Automated Driving Toolbox™ Release Notes

MATLAB®&SIMULINK®

MathWorks®

How to Contact MathWorks

Latest news:	`www.mathworks.com`
Sales and services:	`www.mathworks.com/sales_and_services`
User community:	`www.mathworks.com/matlabcentral`
Technical support:	`www.mathworks.com/support/contact_us`
Phone:	508-647-7000

The MathWorks, Inc.
1 Apple Hill Drive
Natick, MA 01760-2098

Automated Driving Toolbox™ Release Notes

Contents

R2022a

R2018b

R2018a

R2017a

R2022a

Ground Truth Labeling

Labeler Enhancements: 3D line ROI labels for point clouds

The following table describes enhancements for these labeling apps:

- **Image Labeler**
- **Video Labeler**
- **Ground Truth Labeler**
- **Lidar Labeler**

Enhancement	Image Labeler	Video Labeler	Ground Truth Labeler	Lidar Labeler
Draw, visualize, and export semantic labels in the point cloud.	No	No	No	Yes
Create training data for object detection in point clouds by using the `lidarObjectDetectorTrainingData` function.	No	No	No	Yes
Import multiframe DICOM images.	Yes	No	No	No
Create 3-D line ROI for point cloud data.	No	No	Yes	Yes
Create voxel ROI for point cloud data.	No	No	No	Yes
Show or hide pixel labels in a labeled image or video.	Yes	Yes	Yes	No

File I/O

ADTF File Reader: Read data from Automotive Data and Time-Triggered Framework (ADTF) DAT file

Automated Driving Toolbox now supports reading data from files stored in the Automated Data and Time-Triggered Framework (ADTF), developed by Elektrobit for automated driving applications. Use the `adtfFileReader` object to read stream information and inspect the contents of an ADTF DAT file. To select the messages of a specific sensor from this file, use the `select` function. You can then use the `read` or `readNext` function to read the messages contained in the file, and use these messages in automated driving workflows. For an example, see “Read Data From ADTF DAT Files”.

Reading ADTF DAT files is not supported for Mac platforms.

Cuboid Scenario Simulation

Ultrasonic Sensor Model: Generate synthetic range measurements from programmatic driving scenarios and Driving Scenario Designer app

Use the `ultrasonicDetectionGenerator` System object™ to model an ultrasonic sensor and generate synthetic range data for actors in a `drivingScenario` object. To visualize the ultrasonic detections on a bird's-eye plot, create a `rangeDetectionPlotter` object, and then plot the set of ranges using the `plotRangeDetection` function.

In the **Driving Scenario Designer (DSD)** app, you can now model an ultrasonic sensor and generate synthetic range data from a driving scenario. The bird's-eye-plot in the DSD app visualizes the ranges detected by the ultrasonic sensor as arcs. When you export a scenario containing an ultrasonic sensor to MATLAB®, the sensor is represented as an `ultrasonicDetectionGenerator` System object.

Bird's-Eye Scope Enhancement: Run simulations from previously saved models without finding signals again

When visualizing signals in Simulink® models by using the **Bird's-Eye Scope**, you can now immediately visualize signals logged from the last time you saved and closed the model. Previously, when reopening a model, you had to click **Find Signals** to find all signals in the model again before visualizing them. To find and visualize new signals in a reopened model, click **Update Signals**.

Radar Sensor Performance Enhancement: Simulate driving scenarios with radar sensors faster in MATLAB and Simulink

Radar sensors modeled using `drivingRadarDataGenerator` system object or Driving Radar Data Generator block now have improved simulation performance in complex driving scenarios with extended targets. For a driving scenario containing 7 radar sensors, a 42% average performance improvement has been observed on Windows 10 platform.

ASAM OpenSCENARIO Export Enhancements: Export road networks, actors, and trajectories to ASAM OpenSCENARIO file version 1.1

You can now export a driving scenario to ASAM OpenSCENARIO® file version 1.1 by using the **Driving Scenario Designer** app or the `export` function of the `drivingScenario` object.

Use the `OpenSCENARIOVersion` name-value argument of the `export` function to specify the version for the file. For example:

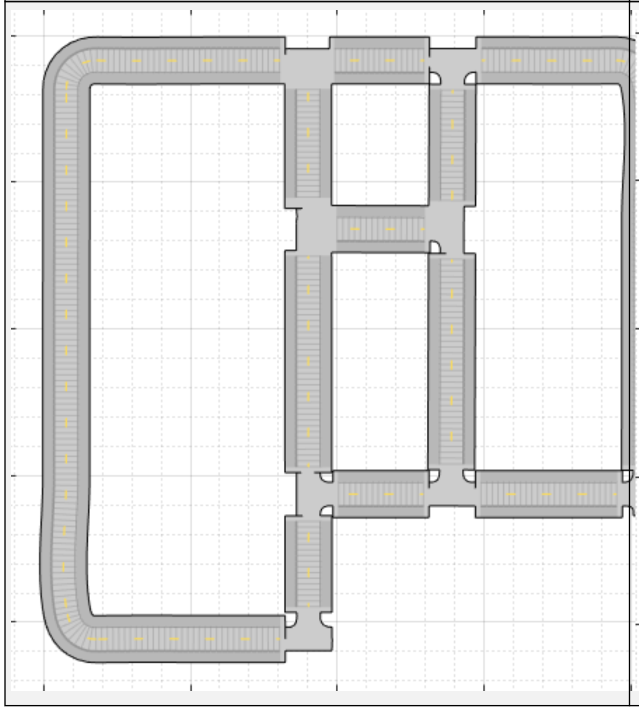
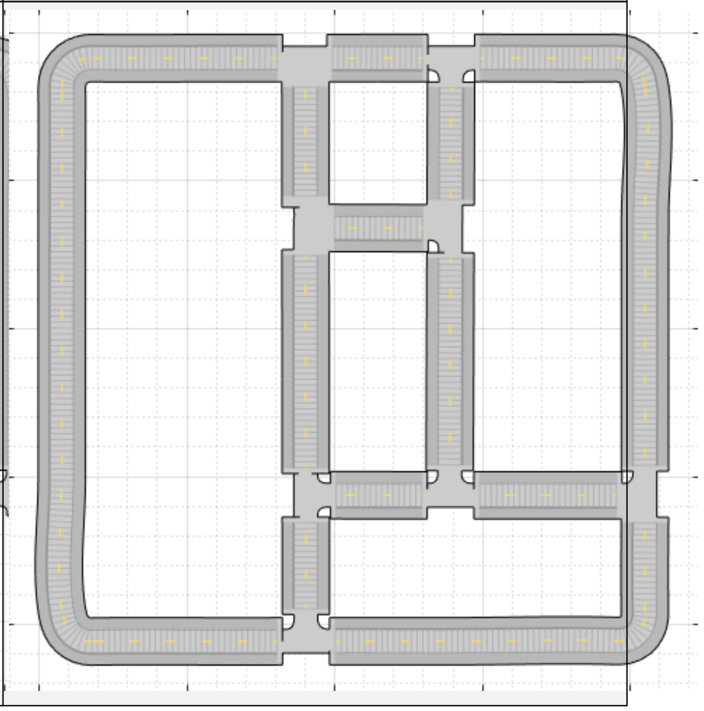
```
filename = "newfile.xosc";
export(scenario,"OpenSCENARIO",filename,OpenSCENARIOVersion=1.1);
```

Sharp Curvature Roads: Create or import roads with sharp curvature

You can now create or import roads with sharp curvature using the `road` function or `roadNetwork` function, respectively, of the `drivingScenario` object. Previously, creating or importing sharp curvature roads was not supported.

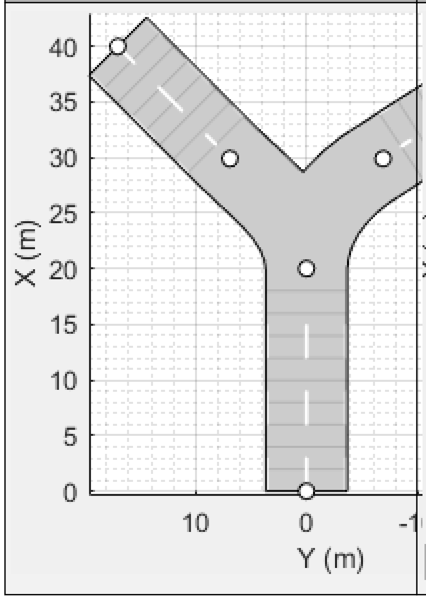
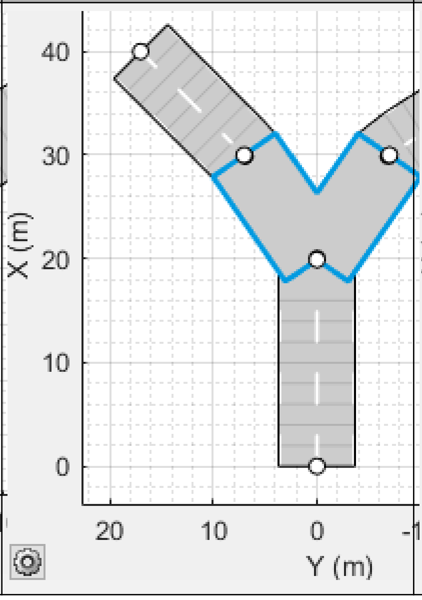
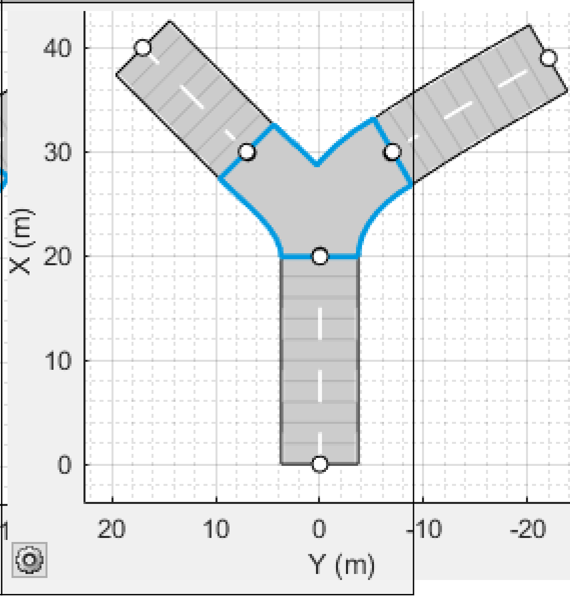
You can also interactively create or import roads with sharp curvature using the **Driving Scenario Designer** app.

This table shows an example of enhanced ASAM OpenDRIVE® road network imported using R2022a compared to the road network imported using R2021b.

R2021b	R2022a
	

Road Group Enhancements: Import heading angle information of road groups into the Driving Scenario Designer app

When you import a `drivingScenario` object into the **Driving Scenario Designer** app, you can now import heading angles of road segments stored within the `RoadGroup` object. Previously, heading angle information of road groups was not imported into the app. In addition, you can also export the heading angle information of road groups to a MATLAB function from the app. This heading angle information enables you to accurately match the shapes of road groups across programmatic and interactive workflows as shown in this figure.

Scenario created using `drivingScenario` object	Scenario imported in the app using R2021b	Scenario imported in the app using R2022a
		

Ego Localization Example: Correct ego vehicle localization using recorded sensor data

The “Improve Ego Vehicle Localization” example shows how to correct ego vehicle localization and generate an accurate ego trajectory by fusing global positioning system (GPS) and inertial measurement unit (IMU) sensor data. The example also shows how to compose a virtual driving scenario using a localized ego trajectory and OpenStreetMap® road network.

Unreal Engine Scenario Simulation

Simulation 3D Lidar Reflectivity: Model surface reflections in Unreal Engine environment

In the Simulation 3D Lidar block, use the **Reflectivity** output port to output the reflectivity of surface materials in the Unreal Engine® environment.

OpenCV Radial Distortion in Simulation 3D Camera Block: Simulate cameras with OpenCV supported radial distortion model in Unreal Engine Environment

In the Simulation 3D Camera block, you can now use the OpenCV six-coefficient formula for modeling radial distortion. Specify the formula to the **Radial distortion coefficients** parameter. This is in addition to the two-coefficient and three-coefficient models already supported by camera calibration tools in Computer Vision Toolbox™. For more information on calibrating a camera using the six-coefficient formula, see Camera Calibration and 3D Reconstruction in the OpenCV documentation.

Simulation 3D Camera Performance Improvements: Run cameras at improved speeds during Unreal Engine simulation

The Simulation 3D Camera block now has improved simulation performance and runs at higher frame rates. This table shows the increase in frames per second (FPS) for each camera in an Unreal Engine simulation.

Number of Cameras in Simulation	Frame Rate per Camera (R2021b)	Frame Rate per Camera (R2022a)	Percent Improvement per Camera
1	75.85 FPS	88.45 FPS	16.6%
4	30.51 FPS	32.80 FPS	7.5%

These simulations were timed on a Windows 10, Intel® Xeon® W-2133 CPU @ 3.60 GHz, with 64 GB of RAM and a GPU with 8 GB of on-board RAM.

These improvements enable you to run cameras at real-time speeds, provided that your system meets the requirements specified by the “Unreal Engine Simulation Environment Requirements and Limitations”.

Simulation 3D Environment Upgrade: Run 3D simulations using Unreal Engine 4.26

The 3D visualization engine that comes installed with Automated Driving Toolbox has been updated to Unreal Engine 4.26. Previously, the toolbox used Unreal Engine 4.25.

For information about using Unreal Engine to create custom scenes, see “Customize Unreal Engine Scenes for Automated Driving” and “Unreal Engine Simulation Environment Requirements and Limitations”.

Version History

If your Simulink model uses an Unreal Engine executable or project developed using a prior release of the Automated Driving Toolbox Interface for Unreal Engine 4 Projects support package, the simulation may return an error. To migrate the project so that it is compatible with the R2022a version of the support package, see “Migrate Projects Developed Using Prior Support Packages”.

Functionality being removed or changed

Updated Large Parking Lot scene
Behavior change

Starting from R2022a, the Large Parking Lot scene in the Unreal Engine 3D environment is rendered using RoadRunner. As a result, the locations of scene objects, including cones and parked vehicles, are moved from their pre-R2022a locations.

