

RADAR Signal Simulation

SAE project

By

Dayananda BN

Tahsin Khan

Santosh Kumar

Sharan Venkatesh

Table Of Contents:

1. Introduction
2. Sensor Fusion Using Synthetic Radar and Vision Data
 - Generating the Scenario
 - Defining Radar and Vision Sensors
 - Creating a Tracker
 - Simulating the Scenario
 - Summary
3. Track-Level Fusion of Radar and Lidar Data
 - Setup Scenario for Synthetic Data Generation
 - Radar Tracking Algorithm
 - Lidar Tracking Algorithm
 - Set Up Fuser, Metrics, and Visualization
 - Running Scenario and Trackers
 - Summary
4. References

Introduction

In this project we aim to understand to customize an onboard LiDAR model to the specification of a desired RADAR field of view, resolution, and range and then utilizes a density-based clustering algorithm to generate the RADAR output on an open-source graphical engine such as Unreal Engine(UE).

1. A survey of the available open-source simulators reveals that RADAR (Radio Detection and Ranging) sensor modelling has received minimal attention.
2. High-fidelity RADAR models are already accessible in private simulation tools like MATLAB's automated driving toolbox, but not in open-source simulators like AirSim, necessitating the construction of suitably realistic automobile RADAR sensor models.
3. A LiDAR, on the other hand, is more susceptible to weather conditions such as rain and dust. RADAR, on the other hand, operates with frequency modulated continuous wave (FMCW), which is unaffected by these conditions but susceptible to a different set of environmental variables, such as temperature.
4. As a result, we were able to create a LiDAR sensor model to replicate RADAR detection while also giving it with a suitable range.

Differences between RADAR AND LiDAR

1. With the help of lidar we can detect little objects with a short wavelength. A LIDAR can create a 3D monochromatic image of an item that is extremely accurate. Disadvantages of Using LIDAR When it's dark or overcast outside, use it sparingly. The operating height is between 500 and 2000 meters. This is a high-priced technology.
2. In overcast weather and at night, RADAR can readily function. Operating distance is long. Use of RADAR also has its Drawbacks Smaller objects cannot be detected at shorter wavelengths. Because of the longer wavelength, RADAR cannot deliver a precise image of an item to the user.

Open-source and closed source simulators that use sensor model integration

1. AirSim: Developed by Microsoft originally for autonomous flying devices, the simulator now includes driving applications as well. However, it is still under development for robustness in the autonomous driving application
2. GTA V: Developed as a computer game, nevertheless, it is far more ahead in the physics implementation of a car when integrated with ScriptHook. Irrespective of the fact, GTA V lacks sensor emulation and multiple viewpoints and it cannot be used for commercial purposes.

3. CARLA Simulator: The most widely used open-source simulator for autonomous driving. It has the best autonomous driving algorithms built and are constantly upgrading to increase performance and robustness of the Simulator.

LiDAR to RADAR Translation

1. We first propose a conditional model called L2R Generative Adversarial Network to translate LiDAR data into RADAR
2. The GAN helps in image to image conditional translation and gives three different radar data representations
 1. Raw Polar RADAR Spectrum
 2. RADAR Spectrum in Cartesian Coordinates
 3. 3D RADAR points clouds and 2D RADAR pins
3. The original RADAR data obtained is in 2D array form and is not available to the users. Nevertheless, the above issue is countered by using Digital Signal Processing (DSP) algorithms such as Fast Fourier Transform (FFT) and Multiple Signal Classification (MUSIC) to obtain polar RADAR data.
4. For the next part, we implement the baseline for pix2pix and pix2pixHD to generate a high-resolution RADAR data/ spectrum from the LiDAR point cloud. The method is a framework for GAN to create image-to image translation and adding loss.
5. The research's conclusion is that the L2R GAN neural network model is significantly promising in generating

RADAR images using global for large regions and local generators for small regions. Also, the model can be used to generate RADAR data for emergency situations such as pedestrian collision warning. The research is a good reference for further learning in ADAS applications.

Sensor Fusion Using Synthetic Radar and Vision Data

Generating the Scenario

Scenario generation comprises generating a road network, defining vehicles that move on the roads, and moving the vehicles.

In this example, we test the ability of the sensor fusion to track a vehicle that is passing on the left of the ego vehicle. The scenario simulates a highway setting, and additional vehicles are in front of and behind the ego vehicle.

```
% Define an empty scenario  
  
scenario = drivingScenario;  
  
scenario.SampleTime = 0.01;
```

Add a stretch of 500 meters of typical highway road with two lanes. The road is defined using a set of points, where each point defines the centre of the road in 3-D space, and a road width.

```

roadCenters = [0 0; 50 0; 100 0; 250 20; 500 40];

roadWidth = 7.2; % Two lanes, each 3.6 meters

road(scenario, roadCenters, roadWidth);

```

Create the ego vehicle and three cars around it: one that overtakes the ego vehicle and passes it on the left, one that drives right in front of the ego vehicle and one that drives right behind the ego vehicle. All the cars follow the path defined by the road waypoints by using the path driving policy. The passing car will start on the right lane, move to the left lane to pass, and return to the right lane.

```

% Create the ego vehicle that travels at 25 m/s along the road

egoCar = vehicle(scenario, 'ClassID', 1);

path(egoCar, roadCenters(2:end,:) - [0 1.8], 25); % On right lane

% Add a car in front of the ego vehicle

leadCar = vehicle(scenario, 'ClassID', 1);

path(leadCar, [70 0; roadCenters(3:end,:)] - [0 1.8], 25); % On right lane

% Add a car that travels at 35 m/s along the road and passes the ego vehicle

passingCar = vehicle(scenario, 'ClassID', 1);

waypoints = [0 -1.8; 50 1.8; 100 1.8; 250 21.8; 400 32.2; 500 38.2];

path(passingCar, waypoints, 35);

% Add a car behind the ego vehicle

chaseCar = vehicle(scenario, 'ClassID', 1);

path(chaseCar, [25 0; roadCenters(2:end,:)] - [0 1.8], 25); % On right lane

