UWAFT Sensor Fusion for Multi-Vehicle Detection

Final Report Submission for ME 599 Hybrid Vehicles

22 December 2021

Ethan Thompson

20717462

Department of Engineering

Table of Contents

1 Introduction

This report summarizes the findings and modifications made to the sensor fusion algorithm of UWAFT's hybrid-electric vehicle project as part of the EcoCAR Mobility Challenge (EcoCAR).

1.1 Competition Requirements

EcoCAR is a four-year long competition hosted by the US Department of Energy, challenging 11 University teams across North America [1]. The University of Waterloo competition team, UWAFT, is developing vehicle propulsions systems and autonomous solutions for a 2019 Chevrolet Blazer.

The relevant competition requirement is the implementation of SAE Level 2 automation. As described in Figure 1, level 2 autonomy provides driver support features such as Adaptive Cruise Control (ACC), Automated Emergency Braking (AEB), and Lane Centering (LC). To achieve this, the Blazer has been outfitted with sensing hardware such as the Intel Mobileye 6 and Bosch Radar. Inputs from these sensors will be consolidated by the sensor fusion algorithm in real-time to provide these autonomous driver support features.

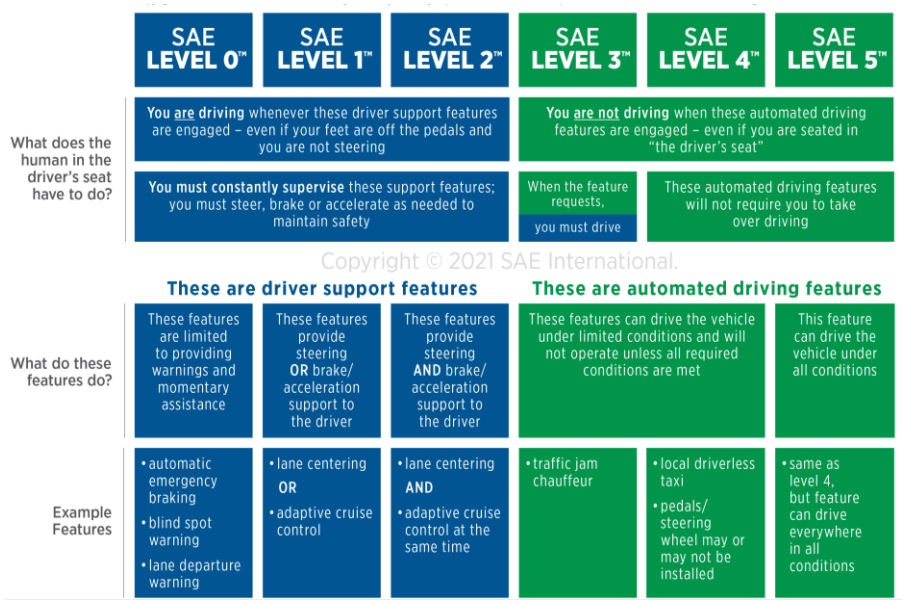

Figure 1 Levels of vehicle autonomy defined by SAE

1.2 UWAFT Integration

The sensor fusion algorithm lives in *kaiROS*, the primary code repository for the Connected and Autonomous Vehicles (CAV) team. During testing, the repository will be cloned onto the Intel Tank, the Blazer's on-board computer, to run the sensor fusion nodes in real-time. The code described in this report can be found on branch *UWAFT-3661* in the *kaiROS* repository. When the code in this branch has been fully validated against the OxTS ground truth data it can be merged into the main branch to be used for real-time CAV tasks.

1.3 Background

The objective of a sensor fusion algorithm is to take inputs from multiple sources and combine them to get a better understanding of the system. For UWAFT's CAV team, this means taking inputs from the Mobileye and radars, and determining the position and trajectories of surrounding objects. In any real-world system, there will be many factors that need to be accounted for in the sensor fusion algorithm. For instance, night driving will produce a different sensor response than daytime driving, rain may instigate false positive readings, and vehicles come in different shapes and sizes. The following sections will discuss the design considerations for successful sensor fusion.

1.3.1 Sensor Hardware

The Blazer is equipped with four sensors, an Intel Mobileye which is located at the top center of the windshield, and three Bosch radars which are located at the front corners and the center of the grille.

The Mobileye has an onboard processor which runs its own computer vision software to identify objects in real time. It extracts relevant object features such as the object's position relative to the ego vehicle, dx and dy, as well as the relative velocity along the longitudinal axis, vx. The Mobileye can detect up to 10 objects as far as 150 m within a 38° field of view (FOV) [2].

The front radar is the Bosch Evo14 mid-range radar (MRR) with a range of 160 m and a FOV of 90° [3]. The corner radars are both rear MRR with a range of 100 m and FOV of 150 degrees. The radar will transmit a 76-77 GHz radio frequency signal which will reflect off nearby objects. The time of flight for each signal can be used to calculate the distance. Each radar can track up to 32 objects.

A Bird's eye plot (BEP) is used to visualize the coverage of the sensors on a vehicle. The BEP of the Blazer is shown in Figure 2 using Driving Scenario Designer (DSD) – a software used to simulate driving scenarios and collect realistic sensor data.

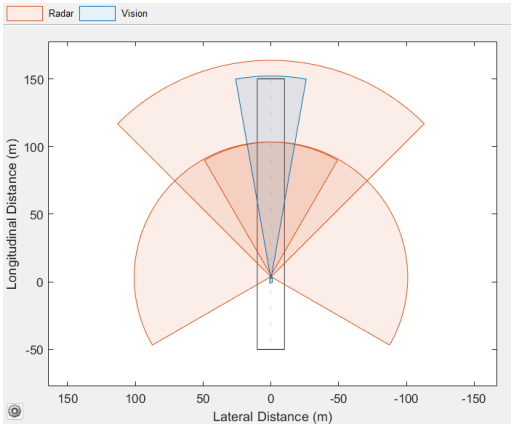

Figure 2 BEP of the Chevrolet Blazer with Mobileye (blue) and radars (orange)

1.3.2 Sensor Fusion Theory

The sensor fusion workflow can be decomposed into 5 main components, described in Figure 3 [4]. First, the sensors will collect observations about the surrounding environment. Second, each data point is assigned to an object. This data point can be a real object, sensor noise, or roadside infrastructure that does not require tracking. The decision to keep or discard a piece of data is determined by step three, track maintenance. Once the observations have been appropriately filtered, they will be consolidated in the fourth step, estimation filtering. This node looks at all the data points for each object and statistically determines the most probable location of the object. Finally, a gating step is performed to discard any values that exceed pre-defined thresholds.

All sensor fusion algorithms will implement some combination of these steps depending on the use-case. For multi-vehicle detection (MVD), assignment and track maintenance become the most important steps. This is because it is challenging to decouple signals from objects very close to one another. Therefore, the primary focus for MVD is data association – associating the detections with the correct objects.