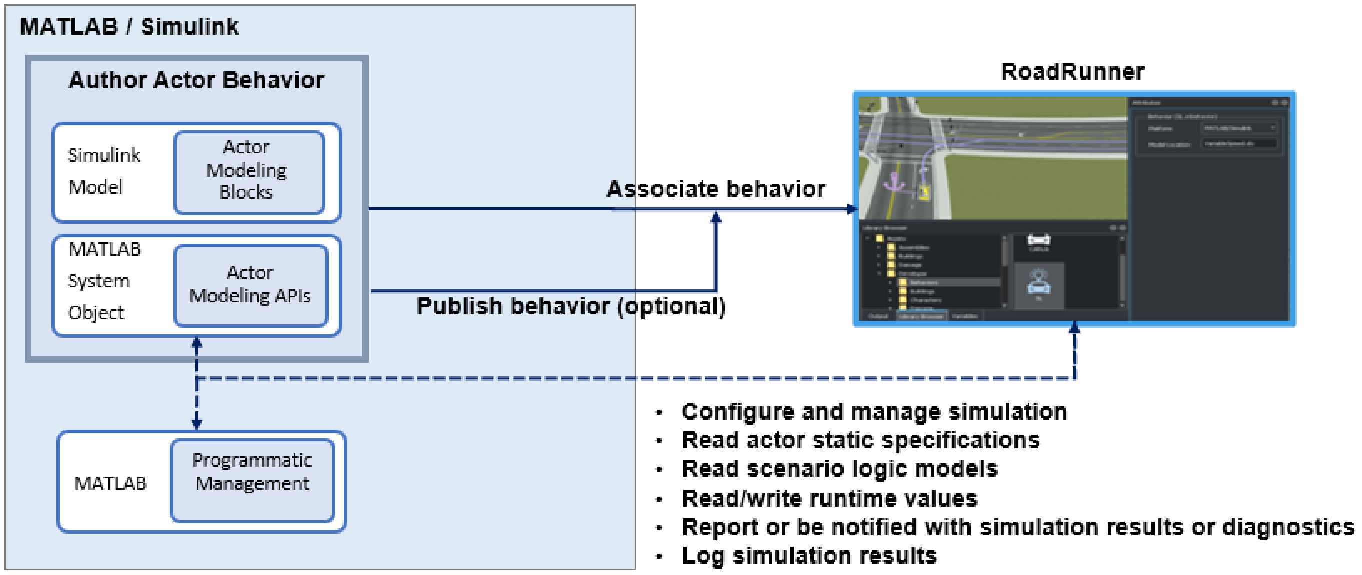
RoadRunner Scenario Simulation

Simulate RoadRunner scenarios with MATLAB and Simulink

RoadRunner is an editor that enables you to design 3D scenes for simulating and testing automated driving systems. In R2022a, Automated Driving Toolbox provides a cosimulation framework for simulating scenarios in RoadRunner with actors modeled in MATLAB and Simulink.

These are the steps of the simulation workflow:

- Author RoadRunner actors in MATLAB and Simulink. For more information, see “Simulate RoadRunner Scenarios with Actors Modeled in Simulink” and “Simulate RoadRunner Scenarios with Actors Modeled in MATLAB”.
- Associate actor behavior in RoadRunner. For more information, see “Overview of Simulating RoadRunner Scenarios with MATLAB and Simulink”.
- Optionally, publish an actor behavior. For more information, see “Publish Actor Behavior as Proto File or Package”.
- Tune the parameters defined in MATLAB or Simulink for RoadRunner simulations. For more information, see “Overview of Simulating RoadRunner Scenarios with MATLAB and Simulink”.
- Simulate a scenario using the RoadRunner user interface or control simulations programmatically from MATLAB. For more information, see “Overview of Simulating RoadRunner Scenarios with MATLAB and Simulink”.
- Inspect simulation results using data logging. For more information, see “Overview of Simulating RoadRunner Scenarios with MATLAB and Simulink”.


MATLAB / Simulink
Author Actor Behavior
Simulink Model
Actor Modeling Blocks
MATLAB System Object
Actor Modeling APIs
MATLAB
Programmatic Management
Associate behavior
Publish behavior (optional)
RoadRunner
• Configure and manage simulation
• Read actor static specifications
• Read scenario logic models
• Read/write runtime values
• Report or be notified with simulation results or diagnostics
• Log simulation results

Use these new objects to view and control the attributes of a RoadRunner scenario simulation and its associated actors through MATLAB:

- `Simulink.ScenarioSimulation` — Create, access, and control scenario simulation
- `Simulink.ActorSimulation` — Access and modify runtime specifications of actor

- `Simulink.ActorModel` — View static specifications of actor
- `Simulink.ScenarioLog` — View diagnostic information of scenario simulation

You can use these objects and functions in MATLAB System Objects to create custom RoadRunner actor behaviors.

Use these new blocks to create a model to define custom behaviors for your actors in RoadRunner using Simulink:

- RoadRunner Scenario — Establish the model interface with a scenario
- RoadRunner Scenario Reader — Read the world state, including actor pose, velocity, color, and supervisory actions
- RoadRunner Scenario Writer — Write an actor state to the scenario and report errors and warnings

Explore these examples that demonstrate speed action follower, trajectory follower, and highway lane change planner workflows with RoadRunner Scenario cosimulation:

- The “Speed Action Follower with RoadRunner Scenario” example shows how to design speed action following behavior using MATLAB. You assign this behavior to the ego vehicle in the RoadRunner scenario and control the speed of the ego vehicle to avoid collision with a lead car. The example also shows how to visualize RoadRunner Scenario simulation data using MATLAB.
- The “Trajectory Follower with RoadRunner Scenario” example shows how to control the motion of the ego vehicle in RoadRunner Scenario using Simulink to follow the specified trajectory. The example uses the Stanley controller and 3DOF vehicle dynamics to control the motion of the ego vehicle. The example also shows how to visualize RoadRunner Scenario simulation data using MATLAB.
- The “Highway Lane Change Planner with RoadRunner Scenario” example shows how to simulate a lane change behavior for the ego vehicle in a RoadRunner scenario by using Simulink. The example uses a highway lane change planner Simulink model that finds an optimal collision-free trajectory to navigate the ego vehicle.

These examples require licenses for RoadRunner and RoadRunner Scenario.

MATLAB Functions for RoadRunner Scenes and Scenarios: Import and export RoadRunner scenes and scenarios programmatically

Using the `roadrunner` object and its associated MATLAB functions, you can control the RoadRunner application programmatically. Common programmatic tasks that you can perform include:

- Open and close the RoadRunner application.
- Open, close, and save scenes and projects.
- Import and export scenes.

These MATLAB functions require an Automated Driving Toolbox license. For details on using these functions, see “MATLAB Functions for Scenes” (RoadRunner) and “MATLAB Functions for Scenarios” (RoadRunner Scenario).

Detection and Tracking

YOLO v4 Object Detection: Detect objects in monocular camera images using you only look once version 4 (YOLO v4) deep learning network

The `configureDetectorMonoCamera` function can now configure a monocular camera to use the YOLO v4 object detector, returning an `yolov4ObjectDetectorMonoCamera` object.

Bird's-Eye View Example Update: Generate code for algorithm to create 360° bird's-eye-view image around a vehicle

The “Create 360° Bird's-Eye-View Image Around a Vehicle” example now shows how to generate code for algorithm to create 360° bird's-eye-view image around a vehicle for use in a surround-view monitoring system. It also shows how to verify the generated code before deployment.

PIL Verification of JPDA Tracker Example: Generate embedded code and perform processor-in-loop (PIL) verification of JPDA tracker in highway scenarios

The “Processor-in-the-Loop Verification of JPDA Tracker for Automotive Applications” example shows how to generate embedded code for a joint probabilistic data association (JPDA) tracker configured to process detections from a camera and radar sensor mounted on the front of the ego vehicle in highway scenarios. It also shows how to verify the generated code using processor-in-loop (PIL) simulation on an STM32 Nucleo board using simulated detections.

Functionality being removed or changed

Bug fixes and behavior changes of trackingKF object
Behavior change

As of R2022a the `trackingKF` filter object has these behavior changes:

- If you set the `MotionModel` property to a predefined state transition model, such as `"1D Constant Velocity"`, you can no longer specify the control model for the filter. To use a control model, specify the `MotionModel` property as `"Custom"`.
- You must now specify the control model of the filter when creating the filter. You can no longer specify it after creating the filter.
- You can now specify the process noise for a `trackingKF` object using the `ProcessNoise` property for a predefined motion model. The dimension of the process noise matrix set through the `ProcessNoise` property now differentiates between a predefined motion model and a customized motion model. Specifically,

  - If the specified motion model is a predefined motion model, specify the `ProcessNoise` property as a *D*-by-*D* matrix, where *D* is the dimension of the motion. For example, $D = 2$ for `"2D Constant Velocity"` motion model.
  - If the specified motion model is a customized motion model, specify the `ProcessNoise` property as an *N*-by-*N* matrix, where *N* is the dimension of the state. For example, $N = 4$ if you customize a 2-D motion model in which the state is (x, v_x, y, v_y).

- The orientation of the filter state now matches the state vector that you specify when creating the filter. For example, if you set the initial state in the filter as a row vector, the filter displays the filter state as a row vector. Previously, the filter displayed the filter state as a column vector regardless of initial state.
- You can generate efficient C/C++ code without dynamic memory allocation for `trackingKF`.

Localization and Mapping

Parking Spot Detection Example: Detect empty parking spots in a parking lot using semantic segmentation

The “Perception-Based Parking Spot Detection Using Unreal Engine Simulation” example shows how to detect lane markings and obstacles in a parking lot using semantically segmented camera images, incrementally update detections in the bird’s-eye view, reconstruct parking spots from lane markings, and build a map of parking spots for decision making in an Unreal Engine simulation environment.

LOAM Example: Build map and localize using Lidar Odometry and Mapping (LOAM)

The “Build a Map with Lidar Odometry and Mapping (LOAM) Using Unreal Engine Simulation” example shows how to build a map with lidar data and localize the position of a vehicle on the map using Lidar Odometry and Mapping (LOAM), an algorithm that uses edge and surface points in the point cloud for registration and mapping.

Point Cloud Localization Example Update: Localize with a prebuilt map using NDT algorithm

The “Lidar Localization with Unreal Engine Simulation” example now shows how to localize the position of a vehicle on a prebuilt map using the Normal Distributions Transform (NDT) algorithm.

Visual SLAM Example Update: Reconstruct a parking lot from stereo images using visual SLAM

The “Develop Visual SLAM Algorithm Using Unreal Engine Simulation” example now shows how to perform dense reconstruction using stereo images of a parking lot scene in an Unreal Engine simulation environment.

Applications

Intersection Navigation Examples: Use V2V and V2X communication technologies to build applications for safe navigation through intersections

The “Intersection Movement Assist Using Vehicle-to-Vehicle Communication” example shows how to design and test an intersection movement assist (IMA) application by modeling vehicle-to-vehicle (V2V) communication. In this example, you also study the effect of channel impairments on the IMA application.

The “Traffic Light Negotiation Using Vehicle-to-Everything Communication” example shows how to design and test decision logic using vehicle-to-everything (V2X) communication to negotiate a traffic light to prevent collisions at intersections. This example uses the vehicle-to-vehicle (V2V) and vehicle-to-infrastructure (V2I) modes of V2X communication.

Autonomous Emergency Braking Examples: Integrate high fidelity vehicle dynamics model with autonomous emergency braking (AEB) system and automate testing of AEB system

The “Autonomous Emergency Braking with Vehicle Variants” example enables you to integrate either 3DOF or 14DOF vehicle models with AEB system in a closed-loop environment. Using this example, you can study interactions between an AEB controller and vehicle dynamics model, and analyze the impact of high-fidelity vehicle dynamics on AEB applications.

The “Autonomous Emergency Braking with Sensor Fusion” example has been updated to calculate the steering angle required for an ego vehicle to follow the reference path. This capability enables you to test and validate an AEB system using complex Euro NCAP® test scenarios that contain intersections.

The “Automate Testing for Autonomous Emergency Braking” example shows how to automate testing of the components of an AEB system and verify the generated code using Simulink Test™ software. You can automate testing of the sensor fusion and tracking, decision logic, and controller components.

Real-Time Testing Example: Deploy and test forward vehicle sensor fusion component in real-time

The “Automate Real-Time Testing for Forward Vehicle Sensor Fusion” example shows how to deploy a forward vehicle sensor fusion component of a highway lane following system to a Speedgoat® real-time machine and automate the regression testing of the deployed application.

Highway Lane Change Example Update: Integrate surround vehicle sensor fusion with highway lane change system

The “Highway Lane Change” example now integrates a surround vehicle sensor fusion component that provides a 360-degree view for detecting target vehicles surrounding the ego vehicle, enabling it to perform a lane change maneuver. Before R2022a, the example instead used the ground truth information of target vehicles to perform a lane change maneuver for the ego vehicle, as shown in the “Highway Lane Change Planner and Controller” example.

