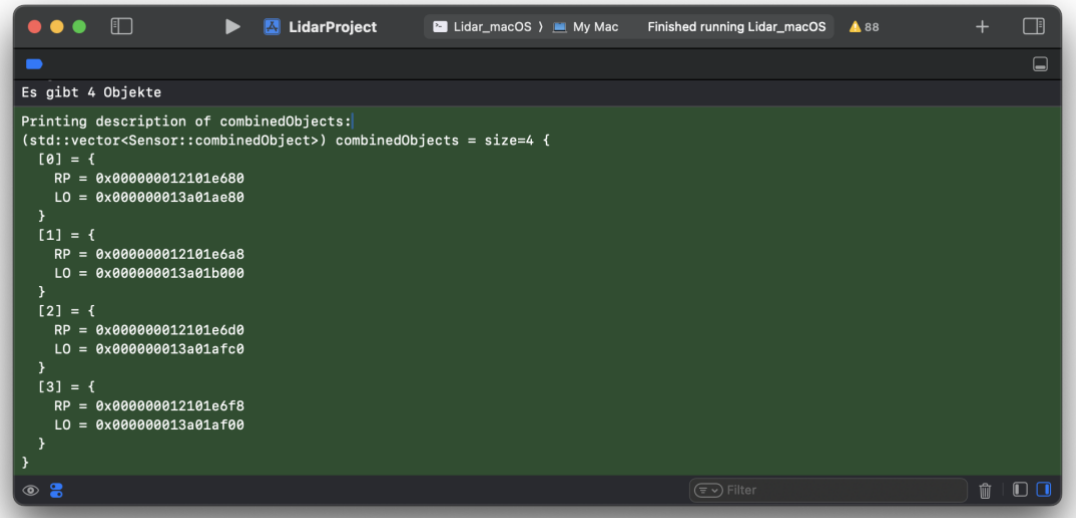

Figure 27: Combined objects.

4.2 Performance Metrics

Measuring performance for the current implementation is done via measuring the execution time of the code and measuring memory usage and leaks. Both are imperative for ensuring accurate detections and error-free execution on hardware. These tests are run locally on an Apple M2 MacBook Pro with 10 (6 performance and 4 efficiency) CPU cores, and 16 GB of Memory using Xcode version 15.4.

4.2.1 Timing Analysis

Execution-time is carried out by measuring execution time of each of the complex blocks of codes individually, in addition to measuring the whole execution cycle for each frame (Radar and LiDAR frame). Measuring processing time is critical to ensure no frames are dropped by either sensor, each processing sequence must occur between the accusation time of the next frame. Measuring overall execution time is sufficient to check that, however, more time-based information helps in optimizing the code and identifying issues in future development.

The most important factor for ensuring no frame drops is to first set the base-line timing requirement for both the LiDAR and the Radar. The Hesai XT32 LiDAR is set at a 10Hz frequency [3], or 10 frames per second, which means each one frames takes 100 ms to be captured and sent through UDP. And with the nature of multi-threading used in getting the frame, a base line of maximum ***100 ms*** is set for processing. For the Smartmicro UMR11 radar, it is calculated throughout the experiment to take on average ***54.98 ms*** Detection time as shown in Table 5, which seems to imply the detection time is independent from the number of detections, thus lowering the baseline to ***54.98 ms.***

Table 5: Detection times vs. number of detections for UMR11 Radar

	Detection Time (ms)	Radar Detections
Max	54,98	9,00
Min	54,66	2,00
Average	54,98	6,41

This means, for optimum performance, the overall processing time including all activities should be less than 54.98 ms. Table 6 shows the most time-consuming blocks of code, including DBSCAN run time, DBSCAN object creation time, Intersection over Union time, and additional processing time which is part of the main executable where LiDAR and radar objects get merged. The time difference between each sensor implies that almost 2 radar frames exist with every one LiDAR frame, since the code requests data packets at the same time, every second radar packet is discarded for presentation purposes but kept for timing analysis, solutions for this will be discussed in chapter 5. Discussion.