```

Defining RADAR and vision sensors

In this example, we simulate an ego vehicle that has 6 radar sensors and 2 vision sensors covering the 360 degrees field of view. The sensors have some overlap and some coverage gap. The ego vehicle is equipped with a long-range radar sensor and a vision sensor on both the front and the back of the vehicle. Each side of the vehicle has two short-range radar sensors, each covering 90 degrees. One sensor on each side covers from the middle of the vehicle to the back. The other sensor on each side covers from the middle of the vehicle forward. The figure in the next section shows the coverage.

```
sensors = cell(8,1);

% Front-facing long-range radar sensor at the center of the front bumper of the
% car.
sensors{1} = radarDetectionGenerator('SensorIndex', 1, 'Height', 0.2, 'MaxRange', 174,
...
    'SensorLocation', [egoCar.Wheelbase + egoCar.FrontOverhang, 0], 'FieldOfView',
[20, 5]);

% Rear-facing long-range radar sensor at the center of the rear bumper of the car.
sensors{2} = radarDetectionGenerator('SensorIndex', 2, 'Height', 0.2, 'Yaw', 180, ...
    'SensorLocation', [-egoCar.RearOverhang, 0], 'MaxRange', 174, 'FieldOfView',
[20,5]);

% Rear-left-facing short-range radar sensor at the left rear wheel well of the car.
sensors{3} = radarDetectionGenerator('SensorIndex', 3, 'Height', 0.2, 'Yaw', 120, ...
    'SensorLocation', [0, egoCar.Width/2], 'MaxRange', 30, 'ReferenceRange', 50, ...
    'FieldOfView', [90, 5], 'AzimuthResolution', 10, 'RangeResolution', 1.25);

% Rear-right-facing short-range radar sensor at the right rear wheel well of the
% car.
```



```

sensors{5} = radarDetectionGenerator('SensorIndex', 5, 'Height', 0.2, 'Yaw', 60, ...
    'SensorLocation', [egoCar.Wheelbase, egoCar.Width/2], 'MaxRange', 30, ...
    'ReferenceRange', 50, 'FieldOfView', [90, 5], 'AzimuthResolution', 10, ...
    'RangeResolution', 1.25);

% Front-right-facing short-range radar sensor at the right front wheel well of the
% car.

sensors{6} = radarDetectionGenerator('SensorIndex', 6, 'Height', 0.2, 'Yaw', -60, ...
    'SensorLocation', [egoCar.Wheelbase, -egoCar.Width/2], 'MaxRange', 30, ...
    'ReferenceRange', 50, 'FieldOfView', [90, 5], 'AzimuthResolution', 10, ...
    'RangeResolution', 1.25);

% Front-facing camera located at front windshield.

sensors{7} = visionDetectionGenerator('SensorIndex', 7, 'FalsePositivesPerImage', 0.1,
...
    'SensorLocation', [0.75*egoCar.Wheelbase 0], 'Height', 1.1);

% Rear-facing camera located at rear windshield.

sensors{8} = visionDetectionGenerator('SensorIndex', 8, 'FalsePositivesPerImage', 0.1,
...
    'SensorLocation', [0.2*egoCar.Wheelbase 0], 'Height', 1.1, 'Yaw', 180);

```

Creating a Tracker

Create a multiObjectTracker to track the vehicles that are close to the ego vehicle. The tracker uses the `initSimDemoFilter` supporting function to initialize a constant velocity linear Kalman filter that works with position and velocity. Tracking is done in 2-D. Although the sensors return measurements in 3-D, the motion itself is confined to the horizontal plane, so there is no need to track the height.

```

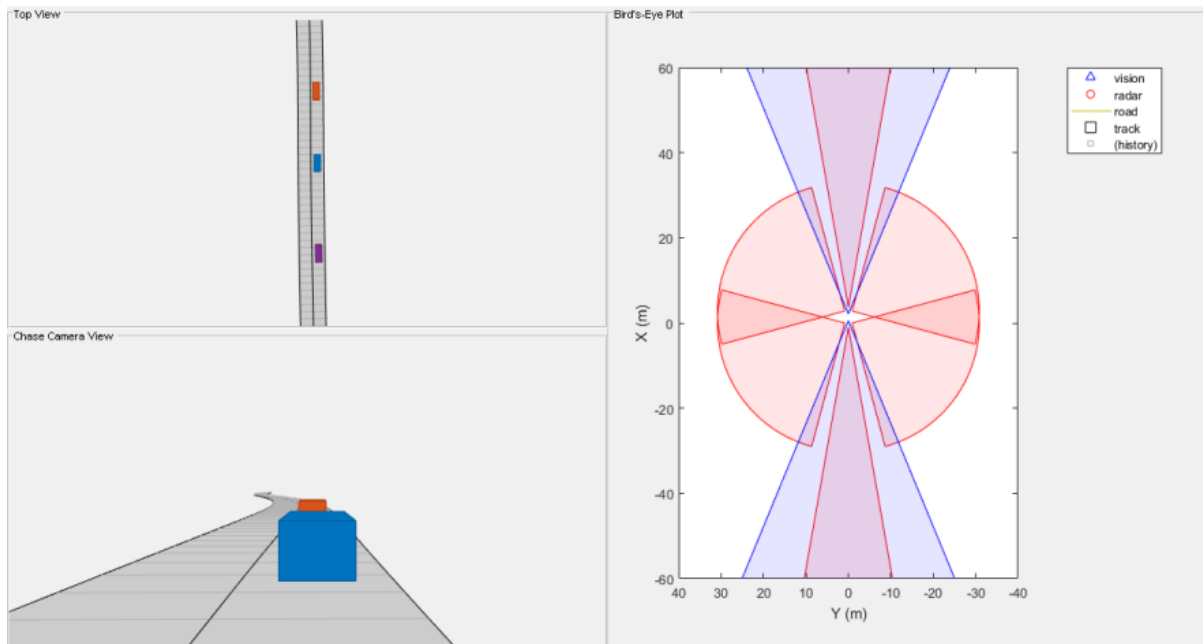
tracker = multiObjectTracker('FilterInitializationFcn', @initSimDemoFilter, ...
    'AssignmentThreshold', 30, 'ConfirmationParameters', [4 5]);

positionSelector = [1 0 0 0; 0 0 1 0]; % Position selector
velocitySelector = [0 1 0 0; 0 0 0 1]; % Velocity selector

% Create the display and return a handle to the bird's-eye plot

BEP = createDemoDisplay(egoCar, sensors);

