WHITE PAPER

Sensor Fusion and Tracking for Autonomous Systems

MathWorks®

This white paper demonstrates how you can use MATLAB® and Simulink® to:

1. Define scenarios and generate detections from sensors including radar, camera, lidar, and sonar
2. Develop algorithms for sensor fusion and localization
3. Compare state estimation filters, motion models, and multi-object trackers
4. Perform what-if analysis with different scenarios
5. Evaluate positional accuracy and track assignment performance versus ground truth
6. Generate C code for rapid prototyping

With this workflow, you can avoid reinventing the wheel with every new autonomous system development project, saving you time and effort. In addition, you can share your models and results both within and outside your organization.

In this white paper, Sensor Fusion and Tracking Toolbox™ and Automated Driving Toolbox™ are used in the associated workflows.

Introduction

Autonomous systems range from vehicles that meet the various SAE levels of autonomy to systems including consumer quadcopters, package delivery drones, flying taxis, and robots for disaster relief and space exploration. Every autonomous system can be described at a high level with the block diagram shown in Figure 1.

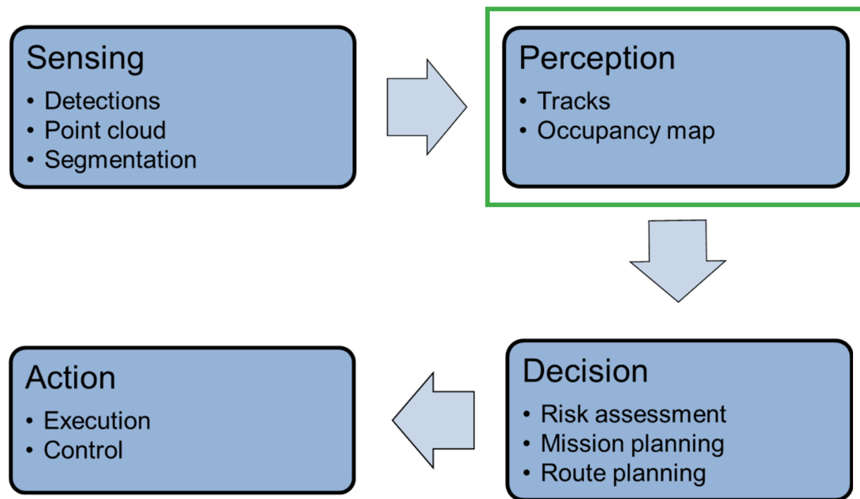

Figure 1. Autonomous systems processing loop with a focus on perception.

The focus of this white paper is on the perception component of the autonomous systems processing loop. Sensor fusion and multi-object tracking are at the heart of perception, and that is where the examples below are focused.

MathWorks®

Developing Perception Systems

Autonomous systems rely on sensor suites that provide data about the surrounding environment to feed the perception system. These sensors include radars and cameras, which provide detections of objects in their field of view. They also include lidar sensors, which provide point clouds of returns from obstacles in the environment, and in some cases, ultrasound and sonar sensors. Autonomous systems must also be able to estimate their position to maintain self-awareness. For this, sensors such as inertial measurement units (IMUs) and global positioning system (GPS) receivers are used.

The sensor system can also perform signal processing, including detection, segmentation, labeling, classification, and oftentimes basic tracking to reduce false alarms. Performing this function is a challenge for systems that must achieve the highest levels of autonomy because a lack of understanding of the occupied space around the autonomous system can have catastrophic results. Similarly, the presence of false tracks results in a confused state in the perception system, which impacts the planning and controls components.

Development teams can save countless hours of field testing through simulation of realistic scenes including ownship motion (also known in some autonomous systems communities as ego vehicle motion). You can build up libraries of scenarios that represent corner cases for your system. These scenarios can then be used to develop and test your sensor fusion and tracking algorithms.

Models and simulations can extend to 2D and 3D environments. You can configure and integrate sensor fusion and tracking algorithms from a complete set of tracker libraries. In addition, you can simulate sensor data, swap trackers and tracker components, and test sensor fusion architectures. With your system models in place, you can integrate metrics to evaluate the overall tracking results as compared with the simulated truth.

Sensor Fusion and Tracking Toolbox and Automated Driving Toolbox provide a complete workflow for perception systems, as shown in Figure 2.

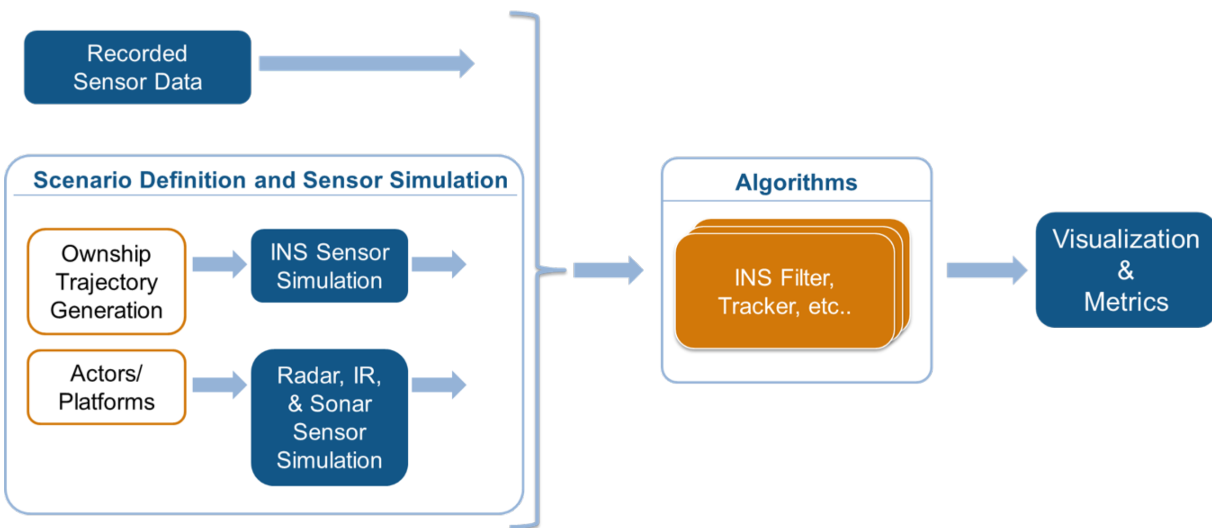

Figure 2. Sensor fusion and tracking workflow.