Table 6: Timing characteristics of execution.

Catagory	DBSCAN RUN	Object Creation	IoU	Additional processing	Overall Time
Min (ms)	19,00	3,00	1,00	3,00	28,00
Max (ms)	121,00	4,00	1,00	6,00	130,00
Average (ms)	55,85	3,05	1,00	4,16	64,06

Rendering time for visualization is excluded as it is handled by a different thread of execution and managed by the OpenCV library, additionally, data accusation time is also not considered as this timing study happens after acquiring the data. From Table 6 it is clear that the most time consuming block of code is the DBSCAN run, which is shown in Figure 13, the Overall run time of each frame is heavily influenced by the DBSCAN run time as the other operations take significantly less time and can be ignored. Based on Table 6, The average execution time is 64 ms which does not hit the baseline mark for optimum execution, however, taking a look at the distribution of the timing provides better understanding for timing analysis shown in Table 7 and Figure 28.

Table 7: processing time distribution.

Time bracket	Range	Frames[2]
30	<=30	3
40	31-40	291
50	41-50	1881
60	51-60	1567
70	61-70	1154
80	71-80	1022
90	81-90	870
100	91-100	666
110	101-110	87
120	111-120	13
more	>120	4

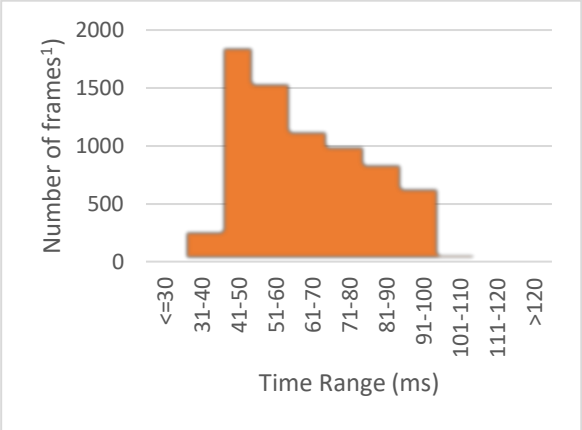

Figure 28: Distribution of frame processing time.

From both Table 7 and Figure 28, the results of processing time can be shown in Table 8 where almost 50% of the radar frames are lost, where almost all LiDAR frames are processed. This percentage difference is discussed in chapter 5. Discussion

Table 8: percentage of processed LiDAR and radar frames

	Radar	LiDAR
on time	**3742,00**	**7454,00**
missed	**3816,00**	**104,00**
percentage	49,51	98,62

[2] One radar frame and one lidar frame

4.2.2 Memory Analysis

Managing memory allocation and preventing memory leaks are crucial for maintaining the performance and stability of software applications. Efficient memory management ensures that programs run smoothly, use resources effectively, and avoid issues like crashes due to excessive memory usage or bad addresses. Many programming languages like Java and python offer built-in garbage collection for memory management, however, in C++ ensuring memory safety is done on the programmer's side by keeping track of the memory allocation system calls, performing the proper cleaning up after usage, and statically allocating and reusing allocated memory. Static and dynamic memory analysis is also important to avoid unforeseen issues.

For this implementation, memory is allocated and reused to ensure a limited memory usage and reduce allocation and deletion time, references (passing by reference) to point clouds and constructs is also used instead of memory copy operations (passing by value). where the required memory is limited during the modeling phase.

Dynamic memory analysis is done using Xcode Instruments tool which provides detailed information regarding memory usage and detecting leaks.