```



Simulating the scenario

The following loop moves the vehicles, calls the sensor simulation, and performs the tracking. Note that the scenario generation and sensor simulation can have different time steps. Specifying different time steps for the scenario and the sensors enables us to decouple the scenario simulation from the sensor simulation. This is useful for modelling actor motion with high accuracy independently from the sensor's measurement rate. Another example is when the sensors have different update rates. Suppose one sensor provides updates every 20 milliseconds and another

sensor provides updates every 50 milliseconds. We can specify the scenario with an update rate of 10 milliseconds and the sensors will provide their updates at the correct time. In this example, the scenario generation has a time step of 0.01 second, while the sensors detect every 0.1 second. The sensors return a logical flag, `isValidTime`, that is true if the sensors generated detections. This flag is used to call the tracker only when there are detections. Another important note is that the sensors can simulate multiple detections per target, in particular when the targets are very close to the radar sensors. Because the tracker assumes a single detection per target from each sensor, we must cluster the detections before the tracker processes them. This is done by the function `clusterDetections`. See the 'Supporting Functions' section.

```
toSnap = true;

while advance(scenario) && ishghandle(BEP.Parent)

    % Get the scenario time

    time = scenario.SimulationTime;

    % Get the position of the other vehicle in ego vehicle coordinates

    ta = targetPoses(egoCar);

    % Simulate the sensors

    detections = {};

    isValidTime = false(1,8);

    for i = 1:8

        [sensorDets,numValidDets,isValidTime(i)] = sensors{i}(ta, time);

        if numValidDets

            for j = 1:numValidDets
```

```

        % Vision detections do not report SNR. The tracker requires
        % that they have the same object attributes as the radar
        % detections. This adds the SNR object attribute to vision
        % detections and sets it to a NaN.

        if ~isfield(sensorDets{j}.ObjectAttributes{1}, 'SNR')
            sensorDets{j}.ObjectAttributes{1}.SNR = NaN;
        end
    end

    detections = [detections; sensorDets]; %#ok<AGROW>
end

end

% Update the tracker if there are new detections
if any(isValidTime)
    vehicleLength = sensors{1}.ActorProfiles.Length;
    detectionClusters = clusterDetections(detections, vehicleLength);

```

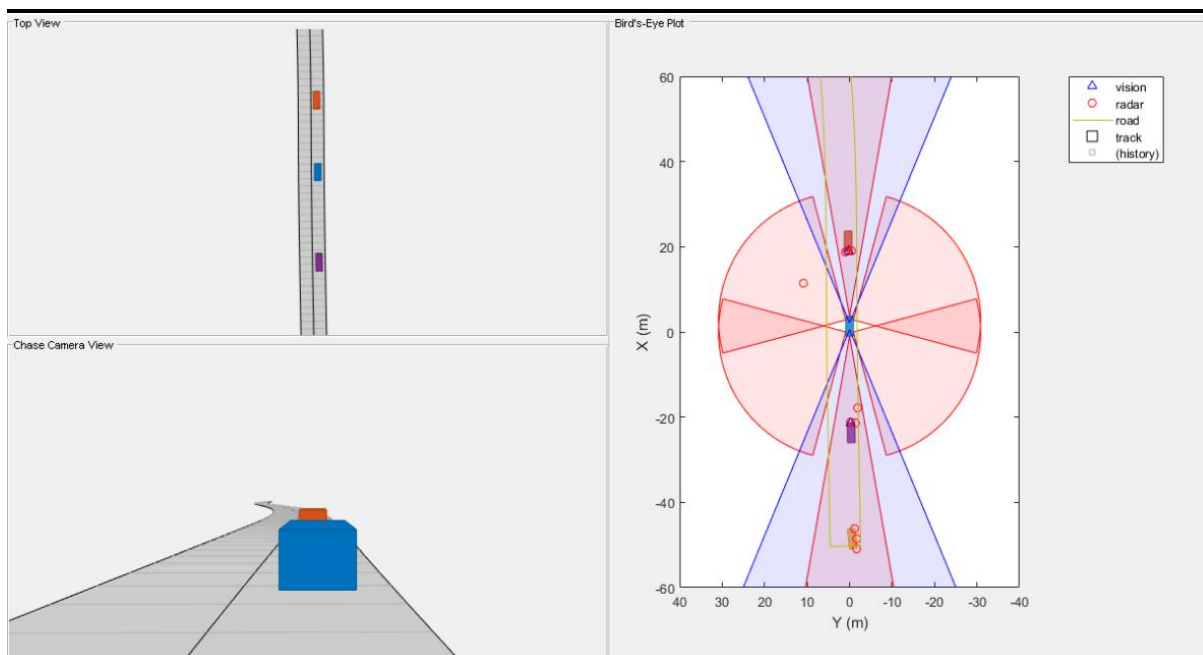
```

confirmedTracks = updateTracks(tracker, detectionClusters, time);

% Update bird's-eye plot
updateBEP(BEP, egoCar, detections, confirmedTracks, positionSelector,
velocitySelector);
end

% Snap a figure for the document when the car passes the ego vehicle
if ta(1).Position(1) > 0 && toSnap
    toSnap = false;
    snapnow
end
end
end

```

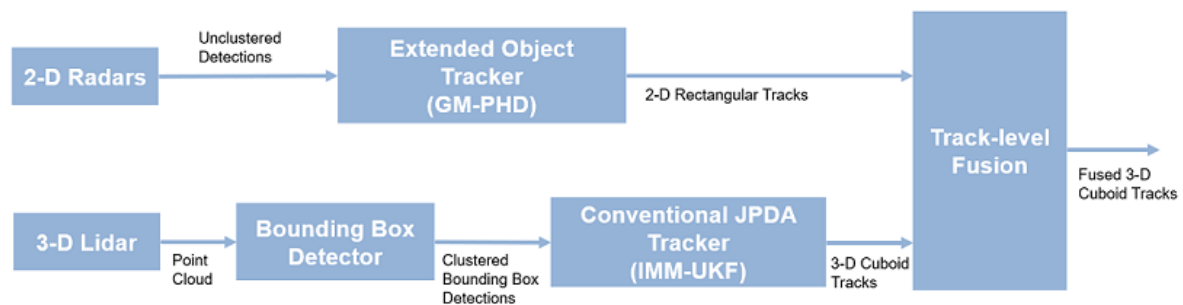


Summary

This example shows how to generate a scenario, simulate sensor detections, and use these detections to track moving vehicles around the ego vehicle. We can try to modify the scenario road, or add or remove vehicles. We can also try to add, remove, or modify the sensors on the ego vehicle, or modify the tracker parameters.

