# Modifying CARLA Simulator's Radar Sensor Model and Implementing DBSCAN for Data Labelling and Machine Learning Algorithms

**ANANYA REGINALD FREDERICK**

**7207003**

Supervisor: **Prof. Dr. Andreas Becker**

Research Thesis

Master of Engineering

Faculty of Information Technology

Fachhochschule Dortmund

Germany

# Contents

# 1. TABLE OF FIGURES

# 2. INTRODUCTION

One of the primary requirements of autonomous driving is a reliable perception of the environment. This is generally provided by highly accurate sensors like Lidar, Radar, and Camera. Also, the data is often provided as 3D point clouds. We can effectively derive important information from these point clouds using different algorithms.

Point Clouds are just datasets of points that represent objects or space in 3D format. Each point in the dataset comprises X, Y, and Z geometric coordinates. They are mostly generated using a LIDAR. Radar gives us the 3D point cloud and another important data: range rate. Point clouds are easier to edit, display, and filter as compared to range-doppler or range-azimuth maps. Also, they are a non-intrusive way to measure an object or its properties. Buildings and sites don't need to be shut down to be measured.

Since LIDARS emit light in all directions by rotating 360 degrees, more points are reflected, resulting in a dense collection of points as opposed to using Radars. The scenes reconstructed from these point clouds are more detailed due to the density of the point clouds.

Radar point clouds, on the other hand, are generally sparse. They are generated by radio waves reflecting from the surface of buildings or objects. However, when it comes to dim lighting and weather conditions, they have an advantage over Cameras and LIDARs. A combination of these three sensors is beneficial for Autonomous Driving.

Any AI neural network requires a lot of labeled data to train the model and work. Also, data for innumerable scenarios and test cases are required. Some of these scenarios are risky to carry out in real life, for example, scenarios in which the car needs to get into uncomfortable proximity to a person or child. Moreover, the sheer volume of data to train a model is difficult and expensive to collect. This is where simulators come to the rescue by providing the ability to create complex and varied scenarios without any threat to human life.

Carla is one such simulator that is open source. It gives us access to unlimited Radar, Lidar, and Camera data for any required scenario. However, there is a drawback with the current Radar in Carla. It does not provide labeled data, which, as mentioned above is required for Machine learning Algorithms. So, the goal is to build Carla and modify the Radar source code so that it returns labeled data that can be used for various research and development projects.

# 3. OBJECTIVES

The project's goal is to modify the Radar code so that it returns the ID of the objects being detected by the Radar. With this, there will be access to labeled data which can be used to train various machine-learning models and applications. After this, different scenarios will be created so that radar data can be collected, moving objects can be filtered, and clustered.

The main tasks that were carried out were:

1. Build Carla and Unreal engine on Windows and Linux PC.
2. Research how to create/make changes to a sensor in Carla and understand the Carla Pipeline.
3. Performing an analysis of radar and lidar code written in C++ and subsequently adding a feature to the radar code that returns labeled radar data.
4. Creating scenarios and collecting simulated Radar data.
5. Filtering stationary and moving objects.
6. Performing clustering of the data using DBSCAN

# 4. OVERVIEW OF CARLA SIMULATOR

Carla (Car Learning to Act) is an open-source simulator that has been developed from scratch to support and facilitate training, development, and validation of open autonomous driving systems. Being open source, it gives free access to digital assets and has sensor models that can be used to create any scenario that is required.

The Carla simulator has a client-server architecture, where the server-side deals with all the simulation-related functions like physics computation, actors, sensor visualization, etc. On the other hand, the client side is made of modules that control the logic of actors on the scene and set the world condition. It is crucial to have a dedicated GPU as Carla aims to make the simulation as real as possible [1].

As the current goal of the project is to modify an existing sensor model, it is necessary to build the Carla and Unreal Engine source code.

# 5. BUILDING CARLA

Building Carla from the source code is necessary if there needs to be any further development. In the current project, additional features are to be added to the Radar sensor model. Carla can be built in both Linux and Windows environments, and it is built in both to check the viability. The Carla versions built are 0.9.10, 0.9.11, and 0.9.12. The most stable version found was 0.9.11.

## 5.1 WINDOWS AND LINUX BUILD

The detailed process for building Carla on both Windows and Linux systems is comprehensively outlined in the Appendix of the Thesis report. Here, you will find a thorough breakdown of the required software components and a step-by-step guide for the entire build process. Chapters 14.1 and 14.2 (Appendix) delve into the specifics of Windows and Linux, respectively.

### 5.1.1 Starting the Simulation

To control the simulation using the Python scripts it is necessary to ‘Play’ the simulation. This can be seen in Figure 5.2.
After this any example, preferably, dynamic weather, as it has obvious changes to the environment, can be used to check if the simulation is working or not.

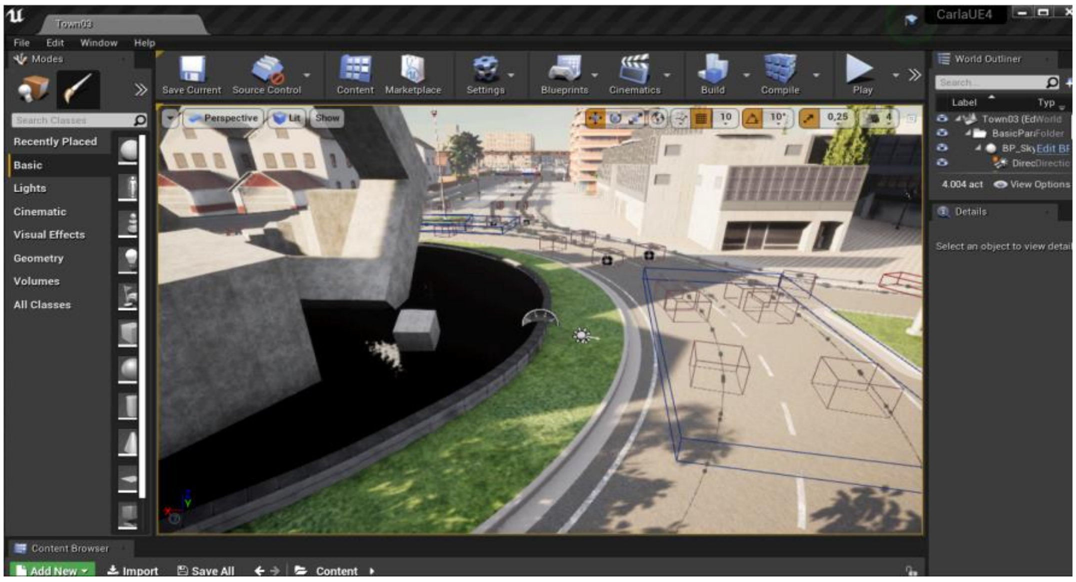
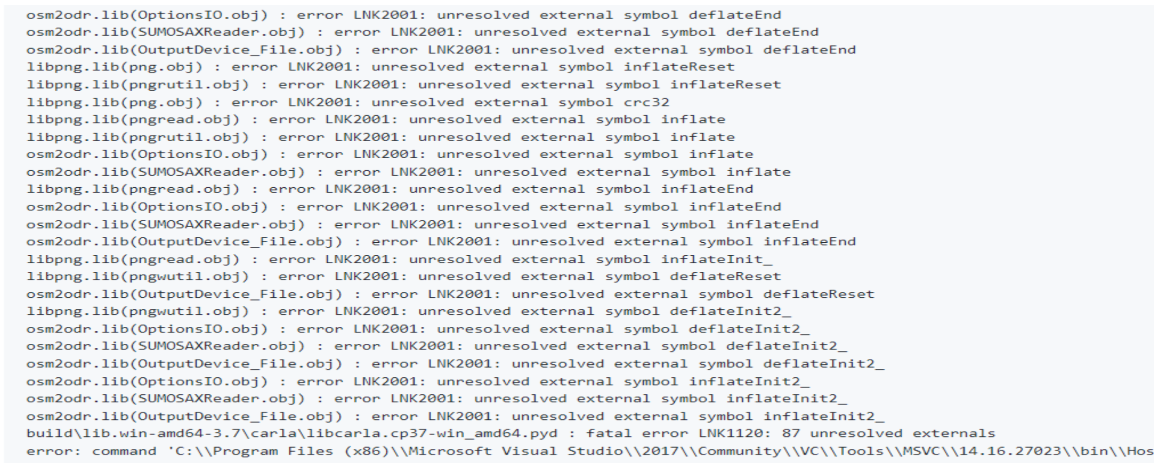


Figure 5.1 The Carla Interface after a successful ‘make launch’ command.

## 5.2 Issues faced with Windows build.

### 1. LNK2001: unresolved external symbol



```
osm2odr.lib(OptionsIO.obj) : error LNK2001: unresolved external symbol deflateEnd
osm2odr.lib(SUMOSAXReader.obj) : error LNK2001: unresolved external symbol deflateEnd
osm2odr.lib(OutputDevice_File.obj) : error LNK2001: unresolved external symbol deflateEnd
libpng.lib(png.obj) : error LNK2001: unresolved external symbol inflateReset
libpng.lib(pngrutil.obj) : error LNK2001: unresolved external symbol inflateReset
libpng.lib(png.obj) : error LNK2001: unresolved external symbol crc32
libpng.lib(pngread.obj) : error LNK2001: unresolved external symbol inflate
libpng.lib(pngrutil.obj) : error LNK2001: unresolved external symbol inflate
osm2odr.lib(OptionsIO.obj) : error LNK2001: unresolved external symbol inflate
osm2odr.lib(SUMOSAXReader.obj) : error LNK2001: unresolved external symbol inflate
libpng.lib(pngread.obj) : error LNK2001: unresolved external symbol inflateEnd
osm2odr.lib(OptionsIO.obj) : error LNK2001: unresolved external symbol inflateEnd
osm2odr.lib(SUMOSAXReader.obj) : error LNK2001: unresolved external symbol inflateEnd
osm2odr.lib(OutputDevice_File.obj) : error LNK2001: unresolved external symbol inflateEnd
libpng.lib(pngread.obj) : error LNK2001: unresolved external symbol inflateInit_
libpng.lib(pngwutil.obj) : error LNK2001: unresolved external symbol deflateReset
osm2odr.lib(OutputDevice_File.obj) : error LNK2001: unresolved external symbol deflateReset
libpng.lib(pngwutil.obj) : error LNK2001: unresolved external symbol deflateInit2_
osm2odr.lib(OptionsIO.obj) : error LNK2001: unresolved external symbol deflateInit2_
osm2odr.lib(SUMOSAXReader.obj) : error LNK2001: unresolved external symbol deflateInit2_
osm2odr.lib(OutputDevice_File.obj) : error LNK2001: unresolved external symbol deflateInit2_
osm2odr.lib(OptionsIO.obj) : error LNK2001: unresolved external symbol inflateInit2_
osm2odr.lib(SUMOSAXReader.obj) : error LNK2001: unresolved external symbol inflateInit2_
osm2odr.lib(OutputDevice_File.obj) : error LNK2001: unresolved external symbol inflateInit2_
build\lib.win-amd64-3.7\carla\libcarla.cp37-win_amd64.pyd : fatal error LNK1120: 87 unresolved externals
error: command 'C:\\Program Files (x86)\\Microsoft Visual Studio\\2017\\Community\\VC\\Tools\\MSVC\\14.16.27023\\bin\\Hos
```

Figure 5.2 Figure showing the issue, LNK2001: unresolved external symbol
**Image source**: Carla GitHub Issues *[4]*

This is one of the trickiest issues found during building Carla. As this can occur for various reasons and depends on the system being used at the time, it takes many trials to get it right.

1. Reasons for this issue:
   - Multiple Python versions installed on the local system.
   - Wrong version of Python or other software.
2. Solution
   - Uninstall other versions of Python.
   - Install only the correct version of Python. (in the case of Carla 0.9.11, the most stable python version is python 3.8.8)
   - Restore the PC to default factory settings and install all software from scratch. This solution worked exceptionally well in the current project.

### 1. RuntimeError: time-out of 2000ms while waiting for the simulator.

This error occurs if the play button is not pressed in the Unreal Engine. The simulation environment is not activated and hence the scripts cannot find a simulator to run on.
Also, If the PC doesn't have a good enough GPU or RAM that's required for a particular version of Carla then it might take longer than 2000ms for the simulator to load. In such a case, increasing the time from 2000ms can solve these issues [5].

### 2. Module 'Carla' has no attribute 'Client'.

The reason for this is due to an issue with the latest version of the setup-tools library in Python 3 that builds the .egg files.

The issue can be resolved by using an earlier version of the library. In the above case, the command ‘pip3 install -Iv setuptools==47.3.1’ was used to install the version of ‘setuptools’ that would solve the error [6].

## 5.3 Conclusion

There were relatively few issues faced while compiling Carla in Linux OS. Comparatively, it was observed that the number of issues while compiling in Windows OS was just too many. Also, building Carla on Windows is extremely version-oriented. Therefore, any mismatch in version results in errors that cannot be resolved easily. Hence, Carla should be built on a Linux system to avoid these issues.

# 6. SENSOR ADDITION/MODIFICATION IN CARLA

The project aims to extract more comprehensive data from the radar sensor in Carla. To achieve this, it is necessary to understand how sensors work and the changes that are required to be done in the code. Understanding the process of sensor creation offers clarity in workflow, facilitating radar sensor modification.

In Carla, sensors are a special type of actor that produces a stream of data. This stream of data can be produced continuously every time the sensor is updated or in the form of interrupts that only produce data after specific events. For example, a Radar sensor produces point clouds on every update, but a collision sensor only returns data when a collision happens.

There are 2 types of sensors in Carla: Client-side sensors and Server-side sensors. The Client-side sensors don't need to interact with the Unreal Engine (Server-side). The best example is the LaneInvasion sensor, which notifies every time a lane mark has been crossed. Whereas the Server-side sensors run inside the UE4 and send data back to the Python Client. To do this they need to cover the whole communication pipeline which is shown in Figure 6.1***.*** The creation of a server-side sensor is what is required in this case.

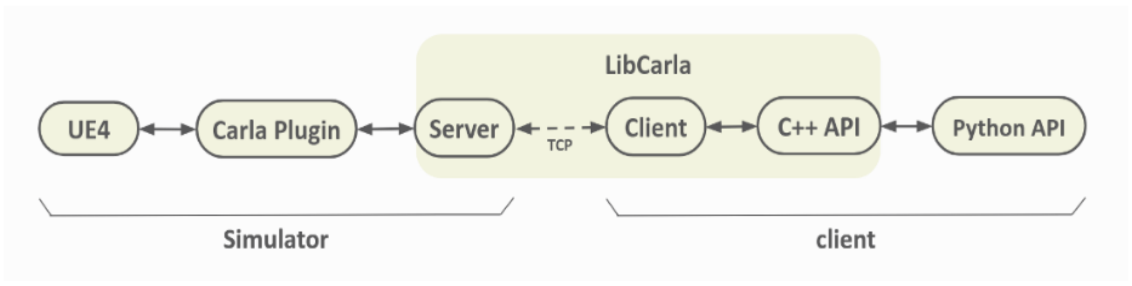


Figure 6.1 Communication pipeline of Carla Simulator. **Image source**: Carla's official Website *[7]*.

The major changes that need to be done in the pipeline can be divided into 4 major parts.

1. Sensor Actor
   The sensor actor belongs to the 'AActor' class as it is a special actor that is used to measure or simulate data. The user can have access to it in the form of a Sensor actor.
   - It is a part of the UE4 framework which is the server side.
   - As discussed above, the sensor actor is a special actor and therefore, it derives from the 'AActor' class (which is available in UE4).

- The ‘AActor’ class has a virtual method called Tick which can be used to update the sensor on every update from the simulator.
- Since an actor is any object that can be dropped into the world, the sensor also inherits this property.
- The code (.cpp and .h) files that govern the sensor are found in the path mentioned below. If a new sensor is to be created the .cpp and .h files need to be created in this path. Since, the goal here is to modify the Radar sensor, the path to the radar sensor model files is given below.
  *‘Unreal/CarlaUE4/Plugins/Carla/Source/Carla/Sensor/Radar.h’*
- The constructor can be used to create the trigger box. The tick can be enabled or disabled here. For this project, the tick is required.
- Carla needs to be told what attributes the sensor has. This is done in the ‘GetSensorDefinition’ function. The function is defined in ‘ActorBlueprintFunctionLibrary.cpp’ whose location is provided below.
  ‘*Unreal\CarlaUE4\Plugins\Carla\Source\Carla\Actor\ActorBlueprintFunctionLibrary.cpp*’
  These attributes are required to spawn the sensor. For example, some of the attributes of radar are horizontal field of view, vertical field of view, range, etc.
- Set Function: It is used to create a sensor on user demand. As soon as the sensor is connected, the set function is called with the parameters requested by the user.
- The Tick function is now used, and data is sent back to the client,

Every sensor has a stream that is used to send data to the client. While using ‘sensor.listen(callback function)’, this stream is what it is subscribed to. The callback function gets triggered every time on the user side when some data is sent. However, before this data can be accessed it needs to be serialized [7].