Defining a Scenario

With the scene generation tools, it is possible to focus on corner cases that may be difficult to capture with data recorded in the field. Complex scenes can be set up and detections can be synthesized directly to test algorithms and configurations to augment live testing.

You can use the scenario generation tools to define the ground truth. Platforms can be represented as either point objects or extended objects. Position, including pose and orientation, in space over time, is defined by attaching a trajectory to an object. Objects can be assigned multiple signatures. These signatures define how that object interacts with the sensor simulation. You can associate sensors with the platform, which allows them to sense other platforms in the environment. Detection generators, which are based on configurable statistical sensor models, are available for radar, camera, EO/IR, and sonar sensors.

Due to the physical mounting of the sensors on the vehicle, each sensor can have a different orientation relative to the vehicle. Each sensor can also have a defined field of view with a configurable update rate. Vision sensor models return a single detection from each detectible vehicle, assuming a monocular camera model and a detector that uses classification to limit false alarms. Radar sensors return multiple detections from each object, depending on their azimuth and range resolutions.

Once the platforms and sensors are created, scenarios can be defined. Figure 3 illustrates how you can generate a driving scenario for autonomous ground vehicles. Here, waypoints (shown in Figure 3 as blue dots) are used to generate the trajectory, and the scenario can be played out.

```
% Create driving scenario
s = drivingScenario('SampleTime', 0.05);

% Add road
roadCenters = [0 0; 10 0; 40 20; 50 20]; % (m)
roadWidth = 5; % (m)
road(s,roadCenters,roadWidth)
plot(s)

% Add vehicle
egoCar = vehicle(s);
waypoints = roadCenters; % (m)
speed = 13.89; % (m/s) = 50 km/hr
trajectory(egoCar, waypoints, speed);

% Play scenario
while advance(s)
    pause(s.SampleTime);
end
```

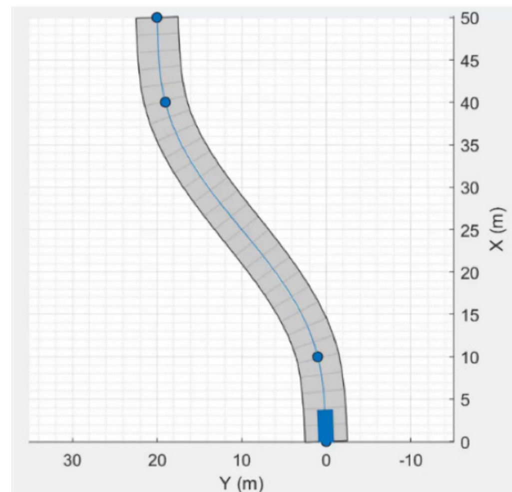

Figure 3. Example of programmatic generation of road scene (left) and the corresponding visualization (right).

Generating this type of driving scene is also possible using the Driving Scenario Designer app, which is shown in Figure 4. Note that the scenario can be defined interactively in the app. Once the scenario is defined, you can export the scene to a MATLAB script to recreate the same scenario programmatically.

MathWorks®

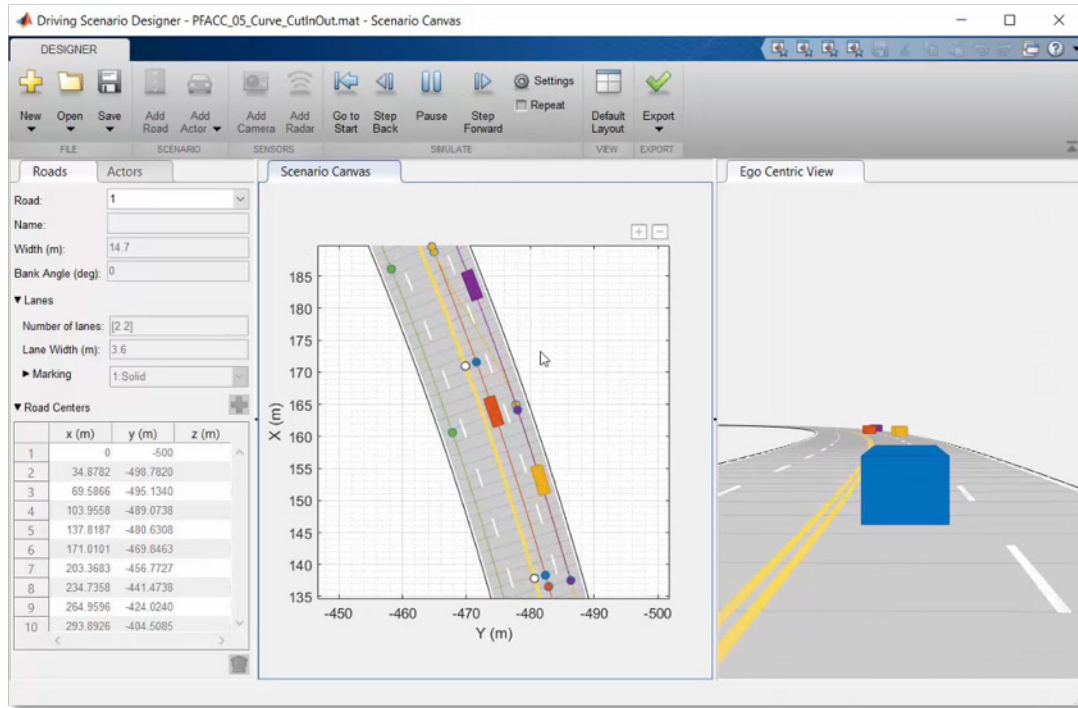

Figure 4. Generating scenarios using the Driving Scenario Designer app.