Track-Level Fusion of Radar and Lidar Data

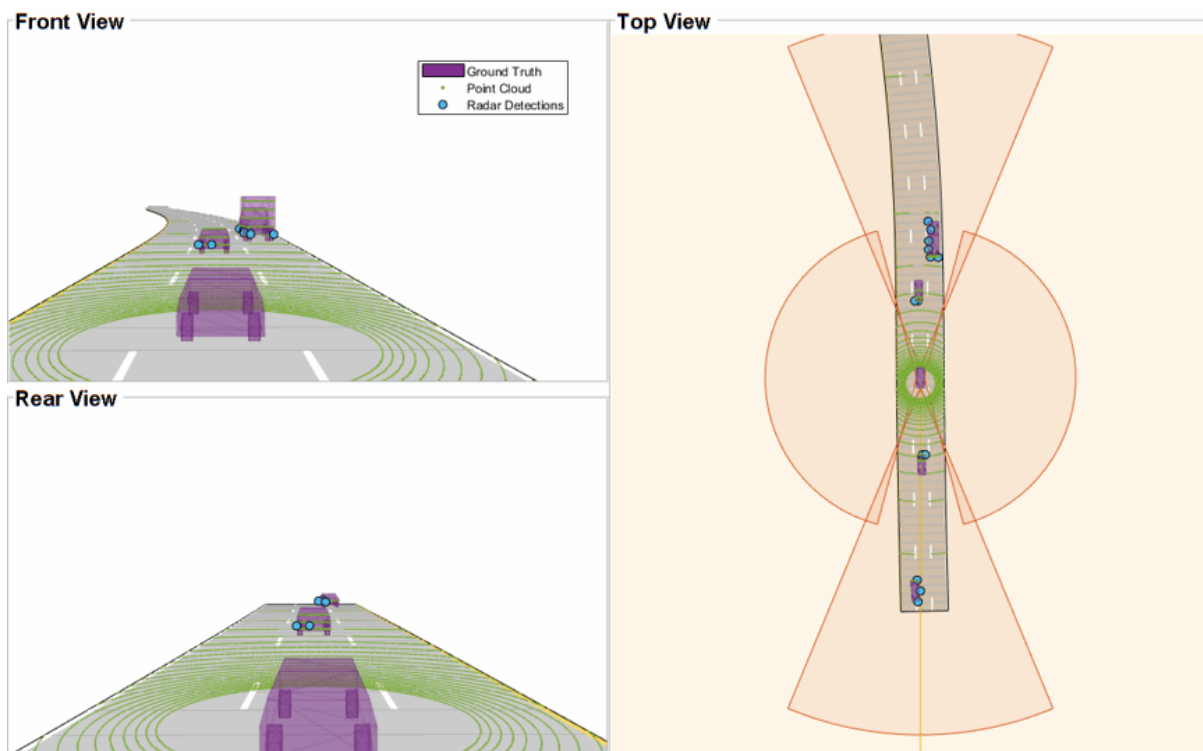
This example shows we how to generate an object-level track list from measurements of a radar and a lidar sensor and further fuse them using a track-level fusion scheme. We process the radar measurements using an extended object tracker and the lidar measurements using a joint probabilistic data association (JPDA) tracker. We further fuse these tracks using a track-level fusion scheme. The schematic of the workflow is shown below.



Setup Scenario for Synthetic Data Generation

The scenario used in this example is created using drivingScenario (Automated Driving Toolbox). The data from radar and lidar sensors is simulated using drivingRadarDataGenerator (Automated Driving Toolbox) and lidarPointCloudGenerator (Automated Driving Toolbox), respectively. The creation of the scenario and the sensor models is wrapped in the helper function helperCreateRadarLidarScenario. For more information on scenario and synthetic data generation, refer to Create Driving Scenario Programmatically (Automated Driving Toolbox).

The ego vehicle is mounted with four 2-D radar sensors. The front and rear radar sensors have a field of view of 45 degrees. The left and right radar sensors have a field of view of 150 degrees. Each radar has a resolution of 6 degrees in azimuth and 2.5 meters in range. The ego is also mounted with one 3-D lidar sensor with a field of view of 360 degrees in azimuth and 40 degrees in elevation. The lidar has a resolution of 0.2 degrees in azimuth and 1.25 degrees in elevation (32 elevation channels). Visualize the configuration of the sensors and the simulated sensor data in the animation below. Notice that the radars have higher resolution than objects and therefore return multiple measurements per object. Also notice that the lidar interacts with the low-poly mesh of the actor as well as the road surface to return multiple points from these objects.



Radar Tracking Algorithm

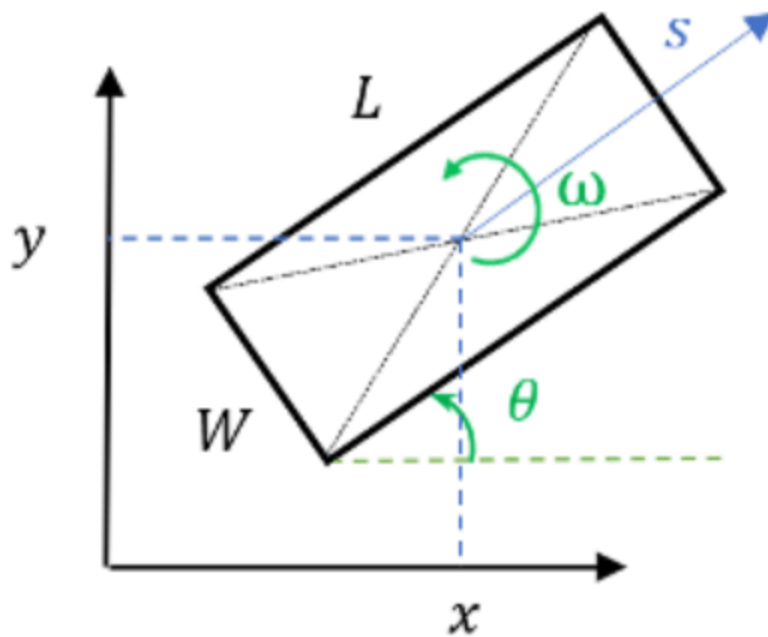
As mentioned, the radars have higher resolution than the objects and return multiple detections per object. Conventional trackers

such as Global Nearest Neighbour (GNN) and Joint Probabilistic Data Association (JPDA) assume that the sensors return at most one detection per object per scan. Therefore, the detections from high-resolution sensors must be either clustered before processing it with conventional trackers or must be processed using extended object trackers. Extended object trackers do not require pre-clustering of detections and usually estimate both kinematic states (for example, position and velocity) and the extent of the objects. For a more detailed comparison between conventional trackers and extended object trackers, refer to the Extended Object Tracking of Highway Vehicles with Radar and Camera (Sensor Fusion and Tracking Toolbox) example.

In general, extended object trackers offer better estimation of objects as they handle clustering and data association simultaneously using temporal history of tracks. In this example, the radar detections are processed using a Gaussian mixture probability hypothesis density (GM-PHD) tracker (trackerPHD (Sensor Fusion and Tracking Toolbox) and gmphd (Sensor Fusion and Tracking Toolbox)) with a rectangular target model. For more details on configuring the tracker, refer to the "GM-PHD Rectangular Object Tracker" section of the Extended Object Tracking of Highway Vehicles with Radar and Camera (Sensor Fusion and Tracking Toolbox) example.