2. Serializer
   It has functions that are used to serialize and deserialize the data generated by the sensor. It runs in the LibCarla which connects both the server and client.
   - These are 2 files that belong to the LibCarla part of the pipeline. It consists of 2 static methods, Serialize and Deserialize. Below are the 2 files and their location.
     ‘*LibCarla/source/carla/sensor/s11n/SafeDistanceSerializer.h*’
     ‘*LibCarla/source/carla/sensor/s11n/SafeDistanceSerializer.cpp*’
   - Serialize function: This function has 2 arguments one of which is the sensor, and the other is a return buffer. The function gets the arguments that are passed to the ‘Stream.Send(…)’ function. The return buffer is ‘carla::Buffer’ which is just a dynamically allocated piece of raw memory. This is used to send raw data to the client.

- Deserialize function: The next step is to deal with the buffer that is coming to us from the serialize function, but this time it is on the client side. This data is deserialized and packed into a ‘RawData’ object.

  But before this can happen an event needs to be defined so that this can be executed. It is done in the next stage i.e. Sensor data object.

3. Sensor Data
   It stores the data generated by the sensor. This data will be sent to the final user. To accomplish this, a data object is created which represents the data of the sensor.
   ‘*LibCarla/source/carla/sensor/data/RadarData.h*’
   In the case of radar, the buffer created in ‘Serialize’ will be interpreted as a structure containing information about range, azimuth, altitude, and radial velocity. The goal is to add one more variable to be returned.

4. Register the Sensor
   The pipeline has been completed. This sensor can now be registered. This is done in the file ‘*LibCarla/source/carla/sensor/SensorRegistry.h*’. After this, the data can be dispatched to the corresponding serializer.

# 7. MODIFYING RADAR SENSOR MODEL

Carla mimics a Lidar or Radar sensor by using Ray-casting to get point clouds of the surrounding environment. The Lidar sensor already has the feature in which it returns the object IDs, so it was analyzed to get an understanding of how to proceed with getting the additional information from the Carla radar sensor model.

## 7.1 Understanding The Lidar Sensor Model

As mentioned above, the Lidar sensor in Carla uses the technique of 'raycasting' to simulate behavior like real-world Lidar. Raycasting is a method in which virtual rays are cast from the Lidar sensor into the environment to measure the distance to the objects. Each of these rays covers a designated field of view and each ray moves in a specific direction. Whenever a ray intersects an object, the intersection point and the required properties of the object are recorded. After this Carla assigns semantic labels to the detected objects. Semantic labeling is the process in which Carla assigns specific classifications to roads, sidewalks, pedestrians, etc. These labels provide contextual information about the objects detected by the Lidar.
Using the data collected by the intersection points and the semantic labeling associated with them, Carla can generate a point cloud representation of the scene. Each point cloud represents a location in 3D space and includes information such as intensity, position, range, and semantic label. This data is then sent via the client-server pipeline of Carla to be accessed by the user.

Figure 7.1 shown below depicts the functional overview of the Ray cast semantic lidar code.
A detailed working of the Lidar code will be discussed below and the part of the code that deals with extracting the object ID of the objects intersected by the lidar rays will be analyzed so that the same feature can be applied to radars.

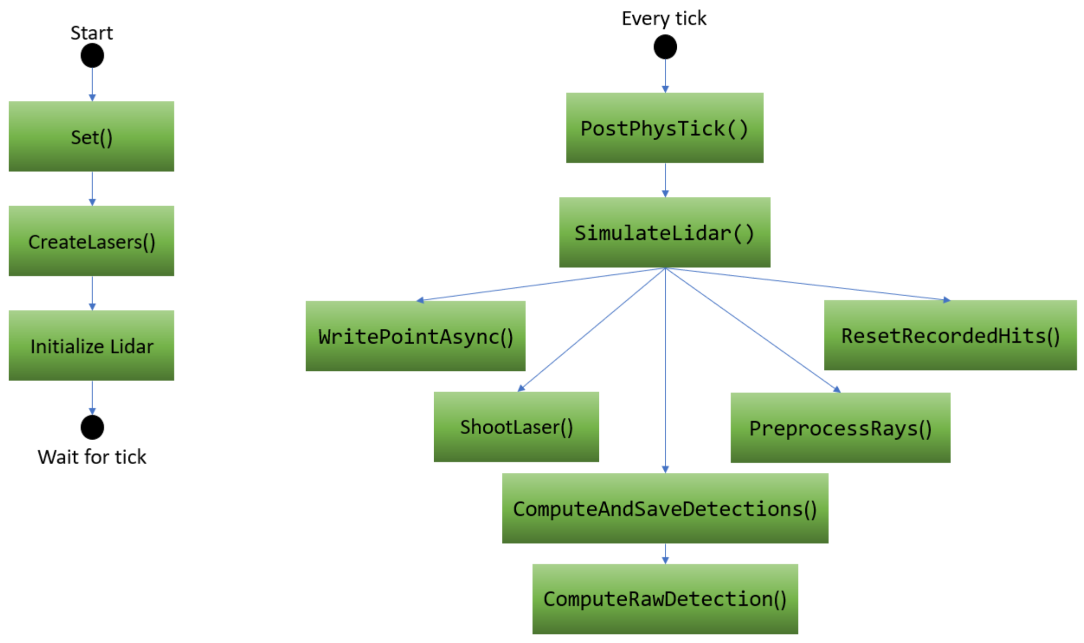


Figure 7.1 Flowchart showing the working of the Carla Semantic Lidar Sensor

Let's go through the code and explain the purpose of each function:

1. GetSensorDefinition():
   This function returns the definition of the LIDAR sensor as a ‘FactorDefinition’ object. It is used to define the sensor in the Carla simulator.

2. ARayCastSemanticLidar():
   This is a constructor that sets the primary actor tick to be enabled. This enables the tick() function to be called on every frame.

3. Set():
   This function sets the properties of the LIDAR sensor based on the provided FLidarDescription. It is called the CreateLaser() function which creates the lasers for the sensor based on the number of channels specified and initializes some data structures.

4. CreateLasers():
   This function creates the lasers for the LIDAR sensor based on the number of channels specified in the LIDAR description. It calculates the vertical angles for each laser based on the upper and lower field of view limits and stores these points to be used later by the SimulateLidar() function. With this function, Initializing the Lidar is done, and the code waits for the physics tick of the world which then calls the PostPhysTick() function.

5. PostPhysTick():
   This function is called after the physics tick of the world. It calls the SimulateLidar() function which is responsible for simulating the LIDAR sensor and sending the sensor data to the data stream.

6. SimulateLidar():
   This function performs the LIDAR simulation for the given delta time. It does so by casting rays from each laser, detecting hits with objects, recording the hits, and generating semantic detections based on the recorded hits. It also handles the synchronization and parallelization of the lidar simulation process. This is a brief overview of how it works:
   - Based on the points per second mentioned in the Lidar attributes and the frame’s delta time, it calculates the number of points to scan with each laser in that particular frame. This ensures that there is a consistent number of points per frame being generated regardless of frame rate.
   - If the number of points to scan with one laser is zero or negative then it means that no points were requested in that frame, and in this case, a warning log message is printed.
   - It gets the current horizontal angle of the lidar and converts it to degrees to represent the lidar's rotation around the vertical axis. It resets the recorded hits so that new lidar detection results can be stored.
   - After all this information is collected the lidar is ready to shoot traces and record detections. To do this parallel processing is used to iterate over each channel and each point to scan with one laser. Using parallel processing distributes the work across multiple threads to enable efficient computation.
   - The ‘HitResult’ object stores the result of the ray cast done in the shoot laser function. If the preprocess condition is ‘true’ and the laser successfully hits an object, the details about the lidar point stored in the ‘HitResult’ object are written asynchronously to the recorded hits vector.

   It calls functions ResetRecordedHits(), PreprocessRays(), WritePointAsync(), ComputeAndSaveDetections(), and ShootLaser() functions to complete the above tasks. These functions will be discussed in detail below.

7. ShootLaser():
   In Semantic Raycast Lidar, the ShootLaser function casts a trace from the lidar body to simulate the laser's detection. It returns “true” if the trace hits an object.
   Here’s a brief overview of the function which demonstrates its working.
   - Create an object named ‘HitInfo’ of the class ‘FHitResult’ which will store the hit results of the trace.

- Retrieve the location and rotation of the Lidar body. This will help in determining the location and orientation of the lidar so that the traces can be shot in the right direction.
- Use the range from the lidar's 'Description' and the forward vector of the laser rotation to calculate the endpoint of the traces.
- With the above information, the traces can be shot and if the trace hits an object, the information is stored and the function returns 'true'.
- If the trace doesn't hit an object the 'HitInfo' is not stored and the function returns 'false'.

8. ResetRecordedHits():
   This function resets the recorded hits data structure to prepare for storing new hits. It resizes the data structure based on the number of channels and the maximum points per channel.

9. PreprocessRays():
   This function initializes the preprocess conditions for the rays. It sets all the conditions to true initially.

10. WritePointAsync():
    This function asynchronously writes a hit point to the recorded hits data structure for a specific channel.

11. ComputeAndSaveDetections():
    This function computes and saves the detections based on the recorded hits. It iterates over the channels and hits, computes the raw detections, and writes them to the semantic LIDAR data structure.

12. ComputeRawDetection():
    This function computes the raw detection based on the hit information and sensor transformation. It calculates the detection point, incident angle, and object information for the hit.

## 7.2 Understanding The Radar Sensor Model

In the real world, radar sensors work by emitting radio waves and measuring the reflections or echoes from objects in the environment. Using these reflections, the information about the distance, position of the detected objects, and relative velocity can be determined.

In the Carla simulator, this behavior is imitated by using ray casting. This means that traces are shot from the position where the radar is positioned in the environment and the details of the objects being hit by the traces are stored as detections. As discussed above, this is also like the working of the Lidar.

Here's an overview of how the Radar sensor works in CARLA:
The radar sensor is used by placing the radar in the Carla virtual environment with the help of Python APIs that are available in Carla. The radar can be attached to a vehicle or just be placed in the environment as per the requirements. Various attributes that are specific to the radar sensor need to be defined here. These include field of view, points per second, range, and sensor tick which define the horizontal and vertical field of view in degrees, points generated by all lasers per second, maximum distance to ray cast in meters, and simulation seconds between sensor ticks respectively.

To replicate the predefined pattern of the radio waves in the field of view, the simulation performs line traces from the radar sensor location to the maximum detection range. The number of rays emitted per second is determined by the “points per second” attribute that we initially defined while initializing the sensor.

These traces check for collision with objects in the virtual world. If a ray hits an object, then a blocking hit or collision is detected. For each successful collision that is detected by the radar sensor, the hit result is then stored data structure and sent to a data stream to be accessed by the user. Using the listen() function this data stream can be accessed and the detections from the radar can be collected. Each detection point has its azimuth, elevation, depth, and radial velocity.

After implementing the changes to the radar sensor model code, the Object ID will also be available. This will help with getting labeled data which is useful for various Machine Learning models.

By simulating the behavior of real-world radar sensors, the CARLA Radar sensor provides crucial information for perception and object detection algorithms in autonomous driving systems. The collected radar data can be used for various purposes, such as object tracking, collision avoidance, and scene understanding.

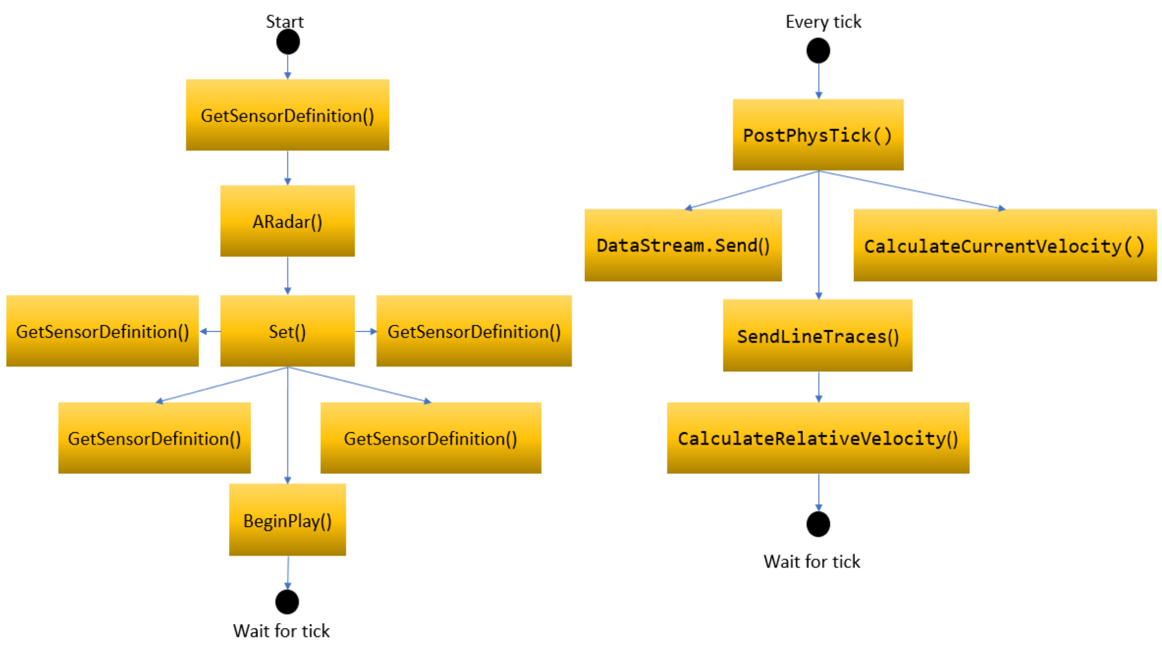


Figure 7.2 Flowchart showing the working of the Carla Radar Sensor

The flowchart in Figure 7.2 above shows the major functions of the radar sensor model and the order in which they are called. The functions are described in detail below.

1. GetSensorDefinition():
   This function returns the definition of the Radar sensor as a ‘FactorDefinition’. It calls the function ‘MakeRadarDefinition’ defined within ActorBlueprintFuntionLibrary.cpp. This function is used to define the sensor’s properties and uses default values to initialize the sensor attributes. By invoking the 'MakeRadarDefinition' function, the sensor attributes are systematically defined, adhering to professional standards. These values can be subsequently customized to align with specific user requirements.

2. ARadar():
   This is the constructor of the ‘ARadar’ class which is used to enable ticking by setting ‘PrimaryActorTick.bCanEverTick’ to true.

3. Set():
   The Set() function calls the SetRadar() function defined in ActorBlueprintFunctionLibrary.cpp and sends it the attributes specified by the user. The SetRadar() function assigns these values to the attributes by using the functions SetHorizontalFOV(), SetVerticalFOV(), SetRange(), and SetPointsPerSecond().

4. SetHorizontalFOV():
   This function sets the horizontal field of view (FOV) of the Radar sensor.

5. SetVerticalFOV()
   This function sets the vertical FOV of the Radar sensor.

6. SetRange():
   This function sets the maximum detection range of the Radar sensor.

7. SetPointsPerSecond():
   This function sets the number of radar points or rays emitted per second.

8. BeginPlay():
   This function is called when the Radar sensor actor begins play. It initializes the ‘PrevLocation’ member variable with the initial location of the actor.

9. PostPhysTick():
   The function is called after every physics tick of the world.
   - It calls CalculateCurrentVelocity() which returns the current velocity of the radar.
   - The data structure used to store detections of the previous frame is reset.
   - Calls the SendLineTraces() function which simulates the entire radar. It shall be explored in detail later.
   - Lastly, it sends collected radar data through the data stream.

10. CalculateCurrentVelocity(const float DeltaTime):
    The function calculates the current velocity of the Radar sensor based on its location changes. It uses the previous and current location of the sensor to estimate the velocity.

11. SendLineTraces(float DeltaTime):
    - It takes ‘DeltaTime’ as a parameter so that it can shoot traces and gather detections for this period.
    - The max radar radius in the horizontal and vertical direction is calculated based on the initial attributes given by the user, i.e., horizontal and vertical field of view (FOV) angles and range of the radar.
    - The number of traces to be generated for raycasting is calculated by multiplying the delta time and points per second and then converting it to an integer.
    - A critical section (‘Mutex’) is acquired to ensure thread safety. Read access is locked in the physics scene.
    - A parallel loop is run for all the rays that must be traced.
    - The end location of each ray is determined based on factors such as the radar's position, rotation, range, and maximum radii in the horizontal and vertical directions.