You can also generate a scenario from recorded vehicle data. Using data you import into MATLAB, you can visualize video from cameras and import OpenDRIVE roads, GPS positioning data, and object lists. This data can be used to create a scenario that can then be used to drive your simulation.

For more generic scene generation with autonomous systems, you have two options. Trajectories can be defined kinematically using specified acceleration and angular velocity. An alternate approach is to use specified waypoints with the time of arrival, velocity, and orientation at each waypoint. Figure 5 shows an example of how these can be used to generate complex trajectories for six airborne platforms.

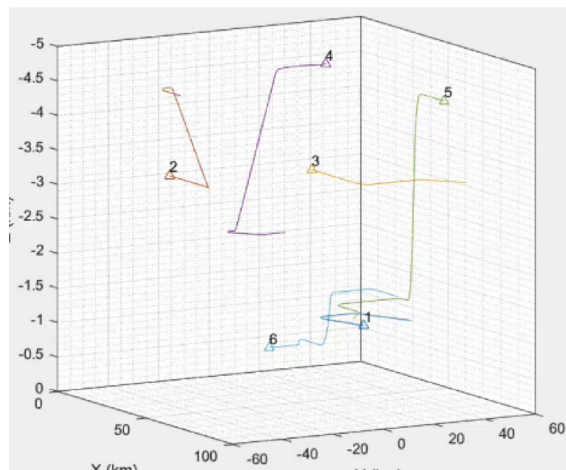

Figure 5. Trajectories for maneuvering airborne platforms.

Localization

Autonomous systems must be able to estimate their location to traverse a route or to share information in a global frame with other vehicles in their environment. With Sensor Fusion and Tracking Toolbox, you can develop a localization system for an autonomous vehicle including a simulation of a vehicle pose in space. This can be done with streaming and recorded data. You can also use sensor models in the toolbox to simulate accelerometer, gyroscope, magnetometer, GPS, and altimeter data. Libraries of algorithms are available to fuse the inputs from simulated or real sensors. You can visualize and evaluate the results of the algorithm. The overall workflow is shown in Figure 6.

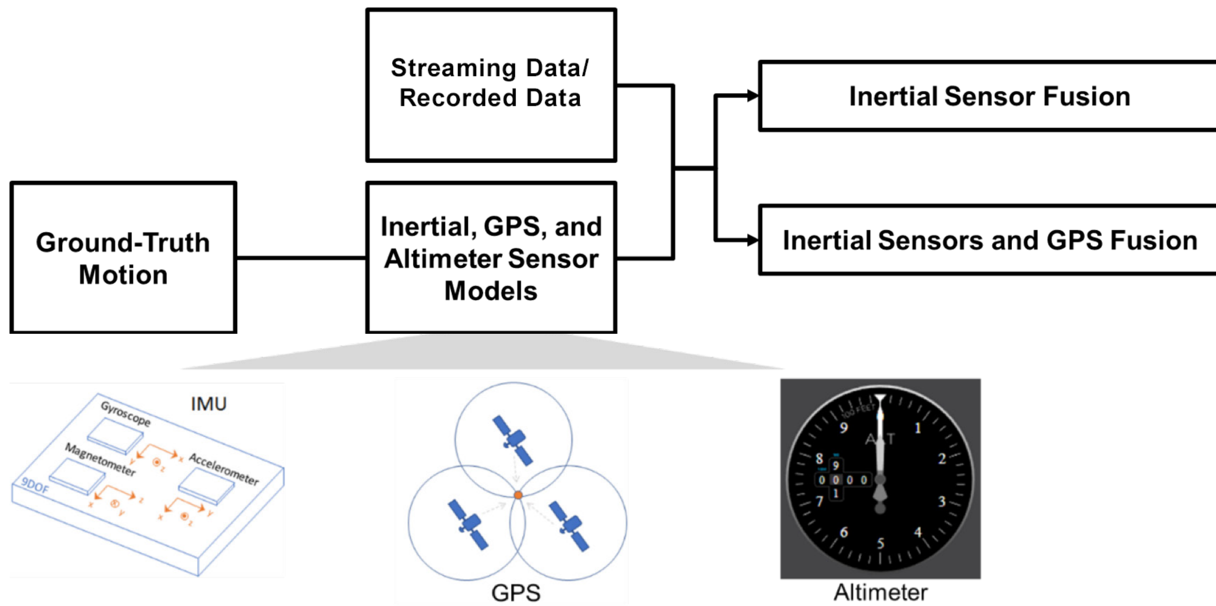

Figure 6. Workflow for localization of an autonomous system.

In the case of GPS-denied environments such as an urban canyon, inertial measurements from an IMU can be fused with other global reference sources such as visual odometry to provide improved positional accuracy, as shown in Figure 7.