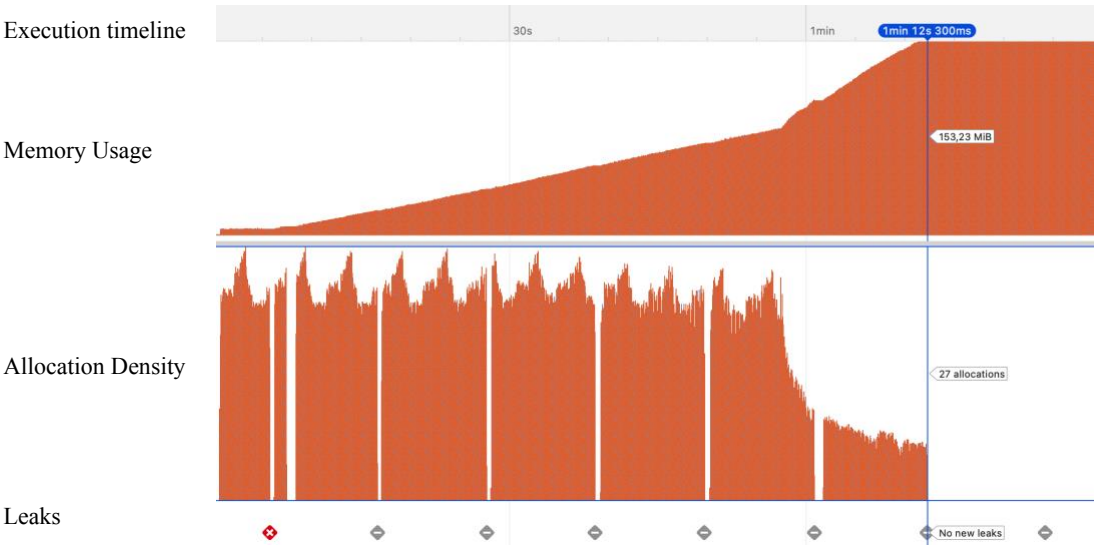

Figure 29: Memory Usage and leaks.

Figure 29 shows a timeline of the execution with the overall memory usage of the executable, the allocation density (where the executable requests memory from the operating system) and the number of leaks detected every period of time. The memory usage climbs steadily towards approximately 153 MB over the execution time of 1 minute and 12 seconds. this is when the code is requesting memory for the first time for each object it is using. This is also shown by the constant density of allocations after the 1 minute and 12 seconds mark, the executable does not request any further memory and thus does not consume any more memory. Restricting leaks is critical in any application for security, stability, and performance purposes, especially with large data constructs such as point clouds. From Figure 29 we can see one instance of memory leak detected at the beginning of execution; this is related to a string type of a file name related to the pcap file containing the logged LiDAR data. However, no memory leaks are detected afterwards during the execution of the code, and since no more memory is allocated after the 1 minute and 12 seconds mark, no

more leaks would be detected. It is necessary to note that the 1 minute and 12 seconds mark is not a hard-coded time stamp, but an affect that happens after a certain execution time and it might differ from one platform to another or on multiple runs. However, it will always reach a point in time where no more allocation is required.

5. Discussion

5.1 Interpretation of results

The result of this experiment shows that adding Radial speed information to LiDAR point cloud is achievable with some considerations.

5.1.1 Detection Area Coverage

The LiDAR used in this experiment covers more area than the radar used. However, in a typical vehicle setup, radars have of coverage usually by using multiple ones thus in a real-life situation this would not be a problem. Extending the setup multiple Radar Sensors requires additional timing analysis however due to the very short processing time of Radar point, a scale in number of processing threads should be the approach to be tested first.

5.1.2 Memory Management

Effective memory management was demonstrated by maintaining a steady memory usage by not creating any more memory during runtime. As a numerical metric, approximately 153 MB was used throughout the execution. The value of 153 MB is not constant across different execution cycles, it depends on Memory fragmentation state of the machine, the operating system, and other system-related variables. The use of Xcode Instruments tool confirms that no significant memory leaks were detected, ensuring the stability and performance of the system. The initial memory leak detected was minor and related to a string-type for file naming managed by an external library. However, the amount of memory allocations is constant and limited after a certain execution time of each cycle and thus no memory leaks would show up later since all memory is being cleared and reused every time a new LiDAR or radar frame arrives eliminating the possibility of memory leaks.