R2021b

Ground Truth Labeling

Labeler Enhancements: Edit cuboid ROI labels more easily in top, side, and front 2-D view projections, segment ground from lidar data using SMRF algorithm

The following table describes enhancements for these labeling apps:

- **Image Labeler**
- **Video Labeler**
- **Ground Truth Labeler**
- **Lidar Labeler** (Lidar Toolbox)

Enhancement	Image Labeler	Video Labeler	Ground Truth Labeler	Lidar Labeler
Show or hide labels and sublabels of type `Rectangle`, `Line`, `Polygon`, and `Projected cuboid` in a labeled image or video.	Yes	Yes	Yes	No
Show or hide labels of type `Cuboid` in a labeled point cloud or point cloud sequence.	No	No	Yes	Yes
View and edit cuboid ROI labels using top, side, and front 2-D view projections by selecting **Projected View**.	No	No	Yes	Yes
Segment ground from lidar data using the simple morphological filter (SMRF) algorithm. For more information about the algorithm parameters, see the `segmentGroundSMRF` (Lidar Toolbox) function.	No	No	Yes (only with Lidar Toolbox™ license)	Yes
Extract video scenes and corresponding labels from a `groundTruth` or `groundTruthMultisignal` object.	No	Yes	Yes	No
Digital Imaging and Communication in Medicine (DICOM) image format.	Yes	No	No	No

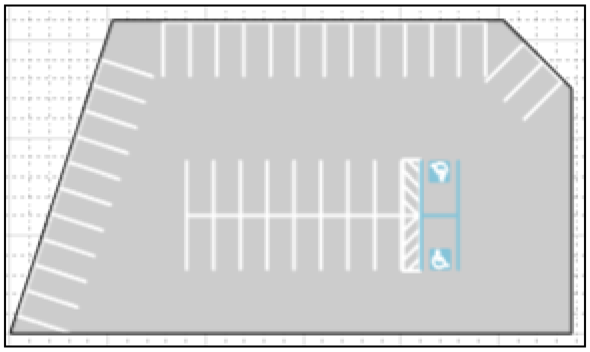
Velodyne Lidar Sources: Load data from Velodyne VLS-128 lidar device into Ground Truth Labeler app

Load data captured using the Velodyne® VLS-128 lidar device into the **Ground Truth Labeler** app. Use the `vision.labeler.loading.VelodyneLidarSource` class to load signals from the packet capture (PCAP) file data source by setting the `DeviceModel` field of the `SourceParams` property to `"VLS-128"`.

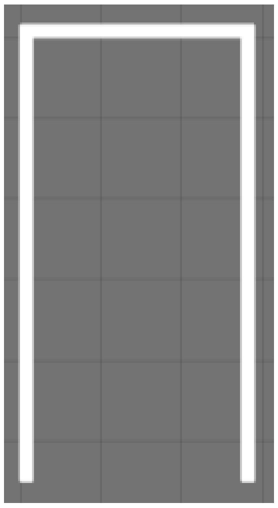
Cuboid Scenario Simulation

Parking Lots: Add parking lots to driving scenarios programmatically

In `drivingScenario` objects, use the `parkingLot` function to create parking lot environments in which to test your automated driving algorithms. You can choose from a variety of predefined parking lot layouts or design a custom layout.

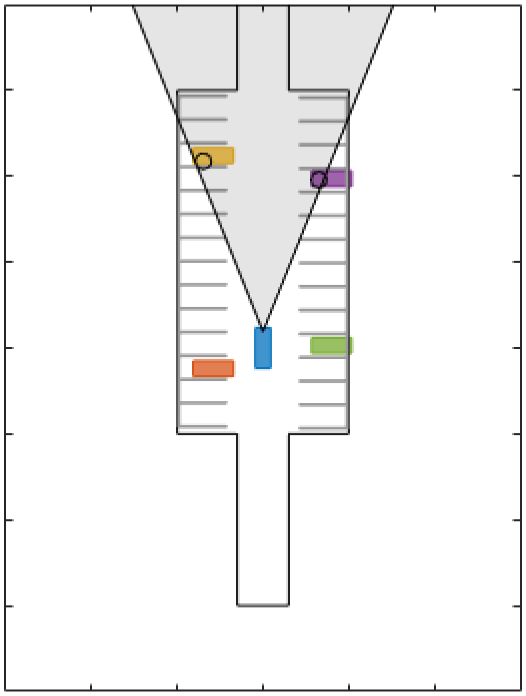


To customize the design of the parking spaces, create `parkingSpace` objects and visualize them by using the `plot` function.



To add parking spaces along the edges of parking lots or to add parking grids at specific positions or orientations, use the `insertParkingSpaces` function.

You can also visualize parking lanes on a bird's-eye plot. First, create a lane marking plotter. Then, obtain the parking lane vertices by using the `parkingLaneMarkingVertices` function and plot the lanes by using the `plotParkingLaneMarking` function.



For examples that use scenario and sensor simulation in parking lots, see Simulate Vehicle Parking Maneuver in Driving Scenario and Visualize Automated Parking Valet Using Cuboid Simulation.

Note The creation of parking lots using the **Driving Scenario Designer** app is not supported. The import of parking lots into the app is also not supported. For more details on parking lot limitations, see the `parkingLot` reference page.

ASAM OpenDRIVE Import Enhancements: Import a road network using OpenDRIVE file version V1.5 and ASAM OpenDRIVE V1.6

You can now import a road network from OpenDRIVE® file version V1.5 and ASAM OpenDRIVE V1.6 into a driving scenario by using the `roadNetwork` function of the `drivingScenario` object, or by using the **Driving Scenario Designer** app. In addition, you can now add roads and export a MATLAB function after importing the road network into the app.

ASAM OpenDRIVE Export Enhancements: Export a road network to OpenDRIVE file version V1.5 and ASAM OpenDRIVE V1.6

You can now export a driving scenario to OpenDRIVE file version V1.5 and ASAM OpenDRIVE V1.6 by using the **Driving Scenario Designer** app or the `export` function of the `drivingScenario` object.

Use the `OpenDRIVEVersion` name-value argument of the `export` function to specify the version of the file. You can also specify whether to export actors by using the `ExportActors` name-value argument. For example:

```
filename = "newfile.xodr";
export(scenario,"OpenDRIVE",filename,OpenDRIVEVersion=1.6,ExportActors=false);
```

ASAM OpenSCENARIO Export Enhancements: Export the routes of actors using instances of Trajectory element

When you export a driving scenario to an ASAM OpenSCENARIO file, the file now specifies the routes of actors using instances of the `Trajectory` element. Previously, the routes of actors were exported separately using a `RouteCatalog` file containing instances of the `Route` element.

Scenario Reader Block: Obtain position, velocity, orientation, and acceleration information from Ego Vehicle State port

The Scenario Reader block now outputs ego vehicle state information that includes the position, velocity and acceleration measurements of the ego vehicle in world coordinates. This information can be used as ground truth data for simulating sensor models, such as an INS sensor. This output is available only in open-loop workflows without ego vehicle pose input to the Scenario Reader block.

INS Block: Generate synthetic readings from an inertial navigation and GPS sensor in driving scenarios in Simulink

Use the INS block to simulate an INS sensor in Simulink. Obtain the state of the ego vehicle from the **Ego Vehicle State** output port of the Scenario Reader block. State information includes the position, velocity, orientation, and acceleration of the vehicle. Pass this ego vehicle state information as ground truth to the INS block, which then generates sensor readings at each simulation time step. For an example, see Generate INS Measurements from Driving Scenario in Simulink.

You can now also export scenarios that model INS sensors modeled using the **Driving Scenario Designer** app to Simulink. For more information, see Generate INS Sensor Measurements from Interactive Driving Scenario.

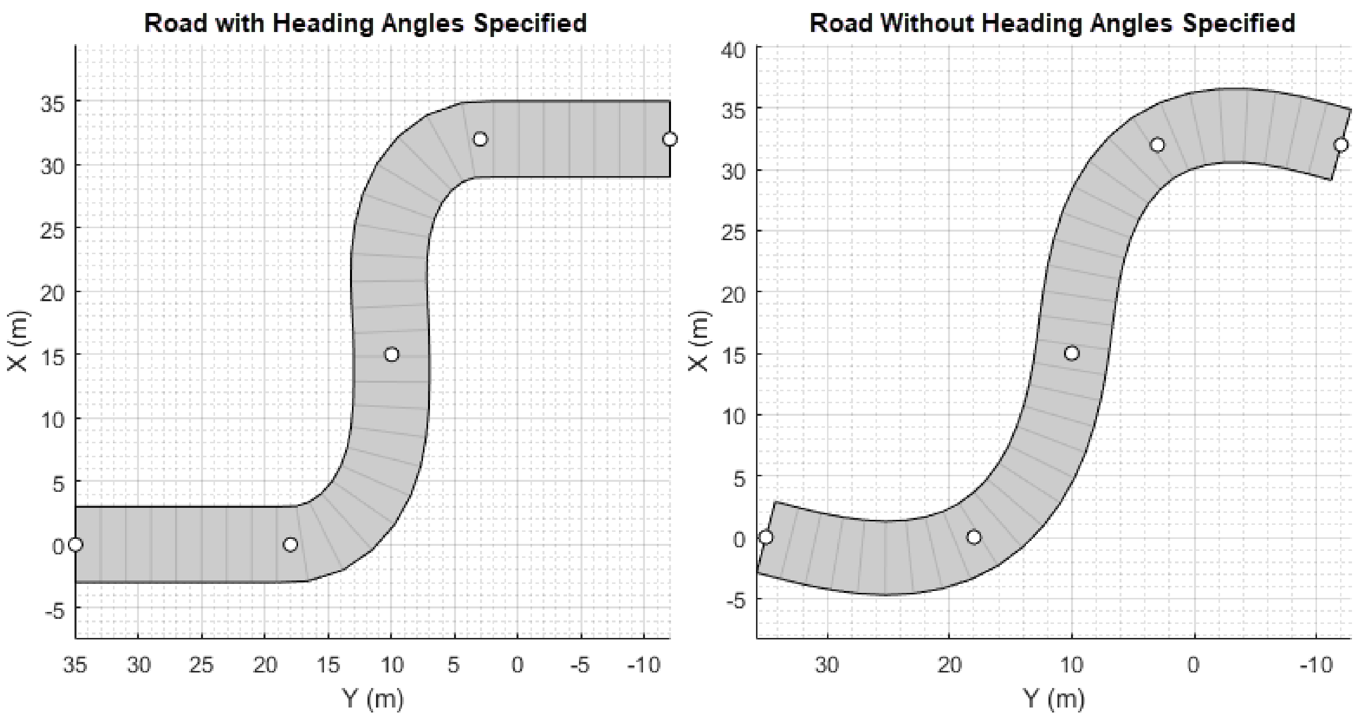
Road Heading Angles: Create more precise roads using fewer road centers

You can now specify heading angles at road centers to create roads. Specifying heading angles as a constraint to road center points enables finer control over the shape and orientation of roads using fewer road centers.

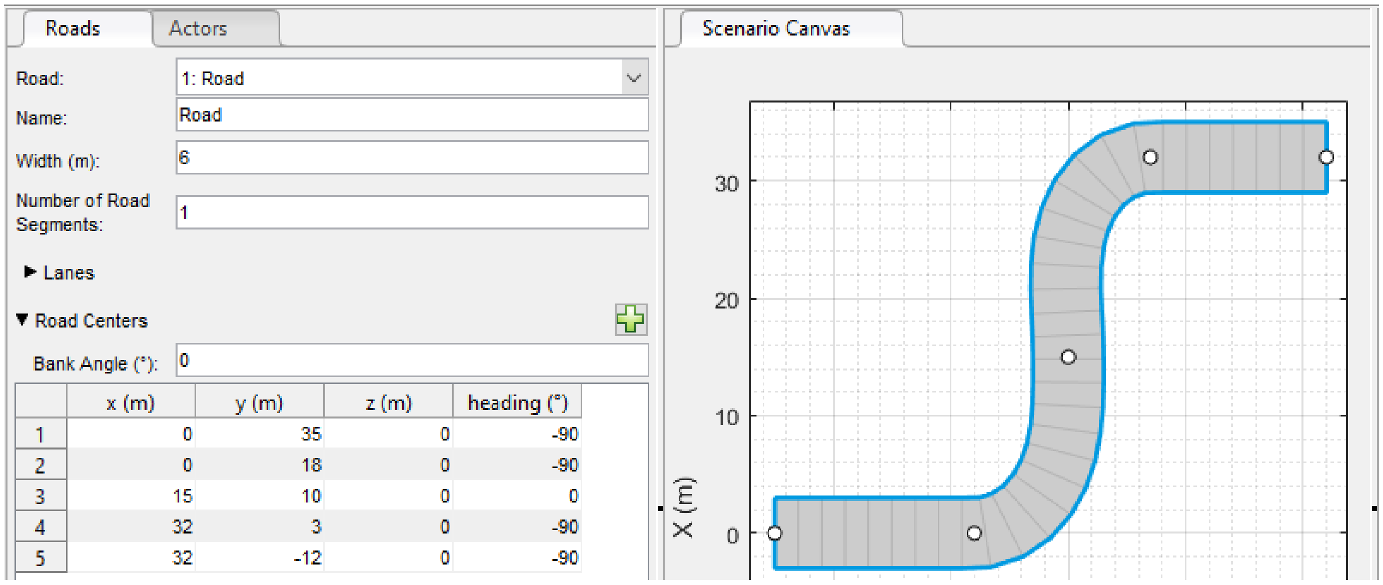
To programmatically add roads with heading angles to a `drivingScenario` object, use the `Heading` name-value argument of the `road` function. Specify the heading value as a column vector of angles in the range [–180, 180] degrees. For example:

```
road(scenario,roadCenters,Heading=[-90;-90;0;-90;-90]);
```

This figure shows two roads with the same road centers, but one has specified heading angle values and the other does not.



You can also use the **Driving Scenario Designer** app to specify road heading angles. Use the **heading** column in the **Road Centers** table to specify the heading angles at each road center.



Road: 1: Road
Name: Road
Width (m): 6
Number of Road Segments: 1
Bank Angle (°): 0

	x (m)	y (m)	z (m)	heading (°)
1	0	35	0	-90
2	0	18	0	-90
3	15	10	0	0
4	32	3	0	-90
5	32	-12	0	-90

Lane Generation Example: Add lane information to map imported road network

The Generate Lane Information from Recorded Data example shows how to generate lane information using recorded data from a camera and a GPS sensor. Use this example to add lane information to a road network imported from SD map data.

Scenario Generation Examples: Generate scenario from recorded sensor data and scenario variants from seed scenario

The Generate Scenario from Recorded GPS and Lidar Data example shows how to automatically generate a driving scenario from the data recorded by global positioning systems (GPS) and lidar sensors. You can use the generated scenario as input data to model and simulate an automated driving system.

The Automatic Scenario Variant Generation for Testing AEB Systems example shows how to automatically generate variants of a seed scenario in which two actors collide. You can generate random variants of a collision scenario and use them to design and validate an automated emergency braking (AEB) system.