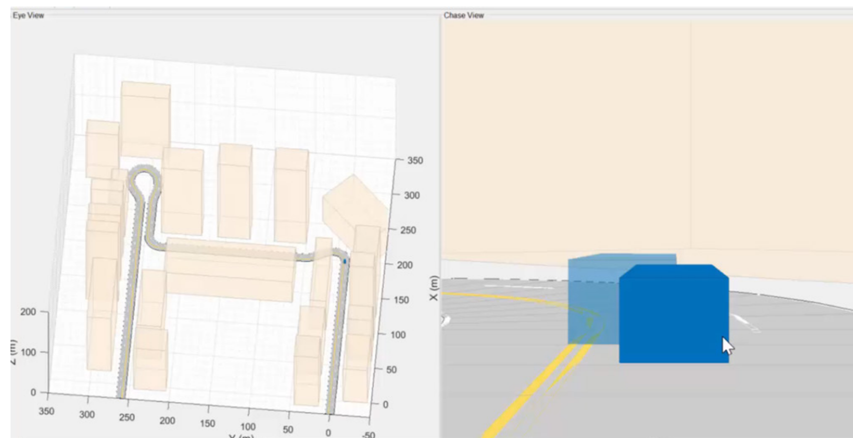

Figure 7. Urban canyon where GPS signal is lost (left) and ground truth vs. estimated position after visual odometry-inertial fusion (right).

MathWorks®

Interfacing to the Trackers

Localizing an autonomous vehicle is necessary, but insufficient, to make decisions and plan the platform motion. Autonomous systems must track the objects around them so that they can predict their motion over time and plan their own motion. In addition, autonomous vehicles require an estimate of the occupied and unoccupied space around them, and whether the occupied space is moving relative to the autonomous vehicles, to avoid collisions. This is where multi-object trackers come into play. Before this is discussed in more detail, it is important to understand the input into the trackers.

To interface with the trackers, a common API for detections is provided that contains the necessary information for the tracker to act on, including time of detection, measurement, uncertainty, classification, and information about the sensor that made the detection along with its coordinate system. The format and definition are shown in Figure 8. Note that no assumptions are made, which means you can set how a detection is defined and measured.

```
objectDetection with properties:

                   Time: 1
            Measurement: [3×1 double]
       MeasurementNoise: [3×3 double]
            SensorIndex: 1
          ObjectClassID: 0
  MeasurementParameters: [1×1 struct]
       ObjectAttributes: {[1×1 struct]}
```

Property	Description	Importance
Time	Time at which measurement was taken	Needed to predict tracks to exact time of measurement and calculate distance from it
SensorIndex	Unique identifier of sensor in the system	Needed for tracker to distinguish between sensors in multi-sensor system and assign up to one detection per sensor to a track
Measurement	What the sensor measures	Used for evaluating and correcting the filter
MeasurementNoise	Uncertainty covariance of measurement	Used for evaluating and correcting the filter
MeasurementParameters *(Optional)*	List of parameters required by nonlinear measurement function	Needed for evaluating and correcting nonlinear filters if the measurement function requires it
ObjectClassID *(Optional)*	Integer representing the classification of the object, e.g. unknown vs. car vs. pedestrian	Detection with any known class that is assigned to a track immediately confirms the track Tracks with known class cannot be assigned to detections with different known class (e.g., no assignment of car to pedestrian)
ObjectAttributes *(Optional)*	Additional information the sensor can provide (will not be used by the tracker)	Pass additional information from the detection to the track

Figure 8. Defining properties using the objectDetection class.

Sensor models included in the toolboxes generate detections that conform to the API. Examples are provided for active and passive sensor modalities, where the detections vary based on the existence of measured quantities (e.g., range).

Because the API into objectDetection is generic, it is possible to format detections from real sensors to match the API such that all downstream tracking and fusion algorithms, visualization, and metrics can be used directly.

Tracking Algorithms

Configurable multisensor, multi-object trackers are available to start your development. Each multi-object tracker handles the full life cycle of tracks including track creation, confirmation, maintenance, and deletion. The trackers are set up to run out of the box, but all the tracker components are highly configurable. Moving between trackers is easy because of the programming interface across the library.

Configurability at the tracker component level is also important because dynamic applications require estimation filters to smooth noisy detection information and convert it to the state representation appropriate to the application. There are many different aspects in the choice of a filter, such as the underlying motion of the objects in the

MathWorks®

environment and their maneuverability, the numerical stability of the calculation in single or double precision, and the computational complexity vs. processing capability of the autonomous platform.

Tracking filters are provided as building block components, and they can be used across the tracker library to optimize system performance and computational requirements. These filters include:

- Linear Kalman filter (KF)
- Extended Kalman filter (EKF)
- Unscented Kalman filter (UKF)
- Cubature Kalman filter (CKF)
- Alpha-beta-gamma filter
- Particle filter
- Range- and angle-parameterized EKF
- Interacting multiple model (IMM) filter

To make it easy to swap filters and to extend the list even further, a common interface is used for all the filters. For example, if you want to compare the results of a constant velocity (CV) extended Kalman filter with the results of an interacting multiple model filter that has both constant velocity and constant turn models, the what-if analysis is done by simply modifying one property of the tracker. With this interface, you can also provide a function handle to your own tracking filters and motion models for integration with the trackers in the toolbox library.

Beyond the filters, confirmation and deletion logic based on track history and track score is provided. In addition, a complete set of data association algorithms that solve both 2D and S-D association for 1-best or k-best solutions are also available as building blocks.

The output of the multi-object trackers consists of lists of tracks, which are estimates of truth objects divided into confirmed and tentative tracks.