5.1.3 Performance metrics

Execution time and Timing analysis are the important terms to consider in this section. Execution time with respect to processing radar frames and LiDAR frames independently was measured to be suitable for each sensor individually within the time restrictions of each. However, Timing Analysis shows that mis-matched frequency of the sensors causes dropped frames in the higher-frequency sensor, in this case the Radar, the solution in this case would be to change the frequency of LiDAR data, making it faster, for instance 20Hz that matches the radar frequency, or use time stamps to choose the right radar point cloud.

5.1.4 Combined objects

The structure of combined radar and LiDAR objects contains a reference, not a copy, of the radar point cloud object and the LiDAR DBSCAN-clustered object. A reference was used

in order to maintain memory management goals by eliminating memory allocation calls and the possibility of memory leaks and reduce the CPU time spent in memory management as a side effect. The structure of combined objects can be easily extended in future work to encapsulate any other sensor detections such as Ultrasonic, additional LiDARs or radars, etc.

5.2 Recommendation for Improvements

5.2.1 DBSCAN optimization

Given that the DBSCAN operation consumes the most amount of time in the processing pipeline, optimizing the current implementation or researching more appropriate, timewise, implementations could reduce the overall processing time and would allow the use of faster, framerate-wise, LiDAR setups. Additionally, exploring more primitive data-structures, for instance C-style arrays instead of C++-style vectors, could be more time-efficient.

5.2.2 Data-fusion enhancements

Detection accuracy can be improved by the use of some advanced data fusion techniques such as Kalman filters or some machine-based learning approaches. This would provide better predictions and handle objects more effectively.

5.2.3 Memory management optimization

Although memory usage during execution was effectively managed and controlled to some degree, some improvements can be done to ensure further memory safety in C++ coding. This includes using for instance C++ smart pointers and other modern C++ features. Additionally, some memory management techniques such as RAII (Resource Acquisition Is Initialization) can be used in future iterations. This is important because C/C++ is infamous for memory unsafety and the usage of C/C++ in time-critical or embedded systems is necessary.

5.2.4 Extending to outdoor environments

Radar and LiDAR systems behave differently in an uncontrolled outdoor environment due to variations in weather conditions, larger detection areas, existence of fast unpredictable moving objects, and so on. Some necessary changes in processing and point cloud filtering would be utilized in that case.

5.2.4 Timing optimization

Matching the frame rate of the sensors used is important to ensure accurate sensor fusion of real-time information. Additionally, the usage of the external GPS-Based timing functions provided by the LiDAR and Radar would provide better time tracking of each frame facilitating accurate matching of frames.

5.2.5 Radar point cloud processing

The radar used in this implementation [6] performs internal object detection operations on the initial point clouds generated and returns a single point representing the object. However, radar sensors in general do not perform these operations internally and thus additional processing functions would be required to perform these operations. This includes using DBSCAN operations on the radar point clouds in addition to applying some Bayesian filter to filter the noise and mitigate the inconsistency of speed measurements from the radar systems.

6. Conclusion

In this project thesis, Sensor fusion was performed in order to enhance LiDAR point clouds with radial speed detected by a radar. This is achieved by developing a C++ framework that facilitates all the operations needed from data acquisition all the way to presentation. The developed C++ library interfaces with three LiDAR sensors and one Radar sensor, either directly or through logged detection files. It includes clustering tools for processing the sensor data and fusing the results to create combined objects that encapsulate the detection information of both sensors. Memory management and timing analysis were critical aspects of the implementation, ensuring efficient and reliable execution without significant data leaks or memory overuse. The results show the feasibility and effectiveness of this approach. The system successfully adds radial speed information to LiDAR's clustered point clouds, providing enhanced detection capabilities. The memory management strategy proved effective, with the system maintaining steady memory usage and avoiding significant leaks. Performance metrics indicated that while the execution time for individual sensor processing was suitable, further optimization is needed to address mismatched sensor frequencies that led to dropped frames, particularly from the Radar. The discussion highlighted the importance of optimizing the DBSCAN clustering algorithm and exploring advanced data fusion techniques, such as Kalman filters or machine learning approaches, to improve detection accuracy and efficiency. Recommendations for future work include extending the system to outdoor environments, optimizing memory management with advanced C++ features, and enhancing timing synchronization between sensors. In conclusion, the framework developed in this project thesis provides a foundation for fusing LiDAR and Radar data for automotive perception. It addresses key challenges in memory management, timing analysis, and data fusion, while maintaining a manageable level of code complexity for debugging and optimization.