- The LineTraceSingleByChannel() function performs a line trace from the radar location to the end location and stores the result of the hit status and actor.
- If the trace hits an object the value is set to 'true'.
- The critical section is now unlocked for read access.
- Finally, a loop iterates through the stored ray values, and only the rays that hit an object are retained as detections.

12. CalculateRelativeVelocity():
    This function uses the velocity of the vehicle with the radar sensor and the target object to calculate the relative velocity.

As described above, the ShootLaser() function shoots traces into the virtual environment to collect detections. It does this by using the LineTraceSingleByChannel() function that's available in Unreal Engine. This function returns 'true' if there is a blocking hit.

The first parameter sent by this function named (in this case) 'HitInfo', is of data type 'struct FHitResult', which has a function 'GetActor()' that can return the actor which owns the component that was hit by the trace.

Once, the information of all the detection points has been collected, it is passed back to the SimulateLaser() function, which then calls ComputeAndSaveDetections(). ComputeAndSaveDetections() runs a loop going through the detections. It then sends each detection as a parameter to ComputeRawDetection() which uses the function GetActor() to extract the name of the actor hit by the trace. This name is then run through the Carla directory to ascertain the Object ID associated with the actor's name.

The object ID is now saved in a variable called 'detection' which is of type 'FSemanticDetection'. 'FSemanticDetection' is an alias of the class 'SemanticLidarDetection' whose member variables are used to store the location, angle of incidence of the rays, object ID, and Object tag of the actor hit by the trace.

## 7.3 Modifying Radar Sensor Model

In the above sections, the functionality of the radar and lidar sensors has been described. In this section, the modification to the radar sensor code will be covered along with where the changes need to be made to the entire Carla pipeline. This information will form the backbone of modifying the existing sensors in Carla to add features and tailor them to specific research requirements or use cases. In this case, getting labeled data from the radar sensor can be used for a variety of other machine-learning algorithms.

### 7.3.1 Modifying radar features (radar.cpp)

The moment a trace hits the object in the environment, data about the object gets stored in the 'LineTraceSingleByChannel' function and details can be extracted for further use. This function is called in the 'SendLineTraces' method.

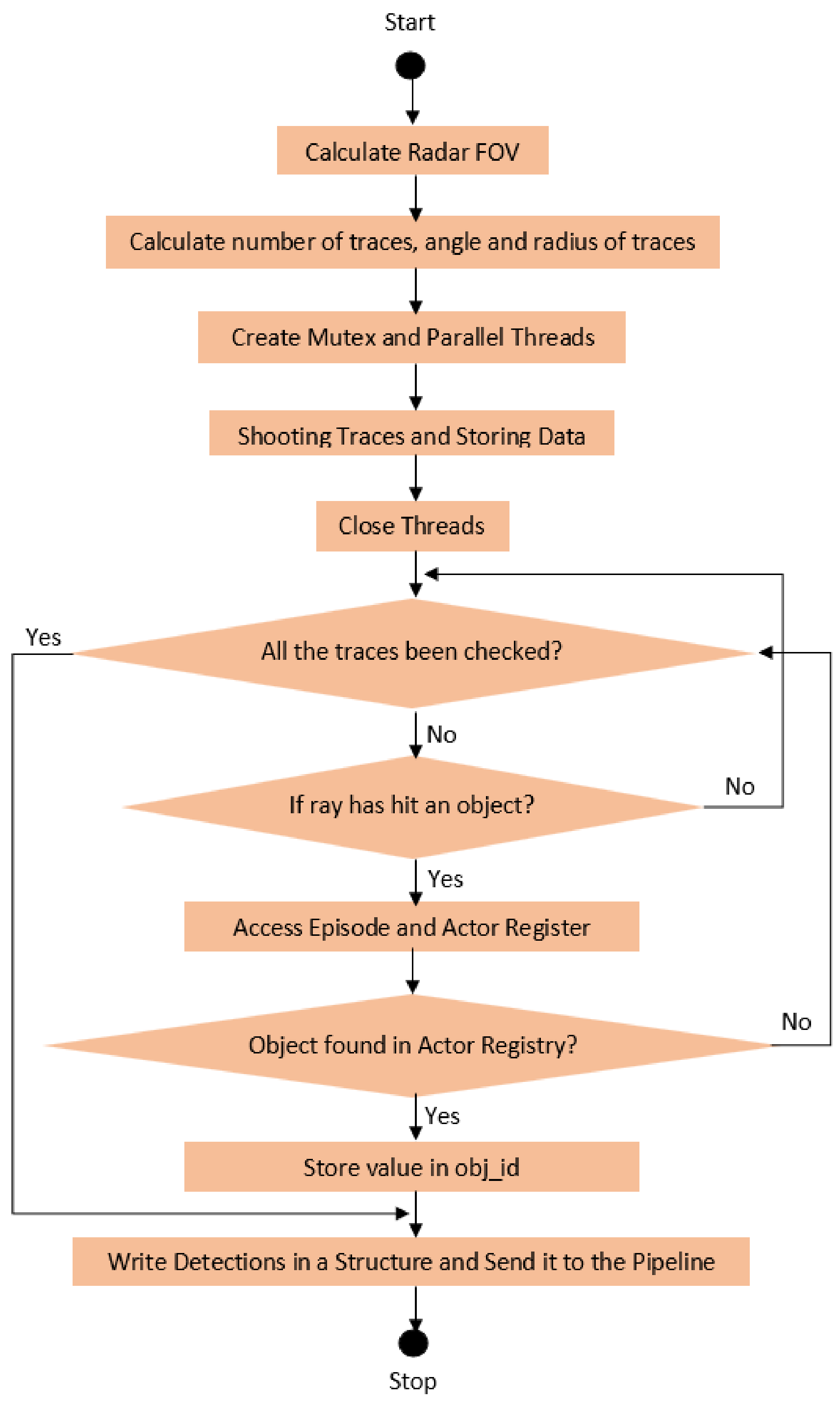
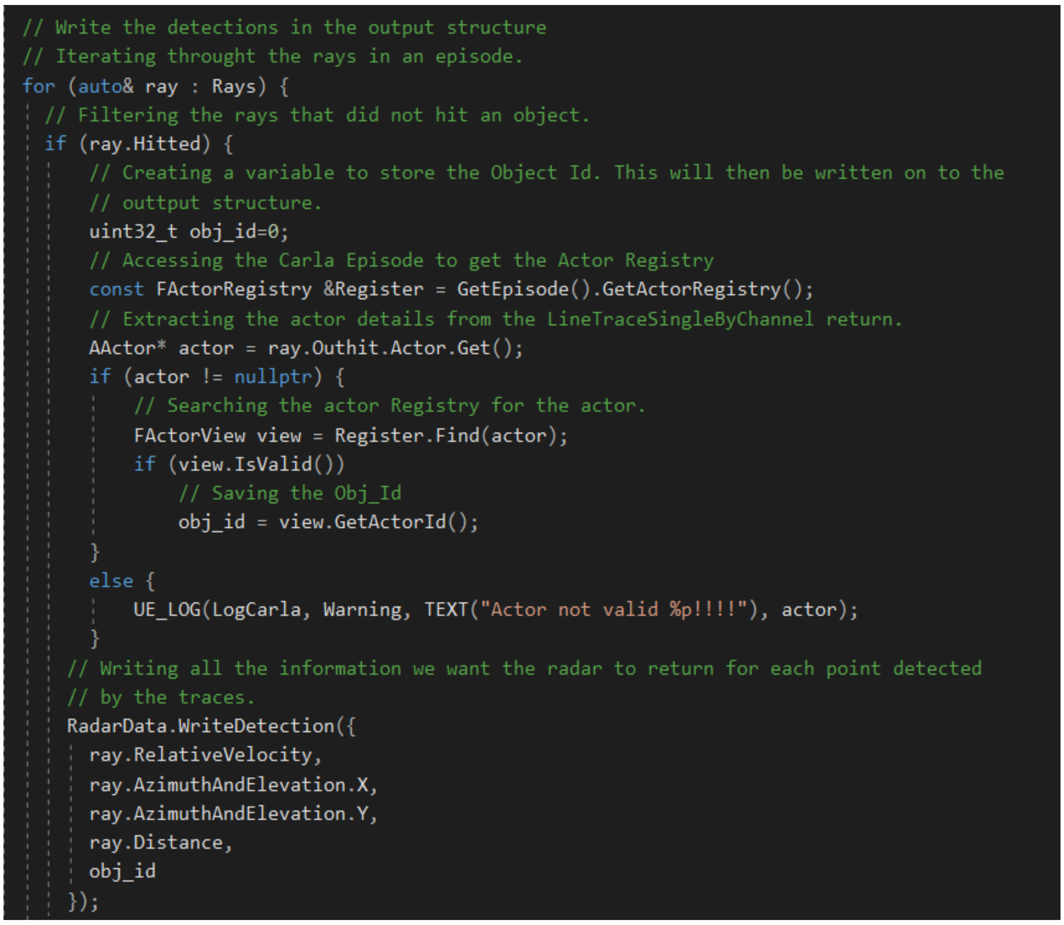

Start
Calculate Radar FOV
Calculate number of traces, angle and radius of traces
Create Mutex and Parallel Threads
Shooting Traces and Storing Data
Close Threads
All the traces been checked?
Yes
No
If ray has hit an object?
No
Yes
Access Episode and Actor Register
Object found in Actor Registry?
No
Yes
Store value in obj_id
Write Detections in a Structure and Send it to the Pipeline
Stop

Figure 7.3 Flowchart of the SendLineTraces() Function.

As the name suggests, the ‘SendLineTraces’ function deals with shooting traces from a small area, outward in multiple directions to mimic the electromagnetic pulse which gets sent out by a radar sensor. This is done by using a different thread for each trace. A mutex (critical section) is also used to ensure thread safety. As read access is locked here, the code modification to add any features that involve extracting actor or episode details shouldn’t be done in this loop. Doing so will result in the simulator crashing without a significantly helpful error message. Once the threads have finished executing and the critical section is free the ’Rays’ array will hold information of all the traces shot by the simulator. This is where the code to add the feature to the sensor will be implemented.

The logic of the ‘SendLineTrace’ function can be seen in Figure 7.3. The code after the ‘Close Threads’ block has been modified. This flowchart shows the logic of the added feature.



```
// Write the detections in the output structure
// Iterating throught the rays in an episode.
for (auto& ray : Rays) {
  // Filtering the rays that did not hit an object.
  if (ray.Hitted) {
      // Creating a variable to store the Object Id. This will then be written on to the
      // outtput structure.
      uint32_t obj_id=0;
      // Accessing the Carla Episode to get the Actor Registry
      const FActorRegistry &Register = GetEpisode().GetActorRegistry();
      // Extracting the actor details from the LineTraceSingleByChannel return.
      AActor* actor = ray.OutHit.Actor.Get();
      if (actor != nullptr) {
          // Searching the actor Registry for the actor.
          FActorView view = Register.Find(actor);
          if (view.IsValid())
              // Saving the Obj_Id
              obj_id = view.GetActorId();
      }
      else {
          UE_LOG(LogCarla, Warning, TEXT("Actor not valid %p!!!!"), actor);
      }
    // Writing all the information we want the radar to return for each point detected
    // by the traces.
    RadarData.WriteDetection({
      ray.RelativeVelocity,
      ray.AzimuthAndElevation.X,
      ray.AzimuthAndElevation.Y,
      ray.Distance,
      obj_id
    });
```

Figure 7.4 The modified code *[8]* extracts and returns the Object ID.

For a ray to be relevant it needs to have collided with an object. The ‘LineTraceSingleByChannel’ function, which is used to create the rays, returns a Boolean value. This will be true if the ray has hit an object. Using this, a check is placed to keep information of only the rays with collision.

Now, the actor details must be extracted from the information stored in the ‘Rays’ array. Individual actor details are stored in the Actor registry which is a part of the Episode.

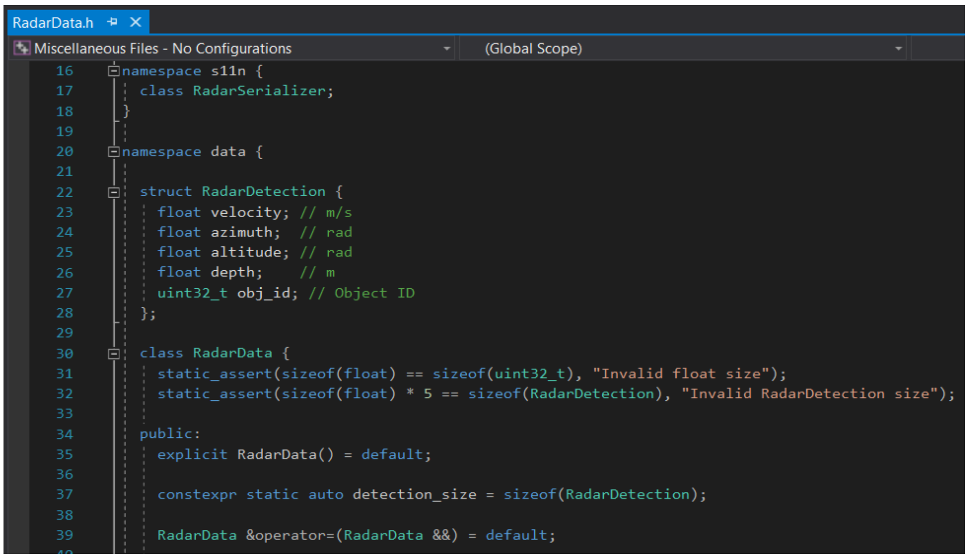
In the Carla simulator, an episode refers to a single run or a discrete run of a simulator scenario each containing a specific scenario or task within the CARLA environment. Every object in Carla has an Episode [9]. During an Episode, various entities such as vehicles, pedestrians, and environmental elements interact based on predefined rules and behavior. Episodes serve as important tools for evaluating and testing algorithms, control strategies, and autonomous driving systems within Carla. It holds all the information for a scenario for that instance of time and is responsible for various aspects in the duration of the Episode. It takes care of the initialization in which it sets up the environment including the configuration of the vehicles, pedestrians, and other objects. During an Episode execution, it handles the parameters and rules of the simulated scenario like the interaction between actors and the world. Data collection takes place simultaneously throughout the episode like sensor readings, vehicle telemetry, and simulation metrics to facilitate analysis and evaluation. Finally, when the Episode concludes, the data is saved, and the simulation environment is reset to prepare for the upcoming episodes.

Therefore, to get the object information, the actor registry in the Carla episode is extracted. The actor information acquired by the ‘LineTraceSingleByChannel’ is matched against the information in the actor registry and the Object ID of the matching actor is stored to return to the User. The code implementation can be seen in Figure 7.4. At this point, the object ID has been extracted and to make it accessible to the user, the data must be provided a path through the pipeline connecting the user to the Carla simulator. This will be described in the next section.

### 7.3.2 Modifying the Carla Pipeline

As shown by Figure 6.1, for a sensor feature to be successfully implemented, the path of the return data from the sensor should be established through the entire Carla Pipeline. The changes made in the above section were only in the simulator end of the pipeline. Files in the Libcarla and PythonAPI folder also need to be changed so that the simulator knows the changes in the structure storing the detections and establishes enough space to transfer it through the TCP protocol.

**Libcarla:**

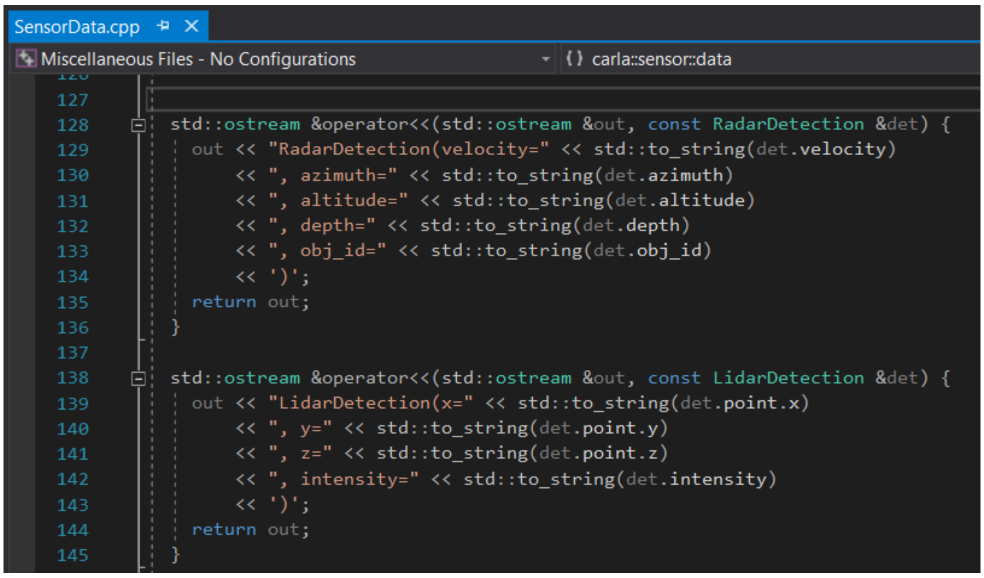


```
namespace s11n {
  class RadarSerializer;
}

namespace data {

  struct RadarDetection {
    float velocity; // m/s
    float azimuth;  // rad
    float altitude; // rad
    float depth;    // m
    uint32_t obj_id; // Object ID
  };

  class RadarData {
    static_assert(sizeof(float) == sizeof(uint32_t), "Invalid float size");
    static_assert(sizeof(float) * 5 == sizeof(RadarDetection), "Invalid RadarDetection size");

  public:
    explicit RadarData() = default;

    constexpr static auto detection_size = sizeof(RadarDetection);

    RadarData &operator=(RadarData &&) = default;
```

Figure 7.5 Structure containing radar detection components. **Image source**: Carla Simulator code *[8]* with modification to include 'obj-id'.