Unreal Engine Scenario Simulation

Unreal Engine Environment Upgrade: Run 3D simulations using Unreal Engine, Version 4.25

The 3D simulation engine that comes installed with Automated Driving Toolbox has been updated to Unreal Engine, Version 4.25. Previously, the toolbox used Unreal Engine, Version 4.23.

For information about using Unreal Engine to create custom scenes, see Customize Unreal Engine Scenes for Automated Driving.

Version History

If your Simulink model uses an Unreal Engine executable or project developed using a prior release of the Automated Driving Toolbox Interface for Unreal Engine 4 Projects support package, the simulation might produce an error. To migrate the project so that it is compatible with the R2021b version of the support package, see Migrate Projects Developed Using Prior Support Packages.

Position Adjustments of Unreal Engine Cameras: Update relative translation and rotation of camera sensors during simulation

In the Simulation 3D Camera and Simulation 3D Fisheye Camera blocks, use the **Translation** and **Rotation** input ports to update the position of the cameras relative to their mounting positions during simulation. You can use these position adjustments to better model actuator dynamics, isolation mounting, and calibration workflows.

Previously, you could set only constant relative positions by using the **Relative translation [X, Y, Z] (m)** and **Relative rotation [Roll, Pitch, Yaw] (deg)** parameters. Now, by selecting **Input** to enable the corresponding ports, the parameters specify the initial position and the ports specify the position during simulation.

Unreal Engine Environment Performance Improvements: Run 3D simulations faster than real-time

Simulink co-simulations with Unreal Engine can now run faster than real-time. Previously, the Unreal Engine frame rate was limited by the inverse of the simulation sample rate. If you want to slow down a 3D simulation to investigate system behavior, you can still use simulation pacing.

Use the Simulation 3D Scene Configuration block parameter **Sample time** to control simulation time. For example, if **Sample time** is `1/30`, then the visualization engine solver tries to achieve a minimum frame rate of 30 frames per second (FPS). However, the real-time graphics frame rate is often lower due to factors such as graphics card performance and model complexity. With sufficient graphics card performance and low model complexity, the frame rate could be greater than 30 FPS, not limited to 30 FPS as in previous releases.

Unreal Engine Visualization Example: Visualize logged data for post-simulation analysis

The Visualize Logged Data from Unreal Engine Simulation example shows how to customize the visualization of logged sensor and simulation data using the Simulation Data Inspector. This example enables you to analyze and debug automated driving test cases after running the simulation.

Detection and Tracking

Perturbations: Perturb object properties using truncated normal distribution

You can now define the perturbation distribution of a property as a truncated normal distribution using the `perturbations` (Sensor Fusion and Tracking Toolbox) function. With offset values bounded by a finite interval, the truncated normal distribution is suitable for perturbing a property whose valid values are confined in a finite interval.

Code Generation: Generate more memory-efficient C/C++ code from trackers and tracking filters

These objects and Simulink blocks now support strict single-precision and static memory allocation code generation:

- `trackingEKF`
- `trackingUKF`
- Multi-Object Tracker

See the **Extended Capabilities** section on each object or block reference page for its code generation limitations.

Radar and Tracking Examples: Fuse radar and camera tracks, track using event-based sensor fusion and retrodiction and track in scenarios with multipath radar reflections in Simulink

The Extended Object Tracking of Highway Vehicles with Radar and Camera in Simulink example shows how to fuse radar and camera measurements to track highway vehicles with multiple extended object tracking techniques and evaluate their tracking performance in Simulink. This example requires the Sensor Fusion and Tracking Toolbox™ software.

The Event-Based Sensor Fusion and Tracking with Retrodiction example shows how to track vehicles using event-based sensor fusion of simulated radar and camera measurements in Simulink. This example requires the Sensor Fusion and Tracking Toolbox software.

The Extended Target Tracking with Multipath Radar Reflections in Simulink example shows how to model and mitigate multipath radar reflections during highway vehicle tracking in Simulink. This example requires the Sensor Fusion and Tracking Toolbox. It closely follows the Highway Vehicle Tracking with Multipath Radar Reflections (Radar Toolbox) example.

Track moving vehicles with multiple lidar sensors using a grid-based tracker in Simulink

The Grid-based Tracking in Urban Environments Using Multiple Lidars in Simulink example shows how to track moving vehicles in urban environments with measurements from multiple lidar sensors using a grid-based tracker in Simulink. This example requires the Sensor Fusion and Tracking Toolbox software. It closely follows the Grid-Based Tracking in Urban Environments Using Multiple Lidars example.

Perform dynamic replanning on highways using tracking in MATLAB

The Object Tracking and Motion Planning Using Frenet Reference Path example shows how to perform dynamic replanning on highways using a Frenet reference path and a joint probabilistic data association (JPDA) tracker in MATLAB. This example requires the Sensor Fusion and Tracking Toolbox and Navigation Toolbox™ software. It is an extension of the Highway Trajectory Planning Using Frenet Reference Path example.

Localization and Mapping

Visual Localization Example: Develop and evaluate a visual localization algorithm in a parking lot scenario

The Visual Localization in a Parking Lot example shows how to develop a visual localization system using synthetic image data from the Unreal Engine simulation environment.

Segment Matching Example: Build Map and Localize Using Segment Matching

The Build Map and Localize Using Segment Matching example shows how to build a map with lidar data and localize the position of a vehicle on the map using *SegMatch*, a place recognition algorithm based on segment matching.

Applications

Message-Based Communication: Establish message-based communication between model components

The Generate C++ Message Interfaces for Lane Following Controls and Sensor Fusion example shows how to establish message-based communication between the controller and sensor fusion components of a highway lane following system. This workflow enables you to integrate system components in a distributed architecture.

Real-Time Testing: Deploy and test highway lane following controller in real-time

The Automate Real-Time Testing for Highway Lane Following Controller example shows how to configure a hardware setup to deploy a lane following controller to a Speedgoat real-time machine. The example also shows how to automate the regression testing of the deployed application.

Automate Testing: Automate testing of components of lane following and lane changing systems

These new examples show how to automate testing and verify generated code for different components of highway lane following and highway lane change systems.

- Automate Testing for Vision Vehicle Detector
- Automate Testing for Forward Vehicle Sensor Fusion
- Automate Testing for Highway Lane Change
- Automate Testing for Highway Lane Following Controller

R2021a

Ground Truth Labeling

Labeler Enhancements: Label object instances for semantic segmentation, automate labeling of multiple signals simultaneously, and additional features

The following table describes enhancements for these labeling apps:

- **Image Labeler**
- **Video Labeler**
- **Ground Truth Labeler**
- **Lidar Labeler** (Lidar Toolbox)

Enhancement	Image Labeler	Video Labeler	Ground Truth Labeler	Lidar Labeler
Label distinct instances of objects belonging to the same class using a polygon label. For more details, see Label Objects Using Polygons.	Yes	Yes	Yes	No
Use superpixel automation to quickly pixel label regions of an image with similar pixel values. For more details, see Label Pixels Using Superpixel Tool.	Yes	Yes	Yes	No
Automate the labeling of multiple signals together within a single automation run. For an example, see Automate Ground Truth Labeling Across Multiple Signals.	No	No	Yes	No
Label very large images (with at least one dimension <8K) that previously could not be loaded into memory. Load these images as blocked images. For more details, see Label Large Images in Image Labeler.	Yes	No	No	No
Use a custom reader function to import any point cloud. For more details, see Use Custom Point Cloud Source Reader for Labeling (Lidar Toolbox).	No	No	No	Yes
Define and view a region of interest (ROI) in the point cloud and label objects in it. For more details, see ROI View (Lidar Toolbox).	No	No	No	Yes
Control the point dimension of the point cloud.	No	No	No	Yes

File I/O

Ibeo File Reader: Read sensor data from Ibeo data container (IDC) files

Ibeo Automotive Systems uses IDC files to record sensor messages from camera, lidar, GPS, and other sensors. Use the `ibeoFileReader` object to read message headers and inspect the contents of an IDC file. To select the messages of a specific sensor from this file, use the `select` function. You can then use the `readMessages` or `readNextMessage` function to read the messages contained in the file, and use these messages in automated driving workflows. For example, you can visualize image, point cloud, and object detection data, or use vehicle state data to specify vehicles in a driving scenario.

The reading of lidar data requires Lidar Toolbox.

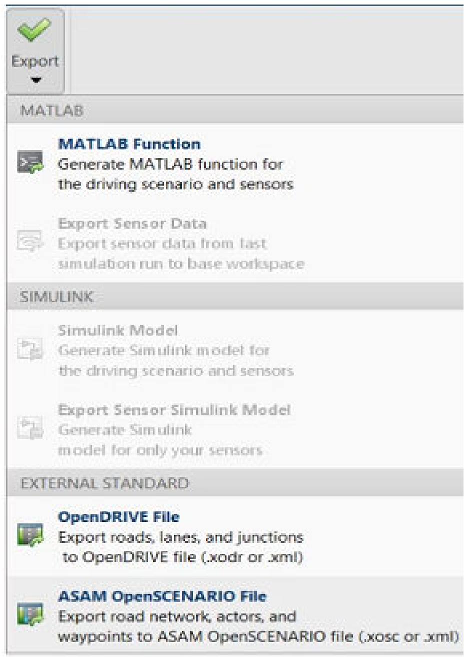
Cuboid Scenario Simulation

ASAM OpenSCENARIO Export: Share a driving scenario using the ASAM OpenSCENARIO 1.0 format

Export a driving scenario to the ASAM OpenSCENARIO format from a `drivingScenario` object or the **Driving Scenario Designer** app.

- To programmatically export a driving scenario from a `drivingScenario` object to the ASAM OpenSCENARIO format, use the `'OpenSCENARIO'` argument of the `export` function and a file name with the `.xosc` extension. For example:

  ```
  filename = 'newfile.xosc';
  export(scenario,'OpenSCENARIO',filename)
  ```
- To interactively export a driving scenario from the **Driving Scenario Designer** app to the ASAM OpenSCENARIO format, select **Export** > **ASAM OpenSCENARIO File**.


Export
MATLAB
MATLAB Function
Generate MATLAB function for the driving scenario and sensors
Export Sensor Data
Export sensor data from last simulation run to base workspace
SIMULINK
Simulink Model
Generate Simulink model for the driving scenario and sensors
Export Sensor Simulink Model
Generate Simulink model for only your sensors
EXTERNAL STANDARD
OpenDRIVE File
Export roads, lanes, and junctions to OpenDRIVE file (.xodr or .xml)
ASAM OpenSCENARIO File
Export road network, actors, and waypoints to ASAM OpenSCENARIO file (.xosc or .xml)

Driving Scenario Import: Create driving scenarios with road data imported from Zenrin Japan Map API 3.0 (Itsumo NAVI API 3.0) service

In the **Driving Scenario Designer** app, you can now generate a road network with data obtained from the Zenrin Japan Map API 3.0 (Itsumo NAVI API 3.0) service. For more details, see Import Zenrin Japan Map API 3.0 (Itsumo NAVI API 3.0) into Driving Scenario.

Programmatically import these roads into a `drivingScenario` object by using the `'ZenrinJapanMap'` syntaxes in the `roadNetwork` function. Manage your credentials by using the `zenrinCredentials` function.

Creating driving scenarios from Zenrin Japan Map API 3.0 (Itsumo NAVI API 3.0) requires Automated Driving Toolbox Importer for Zenrin Japan Map API 3.0 (Itsumo NAVI API 3.0) Service.

To gain access to the Zenrin Japan Map API 3.0 (Itsumo NAVI API 3.0) service and get the required credentials (a client ID and secret key), you must enter into a separate agreement with ZENRIN DataCom CO., LTD.

INS Sensor Model: Generate synthetic readings from an inertial navigation and GPS sensor in driving scenarios

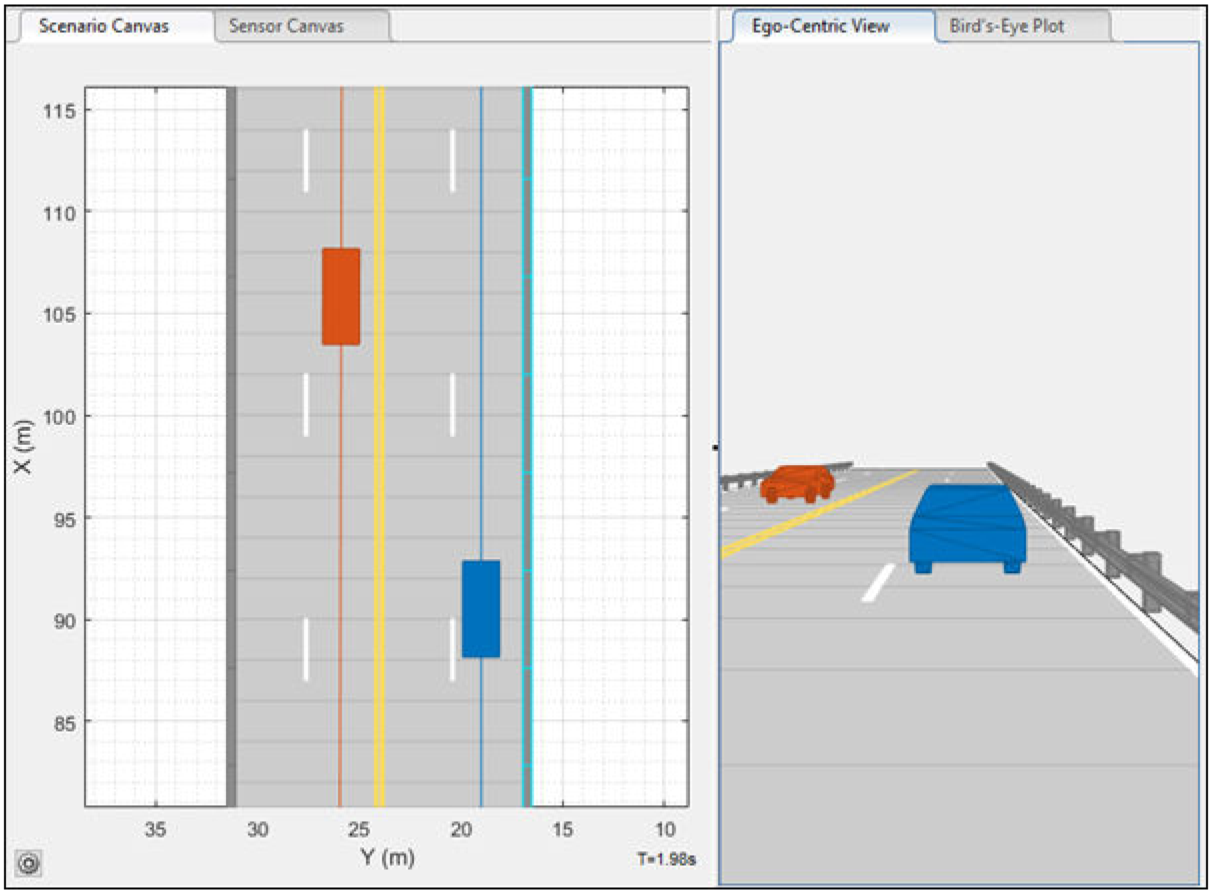
The `insSensor` System object™ models a device that fuses measurements from an inertial navigation system (INS) and global navigation satellite system (GNSS) such as a GPS, and outputs the fused measurements. To model an INS sensor in a programmatic driving scenario, follow these steps:

1. Create a driving scenario by using a `drivingScenario` object. Use the `'GeoReference'` name-value argument to specify the geographic origin of the route that correlates to the inertial navigation and GPS sensor.
2. Add an ego vehicle and generate its trajectory by using the `smoothTrajectory` function. Unlike the `trajectory` function, the `smoothTrajectory` function generates trajectories that avoid discontinuities in acceleration and are more suitable for generating realistic INS readings.
3. Obtain the state of the ego vehicle by using the `state` function. State information includes the position, velocity, orientation, and acceleration of the vehicle.
4. Create an `insSensor` object that is mounted to the ego vehicle, and simulate the driving scenario. Specify the actor state as ground truth data for the sensor. The sensor uses this data to generate sensor readings at each simulation time step.