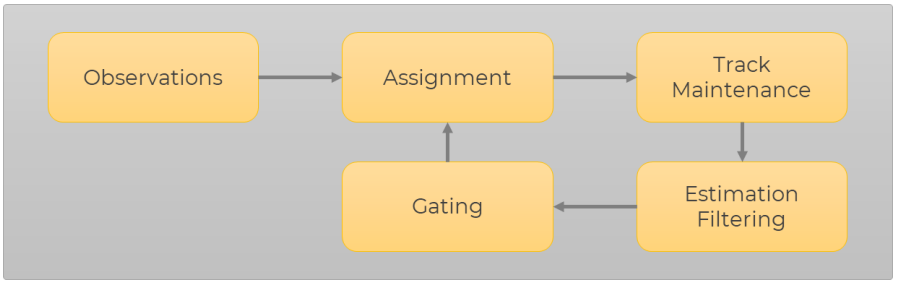

Figure 3 Sensor fusion workflow for multi-object tracking

1.3.3 UWAFT Sensor Fusion Architecture

The specific UWAFT implementation of the sensor fusion workflow is shown in Figure 4. The sensor fusion architecture is implemented in C++ and ROS. The architecture consists of three nodes:

1. **Data Association**: This node receives object data from the `radar_object_data` and `mobileye_object_data` ROS topics. It performs a gating step to filter out poor detections, then assigns each detection to an object.
2. **Kalman Filter**: This node receives the detected objects from Data Association and performs estimation filtering. Based on the object's historical data, it will make a statistical prediction for the future state of the object's location.
3. **Environment State**: This node receives the state-estimated object from Kalman Filter and publishes the object data to be used for ACC, AEB, and LC.

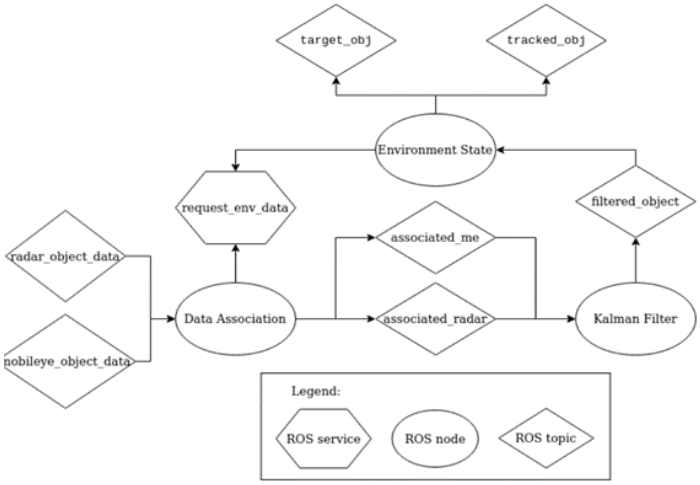

Figure 4 Sensor Fusion Architecture

2 Problem Definition

Currently, the sensor fusion architecture is built only for single vehicle detection. Therefore, track maintenance and data association are simplified. If a detection falls within a range of valid dx, dy, vx values, then it is used to update the object's state. Otherwise, it is classified as sensor noise and discarded. However, multi-vehicle detection introduces a more complex data association problem. There can be any number of detectable objects (up to 32 objects for the Blazer) in the surrounding environment, and each detection must be correctly assigned to the associated object. Multi-vehicle detection is a necessary step towards achieving SAE Level 2 automation because the ego vehicle must know about *all* surrounding objects, not just a single target vehicle to implement ACC, AEB, and LC.

Therefore, there is a need to enhance the current sensor fusion algorithm to accurately detect and track multiple objects, in real-life driving scenarios.

3 Solution and Analysis

At the beginning of this project, no ground truth data had been collected with multiple vehicles. As a result, DSD was used to simulate test environments with multiple vehicles to collect sensor data. This sensor data was then fed into the sensor fusion pipeline to generate pseudo-realistic outputs of target object tracking. The code discussed in this section can be found on branch *UWAFT-3661* in the *kaiROS* repository.

First, the procedure for creating a scenario in DSD, generating sensor data, running it through sensor fusion, and plotting it against ground truth data is provided in section 3.1. Results for single and multi-vehicle detection will be compared. Second, the sensor fusion sandbox created in Python will be discussed in section 3.2. The sandbox functions identically to the ROS/C++ sensor fusion but is written in Python for easier debugging and data visualization. The modifications made to the sensor fusion sandbox and the relevant data association parameters will be investigated.

3.1 Running Sensor Fusion on Simulated Data

3.1.1 Procedure

This section should be used as a guideline for future simulations using DSD and generating outputs from the sensor fusion pipeline.

1. Create a scenario in DSD

DSD can downloaded as a MATLAB application under the “apps” tab. The MATLAB documentation for DSD is a great starting point for generating scenarios. It is possible to simulate Waterloo streets and infrastructure using the open street map (.osm) files that can be imported directly into DSD [5].

2. Import the Blazer sensor configuration into the scenario

The Blazer’s sensor configuration has been saved in a file named `blazer_sensor_config.mat` with the exact position and parameters for each sensor. The Mobileye, and three radars will automatically

be mounted to the ego vehicle upon import. Figure 5 shows the dialogue box that appears asking for the sensor configuration.

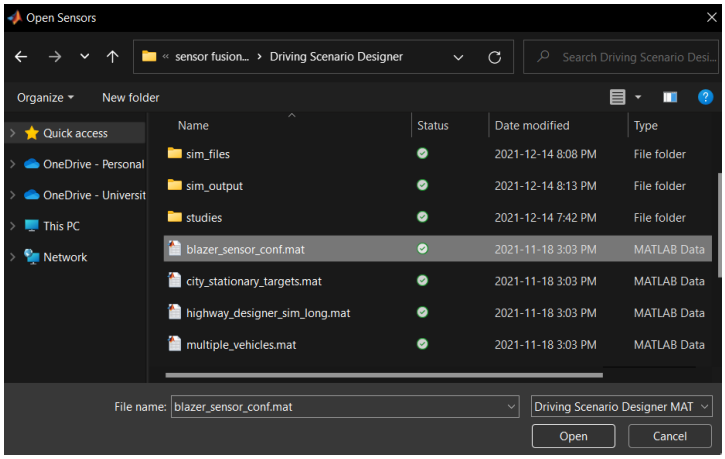

Figure 5 *Import blazer sensor configuration into DSD scenario*

3. Run the scenario to completion

Once the scenario has been designed, click the *Run* button to start the simulation (Figure 6). For all the sensor data to be captured, the simulation must be run to completion.

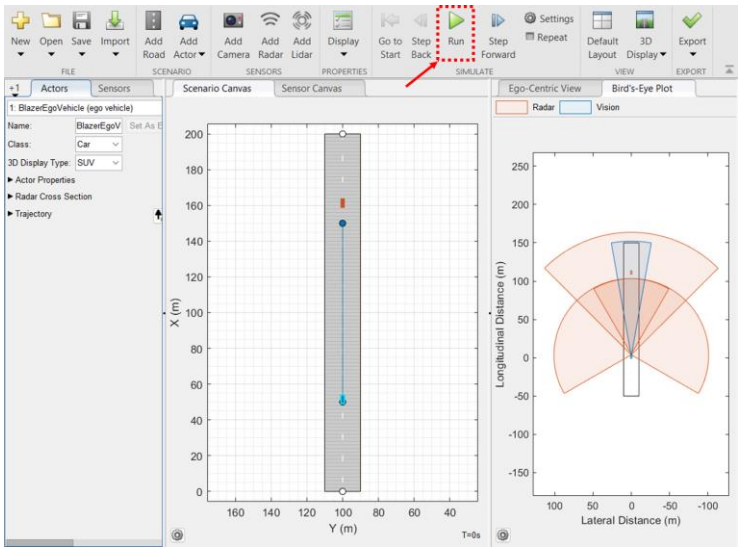

Figure 6 DSD scenario ready to be run

4. Export the sensor data when the simulation has completed.

Use the naming convention: `sensor_data_[n]`, where n is the scenario number. An example is shown in Figure 7.