The 'RadarData.h', file, as shown in Figure 7.5., has the data object for the radar sensor. Until now, the data structure had 4 values, i.e. azimuth, elevation, relative velocity, and range. The 'obj_id' variable needs to be added to this structure and the size of the new structure needs to be updated to include the new variable. This structure template is going to reinterpret the buffer that was created in the Serialize method. The 'serialize' method will now accept variables velocity, azimuth, elevation, range, and Object ID.

**PythonAPI:**



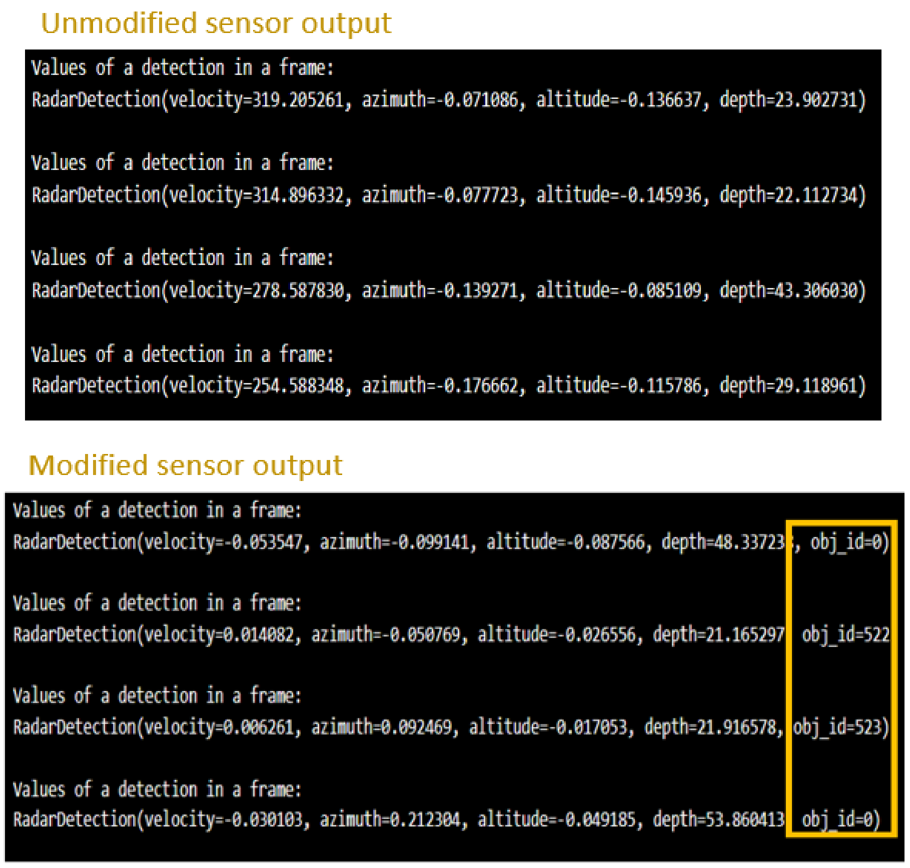
```
std::ostream &operator<<(std::ostream &out, const RadarDetection &det) {
  out << "RadarDetection(velocity=" << std::to_string(det.velocity)
      << ", azimuth=" << std::to_string(det.azimuth)
      << ", altitude=" << std::to_string(det.altitude)
      << ", depth=" << std::to_string(det.depth)
      << ", obj_id=" << std::to_string(det.obj_id)
      << ')';
  return out;
}

std::ostream &operator<<(std::ostream &out, const LidarDetection &det) {
  out << "LidarDetection(x=" << std::to_string(det.point.x)
      << ", y=" << std::to_string(det.point.y)
      << ", z=" << std::to_string(det.point.z)
      << ", intensity=" << std::to_string(det.intensity)
      << ')';
  return out;
}
```

Figure 7.6 Output stream from the sensor to the User. **Image source**: Carla code *[8]* with modification to include 'obj-id'.

Once the data structure in Libcarla has been modified and updated with a new variable, this change should be conveyed to the PythonAPI part of the pipeline. The PythonAPI is closest to the user. It should be able to parse the data correctly and should be updated with the Object_ID.
As shown in Figure 7.6, the file ‘SensorData.cpp’ contains the code that parses the output stream coming from the Unreal engine. The ‘operator<<’ function is used for output stream insertion which allows objects of the ‘RadarDetection’ class to be streamed to an output stream. The variable ‘obj_id’ needs to be added in this section of the code with the other radar detection variables so that the parser knows that the detections have been updated with an extra value.

## 7.4 Results

Now that the code has been modified it can be built and launched. The changes made to the code will reflect in the radar detections. The detection will also show the object IDs of the actors that come into the field of view of the radar Sensor.



```
Values of a detection in a frame:
RadarDetection(velocity=319.205261, azimuth=-0.071086, altitude=-0.136637, depth=23.902731)

Values of a detection in a frame:
RadarDetection(velocity=314.896332, azimuth=-0.077723, altitude=-0.145936, depth=22.112734)

Values of a detection in a frame:
RadarDetection(velocity=278.587830, azimuth=-0.139271, altitude=-0.085109, depth=43.306030)

Values of a detection in a frame:
RadarDetection(velocity=254.588348, azimuth=-0.176662, altitude=-0.115786, depth=29.118961)
```

```
Values of a detection in a frame:
RadarDetection(velocity=-0.053547, azimuth=-0.099141, altitude=-0.087566, depth=48.33723, obj_id=0)

Values of a detection in a frame:
RadarDetection(velocity=0.014082, azimuth=-0.050769, altitude=-0.026556, depth=21.165297 obj_id=522

Values of a detection in a frame:
RadarDetection(velocity=0.006261, azimuth=0.092469, altitude=-0.017053, depth=21.916578, obj_id=523)

Values of a detection in a frame:
RadarDetection(velocity=-0.030103, azimuth=0.212304, altitude=-0.049185, depth=53.860413 obj_id=0)
```

Figure 7.7 The radar detection from the unmodified and modified Carla simulator.

Figure 7.7. depicts the previous radar output and the output after making changes to the Carla Simulator. the object ID can be seen. It is to be noted that some of the Object IDs are ‘0’. This is because Carla returns ‘0’ for stationary objects like trees and buildings [10] whereas, actors created during execution and traffic signs that come with the map have unique Object IDs.

A simple code was written to create a scenario with a couple of actors and the radar mounted on an ego vehicle. This can be used to verify that the Object IDs are correct.

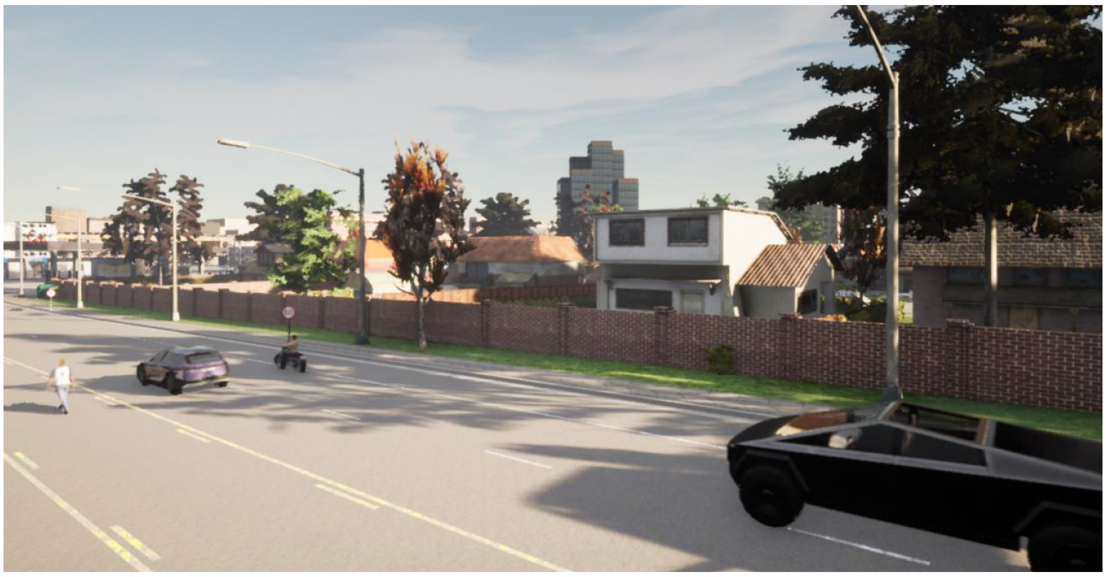
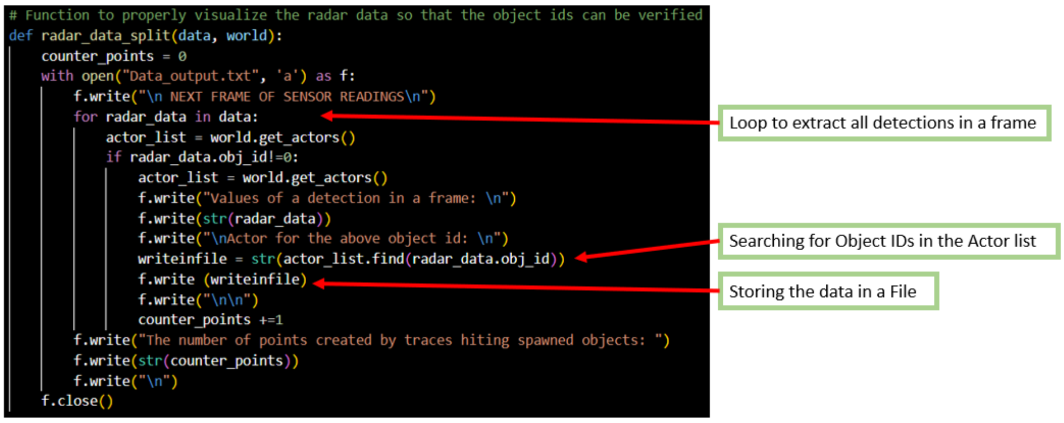


Figure 7.8 A scenario in Carla Simulator. The Cybertruck is the Ego vehicle.

As shown in Figure 6.4.2, the scenario consists of a ‘Tesla Cybertruck’, which is the Ego vehicle and has the radar mounted in front of it. The other three actors in the front are namely, on the right – ‘Harley-Davidson low-rider’, in the middle – ‘Audi Etron’, and on the left – ‘Pedestrian 0002’. These details are provided in the Blueprint Library in Carla [11].

The simulation is executed, and the detections acquired by the radar sensor are stored.
The ‘world’ is an object representing the simulation. It acts as an abstract layer containing the main methods to spawn actors, change the weather, get the current state of the world, etc. [12]. Using the ‘world’ object the list of actors in that simulation can be acquired and the actor matching the Object ID will be returned. If the actors used when creating the scenario match the actor names returned by the Object ID, the code changes made to the simulator can be verified.

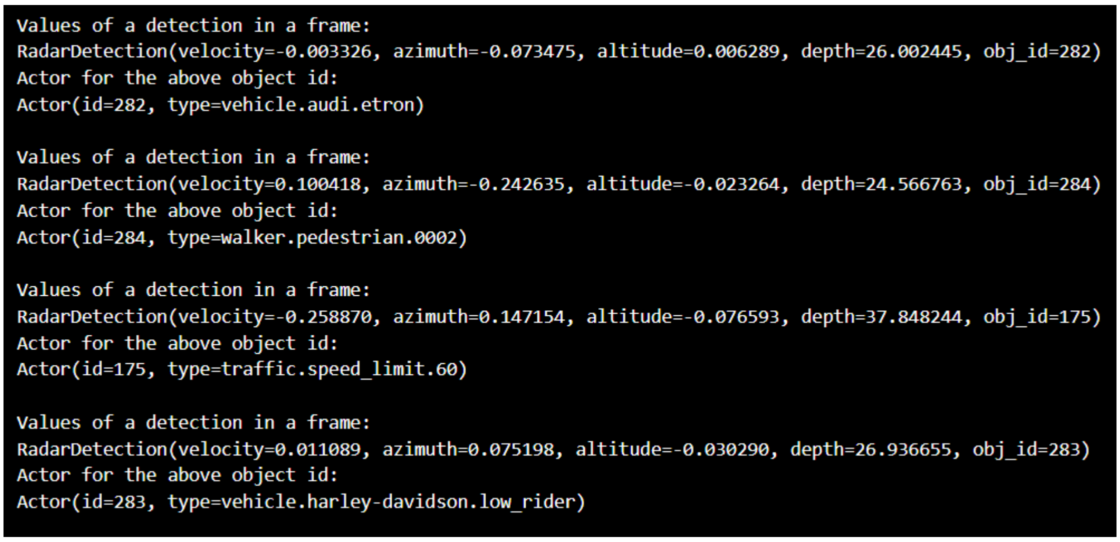


```python
# Function to properly visualize the radar data so that the object ids can be verified
def radar_data_split(data, world):
    counter_points = 0
    with open("Data_output.txt", 'a') as f:
        f.write("\n NEXT FRAME OF SENSOR READINGS\n")
        for radar_data in data:
            actor_list = world.get_actors()
            if radar_data.obj_id!=0:
                actor_list = world.get_actors()
                f.write("Values of a detection in a frame: \n")
                f.write(str(radar_data))
                f.write("\nActor for the above object id: \n")
                writeinfile = str(actor_list.find(radar_data.obj_id))
                f.write (writeinfile)
                f.write("\n\n")
                counter_points +=1
        f.write("The number of points created by traces hiting spawned objects: ")
        f.write(str(counter_points))
        f.write("\n")
    f.close()
```

Loop to extract all detections in a frame
Searching for Object IDs in the Actor list
Storing the data in a File

Figure 7.9 Code which searches the Actor List in the World Object and stores it in a file.

The radar data returned by the Carla simulator has the detections and is stored in an array of frames, where each frame corresponds to one episode in the Carla simulator. The function shown in Figure 6.4.3 gets the detections of one frame at a time. It then goes through all the detections and finds the ones that are not '0', as these detections are stationary objects like trees, buildings, etc. Once it gets a non-zero Object ID, it compares it to the actor list that it acquired from the world object. Upon finding a match it stores the details of the actor in the output file. The contents of the file are shown below in Figure 7.10.



```
Values of a detection in a frame:
RadarDetection(velocity=-0.003326, azimuth=-0.073475, altitude=0.006289, depth=26.002445, obj_id=282)
Actor for the above object id:
Actor(id=282, type=vehicle.audi.etron)

Values of a detection in a frame:
RadarDetection(velocity=0.100418, azimuth=-0.242635, altitude=-0.023264, depth=24.566763, obj_id=284)
Actor for the above object id:
Actor(id=284, type=walker.pedestrian.0002)

Values of a detection in a frame:
RadarDetection(velocity=-0.258870, azimuth=0.147154, altitude=-0.076593, depth=37.848244, obj_id=175)
Actor for the above object id:
Actor(id=175, type=traffic.speed_limit.60)

Values of a detection in a frame:
RadarDetection(velocity=0.011089, azimuth=0.075198, altitude=-0.030290, depth=26.936655, obj_id=283)
Actor for the above object id:
Actor(id=283, type=vehicle.harley-davidson.low_rider)
```

Figure 7.10 Displays the values returned by matching the Object IDs in the Actor list.

As shown in Figure 7.10, the Object ID from a few detections are taken and the actor details extracted from the Actor list are printed below each respective detection. The names of the actors created and placed in the Carla environment as seen in Figure 7.8, can be seen in the output file thus verifying that the Object IDs returned to us by the changes made in the Carla simulator are correct and that they can be used to label data for other implementations.