The algorithm for tracking objects using radar measurements is wrapped inside the helper class, helperRadarTrackingAlgorithm, implemented as a System object™. This class outputs an array of objectTrack (Sensor Fusion and Tracking Toolbox) objects and define their state according to the following convention:

$[x \ y \ s \ \theta \ \omega \ L \ W]$



Lidar Tracking Algorithm

Similar to radars, the lidar sensor also returns multiple measurements per object. Further, the sensor returns a large number of points from the road, which must be removed before used as inputs for an object-tracking algorithm. While lidar data from obstacles can be directly processed via extended object tracking algorithm, conventional tracking algorithms are still more prevalent for tracking using lidar data. The first reason for this trend is mainly observed due to higher computational complexity of extended object trackers for large data sets. The second reason is the investments into advanced Deep learning-based detectors such as PointPillars [1], VoxelNet [2] and PIXOR [3], which can segment a point cloud and return bounding box detections for the vehicles. These detectors can help in overcoming the performance degradation of conventional trackers due to improper clustering.

In this example, the lidar data is processed using a conventional joint probabilistic data association (JPDA) tracker, configured with

an interacting multiple model (IMM) filter. The pre-processing of lidar data to remove point cloud is performed by using a RANSAC-based plane-fitting algorithm and bounding boxes are formed by performing a Euclidian-based distance clustering algorithm. For more information about the algorithm, refer to the Track Vehicles Using Lidar: From Point Cloud to Track List (Sensor Fusion and Tracking Toolbox) example. Compared the linked example, the tracking is performed in the scenario frame and the tracker is tuned differently to track objects of different sizes. Further the states of the variables are defined differently to constrain the motion of the tracks in the direction of its estimated heading angle.

The algorithm for tracking objects using lidar data is wrapped inside the helper class, `helperLidarTrackingAlgorithm` implemented as System object. This class outputs an array of `objectTrack` (Sensor Fusion and Tracking Toolbox) objects and defines their state according to the following convention:

$$[x \ y \ s \ \theta \ \omega \ z \ \dot{z} \ L \ W \ H]$$

The states common to the radar algorithm are defined similarly. Also, as a 3-D sensor, the lidar tracker outputs three additional states, z , \dot{z} and H , which refer to z-coordinate (m), z-velocity (m/s), and height (m) of the tracked object respectively.

Set Up Fuser, Metrics, and Visualization

Fuser

Next, we will set up a fusion algorithm for fusing the list of tracks from radar and lidar trackers. Similar to other tracking algorithms, the first step towards setting up a track-level fusion algorithm is defining the choice of state vector (or state-space) for the fused or central tracks. In this case, the state-space for fused tracks is chosen to be same as the lidar. After choosing a central track state-space, we define the transformation of the central track state to the local track state. In this case, the local track

state-space refers to states of radar and lidar tracks. To do this, we use a `fuserSourceConfiguration` (Sensor Fusion and Tracking Toolbox) object.

Define the configuration of the radar source. The `helperRadarTrackingAlgorithm` outputs tracks with `SourceIndex` set to 1. The `SourceIndex` is provided as a property on each tracker to uniquely identify it and allows a fusion algorithm to distinguish tracks from different sources. Therefore, we set the `SourceIndex` property of the radar configuration as same as those of the radar tracks. We set `IsInitializingCentralTracks` to true to let that unassigned radar tracks initiate new central tracks. Next, we define the transformation of a track in central state-space to the radar state-space and vice-versa. The helper functions `central2radar` and `radar2central` perform the two transformations and are included at the end of this example.

The next step is to define the state-fusion algorithm. The state-fusion algorithm takes multiple states and state covariances in the central state-space as input and returns a fused estimate of the state and the covariances. In this example, we use a covariance intersection algorithm provided by the helper function, `helperRadarLidarFusionFcn`. A generic covariance intersection algorithm for two Gaussian estimates with mean x_i and covariance P_i can be defined according to the following equations:

$$P_F^{-1} = w_1 P_1^{-1} + w_2 P_2^{-1}$$

$$x_F = P_F (w_1 P_1^{-1} x_1 + w_2 P_2^{-1} x_2)$$

where x_F and P_F are the fused state and covariance and w_1 and w_2 are mixing coefficients from each estimate. Typically, these mixing coefficients are estimated by minimizing the determinant or the trace of the fused covariance. In this example, the mixing weights are estimated by minimizing the determinant of positional covariance of each estimate. Furthermore, as the radar does not estimate 3-D states, 3-D states are only fused with

lidars. For more details, refer to the helperRadarLidarFusionFcn function shown at the end of this script.








Metrics

In this example, we assess the performance of each algorithm using the Generalized Optimal SubPattern Assignment Metric (GOSPA) metric. We set up three separate metrics using trackGOSPAMetric (Sensor Fusion and Tracking Toolbox) for each of the trackers. The GOSPA metric aims to evaluate the performance of a tracking system by providing a scalar cost. A lower value of the metric indicates better performance of the tracking algorithm.

To use the GOSPA metric with custom motion models like the one used in this example, we set the Distance property to 'custom' and define a distance function between a track and its associated ground truth. These distance functions, shown at the end of this example are helperRadarDistance, and helperLidarDistance.

Visualization

The visualization for this example is implemented using a helper class helperLidarRadarTrackFusionDisplay. The display is divided into 4 panels. The display plots the measurements and tracks from each sensor as well as the fused track estimates. The legend for the display is shown below. Furthermore, the tracks are annotated by their unique identity (TrackID) as well as a prefix. The prefixes "R", "L" and "F" stand for radar, lidar, and fused estimate, respectively.