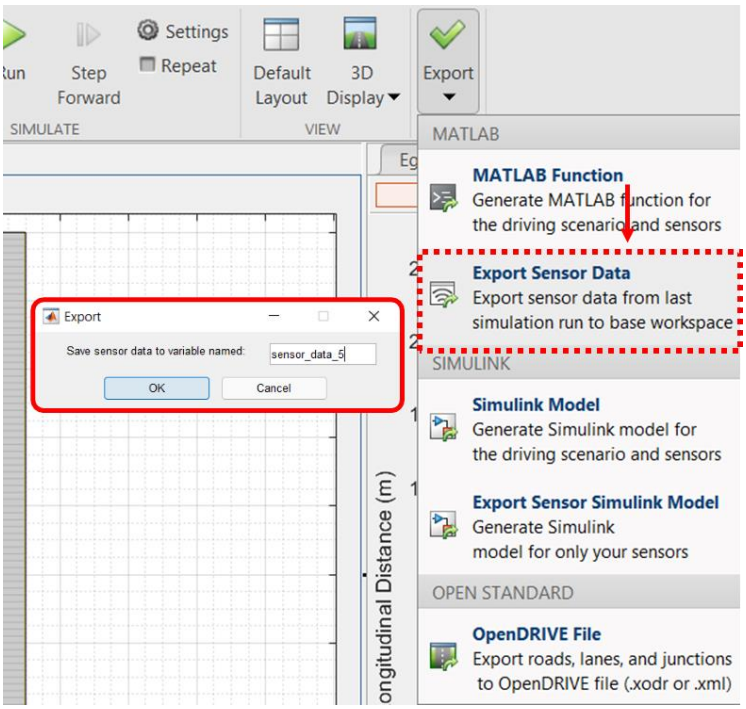

Figure 7 Exporting sensor data from DSD

5. Run the `driving_scenario_data_extraction.m` file to generate the CSV file with sensor outputs

When the sensor data is exported in step 4, it will automatically be imported into the MATLAB workspace. Four changes must be made to this file, as shown in Figure 8.

1. Line 3: This line will save your raw sensor data into the `/sim_output` folder. Make sure to name the file corresponding to the scenario name.
2. Line 5: This will convert the raw sensor data from DSD into a table that can be iterated over
3. Line 80: This is the CSV file that is generated and will contain all the object detections.
    - ***NOTE:*** *This will write the CSV file in append mode – if this script is run more than once, the values will be appended to the bottom of an existing file which may give erroneous results. Therefore, create a new file name or delete the existing file before running.*

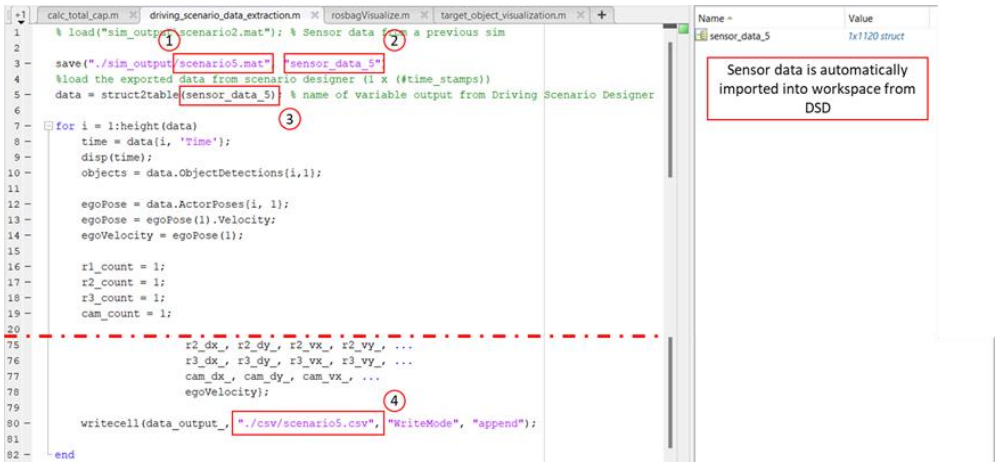

Figure 8 `driving_scenario_data_extraction.m` will convert the raw sensor data into a CSV file

6. Copy the generated CSV file into `kaiROS/src/simulation_sensor_data/csv` on the Intel Tank

Once the CSV file has been created in `kaiROS/sensor fusion testing/Driving Scenario Designer/csv` it must be copied over to the *kaiROS* repo on the intel tank, shown in Figure 9. This is because the sensor fusion pipeline can only be run on a system configured with ROS.

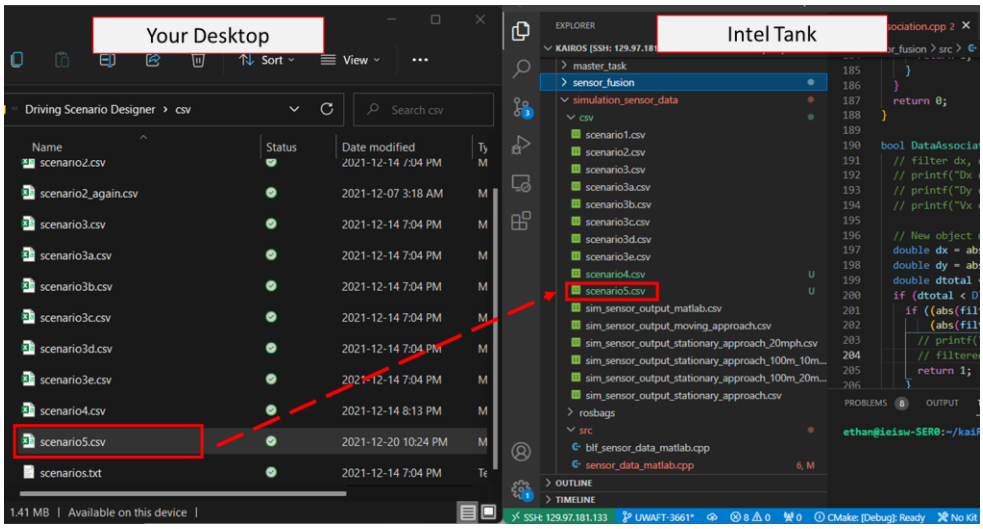

Figure 9 Copy CSV file from your desktop to the Intel Tank

7. Run `kaiROS/src/simulation_sensor_data/src/sensor_data_matlab.cpp`

The sensor fusion pipeline works by receiving messages from the CAN bus via ROS topics to read the object data from Mobileye and radars. These messages can be obtained by converting the object data

in the CSV file into a rosbag with ROS-compatible messages. The *sensor_data_matlab.cpp* file can be run to generate the rosbag. First, two lines (line 108, 118) must be updated to specify the input CSV file, and the name of the rosbag file, shown in Figure 10. The file can be run by typing the following commands into the command line:

```
source devel/setup.bash
catkin build
rosrun simulation_sensor_data sensor_data_matlab
```