To model an INS sensor in the **Driving Scenario Designer** app, see Generate INS Sensor Measurements from Interactive Driving Scenario.

Barriers: Add guardrails and Jersey barriers to driving scenarios

You can now model guardrails and Jersey barriers in cuboid driving scenarios. You can add barriers along an entire road edge or at a specific location within the scenario. This figure shows guardrail barriers added to a scenario in the **Driving Scenario Designer** app.



To add barriers to programmatic driving scenarios, use the `barrier` function. Display barrier outlines on bird's-eye plot using the `plotBarrierOutline` function.

In the **Driving Scenario Designer** app, two new actor classes, `Jersey Barrier` and `Guardrail`, replace the existing `Barrier` class. The class editor features a new property called `Actor Type`. This property allows users to set the actor type of a class to `Vehicle`, `Other`, or `Barrier`. The property serves as a replacement for the **Vehicle** check box.

Version History

When you import existing scenario files with `Barrier` objects into **Driving Scenario Designer**, all barriers are instantiated as `Jersey Barrier` objects by default. If you defined custom actor classes, ensure that their class IDs are not 5 or 6, because actors with those class IDs are instantiated as `Jersey Barrier` and `Guardrail` objects, respectively.

Radar Data Generator: Generate synthetic sensor detections and tracks from a driving scenario

The `drivingRadarDataGenerator` System object is a statistical radar sensor model that generates synthetic data from a driving scenario. This object provides the option to generate tracks, detections, and clustered detections. To model this sensor in Simulink, use the Driving Radar Data Generator block.

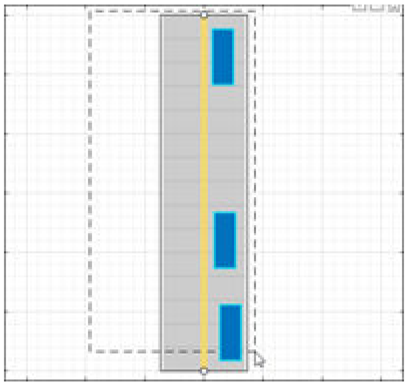
Version History

This System object and block replace the `radarDetectionGenerator` System object and Radar Detection Generator block, unless you require C/C++ code generation. For more details, see radarDetectionGenerator System object and Radar Detection Generator block are not recommended.

Driving Scenario Enhancements: Select multiple actors, align and distribute actors, and additional features

Select Multiple Actors

In the **Driving Scenario Designer** app, hold **Ctrl** and click each actor you want to select. Alternatively, hold **Shift** and click and drag to draw a box around the actors you want to select.

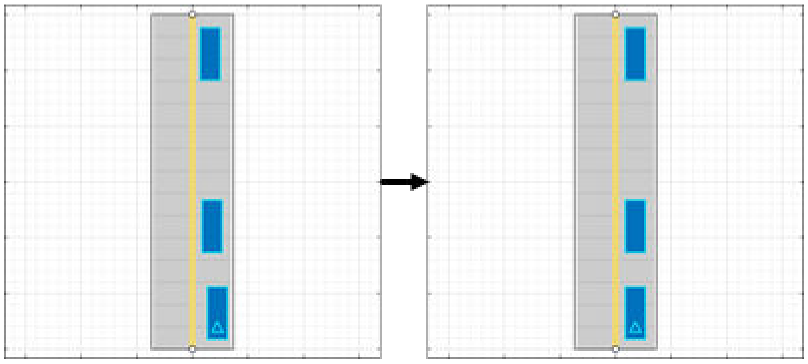


You can then uniformly move, align, or distribute the selected actors.

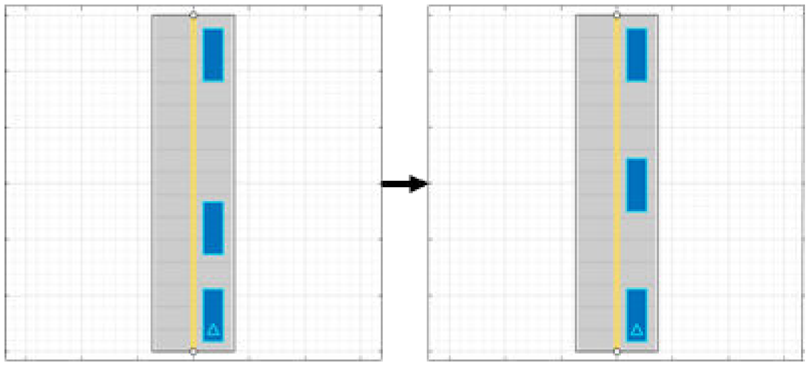
Align and Distribute Actors

In the **Driving Scenario Designer** app, to align selected actors along a specific actor dimension or to distribute actors along a road, right-click one of the actors and select one of the options in the **Align Actors** or **Distribute Actors** menus.

This figure shows actors aligned along their left side.



This figure shows actors distributed vertically along a road.



Specify Maximum Number of Actors and Lane Boundaries in Scenario Reader Block

In the Scenario Reader block, you can now set the maximum number of actors and lane boundaries that you can have in a scenario by using the **Maximum number of actors** and **Maximum number of lane boundaries** parameters, respectively.

Set these maximum values when you want to reuse the same actor or lane boundary buses across scenarios that have varying numbers of actors or lane boundaries. This situation is common when outputting actors or lane boundaries from a referenced model.

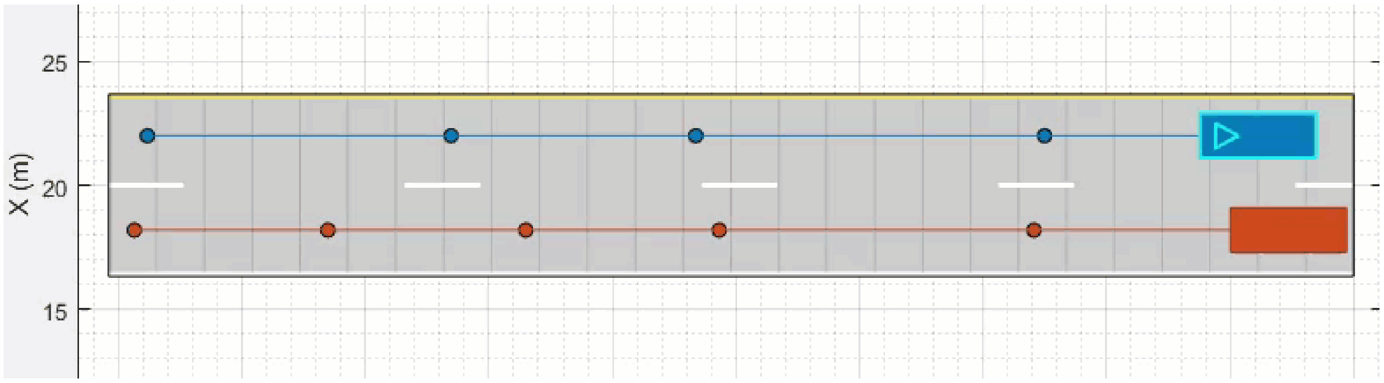
Read Actor Profiles from Scenario Reader Block

In sensor blocks such as the Vision Detection Generator block, you can now read actor profile information directly from the Scenario Reader block that is in your model. Actor profiles are the physical and radar characteristics of the actors in the driving scenario. The sensor blocks use these profiles to generate detections or other scenario data.

Previously, you had to either specify the actor profiles within each sensor block or specify a MATLAB expression that obtained these profiles from the base workspace. Newly created sensor blocks now read the actor profiles from the Scenario Reader block by default.

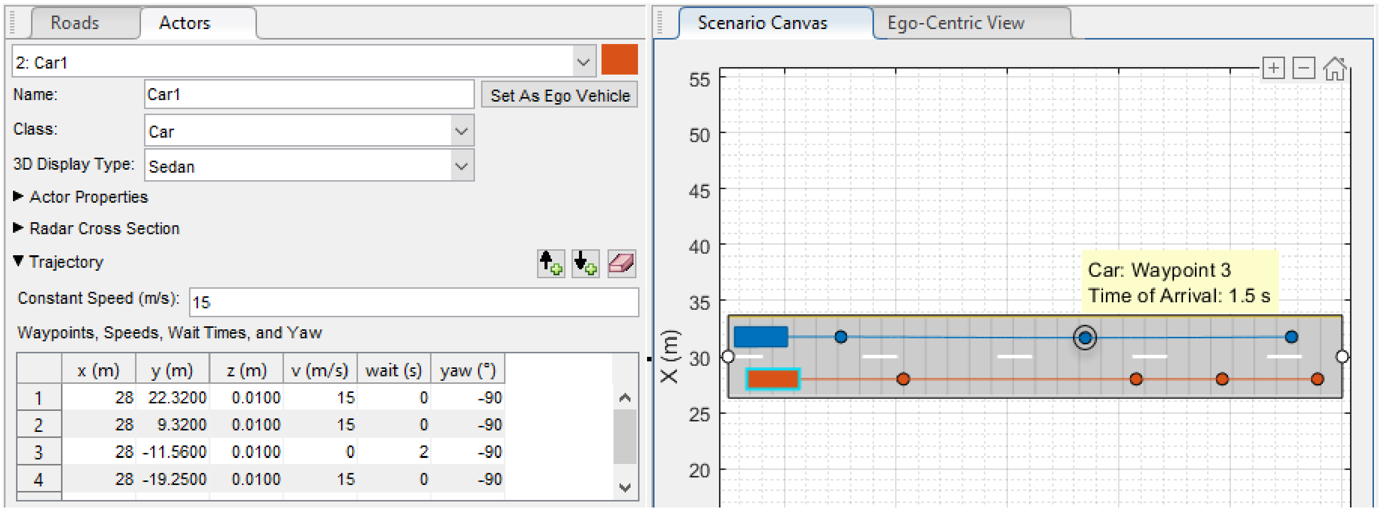
Spawn and Despawn Actors Multiple Times

You can now add or remove actors multiple times during a driving scenario. Specify multiple entry times and exit times for an actor in the **Driving Scenario Designer** app, or by using the `actor` function with a `drivingScenario` object.



Preview Actor Times of Arrival at Waypoints

The arrival time indicator in the **Driving Scenario Designer** app indicates the arrival time of an actor at each waypoint prior to running the simulation. The app enables the arrival time indicator for all actors when either the stop-and-go or the dynamic actor spawn and despawn feature is enabled for at least one actor in the scenario. In the scenario canvas, point to any waypoint along the trajectory of an actor to see the time of arrival of the actor at that waypoint.