# 8. DBSCAN

In the field of radar signal processing, the Density-Based Spatial Clustering of Applications with Noise (DBSCAN) algorithm emerges as a powerful tool, seamlessly addressing the challenges posed by complex and dynamic radar environments. As radar systems become increasingly sophisticated, the need for robust and efficient clustering techniques becomes crucial to extracting meaningful information from vast and intricate datasets [13].

DBSCAN, introduced in the field of data mining, has found a natural application in radar technology due to its ability to identify clusters of radar point clouds based on their spatial density. Unlike traditional clustering algorithms, DBSCAN excels in handling irregularly shaped clusters and is particularly adept at discerning outliers, which is crucial in the presence of noise or unexpected radar reflections [14].

As DBSCAN is a density-based algorithm, it is perfect for defining clusters as regions of high radar point cloud density, enabling the identification of distinct objects or phenomena. This adaptability proves invaluable in scenarios where conventional clustering methods may struggle, such as in the presence of variable target shapes, clutter, or interference [15].

The efficiency and reliability of DBSCAN in radar applications make it an indispensable tool for various purposes, including target detection, tracking, and classification. Its ability to adapt to different radar environments and accommodate varying target densities positions DBSCAN as a versatile solution that enhances the overall performance and accuracy of radar systems [16].

As radar technology continues to evolve, the integration of advanced clustering techniques like DBSCAN signifies a paradigm shift towards more intelligent and adaptive signal processing. The application of DBSCAN in radar not only enhances the capability to discern relevant information from complex data but also paves the way for further innovations in the ever-evolving field of radar technology.

## 8.1 Parameters of DBSCAN

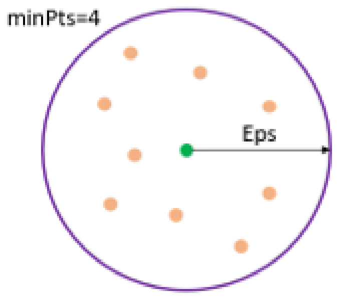


Figure 8.1 Parameters of DBSCAN Algorithm. Image inspired by *[15]*.

- Epsilon: Epsilon is the radius of the circle that needs to be created across each data point to check the density of the cluster.
- *minPoints*: It is the minimum number of data points that need to be contained in a data points circle for it to be classified as a core point [17].

## 8.2 Types of Points

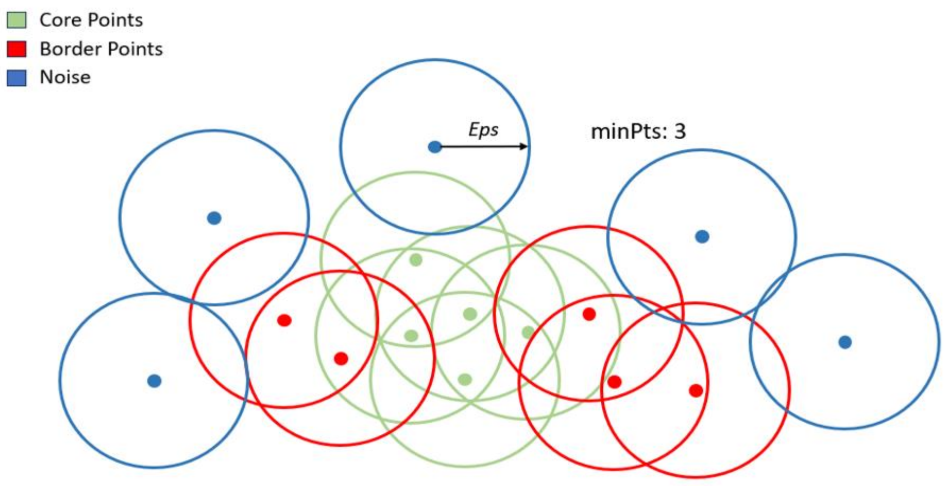


Figure 8.2 Classification of points based on the DBSCAN Parameters. Image inspired by *[15]*.

DBSCAN works by creating a circle of length Epsilon radius around every data point and classifying them into core points border points and noise.

- Core Points: A data point is classified as a core point if it has the number of data points specified by parameters *MinPoints* inside the radius Epsilon.

- Border Points: If a data point has points less than the *minPoints* inside the radius Epsilon, it is classified as a border point [18].
- Noise: If a data point is not in the Epsilon radius of any other point or has no points other than itself inside its neighborhood it is considered Noise.

As shown in Figure 8.2, the *minPoints* have been considered as 3, and a circle of radius Epsilon is drawn around each point. The points depicted in Green are the core points as they have at least 3 points in their circle including themselves.

The data points in Red are the border points and they have less than 3 but greater than 1 point in their circle. Lastly, the data points in Blue are the noise/outliers. These data points have no other points apart from themselves inside the circle.

## 8.3 Reachability and Connectivity

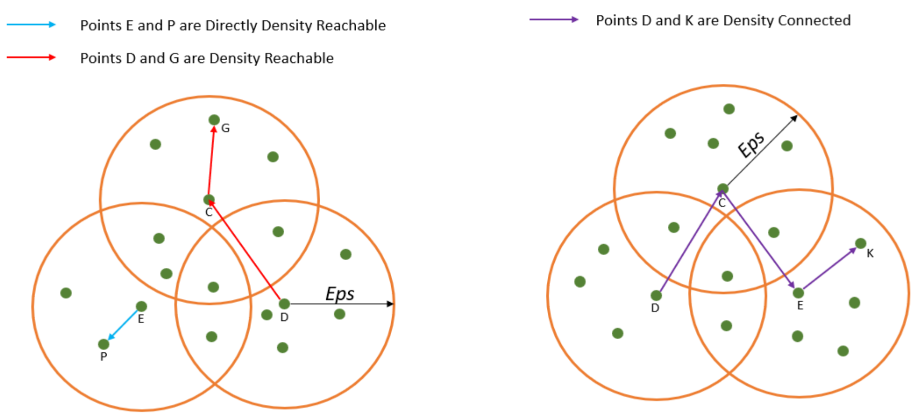

Points E and P are Directly Density Reachable
Points D and G are Density Reachable
Points D and K are Density Connected
G
C
Eps
E
P
D
Eps
C
D
E
K

Figure 8.3 Direct Density Reachable, Density Reachable, and Density Connected points. Image inspired by *[15]*.

If a data point is reachable from another data point directly or indirectly, it’s called reachability. Whereas connectivity defines if two data points belong to the same cluster or not [18].

- Direct Density Reachable: A point is said to be Direct Density Reachable if it falls within the neighborhood of a core point.
- Density Reachable: If a point is connected to another point through a series of core points, it is said to be Density-reachable.
- Density Connected: If there is a core point that is density reachable from both points, then they are said to be Density Connected. It shows a symmetric relationship that demonstrates connectivity between 2 data points in a cluster.

# 9. APPLYING DBSCAN ON CARLA RADAR DATA

The primary goal thus far has been modifying the CARLA simulator's source code to facilitate the generation of labeled data. Having accomplished this goal, the focus shifts towards harnessing the radar data for broader algorithmic utilization. This involves preprocessing steps such as filtering and clustering point clouds to create a more informative labeled dataset for diverse machine learning algorithms.

The initial step includes trimming the data set so that any incomplete frames of data are removed. This is followed by filtering the points based on relative velocity. This helps distinguish between stationary and moving data. This step lays the foundation for further processing aimed at enriching the dataset's content and accuracy.

The next phase is to apply a clustering algorithm to the radar data to identify and group spatially proximal data points. In this case, DBSCAN is being used, and it facilitates the grouping of data points that probably represent the same thing as a car or a pedestrian. Machine learning algorithms like PointNet++ benefit from a data set that already has a clustering algorithm like DBSCAN applied to it. It improved the detection and classification of moving road users [19]. Also, keeping next-generation radar sensors in mind, DBSCAN clustering has already been shown to drastically increase its performance for less sparse radar point clouds [20].The implementation of these individual steps will be discussed in detail in the following sections.

## 9.1 Scenarios

Creating simple scenarios in the Carla Simulator is a very efficient way to design and simulate various traffic situations. This enables the collection of diverse data sets for analysis and algorithm development. It will also aid in predicting and gaining a better understanding of how the simulated radar sensor works in complicated situations, like the impact of modifying different radar attributes and the point cloud generated, thereby providing insight into the different functionalities and limitations of the radar sensor.

These scenarios will test if the radar can capture detections for cars as well as smaller obstacles like bikes, cyclists, or pedestrians. This also includes a scenario in which the target vehicles are moving perpendicular to the ego vehicle.

In all the scenarios, the vehicle used as the ego vehicle is a Tesla Cybertruck and the radar is placed in the front center of the Ego vehicle.

### 9.1.1 Scenario 1

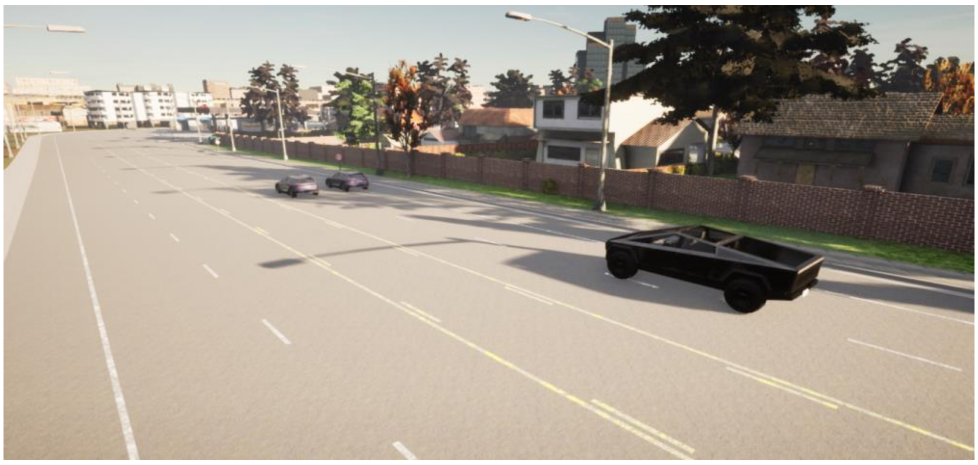


Figure 9.1 A Scenario showing two target vehicles in front of the ego vehicle.

This is a typical urban driving environment with moderate traffic. There are two cars positioned ahead of the Ego vehicle, one on the left lane and one on the right. All the 3 vehicles are moving forward at the same speed. As the Ego vehicle approaches the two cars, the radar system detects their presence and starts updating the ego vehicle with real-time data about their position, velocity, and object IDs. This data can be accessed by the user via the Carla pipeline.

Such a scenario is quite common, and the data acquired should be tested to see if filtering and clustering algorithms can be applied to it such that they can be identified from among the other points in the point cloud.

### 9.1.2 Scenario 2

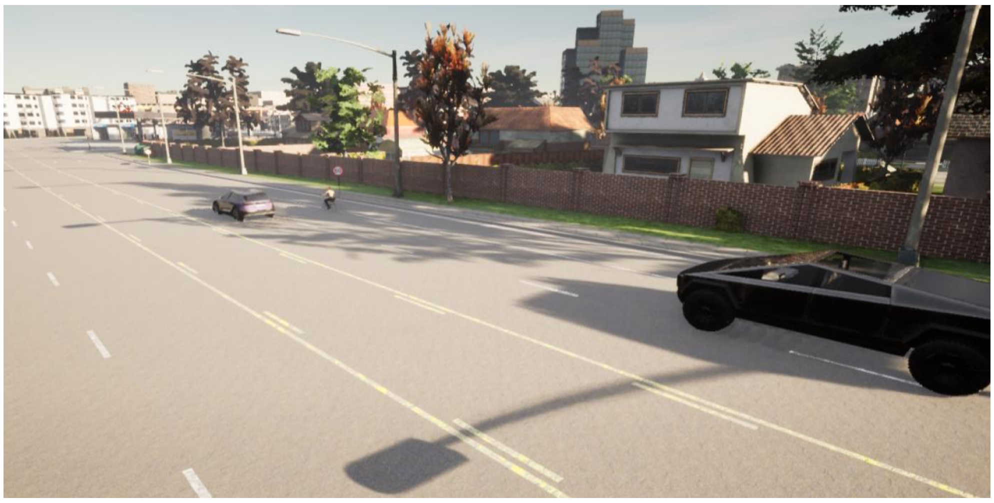


Figure 9.2 A scenario showing a car and cyclist in front of the ego vehicle.

This is a more complex scenario in which a cycle and a car will be placed in front of the ego vehicle. They both will be on parallel lanes and traveling at the same speed. The ego vehicle will maintain a safe distance from the vehicles in front. This can be seen in Figure 9.2.

The goal will be to observe how the radar perceives the objects and whether it can provide enough detection points for a smaller vehicle like a cyclist. Such scenarios are important to check the behavior of the radar when faced with different objects. Verifying that the radar sensor accurately detects both the vehicle and the cycle is crucial for realistic simulation outcomes.

This will also help to verify the working of the DBSCAN algorithm and to see if it can cluster the points after the stationary data points have been filtered out.

### 9.1.3 Scenario 3

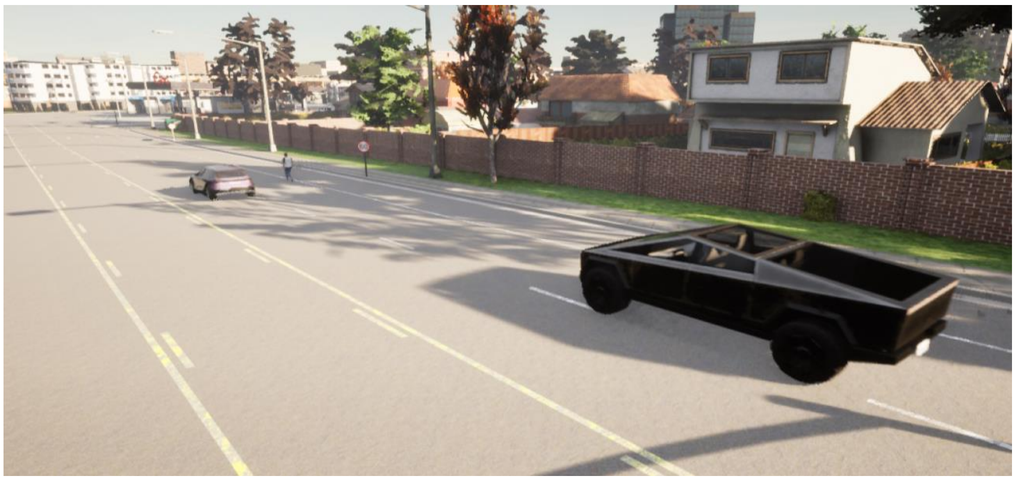


Figure 9.3 A Scenario showing a car and pedestrian in front of the ego vehicle.

As seen in Figure 9.3., in this scenario, the cyclist is being replaced by a pedestrian. Similar to the previous scenario, the Ego vehicle with radar mounted in the front center, moves behind the car and the pedestrian while maintaining a safe distance.

One of the primary concerns in modern automobile development is the improvement of traffic safety which includes pedestrian safety as well [21]. This will assist in enhancing the decision-making algorithms and is crucial to validate how accurately the radar sensor can detect and respond to pedestrians.

### 9.1.4 Scenario 4

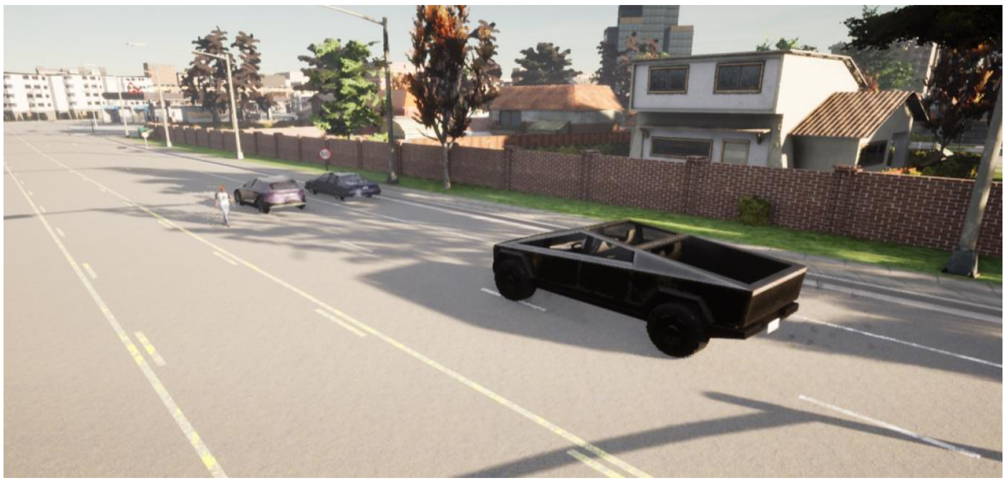


Figure 9.4 Scenario showing 2 cars and a pedestrian in front of the ego vehicle.