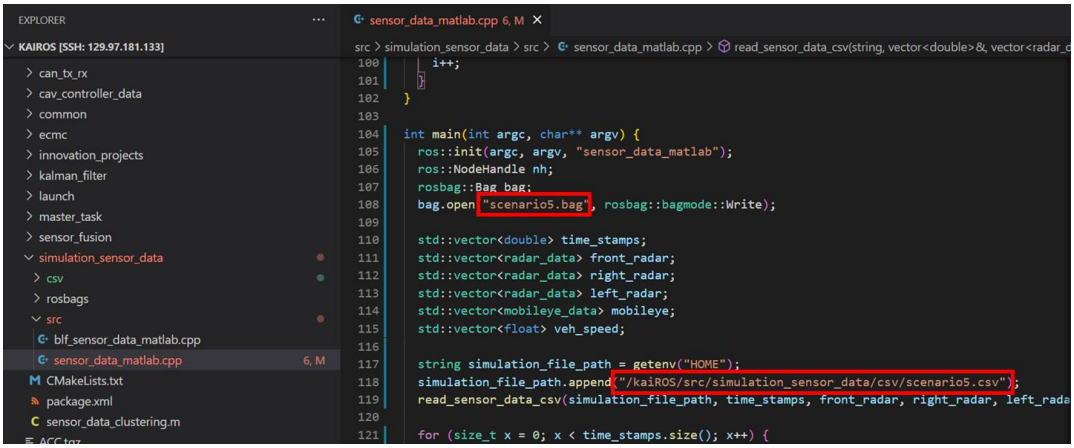

Figure 10 sensor_data_matlab.cpp converts a CSV file into a BAG file

8. Add the generated rosbag into *kaiROS/src/simulation_sensor_data/rosbags*

The .bag file should be generated in the *kaiROS* workspace. Move this file into the */simulation_sensor_data/rosbags* folder to stay organized (Figure 11).

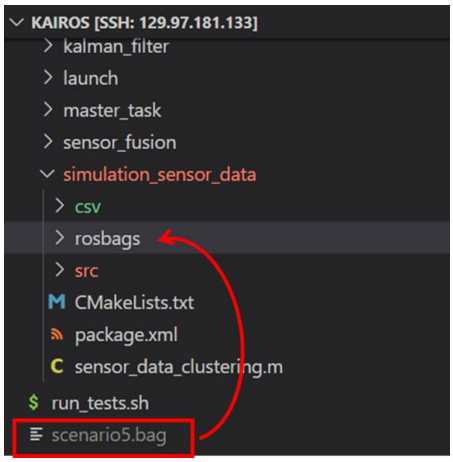

Figure 11 Move the generated rosbag file into the rosbags folder

9. Run the sensor fusion pipeline
    a. Record sensor fusion output into a rosbag

        In a terminal, run `rosbag record -o scenario[n]-output.bag -a` where n is the scenario number.

    b. Run the sensor fusion nodes

        i. Run `rosrun sensor_fusion data_association`

        ii. Run `rosrun sensor_fusion environment_state`

        iii. Run `rosrun kalman_filter kf_node`

    c. Play the rosbag generated in step 6

        In a terminal, run `rosbag play src/simulation_sensor_data/rosbags/scenario[n].bag` where n is the scenario number.

10. Download the rosbag to your local desktop

Once the rosbag has completed, kill the rosbag record terminal and download the output rosbag to your local desktop into `kaiROS/sensor fusion testing/Driving Scenario Designer/bag`, as shown in Figure 12.

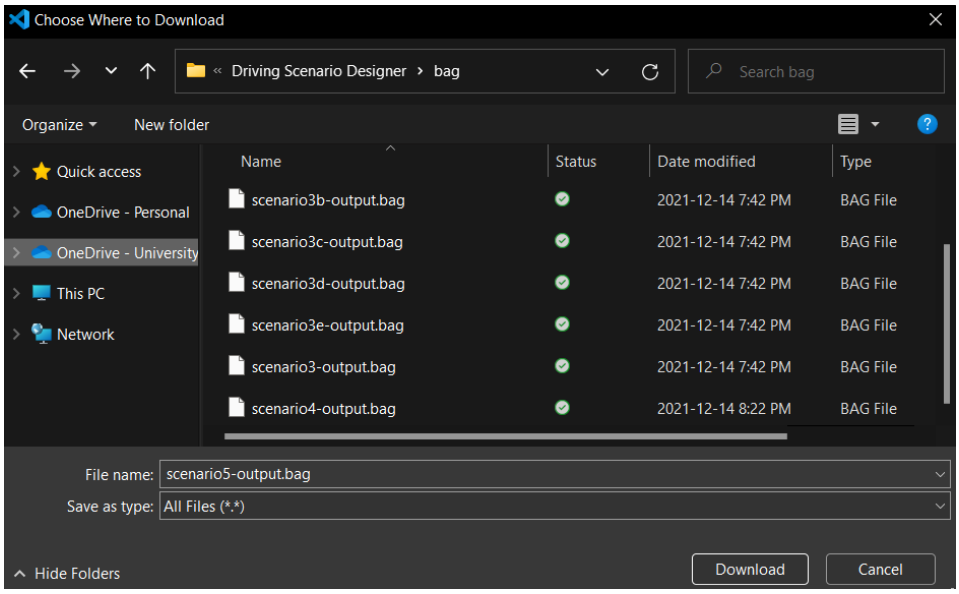

Figure 12 Download sensor fusion output to desktop

11. Generate sensor fusion ground truth data

The raw sensor data generated from the DSD simulation that was saved in step 5 can now be converted to ground truth data. Navigate to `kaiROS/sensor_fusion_testing/Simulation Testing/sim_ground_truth_to_mat.m` and replace line 4 and line 23 with the appropriate scenario number as shown in Figure 13. Run the script to generate the ground truth data.

```
sim_ground_truth_to_mat.m
1 -   clear
2 -   clc
3
4 -   load('../Driving Scenario Designer/sim_output/scenario5.mat');
5 -   detections = struct2table(sensor_data_5);
6
7 -   ground_truth = struct('Time', [], 'Objects', []);
8
9 -   for i = 1:height(detections)
10
11 -      ground_truth(i).Time = detections{i,'Time'};
12 -      actors = detections.ActorPoses{i,1};
13 -      ego = actors(1,1);
14
15 -      for j = 2:size(actors,1)
16 -          ground_truth(i).Objects(j-1,:) = [actors(j,1).Position(1,1) - ego.Position(1,1)
17                                            actors(j,1).Position(1,2) - ego.Position(1,2)];
18
19 -      end
20
21 -   end
22
23 -   save('../Driving Scenario Designer/ground_truth/scenario5_ground_truth.mat', 'ground_truth');
```

Figure 13 Generate ground truth data from the DSD raw sensor output

12. Plot ground truth data

The sensor fusion output can be compared to the ground truth data by modifying the `target_object_visualization.m` file in `kaiROS/sensor_fusion_testing/Simulation Testing` with the appropriate scenario-output.bag file and the ground truth file generated in step 11 (Figure 14). Several functions (in the `/functions` folder) were added to simplify visualization.

```
target_object_visualization.m
1 -   clear all
2 -   clc
3 -   addpath('functions\')
4     %% Script to plot target_obj output against ground truth target data (currently for simulatio
5     % Load rosbag from SF output and ground truth
6
7 -   bag = rosbag('../Driving Scenario Designer/bag/scenario5-output.bag');
8 -   load('../Driving Scenario Designer/ground_truth/scenario5_ground_truth.mat'); %ground_truth
9
10    % Select tracked obj topic
11 -  tracked_obj_bag = select(bag, 'Topic', 'tracked_obj');
12
13    % Save rosbag as struct
14 -  tracked_obj_struct = cell2mat(readMessages(tracked_obj_bag, 'DataFormat', 'struct'));
15
16    %% Visualize ground truth vs. sensor fusion output
17 -  Timestamps = [ground_truth(1,:).Time]';
18 -  num_timestamps = size(Timestamps, 1);
19 -  num_objects = size(ground_truth(1).Objects, 1);
20 -  ground_truth_distance = zeros(num_timestamps, num_objects);
21 -  target_object_distance = [];
```

Figure 14 Run target_object_visualization.m to compare sensor fusion against ground truth data

3.1.2 Results

Three scenarios in DSD were created for this project. The results for sensor fusion vs. ground truth are shown below.