2: Car1

Name: Car1

Class: Car

3D Display Type: Sedan

Constant Speed (m/s): 15

Waypoints, Speeds, Wait Times, and Yaw

	x (m)	y (m)	z (m)	v (m/s)	wait (s)	yaw (°)
1	28	22.3200	0.0100	15	0	-90
2	28	9.3200	0.0100	15	0	-90
3	28	-11.5600	0.0100	0	2	-90
4	28	-19.2500	0.0100	15	0	-90

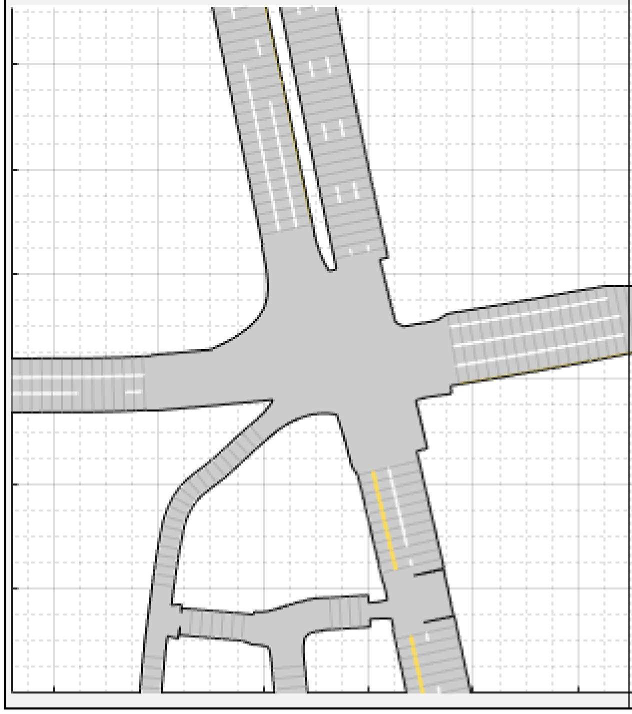
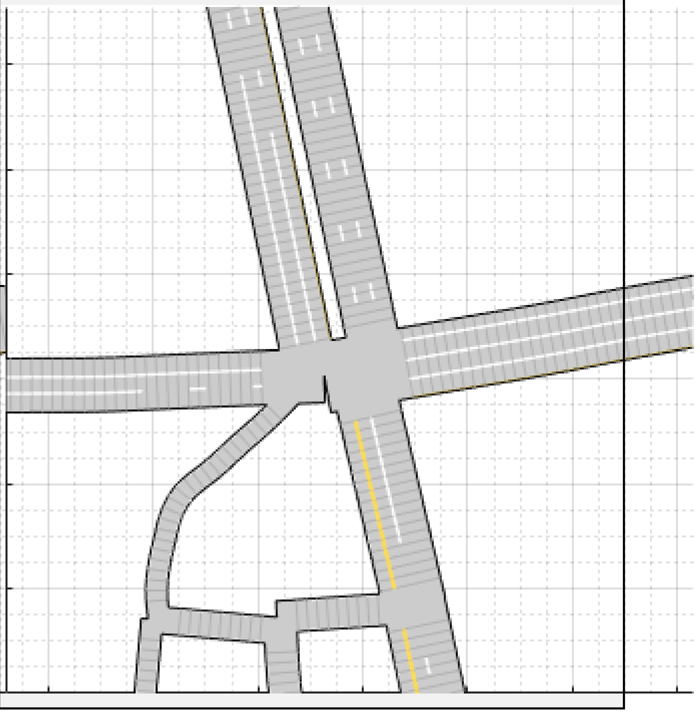
Car: Waypoint 3
Time of Arrival: 1.5 s

HERE HD Live Map Scenario Enhancements: Generate road networks with junctions and specifications for multiple lanes along a single road

In the **Driving Scenario Designer** app, road networks generated with data from the HERE HD Live Map [1] web service now contain road junctions and specifications for multiple lanes along a single road.

This table shows the enhanced road networks available in R2021a compared to the road networks available in R2020b.

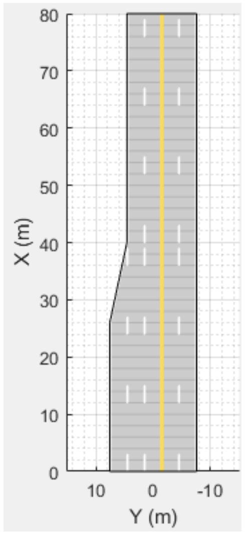
1 You need to enter into a separate agreement with HERE in order to gain access to the HDLM services and to get the required credentials (access_key_id and access_key_secret) for using the HERE Service.

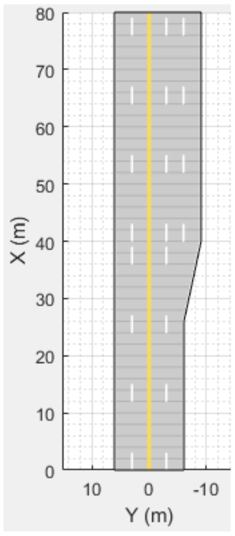
R2021a	R2020b
	

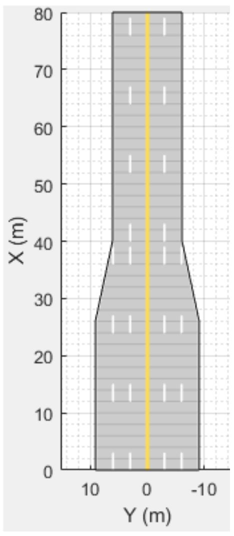
You can also import these road networks into a `drivingScenario` object by using the `'HEREHDLiveMap'` syntaxes of the `roadNetwork` function.

Multiple Lane Specifications: Add or drop lanes along a road

Add or drop lanes along a road by defining multiple lane specifications. To define multiple lane specifications programmatically use `compositeLaneSpec` object with the `road` function.

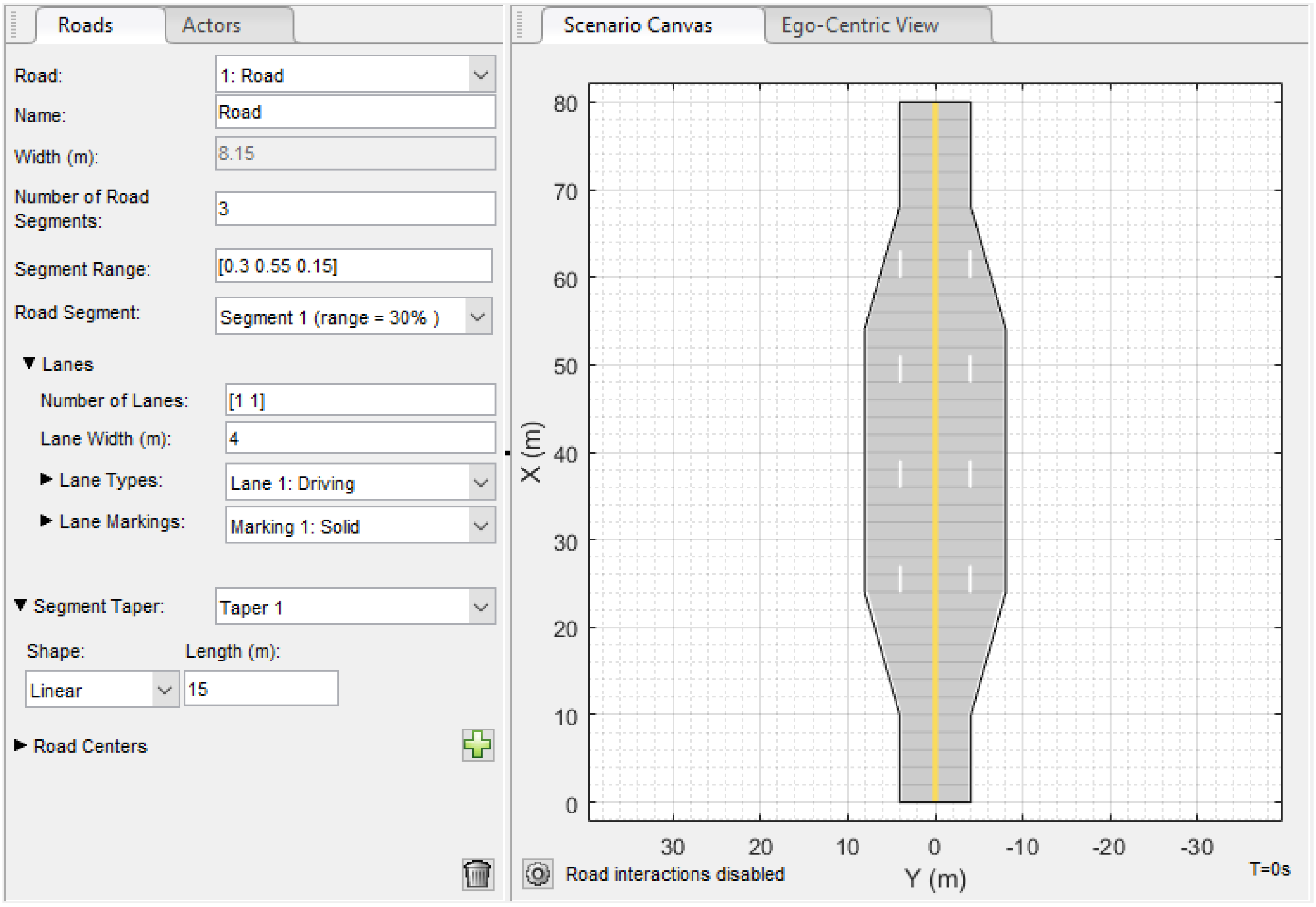






You can also define multiple lane specifications in the **Driving Scenario Designer** app using these new parameters on the **Roads** tab:

- **Number of Road Segments**
- **Segment Range**
- **Road Segment**
- **Segment Taper**



Road: 1: Road
Name: Road
Width (m): 8.15
Number of Road Segments: 3
Segment Range: [0.3 0.55 0.15]
Road Segment: Segment 1 (range = 30%)

▼ Lanes
Number of Lanes: [1 1]
Lane Width (m): 4
▶ Lane Types: Lane 1: Driving
▶ Lane Markings: Marking 1: Solid

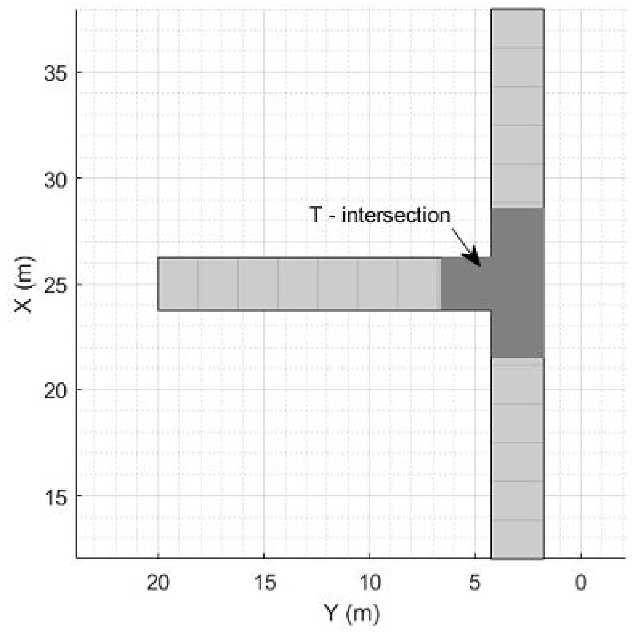
▼ Segment Taper: Taper 1
Shape: Linear
Length (m): 15

▶ Road Centers

Road Groups: Define road intersections

Use the `roadGroup` function to define intersections that connect two or more roads in a `drivingScenario` object.

You can also import a road network containing intersections to the **Driving Scenario Designer** app.



OpenDRIVE Export Enhancements: Export actors to OpenDRIVE format

You can now export the actors in a driving scenario and their properties to the OpenDRIVE file format either by using the `export` function of the `drivingScenario` object or in the **Driving Scenario Designer** app.

Functionality being removed or changed

radarDetectionGenerator System object and Radar Detection Generator block are not recommended

Still runs

The `radarDetectionGenerator` System object and Radar Detection Generator block are not recommended unless you require C/C++ code generation. Instead, use the `drivingRadarDataGenerator` System object and Driving Radar Data Generator, respectively. These new radar sensors provide additional properties for modeling radar sensors, including the ability to generate tracks and clustered detections.

There are no current plans to remove the `radarDetectionGenerator` System object or Radar Detection Generator block. MATLAB code and Simulink models that use these features will continue to run. You can still import `radarDetectionGenerator` objects into the **Driving Scenario Designer** app. However, the app updates the parameters of the imported sensor to reflect the parameters of a `drivingRadarDataGenerator` object. In addition, when you export a scenario containing a `radarDetectionGenerator` sensor to MATLAB code or to a Simulink model, the app exports the sensor as a `drivingRadarDataGenerator` object or Driving Radar Data Generator block, respectively.

Update Code

In MATLAB code, replace all instances of `radarDetectionGenerator` with `drivingRadarDataGenerator`. In addition, update all `radarDetectionGenerator` properties with their equivalent `drivingRadarDataGenerator` properties, as shown in the table. The properties not listed in the table are either specific only to `drivingRadarDataGenerator` or identical in both objects.

`radarDetectionGenerator` **Properties**	**Equivalent** `drivingRadarDataGenerator` **Properties**
`UpdateInterval`	`UpdateRate`

`SensorLocation`
 	
`Height`
	`MountingLocation`

`Yaw`
 	
`Pitch`
 	
`Roll`
	`MountingAccuracy`

`MaxRange`	`RangeLimits`
`MaxNumDetectionsSource`	`MaxNumReportsSource`
`MaxNumDetections`	`MaxNumReports`
`ActorProfiles`	`Profiles`

This table shows sample code for creating a `drivingRadarDataGenerator` object instead of a `radarDetectionGenerator` object.

Discouraged Usage	**Recommended Replacement**

```
radar = radarDetectionGenerator( ...
    'SensorLocation',[-1 0], ...
    'Height',0.2, ...
    'Yaw',180, ...
    'Pitch',0, ...
    'Roll',0, ...
    'MaxRange',50);
```
	```
	radar = drivingRadarDataGenerator( ...
	    'MountingLocation',[-1 0 0.2], ...
	    'MountingAngles',[180 0 0], ...
	    'RangeLimits',[0 50]);
	```

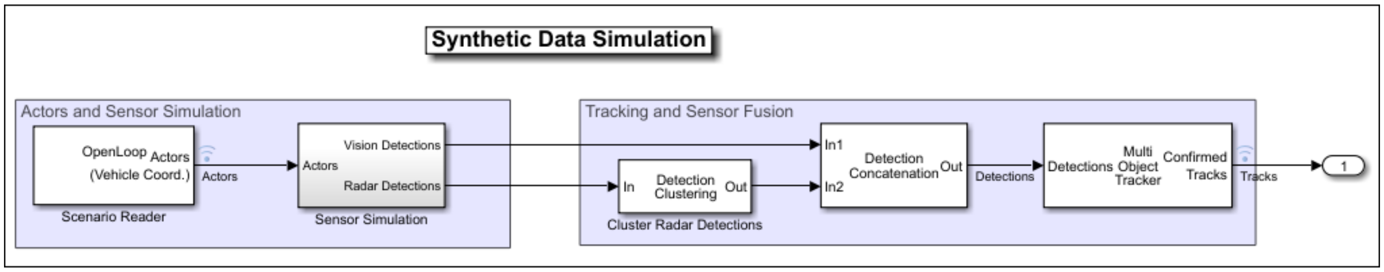
To generate detections from actor poses at each simulation time step, replace the `dets = radarDetectionGenerator(targets,time)` syntax with `dets = drivingRadarDataGenerator(targets,time)`.