-
-  Ground Truth
 -  Radar Tracks
 -  Lidar Tracks
 -  Fused Tracks
 -  Point Cloud
 -  Radar Detections
 -  Lidar Bounding Box Detections
-

Run Scenario and Trackers

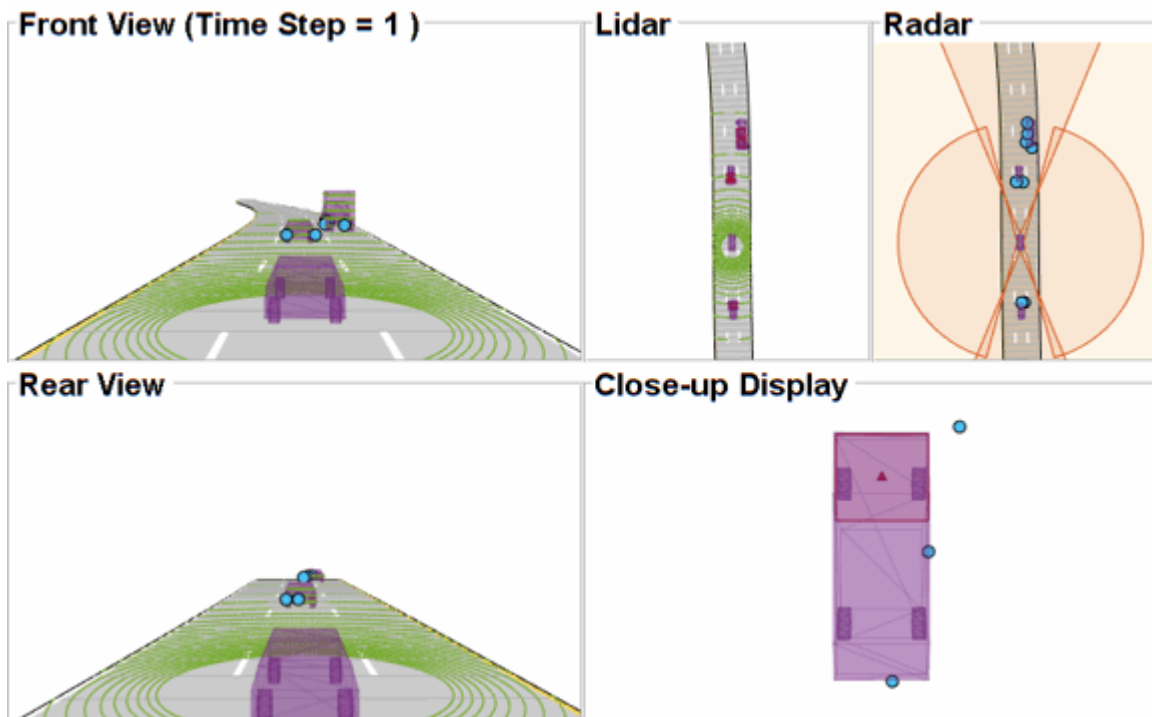
Next, we advance the scenario, generate synthetic data from all sensors and process it to generate tracks from each of the systems. We also compute the metric for each tracker using the ground truth available from the scenario.

Evaluate Performance

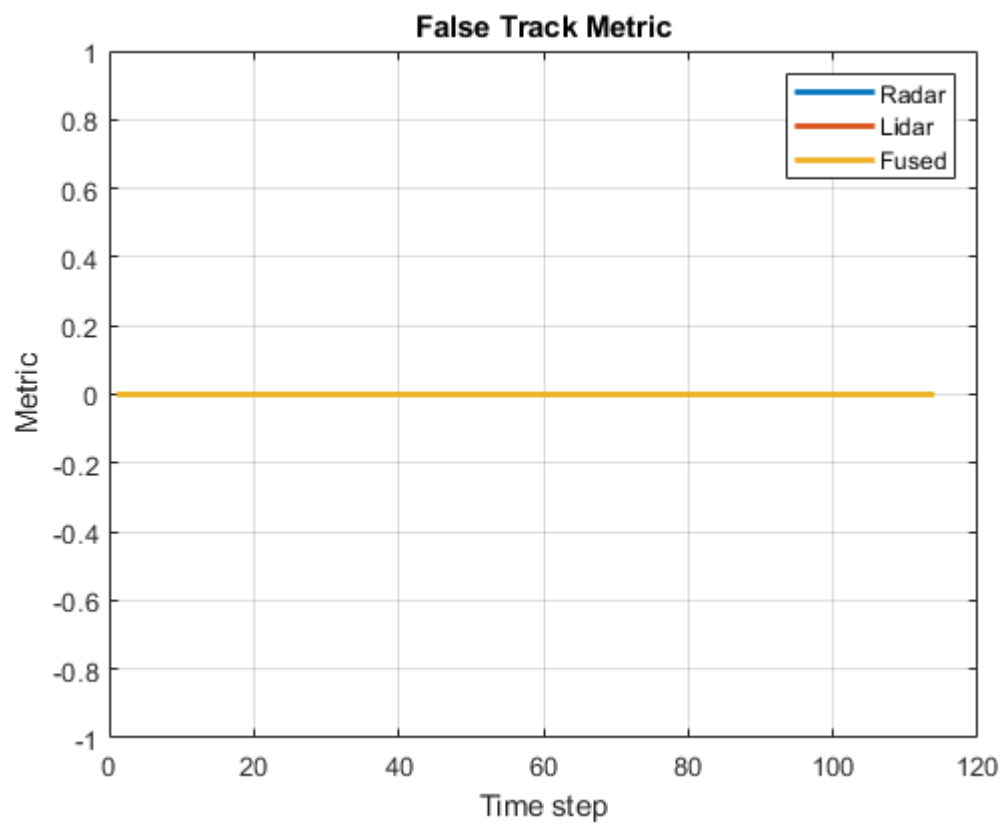
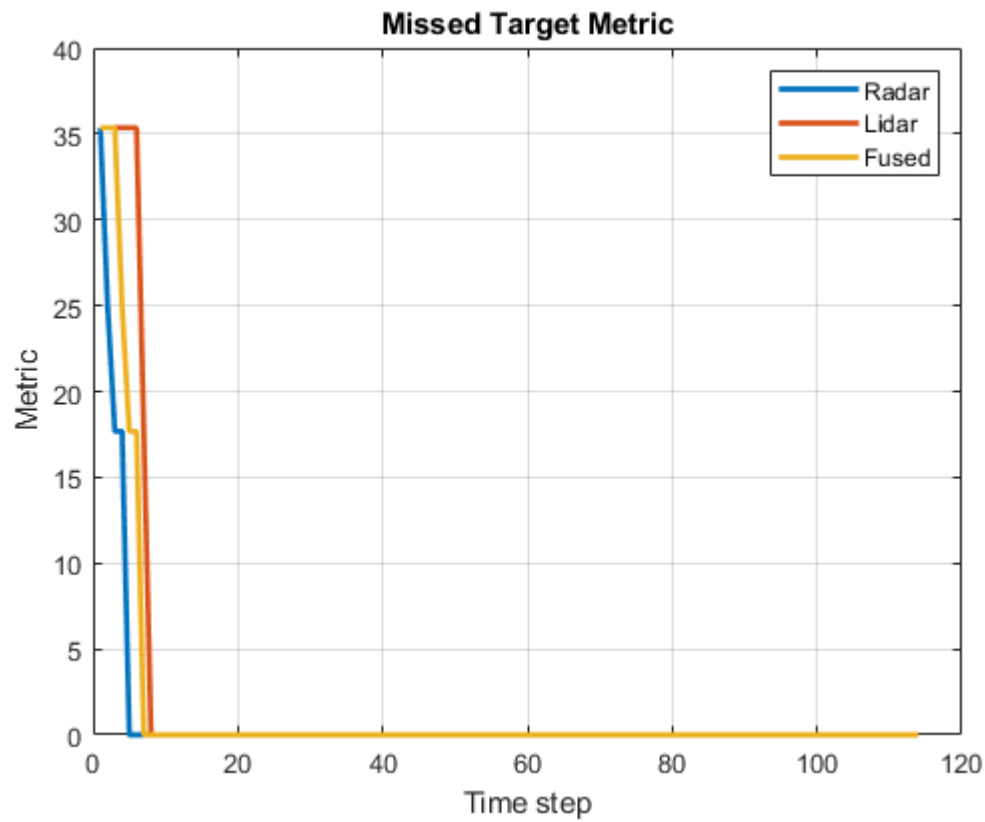
Evaluate the performance of each tracker using visualization as well as quantitative metrics. Analyse different events in the scenario and understand how the track-level fusion scheme helps achieve a better estimation of the vehicle state.