Tracking Objects Around the Autonomous Vehicle

Multi-object tracking is a well-established area with a range of known solutions. Conventional tracking solutions vary based on how they associate a new set of detections with tracked objects. This includes single-hypothesis trackers, multiple-hypothesis trackers, and joint probabilistic detection association (JPDA) trackers. Newer methods, based on random finite sets (RFS), have emerged in recent years and include various flavors of probability hypothesis density (PHD) trackers. All these algorithms are included in Sensor Fusion and Tracking Toolbox and can help you accelerate your development. The overall tracking workflow is similar to the one described for localization in that data can be provided from real sensors or simulated sensors.

The conventional trackers (e.g., trackerGNN, trackerJPDA, and trackerTOMHT) assume one detection per object per scan. In cases where the sensor returns multiple detections per object, these detections must first be clustered into one representative detection that accounts for the cluster of detections. It may provide a single point and larger covariance or may include the shape of the detected object (e.g., by a bounding box). The tracker models the object as a point target and tracks its kinematic state.

MathWorks®

For high-resolution sensors such as lidar, multiple detections of an extended object can be converted into a single parameterized shape detection. The shape detection includes the kinematic states of the object, as well as its extent parameters such as length, width, and height. The shape detection can also be processed by a conventional tracker, which models the object as a bounding box and tracks both the object kinematic state and its dimensions.

In the example shown in Figure 9, lidar detections of vehicles that surround the ego-vehicle are converted into cuboid detections in the form of bounding boxes with a defined length, width, and height. The bounding boxes are fit onto each cluster by using minimum and maximum of coordinates of points in each dimension. The detector wraps the bounding box around the point cloud after it is segmented into objects.

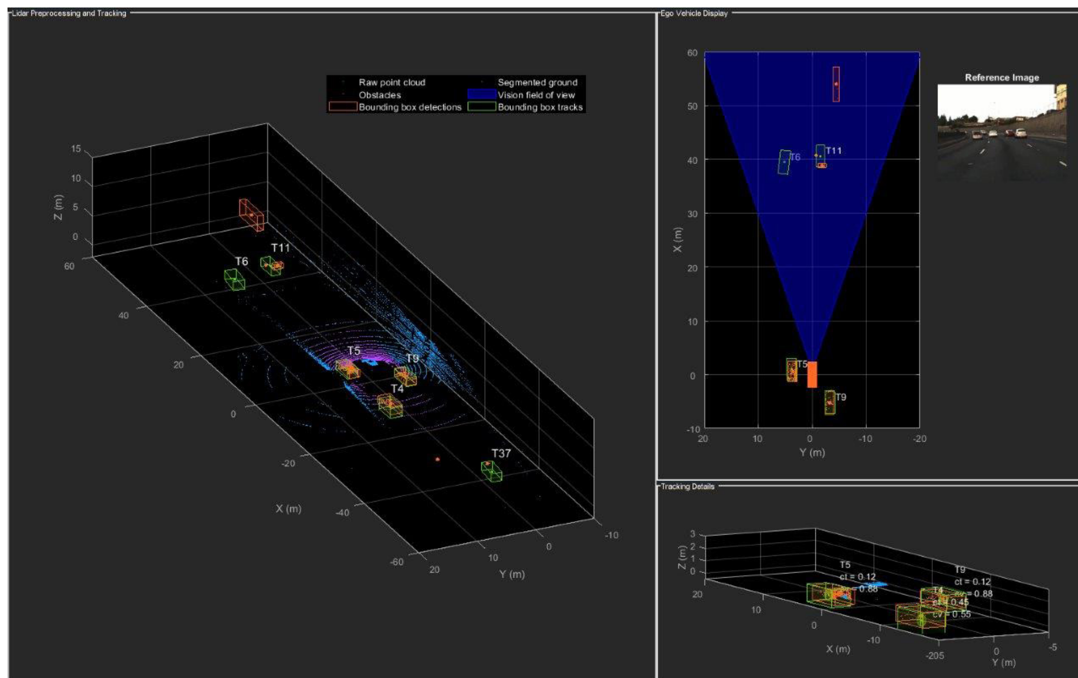

Figure 9. Lidar tracker using bounding box detections.

The JPDA tracker is used to track the position, velocity, and dimensions for all the vehicles with these cuboid detections, as shown in Figure 10.

MathWorks®

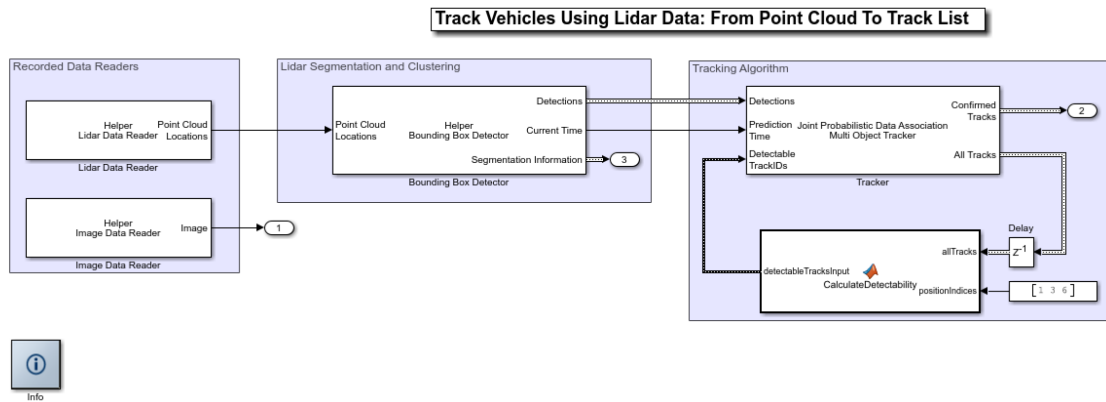

Figure 10. Lidar tracker bock diagram.