Bibliography

[1] R. Ayala and T. K. Mohd, “Sensors in Autonomous Vehicles: A Survey,” *J. Auton. Veh. Syst.*, vol. 1, no. 3, 2021, doi: 10.1115/1.4052991.

[2] “Hesai XT32.” https://www.hesaitech.com/product/xt32/ (accessed Mar. 11, 2024).

[3] HESAI Technology, “PandarXT-32 32-Channel Medium-Range Mechanical LiDAR User Manual,” 2020, [Online]. Available: https://www.oxts.com/wp-content/uploads/2021/01/Hesai-PandarXT_User_Manual.pdf

[4] M. Lidar and U. Manual, “64-Channel Short-Range Mechanical LiDAR User Manual”.

[5] Velodyne, “VLP-16 User Manual”.

[6] Smartmicro, “UMR11 Type 132 Datasheet,” 2021, [Online]. Available: https://www.smartmicro.com/fileadmin/media/Downloads/Automotive_Radar/Sensor_Data_Sheets_76-81GHz/UMRR-11_Type_132_Automotive_Data_Sheet.pdf

[7] “Open Computer Vision.” https://opencv.org (accessed Mar. 19, 2024).

[8] “PcapPlusPlus.” https://pcapplusplus.github.io (accessed Mar. 19, 2024).

[9] W. A. Ahmad and X. Yi, “How Will Radar Be Integrated into Daily Life?: Mm-Wave Radar Architectures for Modern Daily Life Applications,” *IEEE Microw. Mag.*, vol. 23, no. 5, pp. 30–43, 2022, doi: 10.1109/MMM.2022.3148343.

[10] S. M. Patole, M. Torlak, D. Wang, and M. Ali, “Automotive Radars: A review of signal processing techniques,” *IEEE Signal Process. Mag.*, vol. 34, no. 2, pp. 22–35, 2017, doi: 10.1109/MSP.2016.2628914.

[11] J. Liu, Q. Sun, Z. Fan, and Y. Jia, “TOF lidar development in autonomous vehicle,” *2018 3rd Optoelectron. Glob. Conf. OGC 2018*, pp. 185–190, 2018, doi: 10.1109/OGC.2018.8529992.

[12] J. B. Abshire, “NASA’s space lidar measurements of earth and planetary surfaces,” *Optics InfoBase Conference Papers*, 2010. https://ntrs.nasa.gov/citations/20100031189 (accessed Feb. 17, 2024).

[13] M. E. Warren, “Automotive LIDAR Technology C254 C255,” *2019 Symp. VLSI Circuits*, vol. 1, pp. 254–255, 2019.

[14] Velodyne, “Velodyne LiDAR Puck Data Sheet,” *Velodyne LiDAR, Inc.*, 2019, [Online]. Available: https://www.velodynelidar.com/vlp-16.html

[15] K. Na, J. Byun, M. Roh, and B. Seo, “Fusion of multiple 2D LiDAR and RADAR for object detection and tracking in all directions,” *2014 Int. Conf. Connect. Veh. Expo, ICCVE 2014 - Proc.*, pp. 1058–1059, 2014, doi: 10.1109/ICCVE.2014.7297512.

[16] L. Wang *et al.*, “InterFusion: Interaction-based 4D Radar and LiDAR Fusion for 3D Object Detection,” *IEEE Int. Conf. Intell. Robot. Syst.*, vol. 2022-Octob, pp. 12247–12253, 2022, doi: 10.1109/IROS47612.2022.9982123.