In this scenario, there is a car, a Motorcycle rider, and a pedestrian in front of the ego vehicle. The pedestrian is moving in the opposite direction to the other 2 vehicles including the Ego vehicle. This can be seen in Figure 9.4.

This is to see how the radar reacts to objects approaching it and if it can perceive and track objects moving in different directions. This scenario is useful for future developments like implementing decision-making algorithms to account for pedestrians approaching from the opposite direction. Other scenarios that could be created in the future could include changing the speed or suddenly changing directions and observing the radar detections.

### 9.1.5 Scenario 5

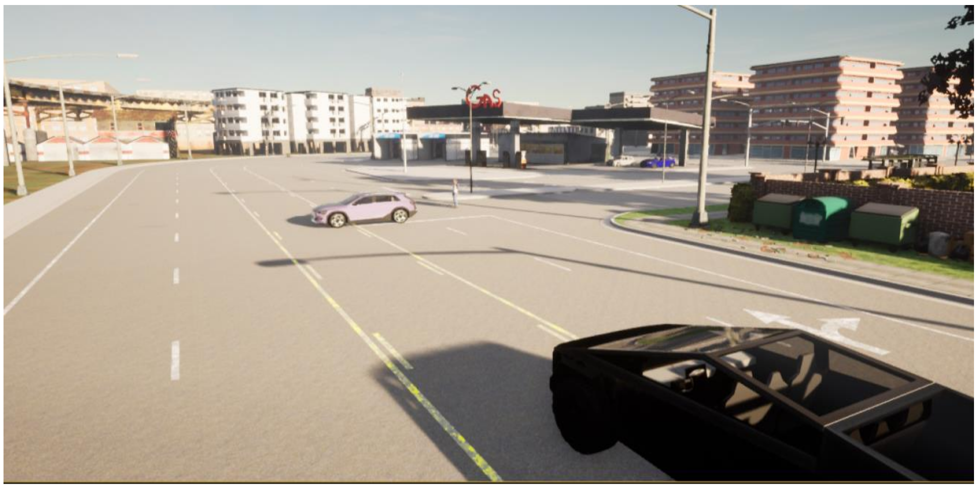


Figure 9.5 A scenario showing a car and a pedestrian crossing in front of the ego vehicle.

In the final scenario, as seen in Figure 9.5., the ego vehicle encounters a moving car and pedestrian crossing perpendicular to its path.

This scenario helps analyze the behavior of the radar when it encounters objects moving perpendicular to its direction of motion. The radar must detect both moving objects. As mentioned previously, this will be useful when tracking, decision-making, or safety algorithms are implemented using radar data.

## 9.2 Filtering

Upon execution of a scenario in the Carla Simulator, the 'listen' function is activated. This function streams the radar data being captured in the simulator. It does so by activating a callback mechanism, effectively storing the data in a queue. A queue has been used here because it is quicker to append.

When a scenario is being executed in Carla many times a few of the initial and ending frames are incomplete or flawed. This disparity arises from the asynchronous spawning and deletion of the actors relative to the radar. This misalignment leads to inconsistent data at the transitional phases of the scenario. Hence, to improve the accuracy and completeness of the data, some of the frames at the beginning and end are trimmed. After the data has been trimmed, the stationary and moving points will be identified.

As discussed previously, the radar sensor gives us the radial velocity of the detection points. What it does is that it gives the radial velocity component of a target velocity vector based on the Doppler-frequency shift [22]. When radar waves reflect off an object, the frequency of the reflected wave is shifted due to the relative motion between the radar sensor and the object. This frequency shift is known as the Doppler effect and because of this, an approaching object results in a reflected wave which is slightly higher frequency than the transmitted wave while a receding object results in a slightly lower frequency. The relative velocity of an object is proportional to the frequency offset [23].

Stationary objects do not move; hence, they cause no frequency shift in the radar waves. As radial velocity is the velocity component of the line of sight between the radar sensor and the detected object, it can be split into its respective x and y components to get the velocity of the target vehicle in both the horizontal (x) and vertical (y) directions.

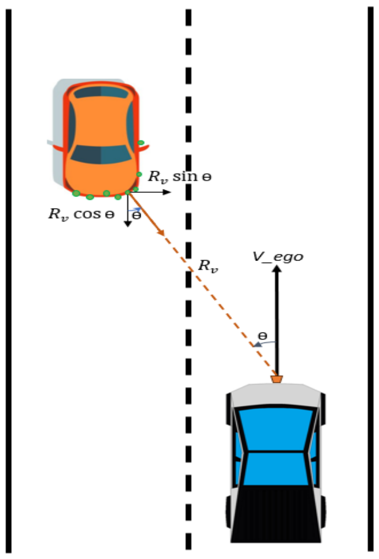


Figure 9.6 Shows how the radial velocity can be split into their respective x and y components so that the velocity of the target vehicle can be determined. (Source: Top view of Tesla cyber truck *[24]*, target vehicle *[25]*)

Figure 9.6., shows a scenario in which a target vehicle approaches the Ego vehicle. The Carla radar sensor returns detections for every trace that hits the target vehicle wherein each point has azimuth, elevation, range, radial velocity, and Object ID. The radial velocity ($R_v$) gives the relative velocity between the Ego vehicle and the target vehicle and azimuth(ɵ) is the horizontal angle between the Ego and the target vehicle. Using these the radial velocity can be split into its x and y components, that is, $R_v \cos \theta$ and $R_v \sin \theta$ respectively.

$$V_x = R_v \cos \theta \quad \text{(Target velocity)}$$

$$V_y = R_v \sin \theta$$

$V_x$ is the target velocity. If this target velocity is approximately the same as the Ego vehicle, falling within a predefined threshold of velocity it can be classified as stationary.

$$V_{ego} - Threshold_{vel} < V_x < V_{ego} + Threshold_{vel}$$

**Algorithm 1** Trimming and identifying moving and stationary detection points

```
Required: Scenarios have been created
while A scenario is running do
    Read sensor data via listen() function          ▷ listen() triggers a callback function
    Store radar data in a Queue
    Store vehicle velocity
Trim Incomplete data frames                         ▷ After a scenario has finished executing
for Each frame in Radar data do
    for Each detection point do
        Extract azimuth, elevation, range, radial velocity, and Object ID
                                    ▷ The above detections are in Spherical Coordinate System
        Calculate speed of target detection point
        Convert the Azimuth, Elevation, and Range to the Cartesian coordinate system
        if Target speed approximately matches the Ego vehicle then
            Store position(x,y,and z coordinate values), and Object ID of stationary points
        else
            Store position(x,y,and z coordinate values), and Object ID of moving points
```

Figure 9.7 Algorithm of the code for filtering and identification of stationary and moving targets.

Figure 9.7. illustrates the algorithm of the code implemented for storing data, trimming frames, and identifying stationary and moving targets. After trimming the frames, the algorithm iterates through the detection points in each frame. Then the radial velocity and azimuth for each point are extracted and split into their respective x and y components. The x component gives the velocity of the target vehicle which is compared with the velocity of the ego vehicle for each frame. This velocity of the Ego vehicle was obtained for each frame during the scenario execution. If the velocity of the target detection approximately matches that of the Ego vehicle, within a specified threshold of velocity, it is deemed to be a stationary object, and the data is stored separately. Whereas, if the velocity is outside these boundaries, then it is considered a moving object.

It is to be noted that the points generated by the Carla simulator are in the Spherical coordinate system and for better representation, they need to be converted to the Cartesian coordinate system. Figure 9.8. represents the detections in the spherical coordinate system and the detections after conversion to the cartesian coordinate system. The scenario used for the plot is Scenario 4 as shown in Figure 9.3. it consists of 3 target objects (a pedestrian and 2 cars), in which the cars are moving in the direction of the Ego vehicle, and the pedestrian is approaching the Ego vehicle.

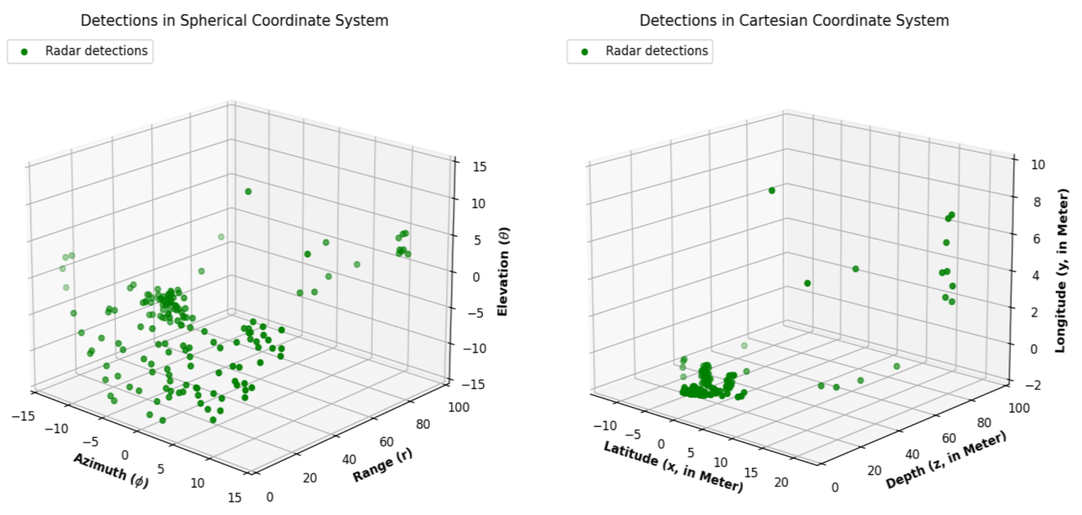


Figure 9.8 Shows the detections in the Spherical Coordinate system and Cartesian coordinate system.

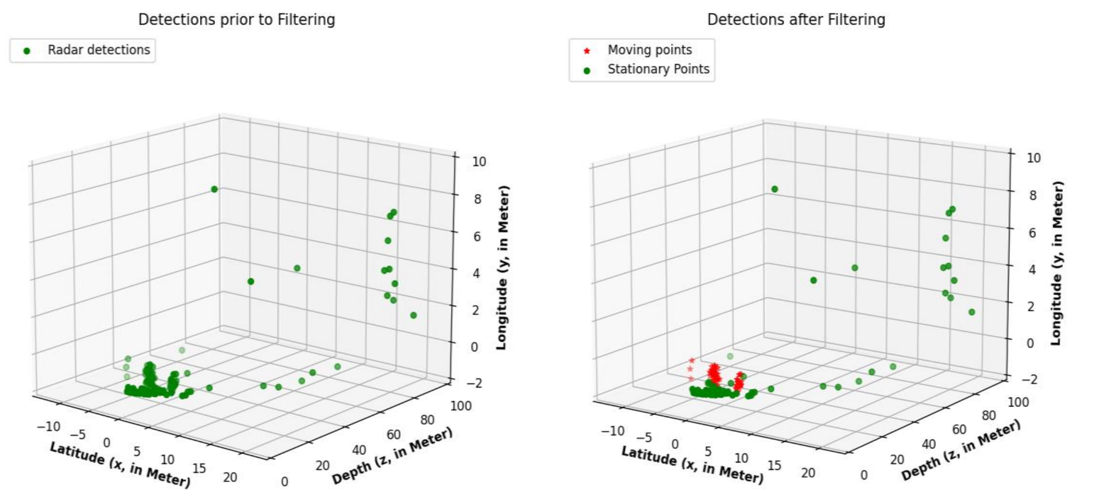


Figure 9.9 Identifying stationary and moving points from radar detections.

Following the conversion of data into the Cartesian coordinate system, the detections need to be segmented into moving and stationary points. As illustrated in Figure 9.6., the target velocity of each detection point is calculated. Those points whose target velocity aligns closely with that of the Ego vehicle are classified as stationary vehicles. Conversely, the remaining points are logically attributed to moving objects. These points are then plotted to see if the detections resemble the scenario being simulated. This can be seen in Figure 9.9. The graph on the left is a plot of all the detections in the frame, whereas the graph on the right shows the points that have been classified as moving and stationary. The detections marked in red are the moving objects. The scenario consists of 3 moving objects in front of the Ego vehicle, and this is represented in the figure.

An important thing to note here is that the default value for the radar attribute ‘points per second’ is 1500 which is too low. This attribute defines the number of points generated by all lasers per minute and the default value provides a very sparse point cloud data. Hence, to improve the results, the ‘points per second value’ was increased. An alternate solution to this would be to fuse the detections of a few consecutive frames into one so that a point cloud of the desired density is achieved.

## 9.3 DBSCAN Clustering

In this subsection, the DBSCAN clustering algorithm is used on the moving and stationary points identified and segregated in the previous section.

The DBSCAN algorithm used here is from ‘Scikit-learn’, which is a versatile machine-learning library in Python. The scikit-learn library provides tools for various tasks, including clustering, classification, regression, etc. Also, its implementation of the DBSCAN algorithm is very user-friendly and efficient [26].

The procedure to use DBSCAN from Scikit-learn involves the following steps:

- Importing the necessary modules from Scikit-learn.
- Creating a DBSCAN object, with desired parameters like ‘*eps*’ and ‘*min_samples*’.
- Fit the model to the dataset.
- After fitting the model, the labels assigned to each data point can be accessed. The label ‘-1’ indicates noise points.
- Finally, the data can be visualized by plotting the points with different labels.

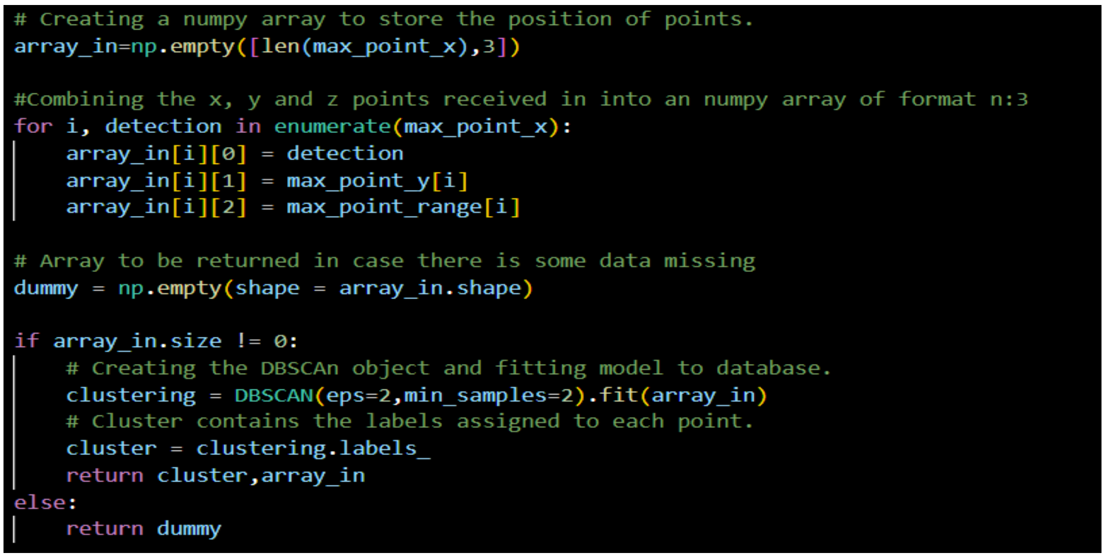
The implementation of these steps for the Carla data will be demonstrated further.

**Algorithm 2** Applying DBSCAN on stationary and moving data points

```
1: Required: Stationary and moving points have been identified and segregated.
2: Import DBSCAN from Scikit-learn library
3: Import matplotlib library                                   ▷ To visualize the result
4: for Each frame in Radar data do
5:                  ▷ The same loop can be run to cluster both moving and stationary data points
6:     Create a 2D array to store the x, y, and z coordinates of stationary/moving data points
7:     Store the x, y, and z coordinates for the points in the frame in the array.
8:     Create an object for DBSCAN                    ▷ Value of eps = 2 and min-samples = 2
9:     Fit model to a dataset
10:    Access cluster results
11:    Assign different colors as per the labels
12:    Visualize data via matplotlib library
```

Figure 9.10 An overview of how the DBSCAN algorithm has been applied to the Carla dataset.