Track Maintenance

The animation below shows the entire run every three time-steps. Note that each of the three tracking systems – radar, lidar, and the track-level fusion – were able to track all four vehicles in the scenario and no false tracks were confirmed.



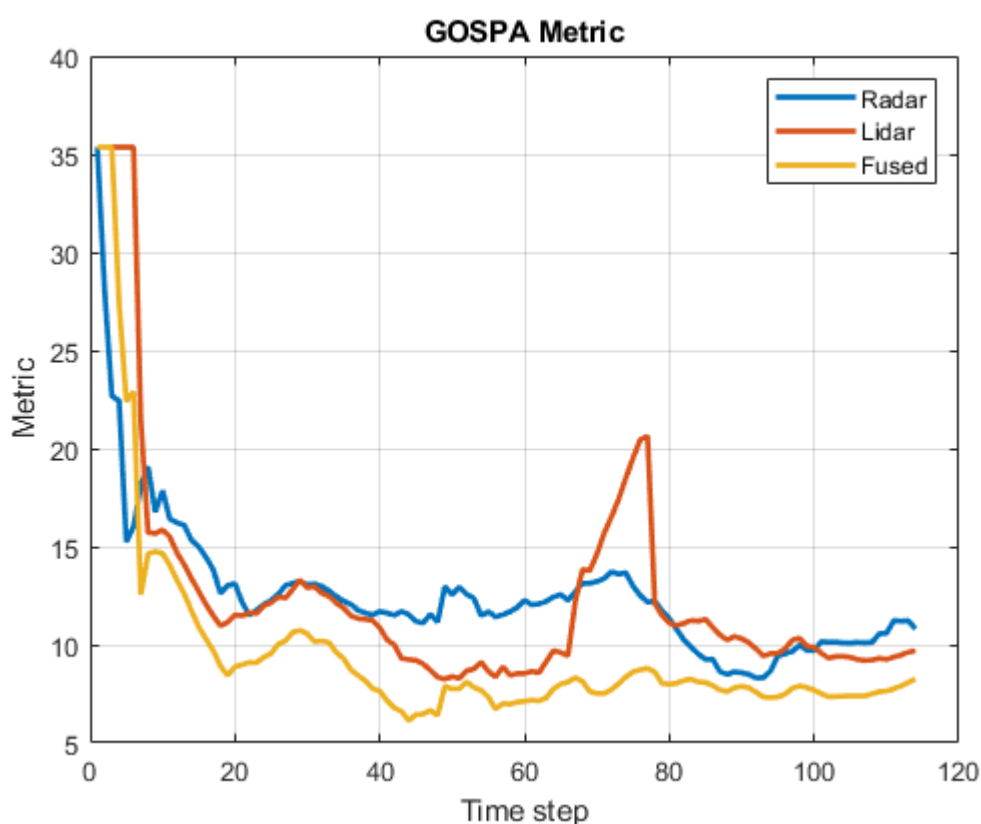
We can also quantitatively measure this aspect of the performance using "missed target" and "false track" components of the GOSPA metric. Notice in the figures below that missed target component starts from a higher value due to establishment delay and goes down to zero in about 5-10 steps for each tracking system. Also, notice that the false track component is zero for all systems, which indicates that no false tracks were confirmed.



Track-level Accuracy

The track-level or localization accuracy of each tracker can also be quantitatively assessed by the GOSPA metric at each time step. A lower value indicates better tracking accuracy. As there were no missed targets or false tracks, the metric captures the localization errors resulting from state estimation of each vehicle.

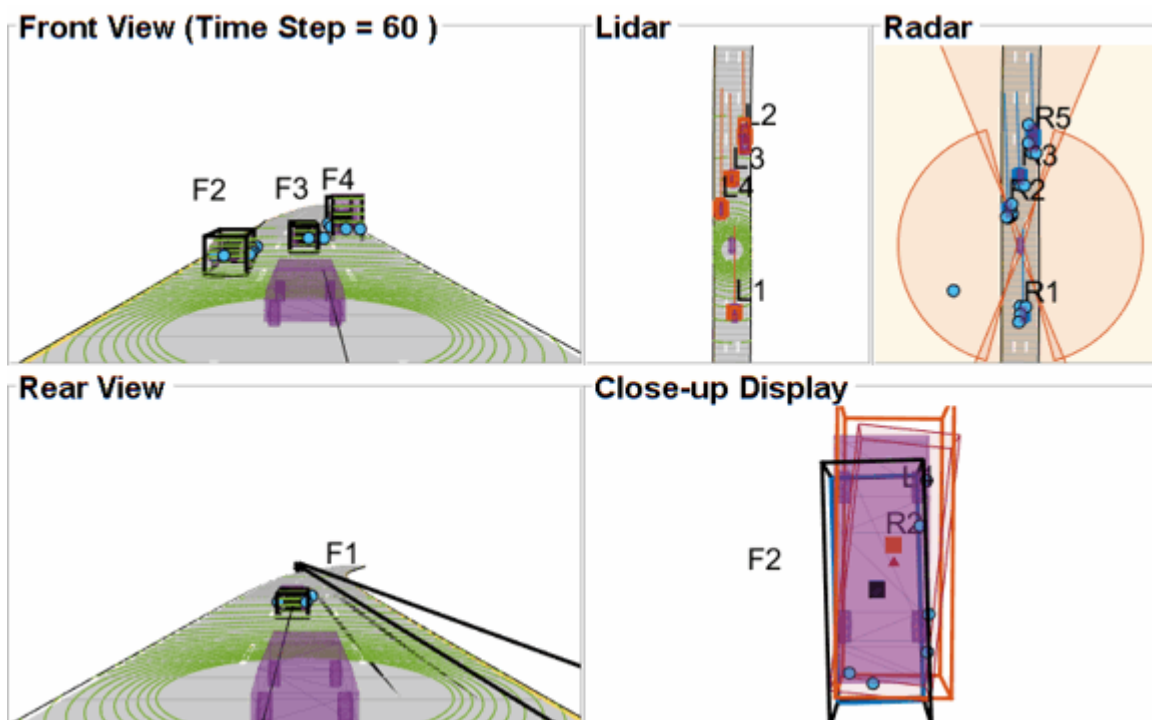
Note that the GOSPA metric for fused estimates is lower than the metric for individual sensor, which indicates that track accuracy increased after fusion of track estimates from each sensor.