[17] OMG, “UML 2.4.1 Superstructure Specification,” *October*, vol. 02, no. August, pp. 1–786, 2004, [Online]. Available: https://www.omg.org/spec/UML/2.4.1/Superstructure/PDF

[18] X. X. Ester Martin, Kriegel Hans-Peter, Sander Jörg, “A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise,” *Proc. 2nd Int. Conf. Knowl. Discov. Data Min.*, pp. 226–231, 1996, [Online]. Available: https://www.dbs.ifi.lmu.de/Publikationen/Papers/KDD-96.final.frame.pdf

[19] A. Alhalabi, “UMR11-132 Driver,” 2024. https://github.com/obi-two-kenobi/UMR11-132-Radar-CAN-driver-for-MacOS.git (accessed Jun. 19, 2024).

[20] “Wireshark.” https://www.wireshark.org (accessed Mar. 28, 2024).

[21] K. Djouzi and K. Beghdad-Bey, “A Review of Clustering Algorithms for Big Data,” *Proc. - ICNAS 2019 4th Int. Conf. Netw. Adv. Syst.*, pp. 1–6, 2019, doi: 10.1109/ICNAS.2019.8807822.

[22] L. Gebraad and A. Fichtner, “Seamless GPU Acceleration for C++-Based Physics with the Metal Shading Language on Apple’s M Series Unified Chips,” *Seismol. Res. Lett.*, vol. 94, no. 3, pp. 1670–1675, 2023, doi: 10.1785/0220220241.

APPENDIX A: Payload structure for Lidars

Hesai XT32

```
struct recvBufferType
{
    char ethernetHeader[42];

    struct PreHeaderType
    {
        uchar SOP0;
        uchar SOP1;
        uchar ProtocolversionMajor;
        uchar ProtocolversionMinor;
        unsigned short Reserverd;
    }PreHeader;

    struct headerType
    {
        uchar LaserNum;
        uchar BlockNum;
        uchar Reserved;
        uchar DisUnit;
        uchar ReturnNumber;
        uchar UDPSeq;
    }header;

    struct blockType
    {
        unsigned short azimuth;
        struct channelType {
            unsigned short distance;
            uchar reflectance;
            uchar reserved;
        }channel[32];

    }block[8];

    struct tailType {
        uchar reserved[10];
        uchar returnMode;
        unsigned short motorSpeed;
        struct DateTimeStruct{
            uchar year;
            uchar month;
            uchar day;
            uchar hour;
            uchar min;
            uchar sec;
        }DateTime;
        unsigned int TimeStamp;
        uchar FactoryInfo;
    }tail;

    struct additionalInfoType {
        unsigned int udpSeq;
    }additionalInfo;

}LidarPayload;
```

Hesai PandarQT

```
struct recvBufferType
{
    char ethernetHeader[42];

    struct PreHeaderType
    {
        uchar SOP0;
        uchar SOP1;
        uchar ProtocolversionMajor;
        uchar ProtocolversionMinor;
        unsigned short Reserverd;
    }PreHeader;

    struct headerType
    {
        uchar LaserNum;
        uchar BlockNum;
        uchar FirstBlockReturn;
        uchar DisUnit;
        uchar ReturnNumber;
        uchar UDPSeq;
    }header;

    struct blockType
    {
        unsigned short azimuth;
        struct channelType {
            unsigned short distance;
            uchar reflectance;
            uchar reserved;
        }channel[64];

    }block[4];

    struct tailType {
        uchar reserved[10];
        unsigned short motorSpeed;
        uchar returnMode;
        struct DateTimeStruct{
            uchar year;
            uchar month;
            uchar day;
            uchar hour;
            uchar min;
            uchar sec;
        }DateTime;
        uchar FactoryInfo;
        unsigned int TimeStamp;
    }tail;

    struct additionalInfoType {
        unsigned int udpSeq;
    }additionalInfo;

}LidarPayload;
```

Velodyne VLP16

```
struct recvBufferType
{
    char ethernetHeader[42];

    struct blockType{
        unsigned short flag;
        uchar azimuth[2];
        struct channelType{
            uchar distance[2];
            uchar reflectance;
        }channelN0[16],
        channelN1[16];

    }block[12];

    int timeStamp;

    uchar returnType;

    uchar lidarType;

}LidarPayload;
```