Once a system is modeled, MATLAB Coder™ can be used to generate C code, as shown in Figure 11. This C code can be used to significantly accelerate simulation times, integrate into a larger simulation framework, or deploy a prototype to a general-purpose processor.

Source Code Generated Successfully

You can now use the C code in your applications. Learn more

Project Summary

Functions	mexLidarTracker.m
Project Type	MATLAB Coder
Numeric conversion	None
Project File	mexLidarTracker.prj

Generated Output

C Code	...R2019a\driving_fusion_vision\TrackVehiclesUsingLidarExample\codegen\lib\mexLidarTracker
Binaries	C:\work\Br2019ad\matlab\bin\win64\tbbmalloc.dll C:\work\Br2019ad\matlab\bin\win64\libmwkdtree.dll C:\work\Br2019ad\matlab\extern\lib\win64\microsoft\libmwkdtree.lib C:\work\Br2019ad\matlab\extern\lib\win64\microsoft\libmwComputeMetric.lib
Example main Files	...iving_fusion_vision\TrackVehiclesUsingLidarExample\codegen\lib\mexLidarTracker\examples
Reports	Code Generation Report

Figure 11. Using MATLAB Coder to generate C code.

MathWorks®

Tracking Extended Objects

As noted previously, when more than one detection is generated, some form of clustering algorithm must be implemented. Two common issues arise with clustering. The first is that you may lose valuable information on size and orientation of the object under track. This also results in inconsistent center of turn positioning, which will vary based on the aspect angle of the sensor with respect to the object being tracked.

The second issue is that imperfect clustering can result in false tracks or dropped tracks. This causes confusion in the perception system and may lead to undesired behavior by downstream algorithms like a decision to perform emergency braking. An alternate approach is to move to an extended object tracker. As shown in Figure 12, extended objects can have multiple detections per object per scan. This type of scenario is typical for a 2D radar. Here, the extended object is mapped to 2D in the form of either an ellipse or a rectangle.

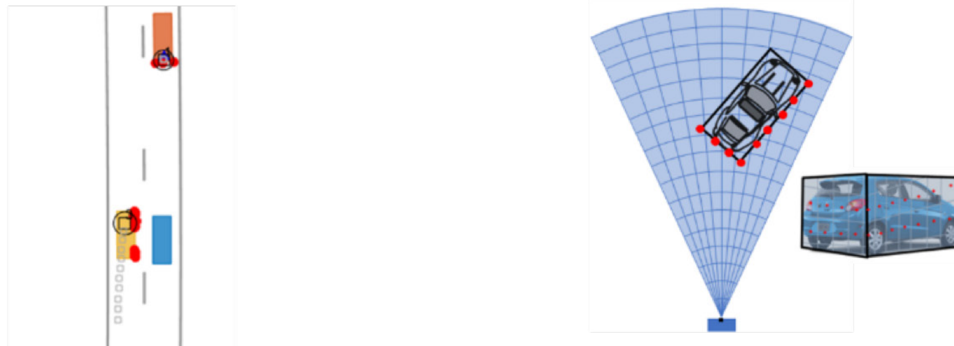

Figure 12. Extended objects represented by more than one detection per object per scan.

These results are shown in Figure 13. Note how the extended object tracker handles multiple detections per object per sensor, without the need to cluster these detections first. By using the multiple detections, the tracker estimates the position, velocity, dimensions, and orientation of each object. The dashed elliptical shape in the figure demonstrates the expected extent of the target. In this example, we use a Gamma Gaussian Inverse-Wishart probability hypothesis density (GGIW-PHD) filter, which assumes that detections are distributed around the target's elliptical center. Therefore, the tracks tend to follow observable portions of the vehicle. Such observable portions include, for example, the rear and front faces of the vehicles directly ahead of and behind the ego vehicle, respectively. In contrast, the length and width of the passing vehicle was fully observed during the simulation. Therefore, its estimated ellipse has a better overlap with the actual shape.

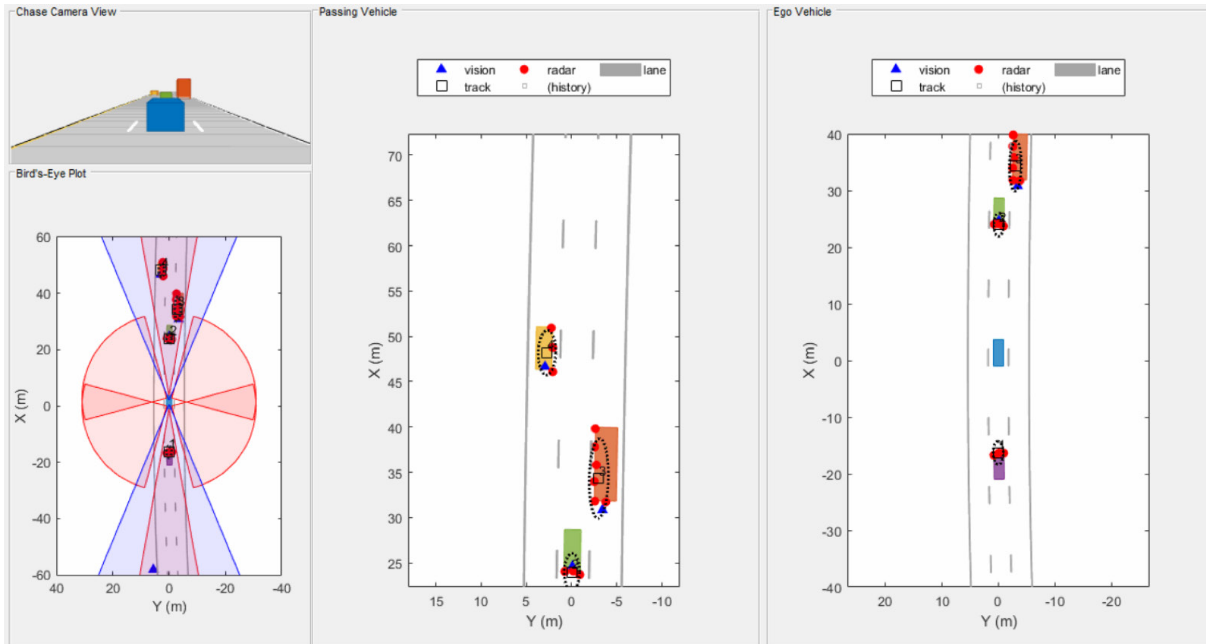

Figure 13. Results of extended object tracker using an ellipse.