Closely-spaced targets

As mentioned earlier, this example uses a Euclidian-distance based clustering and bounding box fitting to feed the lidar data to a conventional tracking algorithm. Clustering algorithms typically suffer when objects are closely-spaced. With the detector configuration used in this example, when the passing vehicle approaches the vehicle in front of the ego vehicle, the detector

clusters the point cloud from each vehicle into a bigger bounding box. We can notice in the animation below that the track drifted away from the vehicle centre. Because the track was reported with higher certainty in its estimate for a few steps, the fused estimated was also affected initially. However, as the uncertainty increases, its association with the fused estimate becomes weaker. This is because the covariance intersection algorithm chooses a mixing weight for each assigned track based on the certainty of each estimate.



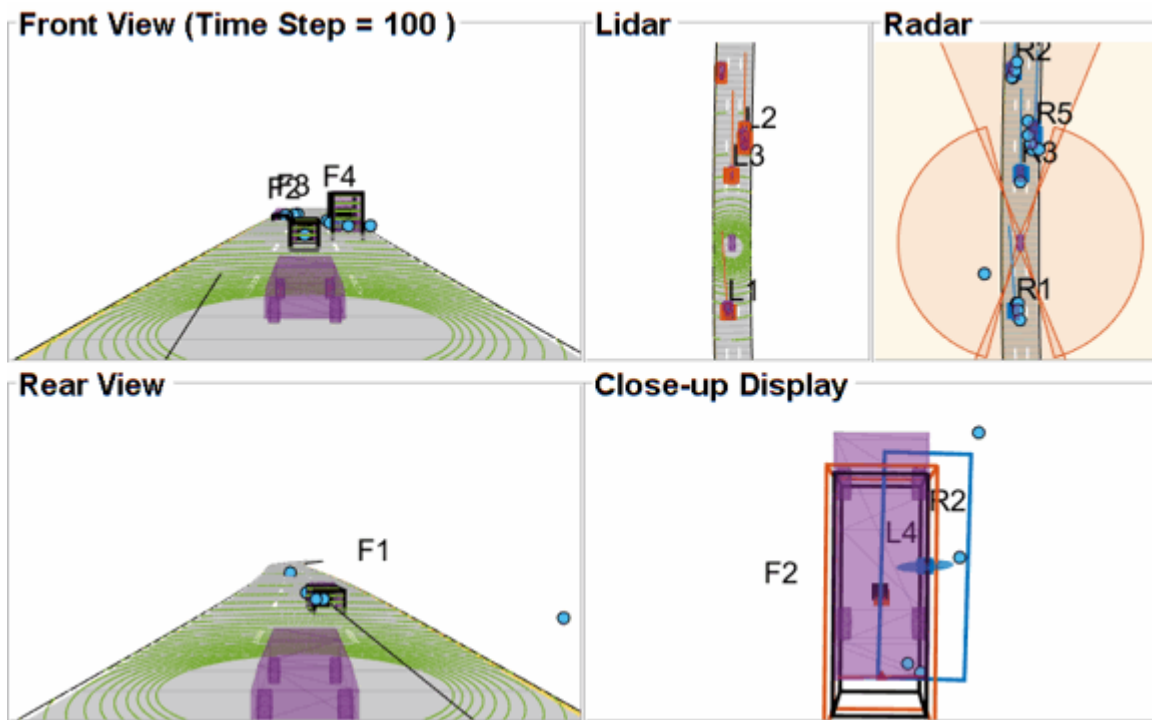
This effect is also captured in the GOSPA metric. We can notice in the GOSPA metric plot above that the lidar metric shows a peak around the 65th time step.

The radar tracks are not affected during this event because of two main reasons. Firstly, the radar sensor outputs range-rate information in each detection, which is different beyond noise-levels for the passing car as compared to the slower moving car. This results in an increased statistical distance between detections from individual cars. Secondly, extended object trackers evaluate multiple possible clustering hypothesis against predicted tracks, which results in rejection of improper clusters and

acceptance of proper clusters. Note that for extended object trackers to properly choose the best clusters, the filter for the track must be robust to a degree that can capture the difference between two clusters. For example, a track with high process noise and highly uncertain dimensions may not be able to properly claim a cluster because of its premature age and higher flexibility to account for uncertain events.

Targets at long range

As targets recede away from the radar sensors, the accuracy of the measurements degrade because of reduced signal-to-noise ratio at the detector and the limited resolution of the sensor. This results in high uncertainty in the measurements, which in turn reduces the track accuracy. Notice in the close-up display below that the track estimate from the radar is further away from the ground truth for the radar sensor and is reported with a higher uncertainty. However, the lidar sensor reports enough measurements in the point cloud to generate a “shrunk” bounding box. The shrinkage effect modelled in the measurement model for lidar tracking algorithm allows the tracker to maintain a track with correct dimensions. In such situations, the lidar mixing weight is higher than the radar and allows the fused estimate to be more accurate than the radar estimate.



Summary

In this example, we learned how to set up a track-level fusion algorithm for fusing tracks from radar and lidar sensors. We also learned how to evaluate a tracking algorithm using the Generalized Optimal Subpattern Metric and its associated components.

References

- https://in.mathworks.com/help/lidar/ug/track-level-fusion-of-radar-and-lidar.html#responsive_offcanvas
- https://in.mathworks.com/content/dam/mathworks/tag-team/Objects/s/80868v00_Sensor_Fusion_Synthetic_Radar_Vision_Data.pdf