1. Scenario 1: Static target vehicle, ego vehicle approaches directly behind at 16 km/h.

Figure 15 shows the sensor fusion output vs. ground truth for scenario 1. There is noise in the output, but in general, sensor fusion does a good job at tracking the position of the target object.

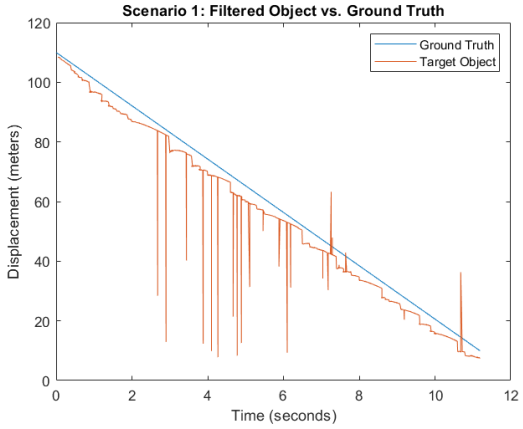

Figure 15 Scenario 1 ground truth vs. sensor fusion output

2. Scenario 2: Static target vehicle, ego vehicle approaches behind starting at 40 km/h with constant deceleration to 0 km/h.

Figure 16 shows the object tracking performance of a slightly more complex scenario. There is considerable noise as the ego vehicle approaches the target object (within 10 meters).

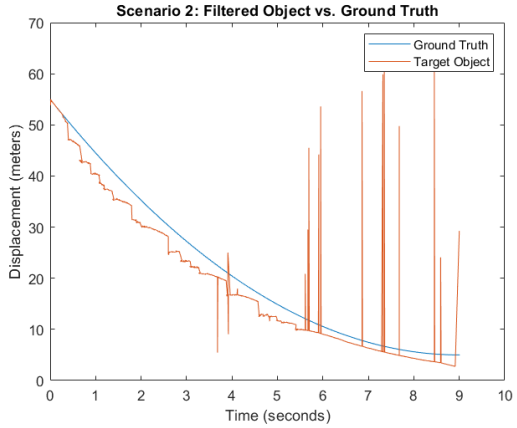

Figure 16 Scenario 2 ground truth vs. sensor fusion output

3. Scenario 3: 3 target vehicles, left at 25 m/s, right at 10 m/s, center at 17.5 m/s. Ego vehicle in center lane traveling at a constant speed of 15 m/s.
    a. Scenario 3a: Scenario 3 but only with the center and right vehicle

For scenario 3, the sensor fusion code had to be modified to output three objects. In particular, the **ME_OBJ** parameter was set to 3; this value and other object tracking parameters can be found in `kaiROS\src\sensor_fusion\include\data_association.h`. These will be discussed further in section 3.2.2.

Figure 17 shows the poor performance of the current sensor fusion algorithm for multi-object tracking. Clearly, there is insufficient logic in the data association node needed to maintain multiple tracks and distinguish between multiple objects.

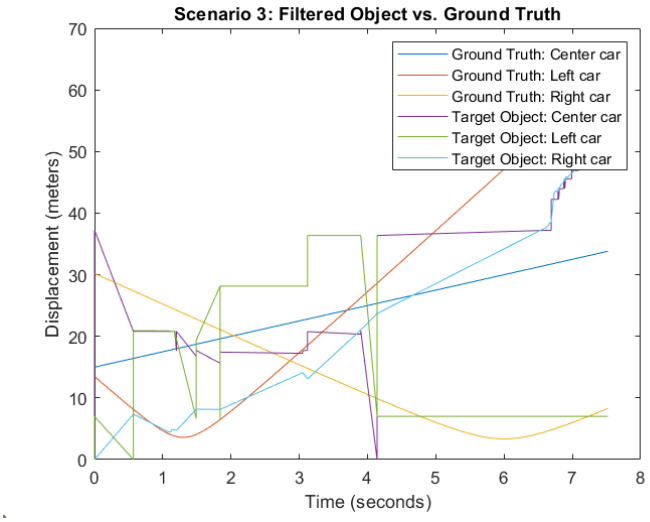

Figure 17 Scenario 3 ground truth vs. sensor fusion output

3.1.3 Identifying accurate sensor ranges

One improvement that was made to sensor fusion was identifying the ranges where each sensor was most accurate. For instance, in Figure 16, there is clearly a lot of noise at close range. Is this due to Mobileye noise or radar noise?

Figure 18 provides a breakdown for the ground truth sensor readings for the Mobileye and Radar in that scenario. Evidently, the Mobileye is more accurate at farther ranges, beyond 13 meters, while the front radar is most accurate within 30 meters. Therefore, these values were used as thresholds for object tracking – detections are considered only if it is beyond 13 meters for the Mobileye and within 30 meters for the radar.

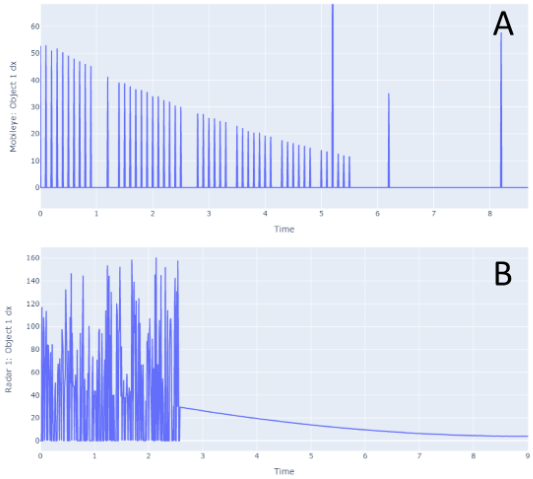

Figure 18 Ground truth sensor readings for (A) Mobileye (B) Front Radar for scenario 2

After implementing these thresholds, the noise from the sensor fusion output was decreased significantly (Figure 19).

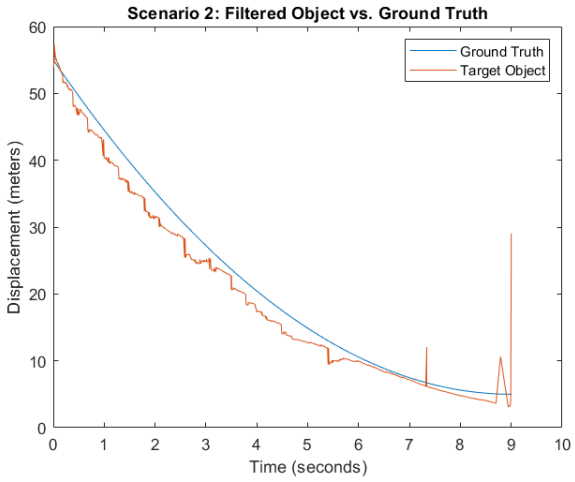

Figure 19 Scenario 2 sensor fusion output after implementing new threshold values

Unfortunately, these thresholds did not change the accuracy of the multi-object detection. Modifications to the code for MVD are discussed in section 3.2.