Figure 9.10. shows the algorithm used to apply DBSCAN on the filtered data set. As discussed above, the procedure to apply DBSCAN from the Scikit-learn library is illustrated. This begins by importing the required libraries for implementing DBSCAN and visualizing (matplotlib) the clustering result. Subsequently, the code iterates through all the frames captured by the radar sensor, thereby, facilitating the frame-by-frame analysis of the data. Both the stationary and moving data points can be clustered as per need and plotted side by side. The DBSCAN algorithm requires an array with all the x, y, and z coordinate data for the points in a frame. It also requires the parameters '*eps*' and '*min_samples*' that determine the maximum distance between 2 sample points for them to be considered a part of the same neighborhood and the minimum number of samples required to form a dense region respectively. They are configured as '2' because this gives the best result. Following this, the DBSCAN model is fitted to the dataset enabling the identification of clusters. Each data point is assigned a label indicating which cluster it belongs to. If a data point is 'noise', then it is assigned a label of '-1'. The implementation can be seen in Figure 9.11.



```
# Creating a numpy array to store the position of points.
array_in=np.empty([len(max_point_x),3])

#Combining the x, y and z points received in into an numpy array of format n:3
for i, detection in enumerate(max_point_x):
    array_in[i][0] = detection
    array_in[i][1] = max_point_y[i]
    array_in[i][2] = max_point_range[i]

# Array to be returned in case there is some data missing
dummy = np.empty(shape = array_in.shape)

if array_in.size != 0:
    # Creating the DBSCAn object and fitting model to database.
    clustering = DBSCAN(eps=2,min_samples=2).fit(array_in)
    # Cluster contains the labels assigned to each point.
    cluster = clustering.labels_
    return cluster,array_in
else:
    return dummy
```

Figure 9.11 Shows the implementation of DBSCAN clustering using Scikit-learn.

## 9.4 Results

After the DBSCAN clustering has been completed and the points have been labeled, they can be plotted using matplotlib. To better visualize the result, the clusters have been assigned different colors and shapes. This can be seen in the figures below.
The moving objects have been depicted by circles and the stationary points have been assigned thin diamonds. The 'x' marks on the graph are the noise and are the points that did not fall in any of the clusters.

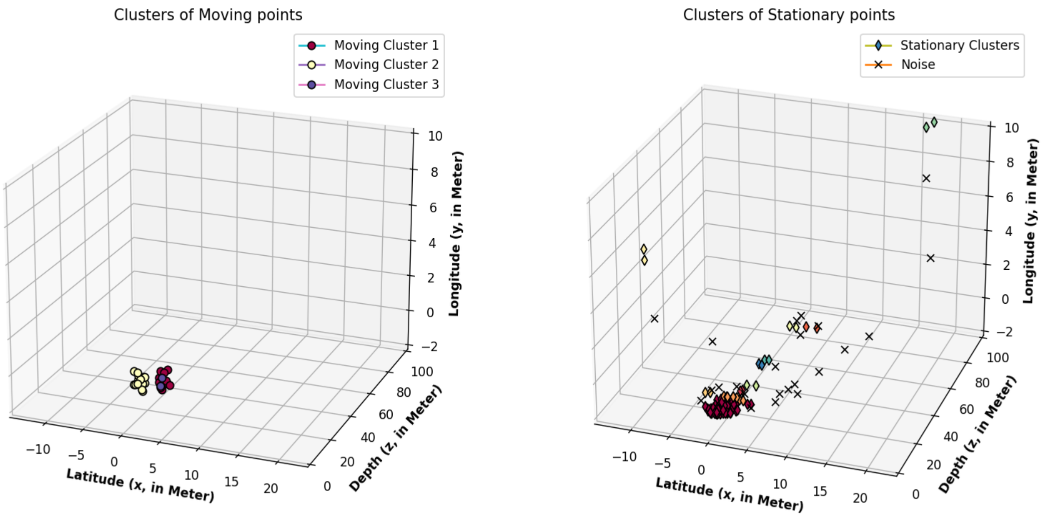


Figure 9.12 Shows the clusters Identified by applying DBSCAN on the point cloud generated by Scenario 1

Figure 9.12. illustrates the result after filtering and applying DBSCAN clustering on the point cloud data obtained from the radar sensor on executing Scenario 1. This scenario is shown in Figure 9.1. The left graph in Figure 9.12, shows the clusters of moving points whereas the right graph shows the clusters of stationary points. Scenario 1 portrays two vehicles moving in front of the ego vehicle. The application of the DBSCAN algorithm results in 2 major clusters of moving points, aligning with the expected outcome. This observation highlights the effectiveness of the applied techniques in accurately discerning and clustering the moving objects within the radar-derived point cloud data.

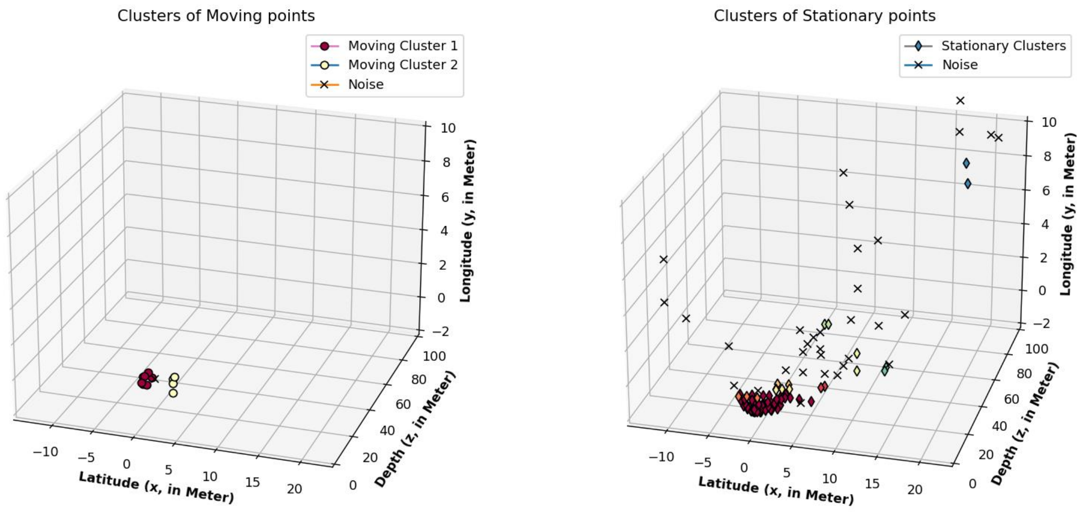


Figure 9.13 Shows the clusters Identified by applying DBSCAN on the point cloud generated by Scenario 2

Scenario 2, as shown in Figure 9.2, illustrates two target objects i.e. a vehicle and a cyclist moving in front of the ego vehicle. The point clouds for this scenario were filtered and clustered. The result is shown in Figure 9.13. As can be seen in the graph of moving objects, there are two objects with one of the two targets having fewer detections. This may suggest that one of the targets is smaller like a cyclist.

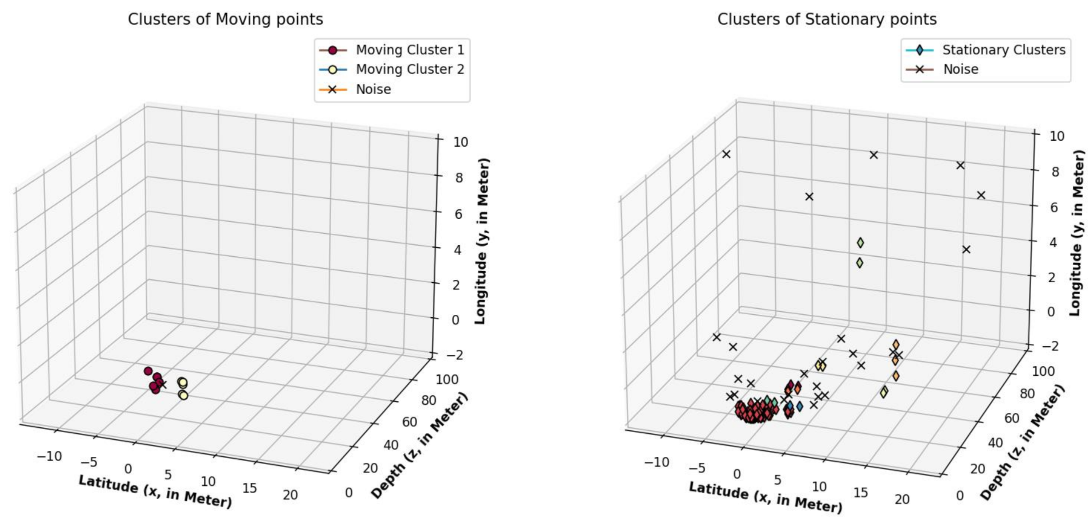


Figure 9.14 Shows the clusters Identified by applying DBSCAN on the point cloud generated by Scenario 3

Like scenario 2, scenario 3 has a vehicle and a pedestrian moving in front of the ego vehicle. The clusters for this scenario are illustrated in Figure 9.14. The graph in Figure 9.14., shows 2 clusters for moving objects. These were identified using the DBSCAN algorithm which aligns with the description of the scenario.

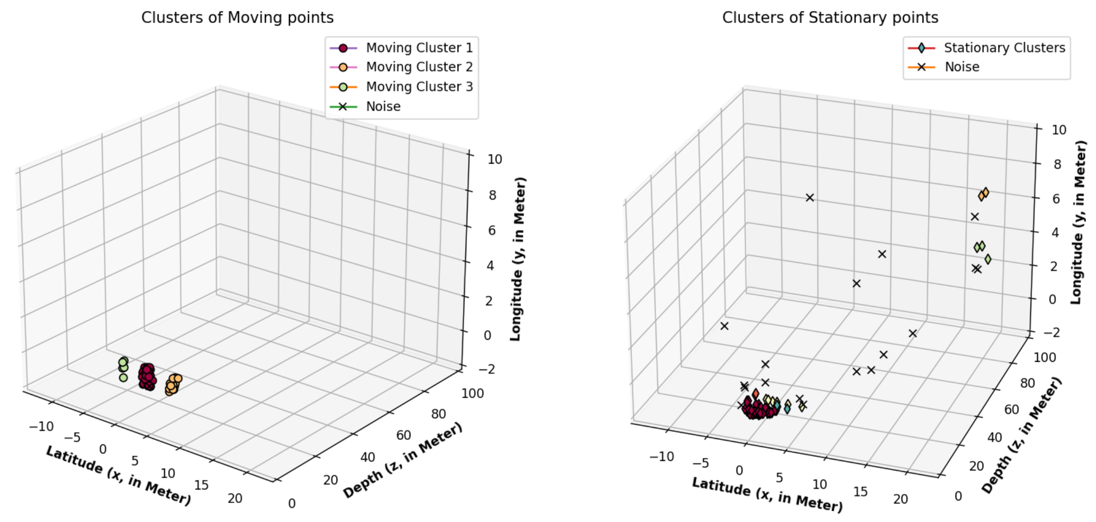


Figure 9.15 Shows the clusters Identified by applying DBSCAN on the point cloud generated by Scenario 4

Scenario 4 has three objects moving in front of the ego vehicle. Two cars are moving in the direction of the ego vehicle whereas 1 pedestrian is approaching the ego vehicle from the forward direction. An image of this scenario can be seen in Figure 9.4. As seen in Figure 9.15, the filtering and clustering of the point cloud show three clusters that were identified among the moving detection points which lends credence to the process.

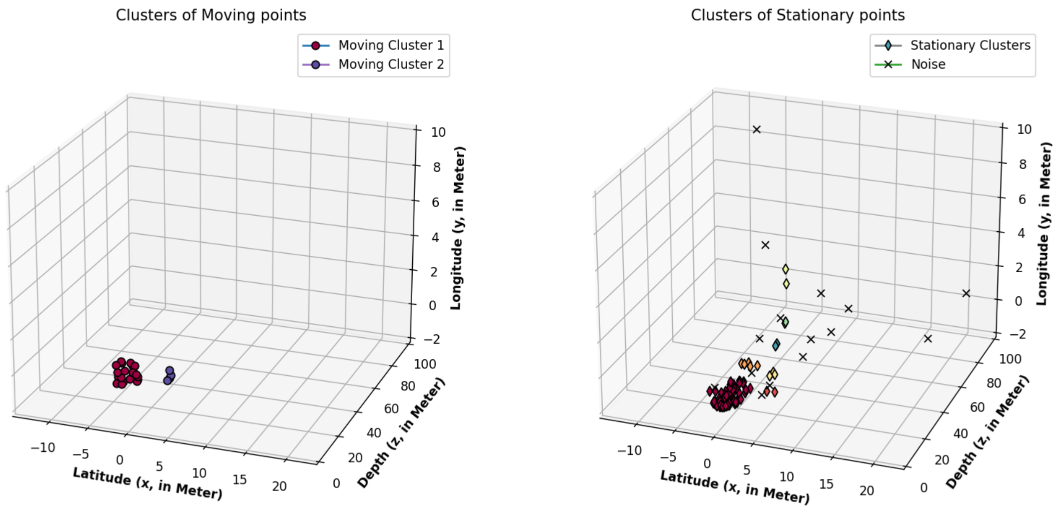


Figure 9.16 Shows the clusters Identified by applying DBSCAN on the point cloud generated by Scenario 5

Lastly, Figure 9.16 shows the clusters obtained by the radar sensor model from scenario 5. As shown in Figure 9.5, scenario 5 was created to check the success rate of the detection of target objects moving perpendicular to the ego vehicle. The target objects in this scenario were a vehicle and a pedestrian, with the pedestrian moving behind the target vehicle. The moving points were successfully segregated and DBSCAN identified two separate clusters which can be seen in Figure 9.16.

# 10. PROBLEMS ENCOUNTERED

A few problems were faced during the implementation of this project, and they are listed below:

1) Hardware availability
2) Version Mismatch on Windows PC
3) Carla Rendering

## 10.1 Hardware Availability

A major factor to consider when building Carla in a PC is the hardware requirements. Carla is a complex software that incorporates graphics, advanced physics, and AI components. It is also heavily dependent on various external libraries which are computationally intensive to build. The build process in Carla consists of compiling thousands of source files, linking libraries, and generating binaries. The advanced graphics used by the Carla simulator to simulate realistic environments and sensor data consume a substantial amount of GPU resources. Lastly, the parallelization of build processes, and reading and writing large amounts of data to disc are also resource expensive.

All the above reasons make it necessary to have more powerful hardware like multicore CPU, ample RAM, and dedicated GPU.

## 10.2 Version Mismatch on Windows PC

Building Carla on a Windows PC leads to a lot of version mismatch errors and hence it is necessary to keep track of the versions of the software being used. Errors occurring because of this are very difficult and time-consuming to debug. These errors were significantly reduced while using Linux OS.

## 10.3 Carla Rendering

Probably because Carla is so resource-intensive, it faces issues rendering sensor data. While executing the code to get radar sensor data on Carla, it was observed that there were anomalies in the data. The number of detections per frame and time interval between consecutive frames was irregular. This also caused problems in the values recorded for the velocity of the ego vehicle.
The solution was to keep Carla's application window as the focus instead of running it in the background. Graphics rendering for Carla requires continuous communication between the CPU and the GPU. The switch in focus of the GPU on rendering other foreground applications causes delays or

interruptions in rendering Carla’s graphics. This competition for CPU, GPU, and memory resources heavily impacts Carla’s performance.
As a result of running Carla in the background, the rendering process was merging data across frames into one and it was doing so inconsistently.

# 11. CONCLUSION

This chapter presents a summary of the initial goals of this thesis and the results obtained. The main goals of the project can be condensed into two points:

1) Build Carla on Windows and Linux PCs, check the feasibility of both, and add a feature to the Radar Sensor Model that allows the radar system to return the object IDs of the detection points.
2) Create different scenarios that depict real-world scenarios and process them so that they can be further used for other algorithms. This includes trimming incomplete/faulty data, identifying stationary and moving data points, and clustering the points using the DBSCAN clustering algorithm.

As discussed previously, the desire for labeled and accurate data is immense. Also, annotating/labeling data is a very time-consuming and expensive process. Hence, to solve this, the first goal of the thesis aimed to modify the Carla simulator so that it could provide labeled data. This goal was accomplished, and with the modification of the Carla simulator pipeline, it now returns labeled radar point cloud data which can be used for various tracking, localization, or machine learning algorithms.

Furthermore, correctly processed data has more information that makes it more valuable for machine learning algorithms, thereby improving the predictions made by the machine learning model. This was what the second goal aimed to achieve. The data set was enriched by trimming, identifying moving/stationary points, and clustering using the DBSCAN algorithm. The process and results for this were demonstrated in Chapter 9.

# 12. FUTURE SCOPE

## 12.1 Improving Radar Sensor Model

There is immense demand for annotated point cloud data in the automotive industry. The current process of labeling data is extremely time-consuming and expensive. Also, it is risky to create real-time scenarios where a person might be endangered while collecting data. This is where the demand for simulated data is crucial. Even though the Carla simulator fills these needs, there is room for improvement in the Radar sensor model so that the generated data is as close to the real-world data as possible.