Update Models

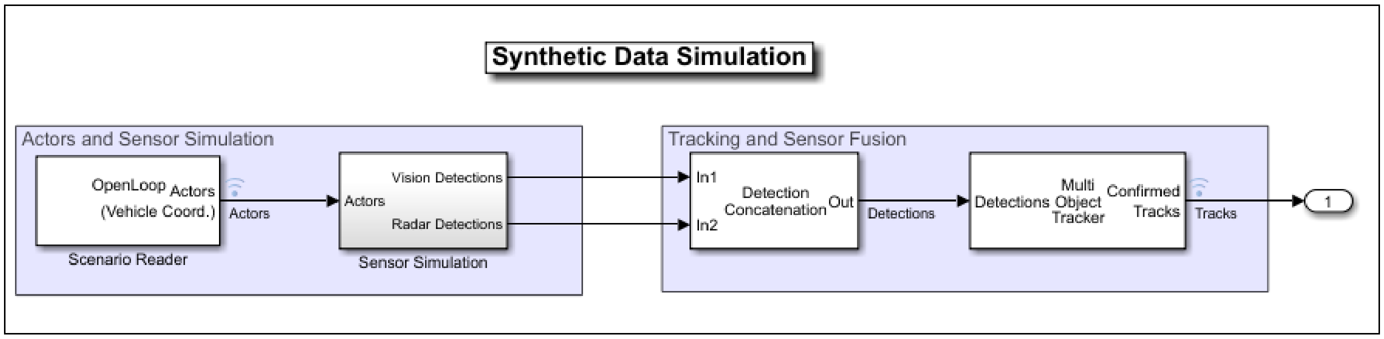
In Simulink models, replace all Radar Detection Generator blocks with Driving Radar Data Generator blocks. In the Driving Radar Data Generator blocks, update the parameter values in the same way you would update the `drivingRadarDataGenerator` property values described in the Update Code section.

If your model contains a separate block that clusters detections, you can remove it because the Driving Radar Data Generator block clusters detections by default.

For example, in this model, the Sensor Simulation subsystem outputs concatenated detections from Radar Detection Generator blocks into a separate block that clusters the detections.



In this model, the Sensor Simulation subsystem outputs concatenated, clustered detections from Driving Radar Data Generator blocks directly into the next part of the model pipeline.



Unreal Engine Scenario Simulation

Unreal Engine Vehicle Enhancements: Import custom meshes and control vehicle lights

You can configure the Simulation 3D Vehicle with Ground Following block to import custom meshes and control vehicle lights.

To	Action
Import custom meshes	**1** Install the Automated Driving Toolbox Interface for Unreal Engine 4 Projects support package. See Install Support Package for Customizing Scenes. **2** On the block **Parameters** tab, set **Type** to `Custom`. **3** In the **Path to custom mesh** field, enter the path to the vehicle mesh in the Unreal Engine project. For example, enter `/MathWorksSimulation/Vehicles/Muscle/Meshes/SK_MuscleCar.SK_MuscleCar`. To create a custom vehicle mesh, see Prepare Custom Vehicle Mesh for the Unreal Editor. **4** Use the vehicle dimensions in the custom mesh to enter the dimensions in the corresponding block parameter fields.
Control vehicle lights	**1** Install the Automated Driving Toolbox Interface for Unreal Engine 4 Projects support package. See Install Support Package for Customizing Scenes. **2** On the block **Light Controls** tab, select **Enable light controls**. **3** Use the enabled parameters to specify the vehicle lights for: • Headlights • Brake lights • Reverse lights • Turn signal lights **4** Connect Boolean light control signals to the `Signal lights` input port.

Unreal Engine Scene Environment: Control weather and sun position

Use the Simulation 3D Scene Configuration block to control scene weather and sun position. Options allow you to create realistic environments when you run maneuvers and test control algorithms in the Unreal Engine 3D simulation environment. The Simulation 3D Camera and Simulation 3D Fisheye Camera blocks receive the image from the 3D simulation environment.

To control scene weather and sun position, on the Simulation 3D Scene Configuration block **Weather** tab, select **Override scene weather**. Use the enabled parameters to change the sun position, clouds, fog, and rain during the simulation.

Detection and Tracking

Out-of-Sequence Measurements Handling: Ignore out-of-sequence measurements of object tracks, or terminate tracking when one is encountered

In the `multiObjectTracker` System object, the `OOSMHandling` property controls whether the tracker neglects out-of-sequence measurements and continues running or stops running as soon as it encounters an out-of-sequence measurement. To handle out-of-sequence measurements in the Multi-Object Tracker block, use the equivalent **Out-of-sequence measurements handling** parameter.

Bird's-Eye View Example: Create a 360° bird's-eye-view image around a vehicle

The Create 360° Bird's-Eye-View Image Around a Vehicle example shows how to create a 360° bird's-eye-view image around a vehicle for use in a surround view monitoring system.

Radar and Tracking Examples: Process radar multipath detections, simulate radar ghosts from multipath detections, and fuse lidar and radar tracks in Simulink

The Simulate Radar Ghosts due to Multipath Return example shows how to generate ghost targets that occur when signal energy is reflected off another target before returning to the radar. This example requires the Radar Toolbox software.

The Highway Vehicle Tracking with Multipath Radar Reflections example shows how to assess and mitigate the impact of multipath radar reflections when you track highway vehicles using an extended object tracker. This example requires the Sensor Fusion and Tracking Toolbox and Radar Toolbox software.

The Track-Level Fusion of Radar and Lidar Data in Simulink example shows how to fuse tracks obtained by radar and lidar sensor measurements in Simulink. This example requires the Sensor Fusion and Tracking Toolbox and Lidar Toolbox software. It closely follows the Track-Level Fusion of Radar and Lidar Data example.

Localization and Mapping

Localization and Mapping Examples: Build an occupancy map from lidar data using SLAM, develop a stereo visual SLAM algorithm, and perform localization using HD map traffic data

The Build Occupancy Map from 3-D Lidar Data Using SLAM example demonstrates how to build a 2-D occupancy map from 3-D lidar data using a simultaneous localization and mapping (SLAM) algorithm.

The Develop Visual SLAM Algorithm Using Unreal Engine Simulation example now shows how to develop a stereo visual SLAM algorithm. This algorithm measures depth information more accurately than the monocular visual SLAM algorithm.

The Localization Correction Using Traffic Sign Data from HERE HD Maps example shows how to use traffic sign data from the HERE HD Live Map service to correct GPS measurements collected by an autonomous vehicle.

Functionality being removed or changed

hereHDLMConfiguration(region) syntax has been removed
Errors

In `hereHDLMConfiguration` objects, the syntax for configuring a `hereHDLMReader` object to search catalogs from a specific region, `hereHDLMConfiguration(region)`, has been removed. Instead, specify the catalog name that corresponds to that region by using the `hereHDLMConfiguration(catalog)` syntax.

Previously, the catalog names for regions such as North America were not available to customers. HERE Technologies now makes these catalog names available through the HERE HD Live Map Marketplace, making the region syntax unnecessary.

Update Code

This table shows a typical usage of the `hereHDLMConfiguration(region)` syntax, and shows how to update that code using the `hereHDLMConfiguration(catalog)` syntax.

Discouraged Usage	Recommended Replacement

`catalog = hereHDLMConfiguration('North America')`	`catalog = hereHDLMConfiguration( ...`
	`'hrn:here:data::olp-here-had:here-hdlm-protobuf-na-2')`

Planning and Control

Motion Planning Example: Plan a path through an urban environment using a dynamic occupancy grid map

The Motion Planning in Urban Environments Using Dynamic Occupancy Grid Map example shows how to perform dynamic replanning in an urban driving scene by using a grid-based tracker. The estimated occupancy map is used for replanning of a Frenet reference path. This example requires the Sensor Fusion and Tracking Toolbox and Navigation Toolbox software.

Applications

Automated Driving Reference Applications: Examples on vehicle sensor fusion, and code generation of vehicle detector, lane following controller, and lane change planner

The Forward Vehicle Sensor Fusion example shows how to implement sensor fusion and tracking from a camera and a radar sensor. You can test the sensor fusion and tracking algorithm using different prebuilt scenarios in a 3D simulation environment that uses the Unreal Engine from Epic Games®.

The Surround Vehicle Sensor Fusion example shows how to implement sensor fusion and tracking from multiple vision and radar sensors that provide 360-degree coverage surrounding an ego vehicle for highway lane change maneuvers.

The Generate Code for Vision Vehicle Detector example shows how to test and generate deployable code for a vehicle detector. The example demonstrates two variants of vehicle detector implemented using an aggregate channel features (ACF) object detector and a pretrained you-only-look-once (YOLO) v2 network. You can generate C++ code for the ACF object detector and CUDA code for the YOLOv2 network.

The Generate Code for Highway Lane Following Controller example shows how to test a highway lane following controller component using ground truth information. This example generates C++ code for the controller and validates the functional equivalence using software-in-the-loop (SIL) simulation.

The Generate Code for Highway Lane Change Planner example shows how to design and test a lane change planner component for a highway lane change application. The example also shows how to generate C++ code, and assess functionality using software-in-the-loop (SIL) simulation.

The Automate Testing for Lane Marker Detector example shows how to automate the testing of a lane marker detector component and verify the generated code using Simulink Test software.

The Automate Testing for Highway Lane Following Controls and Sensor Fusion example integrates sensor fusion and control components of a highway lane following system. The example shows how to automate the testing of this component assembly and verify the generated code using Simulink Test software.

R2020b

Ground Truth Labeling

Labeler Enhancements: Label objects in images and video using projected 3-D bounding boxes, load custom image formats, use additional keyboard shortcuts, and more

This table describes enhancements for these labeling apps:

- **Image Labeler**
- **Video Labeler**
- **Ground Truth Labeler**
- **Lidar Labeler** — Introduced in R2020b

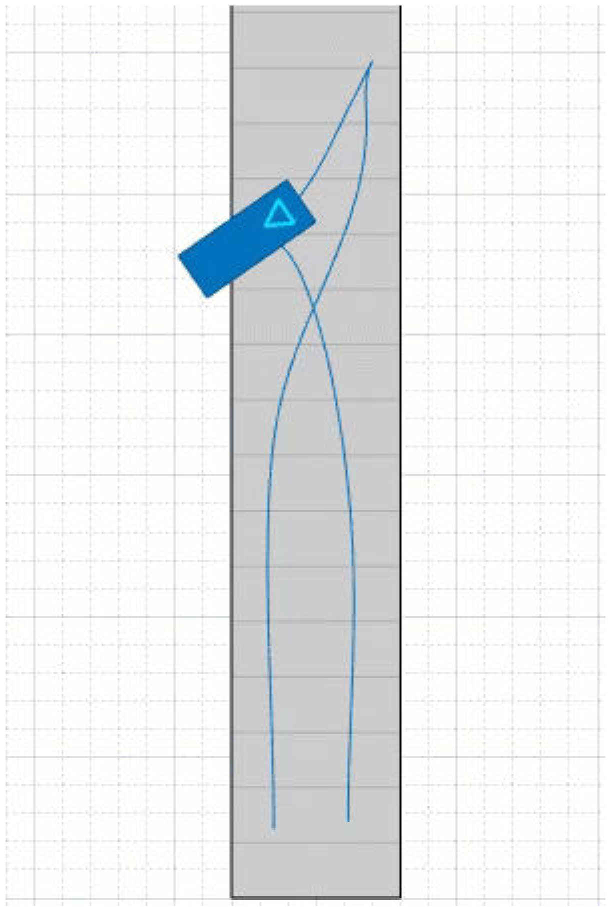
Enhancement	Image Labeler	Video Labeler	Ground Truth Labeler	Lidar Labeler
Load images with custom image formats using an `imageDatastore` object	Supported	Not supported	Not supported	Not supported
Draw projected 3-D bounding boxes around objects in images and video using the projected cuboid label type	Supported	Supported	Supported	Not supported
Delete pixel labels	Supported	Supported	Supported	Not supported
Undo and redo drawing a pixel label an increased number of times	Supported	Supported	Supported	Not supported
Use keyboard shortcuts for selecting drawn labels and resizing bounding boxes	Supported	Supported	Supported	Not supported
Specify attributes for cuboid ROI labels	Not supported	Not supported	Supported	Supported

Enhancement	Image Labeler	Video Labeler	Ground Truth Labeler	Lidar Labeler
Visualize point cloud clusters across all frames, not just individual frames, when **Snap to Cluster** option is selected, by using a new **Cluster Settings** option	Not supported	Not supported	Supported	Supported
Use keyboard shortcuts for panning across the point cloud frame and moving multiple selected cuboids	Not supported	Not supported	Supported	Supported

Cuboid Scenario Simulation

Reverse Motion in Driving Scenarios: Simulate driving maneuvers such as backing into parking spots

In the **Driving Scenario Designer** app, you can now specify reverse motions for actors in a driving scenario. Previously, the app supported only forward motions. Use reverse motion to simulate advanced driving maneuvers such as backing into a parking spot or completing a three-point turn.