3.2 Python Sensor Fusion Sandbox

ROS/C++ does not lend itself to easy debugging and data visualization. For large structural changes that are required for multi-vehicle detection, an exorbitant amount of time would be needed to make the appropriate changes to the current ROS/C++ infrastructure. A language such as Python, which is meant for rapid prototyping, quick debugging, and data visualization could be used to create a sandbox environment for sensor fusion. This is precisely what was done for this project, and this section will document its use for future work on sensor fusion. The code can be found in `kaiROS/python_src` or at https://github.com/uwaft/kaiROS/tree/UWAFT-3661/python_src.

3.2.1 Using the sensor fusion sandbox

1. Setup the python environment

Ensure python 3.8+ is installed on your system. Please visit https://www.python.org/downloads/ to download the software; make sure to add Python to the PATH when prompted. These instructions are also provided in the `README.md` file in the `kaiROS/python_src` folder.

   a. Create a virtual environment in the `kaiROS/python_src` folder
      i. In the terminal, run `python -m venv .venv` to create a new python virtual environment
      ii. In the terminal, run `.\.venv\Scripts\activate` to activate the virtual environment on Windows; the command will be different for Mac users.
   b. Install required packages
      i. In the terminal, run `pip install -r requirements.txt` to install the required packages

2. Modify the `scenario_name` variable in the `main.py` script as shown in Figure 20 . This should follow the naming convention of the scenario file names (e.g., 'scenario1', 'scenario2', 'scenario[n]').

```python
from utils.preprocessing import label_columns, to_long_df, create_stream

if __name__ == '__main__':
    # Read in the raw sensor data
    scenario_name = 'scenario3'  # TODO: Change this value based on the scenario you want to run
    root = r'..\sensor fusion testing\Driving Scenario Designer\csv'
    data = pd.read_csv(Path(root)/Path(scenario_name).with_suffix('.csv'))

    # ================== Pre-process data ==================
    print('\n', '='*100, '\n PRE-PROCESSING DATA \n', '='*100, '\n')

    # 1. Label data columns
    print('\n1. Labeling data')
    data = label_columns(data)
```

Figure 20 Change the scenario name to run the specified scenario in the sensor fusion sandbox

3. Run `main.py` in your terminal to run the sensor fusion sandbox. The terminal should output the sensor fusion steps shown in Figure 21, and the sensor fusion output plots should automatically be generated in the default web browser.

```
(.venv) C:\Users\ectho\OneDrive - University of Waterloo\Fourth Year\ME 599\kaiROS\python_src>python main.py
 ====================================================================================================
 PRE-PROCESSING DATA
 ====================================================================================================


1. Labeling data

2. Removing side radar data

3. Converting to long format

4. Creating scenario data stream


 ====================================================================================================
 RUNNING SENSOR FUSION
 ====================================================================================================

100%|████████████████████████████████████████████████████████████| 752/752 [00:01<00:00, 384.07it/s]

 ====================================================================================================
 PLOTTING DATA
 ====================================================================================================

100%|████████████████████████████████████████████████████████| 8038/8038 [00:22<00:00, 354.88it/s]
100%|████████████████████████████████████████████████████████| 7908/7908 [00:21<00:00, 365.68it/s]
100%|████████████████████████████████████████████████████████| 7908/7908 [00:16<00:00, 482.99it/s]

 ====================================================================================================
 SENSOR FUSION COMPLETE
 ====================================================================================================
```

Figure 21 Sensor fusion sandbox terminal output

3.2.2 Code modifications and results

The plots in section 3.1 show good single-vehicle detection with the old sensor fusion pipeline, but poor performance for MVD. Figure 22A shows a plot generated by the sensor fusion sandbox for scenario 3, with three target objects. The plot shows that up to 5 objects were detected and tracked by sensor fusion. Clearly, there is a lot of noise, and the object tracks are incoherent.

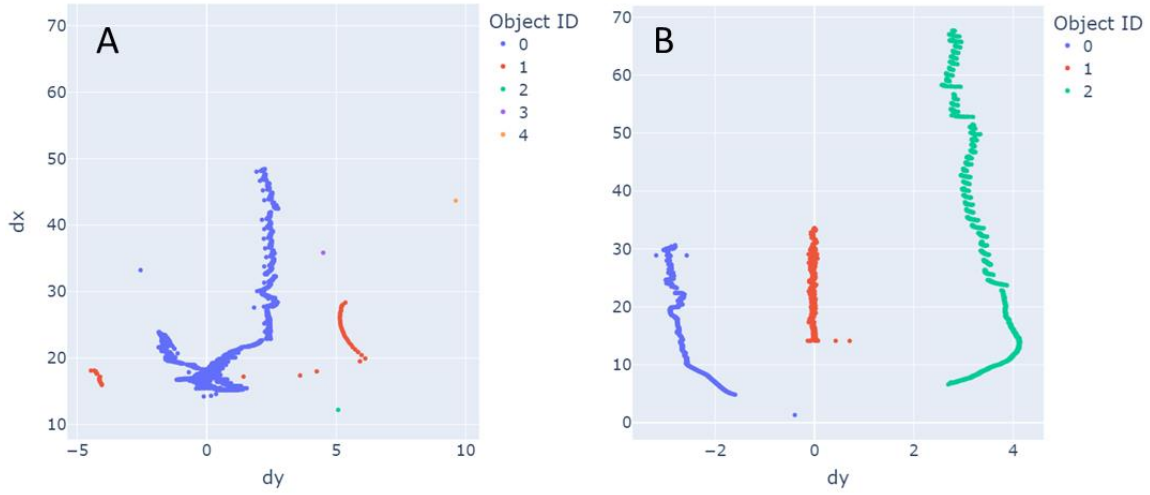

Figure 22 Scenario 3 object tracking with (A) old sensor fusion pipeline (B) new sensor fusion

Figure 22B shows the sensor fusion output with the new sensor fusion pipeline. The performance is significantly better than the old sensor fusion, and it correctly identifies the three target vehicles. The modifications that were made in the Python sensor fusion sandbox include:

1. Tuning the object matching parameters

The object matching parameters are used to determine if a new detection is a new object or matches an existing object. The parameter values used in the old vs. new sensor fusion are shown in Table 1. To explain these parameters, consider a scenario where the old sensor fusion is currently tracking an object with properties $\{dx = 50, dy = 0.2, vx = 10\}$. A new detection arrives from the radar characterized by $\{dx = 100, dy = -4, vx = 14\}$. The matching criteria are as follows

- $\Delta dx <$ DX_TOL $\rightarrow$ 50 m $<$ 100 m **(TRUE)**
- $\Delta dy <$ DY_TOL $\rightarrow$ 4.2 m $<$ 5 m **(TRUE)**

- $\Delta vx < \text{VX\_TOL} \rightarrow 4 \text{ m/s} < 20 \text{ m/s}$ **(TRUE)**

Since all matching criteria are satisfied, this new detection will be used to update the position of the matched object. However, by looking at the values, one would not be convinced that this detection corresponds to the tracked object; it was 4 m to the left and 50 meters ahead of the tracked vehicle. Clearly, the old sensor fusion's matching rules leave too much room for error. Therefore, the sandbox was used to finely tune each of the matching parameters. The new parameters are presented in Table 1. These parameters are small enough to distinguish between two close vehicles, but large enough to prevent a single object from being split up into two different detections.