### 12.1.1 Implementing Multibounce in Radar Sensor Model:

One such improvement could be the addition of multibounce rays in the radar sensor Model. Currently, the Radar Sensor Model works on traces. What this means is that the radar sensor shoots out traces (lines) and when these traces hit a target object then the data of the target object is considered as a detection. But this is not how the radar works in real-time scenarios. The real-world radar sensor sends out a pulse of energy that bounces off an object and is reflected to the radar sensor. This energy can bounce off multiple targets before it returns to the radar sensor. These multiple bounces result in many anomalies in the radar detections, like Ghost Objects [27] [28]. This will not be seen in the Carla Radar Sensor. Therefore, the implementation of multibounce rays will help the Carla Radar Sensor Model mimic a real-world sensor better.

### 12.1.2 Micro-Doppler:

Radar systems can detect and analyze the small Doppler shifts that result from the motion of targets with complex motion patterns, such as rotating and vibrating components. This is called Micro-Doppler. For example, the vibration of an engine, the heartbeat of a human [29], the movement of hands and legs [30], etc. will cause Micro Doppler. These additional motions result in Micro-doppler signatures superimposed on the primary Doppler signature. The analysis of these micro-doppler signatures could help the radar sensor in extracting additional information about the target object, such as activity, orientation, and type, which would play a significant role in many machine learning applications like pedestrian detection or gait detection using radars.
Hence, finding a way to implement the micro-doppler feature in the Carla Radar Sensor would be a great addition to Carla.

## 12.2 Applying Machine Learning Algorithms

PointNet++ algorithm could be applied to the data points clustered using DBSCAN. The methodology proposed in [19] is a valuable reference that describes using PointNet++ along with DBSCAN and LSTM to give promising results. This machine learning model trained on Carla's data could be compared against real-world data including other datasets like the Kitti or nuScenes data set. A comparison could also be made against other algorithm implementations to test which is better. As many scenarios can be created in Carla, a model could be trained for different scenarios.

# 13. Bibliography

[1] A. Dosovitskiy, R. German, F. Codevilla, A. Lopez and V. Koltun, „CARLA: An open urban driving simulator,“ *PMLR,* pp. 1-16, 2017.

[2] Team, Carla, „Windows build,“ 2020. [Online]. Available: https://carla.readthedocs.io/en/0.9.11/build_windows/.

[3] Team, Carla, „Linux Build,“ 2020. [Online]. Available: https://carla.readthedocs.io/en/0.9.11/build_linux/.

[4] „make PythonAPI fails for Windows 10,“ 2021. [Online]. Available: https://github.com/carla-simulator/carla/issues/3621.

[5] Team, Carla, „RuntimeError: time-out of 2000ms while waiting for the simulator,“ 2021. [Online]. Available: https://github.com/carla-simulator/carla/issues/3430.

[6] Team, Carla, „Module 'carla' has no attribute 'Client',“ 2021. [Online]. Available: https://github.com/carla-simulator/carla/issues/3083.

[7] Team, Carla, „How to add a new Sensor,“ 2020. [Online]. Available: https://carla.readthedocs.io/en/latest/tuto_D_create_sensor/.

[8] Carla Team, „Carla Source Code Github,“ [Online]. Available: https://github.com/carla-simulator/carla/tree/0.9.11.

[9] Carla Team, „World and client,“ [Online]. Available: https://carla.readthedocs.io/en/latest/core_world/.

[10] Carla Team, „Semantic Lidar Object ObjIdx Definition and Uniqueness,“ 2020. [Online]. Available: https://github.com/carla-simulator/carla/issues/3191.

[11] Carla Team, „Carla Blueprint Library,“ [Online]. Available: https://carla.readthedocs.io/en/latest/bp_library/ .

[12] Carla Team, „Foundations,“ [Online]. Available: https://carla.readthedocs.io/en/latest/foundations/.

[13] Sohee Lim, Seongwook Lee, Seong-Cheol Kim, „Clustering of Detected Targets Using DBSCAN in Automotive Radar Systems,“ *IEEE,* 2018.

[14] Thomas Wagner; Reinhard Feger; Andreas Stelzer, „Modification of DBSCAN and application to range/Doppler/DoA measurements for pedestrian recognition with an automotive radar system,“ *IEEE.*

[15] Yue Wang; Yilong Lu, „Classification and Tracking of Moving Objects from 77 GHz Automotive Radar Sensors,“ *IEEE,* 2018.

[16] Ankith Manjunath, Ying Liu, Bernardo Henriques, Armin Engstle, „Radar Based Object

Detection and Tracking for Autonomous Driving," *IEEE,* 2018.

[17] N. S. Chauhan, „DBSCAN Clustering Algorithm in Machine Learning," 2022. [Online]. Available: https://www.kdnuggets.com/2020/04/dbscan-clustering-algorithm-machine-learning.html.

[18] A. Sharma, „How to Master the Popular DBSCAN Clustering Algorithm for Machine Learning," [Online]. Available: https://www.analyticsvidhya.com/blog/2020/09/how-dbscan-clustering-works/.

[19] Nicolas Scheiner, Florian Kraus, Nils Appenrodt, Jürgen Dickmann & Bernhard Sick , „Object detection for automotive radar point clouds – a comparison," *SpringerOpen,* 2021.

[20] Nicolas Scheiner, Ole Schumann, Florian Kraus, Nils Appenrodt, Jürgen Dickmann, Bernhard Sick, „Off-the-shelf sensor vs. experimental radar - How much resolution is necessary in automotive radar classification?," *IEEE,* 2020.

[21] Alexey A. Belyaev, Timur A. Suanov, Igor O. Frolov, Dmitriy O. Trots, „The Range of Pedestrian Detection with Automotive Radar," *IEEE,* 2019.

[22] Hermann Rohling, Florian Folster, Henning Ritter, „Lateral velocity estimation for automotive radar applications," 2007.

[23] Hwee Yng, „How Automotive Radars Are Advancing Safety Features," [Online]. Available: https://www.keysight.com/blogs/en/tech/educ/2023/automotive-radar#:~:text=Velocity%20measurement%3A%20Due%20to%20the,relative%20velocity%20of%20the%20object..

[24] „Deviant Art," [Online]. Available: https://www.deviantart.com/bagera3005/art/Tesla-Cybertruck-891003965.

[25] Shendart, „iStock by Getty Images," [Online]. Available: https://www.istockphoto.com/de/vektor/autos-oben-ansicht-vektor-flach-fahrzeugtransporte-symbole-gesetzt-automobilauto-gm1136677792-302811980.

[26] „sklearn.cluster.DBSCAN," [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.cluster.DBSCAN.html.

[27] Florian Kraus, Nicolas Scheiner, Werner Ritter, Klaus Dietmayer, „The Radar Ghost Dataset – An Evaluation of Ghost Objects in Automotive Radar Data," *ARXIV*.

[28] The Radar Ghost Dataset – An Evaluation of Ghost Objects in Automotive Radar Data, „Florian Kraus, Nicolas Scheiner, Werner Ritter, Klaus Dietmayer," *IEEE,* 2021.

[29] JinYi Wei, Libo Huang, PanPan Tong, Bin Tan, Jie Bai, ZhiJun Wu, „Realtime Multi-target Vital Sign Detection with 79GHz FMCW Radar," *IEEE,* 2020.

[30] C.-C. C. Domenic Belgiovane, „Micro-Doppler characteristics of pedestrians and bicycles for automotive radar sensors at 77 GHz," *IEEE,* 2017.

# 14. APPENDIX

## 14.1 WINDOWS BUILD

### 14.1.1 Software prerequisites:

- CMake (3.26.0)
- Python (3.8.8)
- Make (3.81)
- Visual Studio 17: Select 'Windows 8.1 SDK' and 'Desktop development with C++' during installation. Doing this enables the x64 command prompt that will be used to build Carla [2].

The path to CMake, Python, and Make should be added to the Environment variables. Also, it's vital to use the same versions because maintaining the correct versions is crucial while building Carla in Windows. Version mismatches might lead to various Linker issues during the building of Carla which are difficult to solve.

### 14.1.2 Setting up Unreal Engine for version till and including 0.9.11.

- Create an account on Epic Games.
- Run Epic Games Launcher using the Epic Games account.
- Click on the *Unreal Engine* tab and install *version 4.24* or higher.
- After installation add the path to the Environment Variables. To do this, go to *Advanced System settings -> Environment Variables* and create a new variable by clicking '*New*'. Name the variable as '*UE4_ROOT*' and choose the path to the installation folder of the desired UE4 installation [2].

For Carla versions 0.9.12 and above Unreal Engine has a modified fork of 4.26, which needs to be cloned and built on the local PC—the steps for which are given below.

- Link the GitHub account to the Unreal Engine account, so that the fork of Unreal Engine can be downloaded.
- Clone the Carla Branch of Unreal Engine: '*git clone --depth 1 -b carla https://github.com/CarlaUnreal/UnrealEngine.git*'
- Run the configuration scripts '*Setup.bat*', and '*GenerateProjectFiles.bat*'.
- To compile the Unreal engine which has been specially made for Carla, the '*UE4.sln*' file needs to be opened with Visual Studio (verify the Visual Studios version based on the Carla version being built). Then

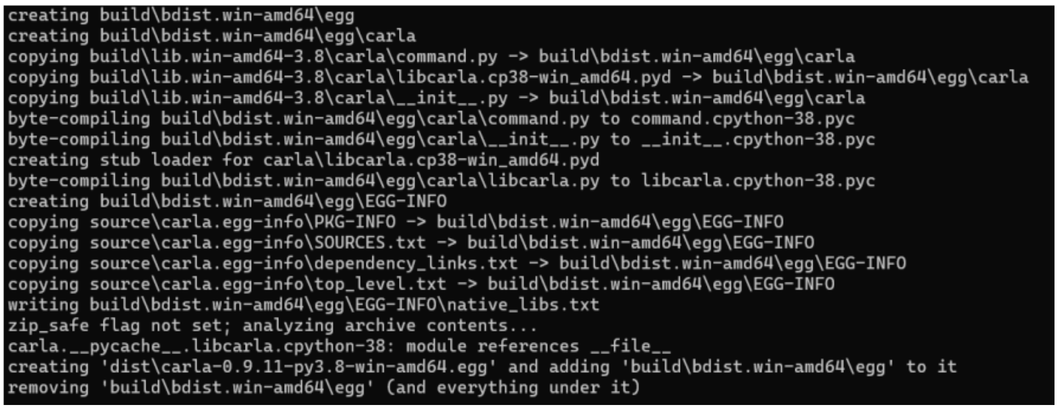
build the UE4 solution keeping these settings: ‘*Development Editor*’, ‘*Win64*’, and ‘*UnrealBuildTool*’.

- To verify the successful build, launch the *UE4Editor.exe* file, and if it has been built correctly all the *.uproject* files should be linked to it. If not, it needs to be added to the Environment variables by creating the variable ‘*UE4_ROOT*’ as mentioned above.

### 14.1.3 Build Carla

After setting the Unreal engine, environment variables, and all required software, Carla can now be built.

- To build Carla the commands must be executed via the ‘x64 Native Tools Command Prompt for VS 2019’.
- For the commands that build Carla to work, it needs to be run in the root CARLA folder, by navigating to it using the ‘cd’ command.
- Now, the PythonAPI Client can be compiled. The PythonAPI Client is required to control the simulation. It gives the tools required to create actors and scenarios which help to create scenarios customized to the desired test cases. Once the Client is compiled, scripts can be written to create scenarios and interact with the simulation. The command used to compile the PythonAPI Client is ‘make PythonAPI’.
- Once the PythonAPI is compiled it will result in 2 files *‘.egg’* and *‘.whl’* files.
  The ‘.egg’ has been used in this case. The creation of these files can be seen in Figure 14.1.
- The next step is to compile the server. The command ‘make launch’ is used next to compile and launch Unreal Engine. Figure 6.2 shows the result of the execution of the ‘make launch’ command.



```
creating build\bdist.win-amd64\egg
creating build\bdist.win-amd64\egg\carla
copying build\lib.win-amd64-3.8\carla\command.py -> build\bdist.win-amd64\egg\carla
copying build\lib.win-amd64-3.8\carla\libcarla.cp38-win_amd64.pyd -> build\bdist.win-amd64\egg\carla
copying build\lib.win-amd64-3.8\carla\__init__.py -> build\bdist.win-amd64\egg\carla
byte-compiling build\bdist.win-amd64\egg\carla\command.py to command.cpython-38.pyc
byte-compiling build\bdist.win-amd64\egg\carla\__init__.py to __init__.cpython-38.pyc
creating stub loader for carla\libcarla.cp38-win_amd64.pyd
byte-compiling build\bdist.win-amd64\egg\carla\libcarla.py to libcarla.cpython-38.pyc
creating build\bdist.win-amd64\egg\EGG-INFO
copying source\carla.egg-info\PKG-INFO -> build\bdist.win-amd64\egg\EGG-INFO
copying source\carla.egg-info\SOURCES.txt -> build\bdist.win-amd64\egg\EGG-INFO
copying source\carla.egg-info\dependency_links.txt -> build\bdist.win-amd64\egg\EGG-INFO
copying source\carla.egg-info\top_level.txt -> build\bdist.win-amd64\egg\EGG-INFO
writing build\bdist.win-amd64\egg\EGG-INFO\native_libs.txt
zip_safe flag not set; analyzing archive contents...
carla.__pycache__.libcarla.cpython-38: module references __file__
creating 'dist\carla-0.9.11-py3.8-win-amd64.egg' and adding 'build\bdist.win-amd64\egg' to it
removing 'build\bdist.win-amd64\egg' (and everything under it)
```

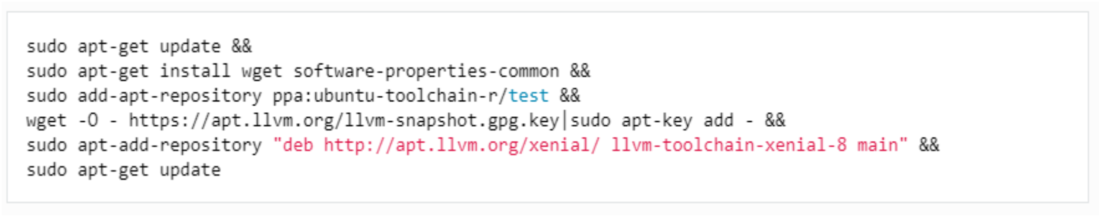
Figure 14.1 Shows the results of a successful build.

## 14.2 LINUX BUILD

### 14.2.1 Software Prerequisites:

- Python (3.8.8)
- Cmake
- Clang

The best way to install all these requirements is to run the commands in Figure 14.2.



```
sudo apt-get update &&
sudo apt-get install wget software-properties-common &&
sudo add-apt-repository ppa:ubuntu-toolchain-r/test &&
wget -O - https://apt.llvm.org/llvm-snapshot.gpg.key|sudo apt-key add - &&
sudo apt-add-repository "deb http://apt.llvm.org/xenial/ llvm-toolchain-xenial-8 main" &&
sudo apt-get update
```

Figure 14.2 Commands to install the requirements to build Carla. **Image Source**: Carla Docs *[3]*

### 14.2.2 Building Unreal Engine

The next step is to get the Unreal Engine ready. For Carla version 0.9.11 Unreal Engine version 4.24 is used. For different versions of Carla use the respective Unreal Engine version [3].

- Clone *Unreal Engine 4.24* on the local computer:- '*git clone --depth=1 -b 4.24 https://github.com/EpicGames/UnrealEngine.git ~/UnrealEngine_4.24*'
- Get to the directory where UE 4.24 has been cloned and download the patch for Unreal Engine.
  '*wget https://carla-releases.s3.eu-west-3.amazonaws.com/Linux/UE_Patch/430667-13636743-patch.txt 430667-13636743-patch.txt*'
  *'patch --strip=4 < 430667-13636743-patch.txt'*
- Build the unreal engine using this command:
  '*./Setup.sh && ./GenerateProjectFiles.sh && make*'
- Set the Unreal Engine environment variable by using the command given below. This will help Carla find Unreal Engine during the launch.
  ' *export UE4_ROOT=~/UnrealEngine_4.24*'
- The variable, UE4_ROOT, should also be added to' ~/.bashrc' or '~/.profile' for it to be accessible session-wide.
  This can be done by :
  (a) Open *'~/.bashrc'* using command '*gedit ~/.bashrc*'.
  (b) Write the environment variable in the *~/.bashrc* file: '*export UE4_ROOT=~/UnrealEngine_4.24*'.
  (c) Now, save the file and reset the terminal.