In Figure 14, you can see the results using a rectangle as the extended object shape.

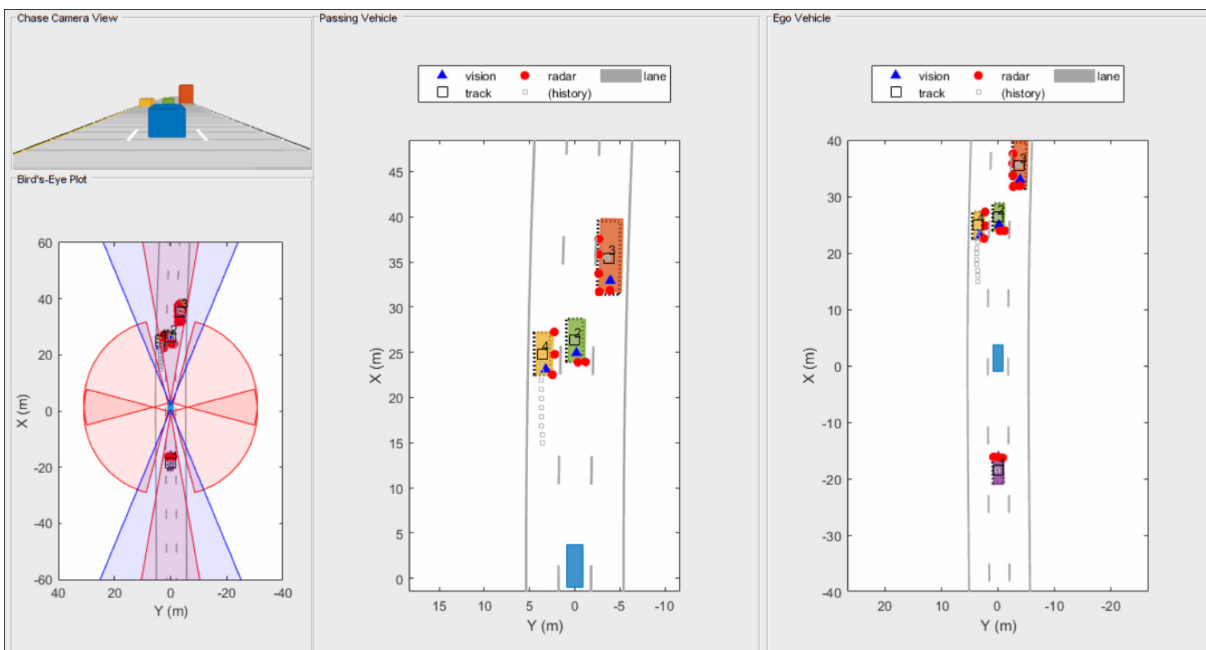

Figure 14. Results of extended object tracker using a rectangle.

MathWorks

The plots in Figure 15 show the average estimation errors for three types of trackers. Because the point object tracker does not estimate the yaw and dimensions of the objects, they are not shown in the dimension and yaw subplots. The point object tracker is able to estimate the kinematics of the objects with a reasonable accuracy. The position error of the vehicle behind the ego vehicle is higher because it was dragged to the left when the passing vehicle overtook this vehicle. This is also an artifact of imperfect clustering when the objects are close to each other.