Table 1 Old and new sensor fusion matching parameters

Parameter	Old sensor fusion	New sensor fusion
DX_TOL	60	8
DY_TOL	5	2
VX_TOL	20	N/A
POTENTIAL_THRESHOLD	5	10
SECONDS_TO_DELETE	3	1

The **POTENTIAL_THRESHOLD** is a parameter that determines the number of detections are required to start a new object track. For instance, we must see 10 detections around the same location before we start a new object track. The higher this value is, the less susceptible sensor fusion is to noise, because it needs more data to be confident that there is truly an object at that location. However, increasing this value too much may prevent an object from being tracked at all, especially if it is at the periphery of the sensor's FOV where the detections are sparse.

The **SECONDS_TO_DELETE** parameter determines the number of seconds to wait before discarding a potential object track. It works closely with the **POTENTIAL_THRESHOLD** parameter. Using the new sensor fusion parameters in Table 1, **10** detections must be made within **1** second for a new object track to be started. Increasing this value will increase the signal-to-noise ratio but may filter out real objects that have sparse detections.

Finally, **VX_TOL** was completely removed as a matching parameter in the new sensor fusion because it was found to be very susceptible to noise and decreases the performance of MVD.

2. Performing lane determination in data association instead of the Kalman filter.

The most important change to sensor fusion was assigning a lane to each detection, and then using that lane to match the detection to an object in environment state. Previously, lane determination was done in Kalman filter after the objects had already been matched. This does not work well for MVD because there is no spatial localization during the data association phase. Instead, data association would blindly assign detections to objects that are within DY_TOL. We know, however, that vehicles will cleanly fall within certain ranges of dy based on which lane it is in. This logic was implemented into the new sensor fusion code.

3. Using a weighted average to determine the dx and dy values of a new object

When creating a new object track, as discussed above, a certain number of similar detections must be made within a certain timeframe. When these criteria are met, the most recent detection will be passed to Kalman filter for state estimation. However, this last detection is not necessarily representative of all the detections that came before. Instead, it would be better to pass the average point across all the detections to Kalman filter. The new sensor fusion takes the second approach and calculates the weighted average of detections to pass to Kalman filter. This is illustrated in Figure 23.

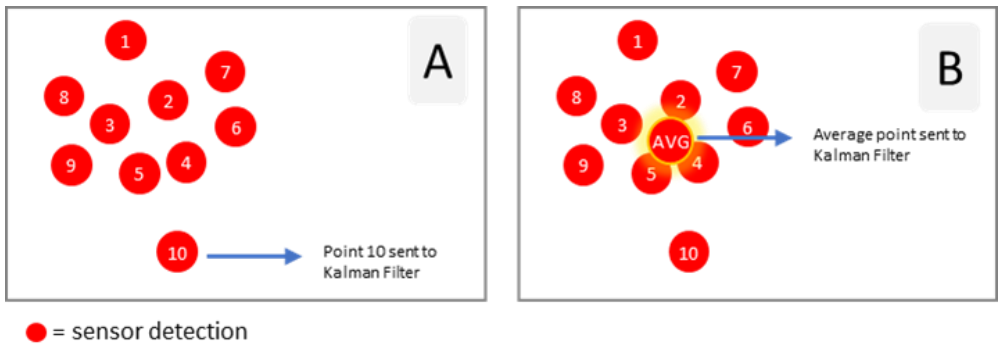

Figure 23 Object matching criteria for (A) old sensor fusion (B) new sensor fusion using weighted average

After implementing these three changes, the new sensor fusion code in the Python sandbox is much more robust for MVD. This is shown in Figure 24 where sensor fusion is compared between the old and new implementations. Clearly the modifications to the code mentioned in this section have improved the performance of both single-vehicle and multi-vehicle detection. There are, however, many modifications that are required to handle real-world driving scenarios. These will be discussed in the next section.

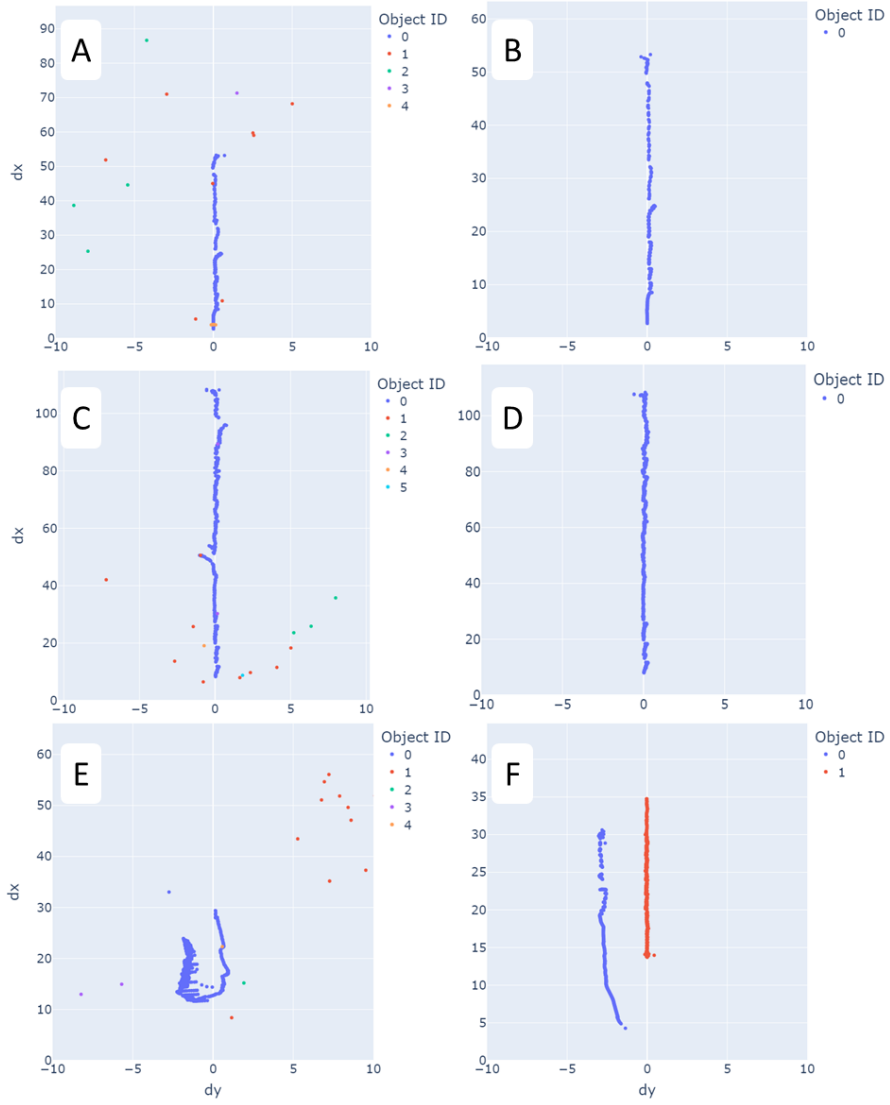

Figure 24 Old sensor fusion results (LEFT) and new sensor fusion results (RIGHT) for scenario 1 (A,B) scenario 2 (C,D) and scenario 3a (E,F)

4 Conclusion

The objective of this project was to modify the existing sensor fusion pipeline to work with MVD for real-world driving scenarios. Unfortunately, this end-goal was not met. However, considerable effort went into the development of the infrastructure required to achieve the end goal.

First, a comprehensive procedure for creating simulated sensor data in DSD was documented. As a result, future members of UWAFT can be quickly onboarded to this project. Each member should be able to create their own scenarios, create a rosbag file from the generated CSV, and run the sensor fusion nodes to obtain an output. Finally, members should be able to plot the sensor fusion output with respect to the ground truth sensor data.

Second, the sensor fusion pipeline was re-written in Python to be used as a sandbox for simple debugging, data visualization, and rapid prototyping. This code is well-documented and easy to follow. Furthermore, this report documents the steps needed to get up-and-running with the sandbox in a matter of minutes. Therefore, new UWAFT members can quickly get a deep understanding of each node in the sensor fusion architecture and how each sensor detection is interpreted in the pipeline. Finally, users can plot the sensor fusion output using the provided plotting functions and identify where the algorithm is performing poorly.

Using the sandbox, three main modifications were made to sensor fusion to qualitatively improve object tracking and data association. First, the object matching parameters were examined and optimized for MVD. A thorough explanation of these parameters is provided. These are the parameters that sensor fusion uses to create new tracks or assign detections to existing objects in environment state. Second, lane determination was moved to data association from Kalman filter. This modification adds intuition about a vehicles' spatial location during object assignment rather than simply relying on the tolerance parameters, dy and dx. The third change was adding weighted averaging to detections when creating a new track. This modification reduces the pipeline's

susceptibility to noise and outliers. All three modifications have increased the robustness of sensor fusion, for both single-vehicle and multi-vehicle detection.

4.1 Lessons learned

This project required the use of DSD, MATLAB, ROS, C++, and Python. I was already proficient with Python and MATLAB, however, there was a considerable learning curve for ROS, C++ and DSD. This was a main driver behind creating the sensor fusion sandbox in Python – it helped me better understand the components of the sensor fusion architecture and how they interact.

This project was very open-ended and there was never an obvious next step for how to improve the algorithm. I learned about the importance of asking lots of questions and getting a good grasp of fundamentals. For instance, a big blocker for me was navigating the ROS/C++ codebase. I could have benefitted from gaining a better understanding of ROS and C++ syntax before trying to make any changes. Asking for guidance from other UWAFT members would have removed blockers earlier. Next, I could have done more research into sensor fusion theory. Knowing certain object assignment algorithms such as global nearest neighbors (GNN) or joint probabilistic data association (JPDA) would have been beneficial to learn at the beginning of the term. As a result, I would have had more time to work out the implementation details in *kaiROS*.

Finally, I learned the importance of good documentation. The decisions that went into making the current sensor fusion pipeline were not well-documented. Lots lots of time was spent trying to reproduce previous results or figuring out how to run a script that was written over a year ago. Therefore, I made it a priority to document the steps needed to reproduce the results in this report. My hope is that future UWAFT members can read this report and have a good understanding of where to go next for MVD.

4.2 Recommendations & Next Steps

This project is still ongoing, the next steps towards improving MVD are:

1. Transfer the version 1 sensor fusion pipeline from the Python sandbox to ROS/C++

Version 0 is the original sensor fusion; version 1 is the updated sensor fusion with new object matching rules. The logic from the sandbox must be transferred over to ROS/C++ and plotted against the sensor ground truth data.

2. Test with simulations with curved roads

One of the limitations of the scenarios used for this report is that they were all done on straight roads. On a curved road, lane determination becomes more complicated. This will require further modification to the lane determination function in data association.

3. Revisit the Kalman filter node

Currently, the Kalman filter node is a black box. Theoretically it is supposed to provide the best likelihood estimate of a position given several measurements. However, the current implementation of Kalman filter is very susceptible to noise when it does not get frequent updates. An example of this is shown in Figure 25A. This figure shows the object tracking for scenario 3a when the **DX_TOL** is set to 10 meters. At 7.14 seconds, a detection at $\{dx = 12.35\ , dy = -2.94\}$ is matched to the vehicle in the left lane at $\{dx = 4.26, dy = -1.36\}$ which was last updated at 5.65 seconds. Therefore, one might expect the updated position to be in the neighbourhood of $\{dx = 10\ , dy = -2\}$. Instead, Kalman filter outputs a new position of $\{dx = 0.52, dy = 5.31\}$ which is on the far side of the opposite lane. This issue is fixed when **DX_TOL** is set to 8 meters (Figure 25B); however, the root cause of the issue is the Kalman filter estimation which remains unresolved. Therefore, more work should go into refining the predictions of this node.

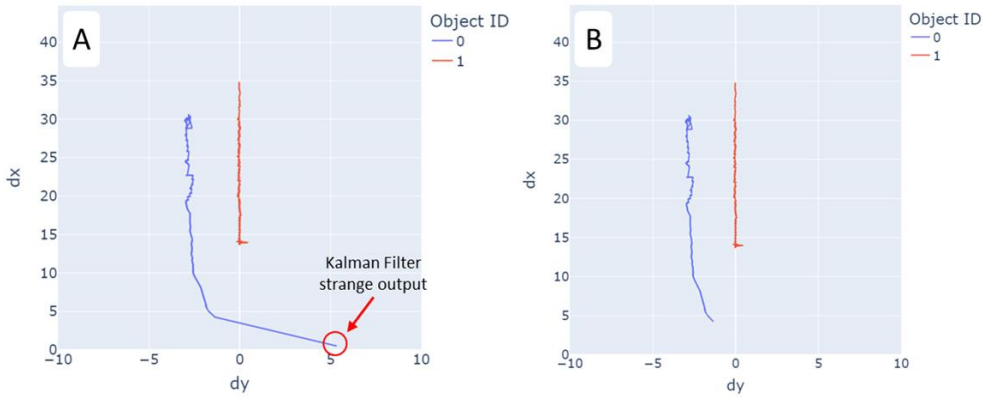

Figure 25 Scenario 3a object tracking using (A) DX_TOL = 10 and (B) DX_TOL = 8

4. Run sensor fusion against real-world OxTS data

The end-goal is to run the MVD sensor fusion inside the Blazer, in real-time. Therefore, real sensor data is required to further refine the accuracy and robustness of the sensor fusion for MVD.

References

[1] "EcoCar Mobility Challenge - AVTC I Advanced Vehicle Technology Competitions." https://avtcseries.org/ecocar-mobility-challenge/ (accessed Dec. 19, 2021).

[2] "Prototyping Perception Systems for SAE Level 2 | Student Lounge," *Student Lounge | The student lounge blog focuses on student success stories. Winning student teams share their knowledge and the MathWorks student programs team shares best practices and workflows using MATLAB and Simulink.*, Jul. 24, 2019. https://blogs.mathworks.com/student-lounge/2019/07/24/perception-level-2-automation/ (accessed Dec. 19, 2021).

[3] "Bosch Radar Deep Dive - UWaterloo Alternative Fuels Team - Confluence." https://uwaterloo.atlassian.net/wiki/spaces/UWAFT/pages/26831750429/Bosch+Radar+Deep+Dive (accessed Dec. 19, 2021).

[4] MATLAB, *Understanding Sensor Fusion and Tracking, Part 5: How to Track Multiple Objects at Once*, (Oct. 29, 2019). Accessed: Dec. 19, 2021. [Online Video]. Available: https://www.youtube.com/watch?v=IIt1LHIHYc4

[5] "Relation: Region of Waterloo (2062151)," *OpenStreetMap*. https://www.openstreetmap.org/relation/2062151 (accessed Dec. 20, 2021).