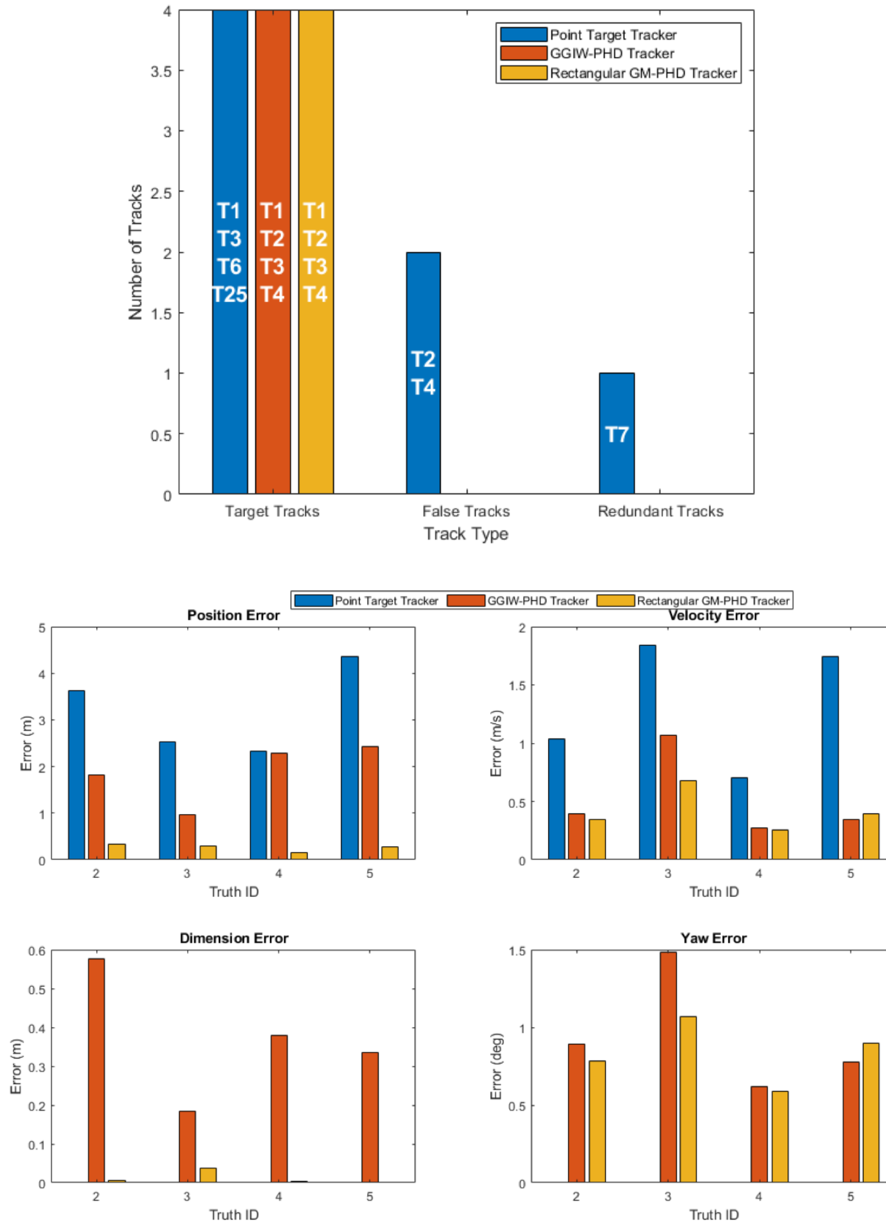

Figure 15. Benchmarks comparing data association and positional accuracy.

MathWorks®

As described earlier, the GGIW-PHD tracker assumes that measurements are distributed around the object's extent, which results in center of the tracks on observable parts of the vehicle. This can also be seen in the position error metrics for TruthID 2 and 4. The tracker is able to estimate the dimensions of the object with about 0.3 meters accuracy for the vehicles ahead of and behind the ego vehicle. Because of higher certainty defined for the vehicles' dimensions, the tracker does not collapse the length of these vehicles, even when the best-fit ellipse has a very low length. As the passing vehicle (TruthID 3) was observed on all dimensions, its dimensions are measured more accurately than the other vehicles'. However, as the passing vehicle maneuvers with respect to the ego vehicle, the error in yaw estimate is higher.

For the extended object tracker that uses a rectangular shaped target, the tracker estimates the shape and orientation more accurately. However, the process is computationally more expensive than using the other techniques.

You can also compare the run times of the trackers (Figure 16), which will give you a sense of the computational intensity of each approach. Comparing run times can help you find the balance between system performance and fitting the algorithm of choice onto your end processor architecture.

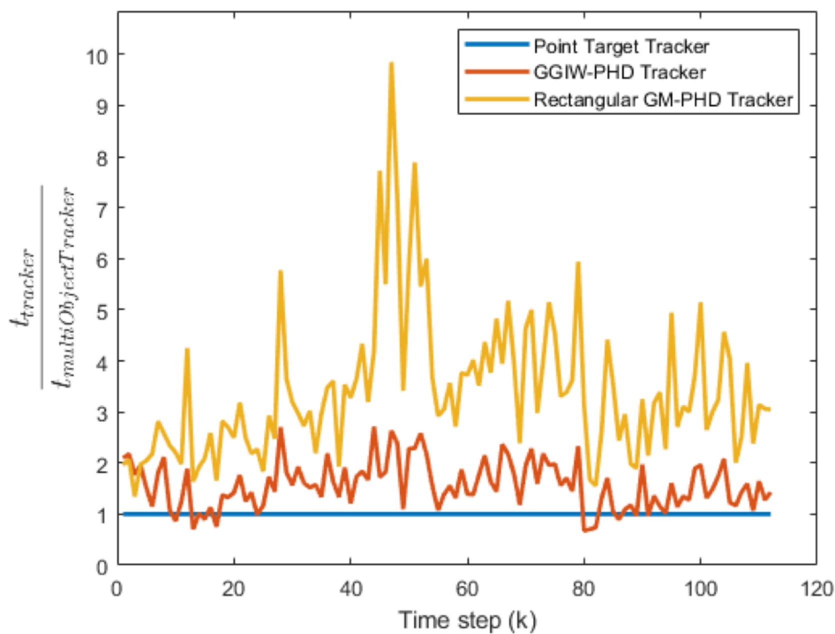

Figure 16. Comparison of performance of tracking approaches.

MathWorks®

Summary

Autonomous systems require sensing, perception, planning, and execution. While sensing is usually done by sensors that are readily available, and execution is controlled by well-known and mature control algorithms, the stages of perception and planning pose the core of research and development efforts for autonomous systems.

Using MATLAB and Simulink, you can focus your efforts on developing more advanced decision and planning algorithms. With a library of algorithms and tracking building blocks, you can find the most suitable perception solution to meet the requirements and sensor suite of your autonomous system. You can share and reuse perception algorithms across researchers and developers, teams, and even organizations.

Next Steps

Get started on your next autonomous systems project with these examples:

Sensor Fusion and Tracking Toolbox - Examples

Automated Driving Toolbox - Examples

© 2019 The MathWorks, Inc. MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See mathworks.com/trademarks for a list of additional trademarks.
Other product or brand names may be trademarks or registered trademarks of their respective holders.