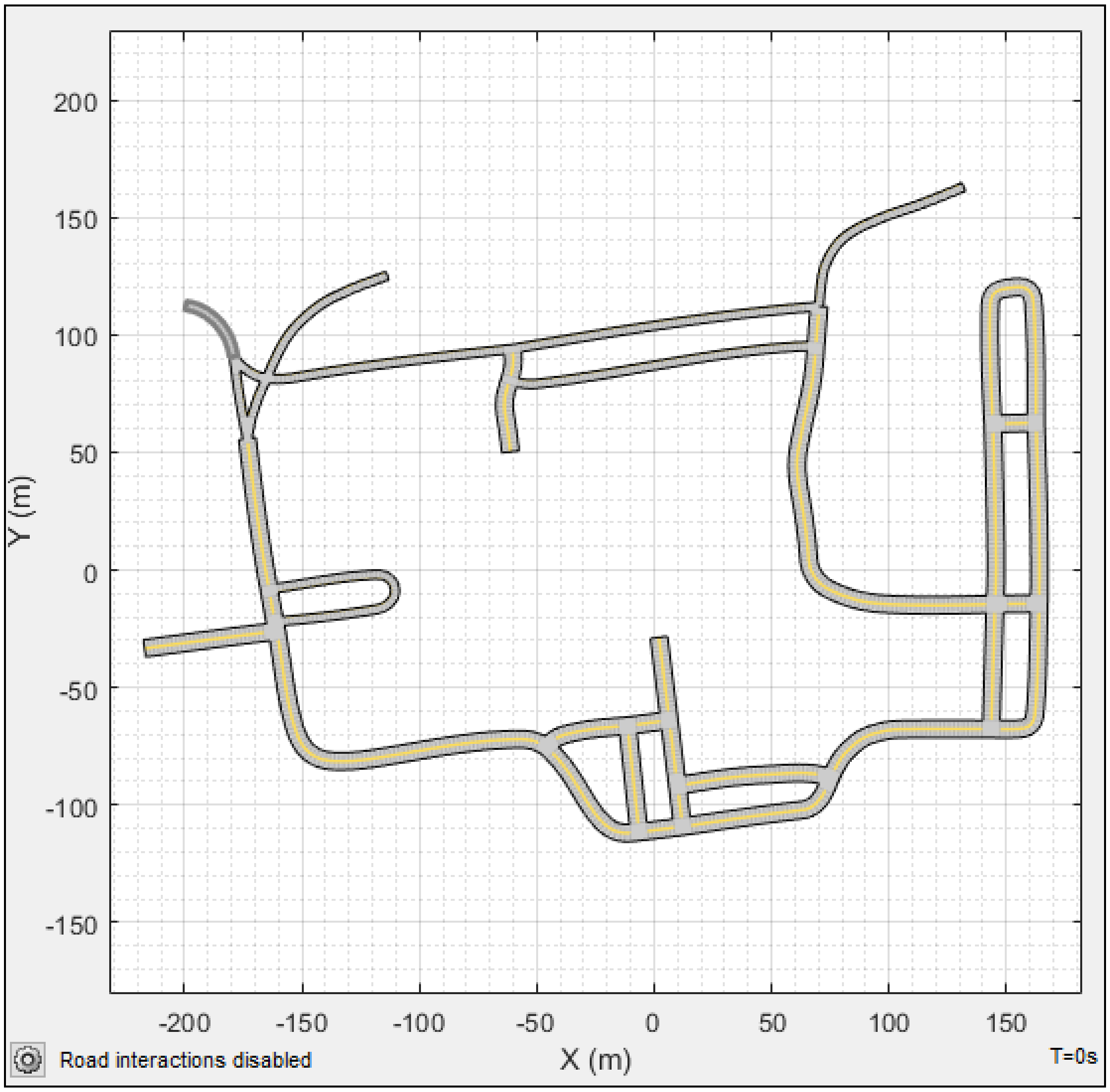


To test reverse motion algorithms, you can use the `Reverse_AEB` scenarios described in Euro NCAP Driving Scenarios in Driving Scenario Designer. To learn how to create your own reverse motion scenarios, see the Create Reverse Motion Driving Scenarios Interactively example.

To simulate reverse motions in programmatic driving scenarios, specify negative speeds for actors in the `trajectory` function.

OpenStreetMap Roads: Create driving scenarios using road data imported from the OpenStreetMap web service

In the **Driving Scenario Designer** app, you can now generate a road network with data obtained from the OpenStreetMap web service.



For more details, see Import OpenStreetMap Data into Driving Scenario.

You can also import these roads into a `drivingScenario` object by using the `'OpenStreetMap'` syntax of the `roadNetwork` function.

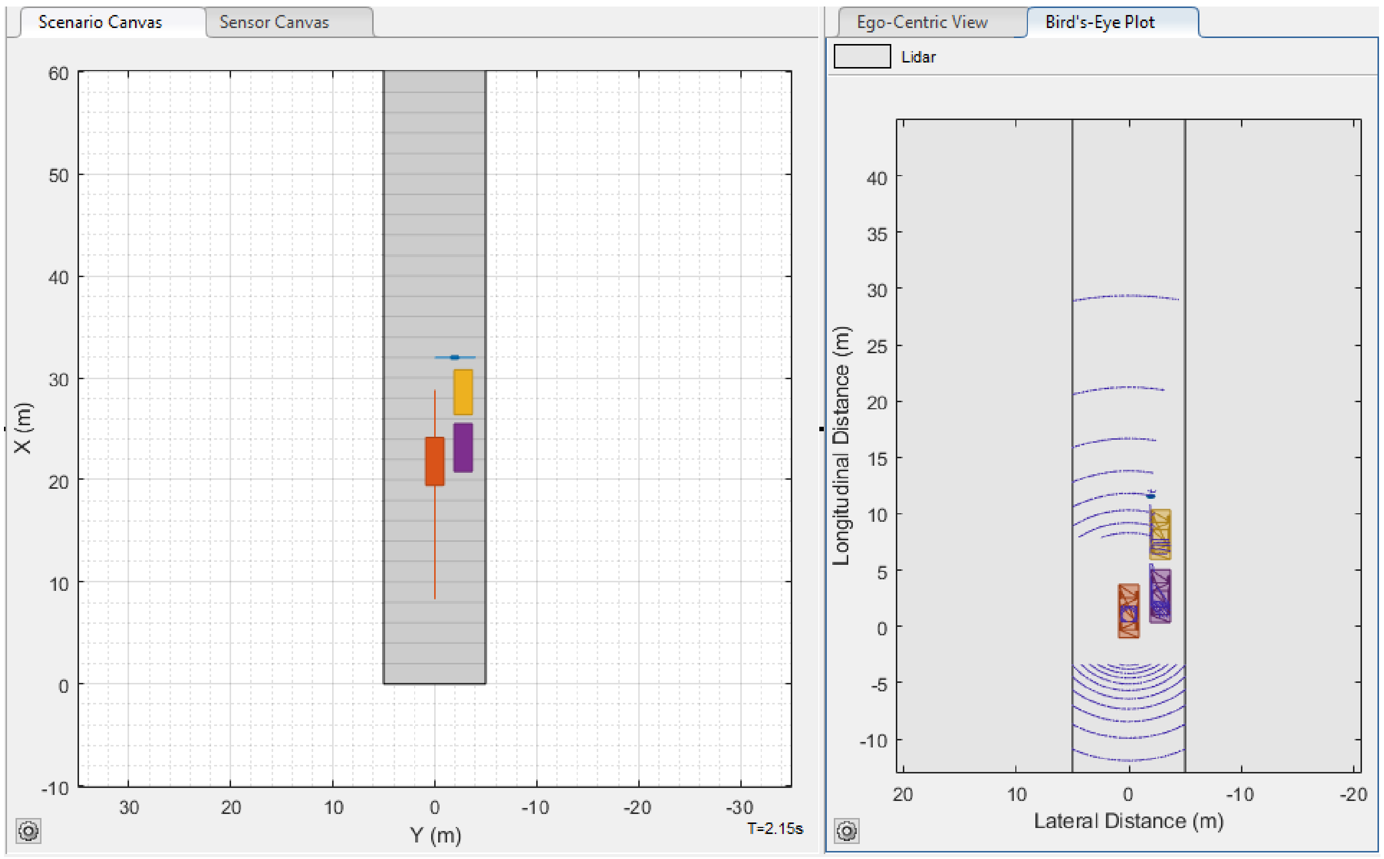
OpenDRIVE Export: Share a driving scenario using the OpenDRIVE format

Use the `export` function with a `drivingScenario` object to programmatically export a driving scenario to OpenDRIVE format.

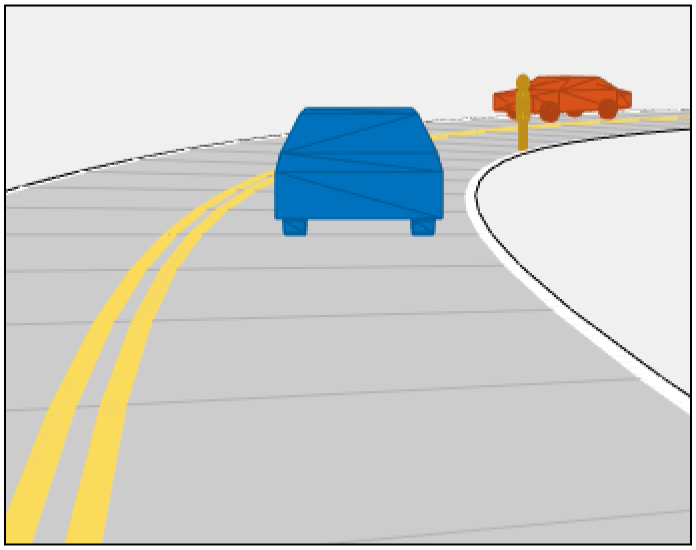
In the **Driving Scenario Designer** app, select the **OpenDRIVE File** menu item in the **Export** menu to export the driving scenario to OpenDRIVE format.

Lidar Sensor Model Extensions: Generate synthetic point clouds from scenarios in Driving Scenario Designer app and in Simulink

In the **Driving Scenario Designer** app, you can now model a lidar sensor and generate synthetic point cloud data from a driving scenario.



This sensor obtains data from mesh representations of the roads and actors within the scenario.



When you export a scenario containing a lidar sensor to MATLAB, the sensor is represented as a `lidarPointCloudGenerator` System object (introduced in R2020a).

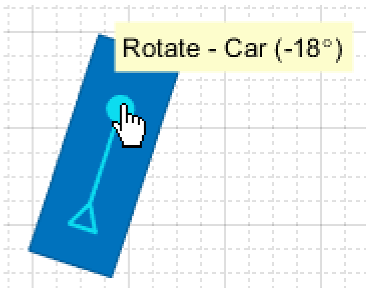
When you export a scenario containing a lidar sensor to Simulink, the sensor in represented as a Lidar Point Cloud Generator block (introduced in R2020b).

Driving Scenario Enhancements: Rotate actors interactively, specify yaw angles with trajectories, and additional features

When creating cuboid driving scenarios using the `drivingScenario` object or the **Driving Scenario Designer** app, you can now use these features.

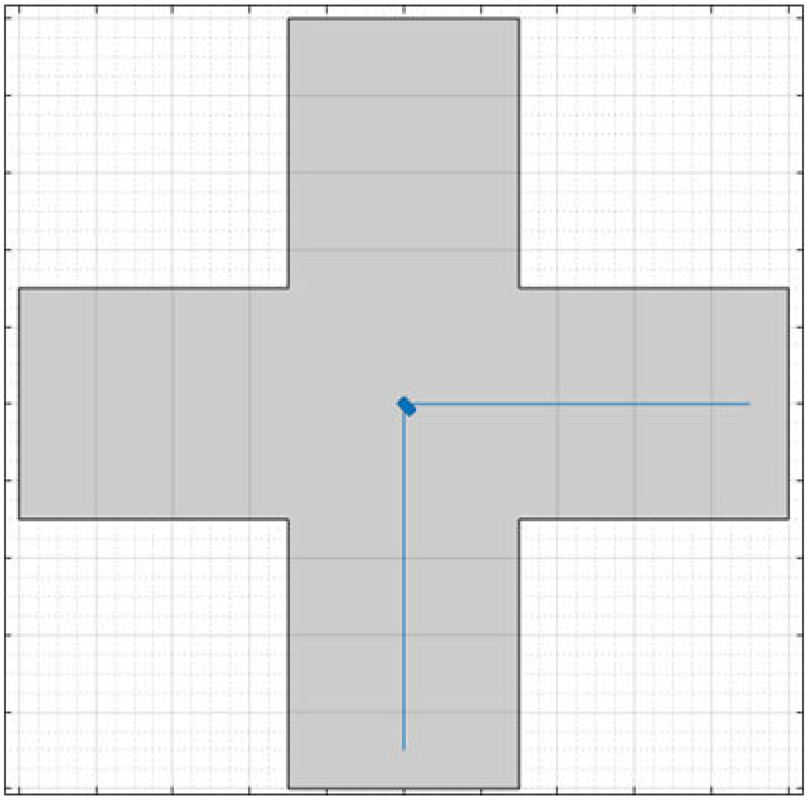
Interactive Actor Rotation

In the **Driving Scenario Designer** app, you can now rotate actors interactively. Previously, to rotate an actor, you needed to specify the **Yaw** value on the **Actors** tab for the selected actor. To rotate actors interactively, on the **Scenario Canvas**, pause your pointer on an actor and move the actor rotation widget in the desired direction.



Yaw Angles for Actor Trajectories

In the **Driving Scenario Designer** app and the `trajectory` function used with `drivingScenario` objects, you can now specify yaw angles for actor trajectories. Specifying yaw angles as a constraint on trajectories enables finer control over actor motions. For example, you can specify more precise motions for vehicles in parking scenarios or specify pedestrians to turn at 90-degree angles.



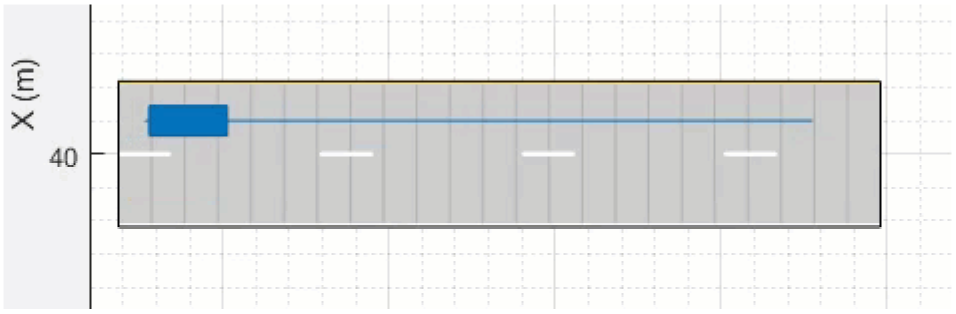
For sample scenarios with specified yaw constraints, see the `AEB_PedestrianTurning` scenarios described in Euro NCAP Driving Scenarios in Driving Scenario Designer.

Actor Spawn and Despawn

You can now add or remove actors dynamically from a driving scenario during simulation.

In the **Driving Scenario Designer** app and the `actor` function used with `drivingScenario` objects, you can specify these options:

- Entry time for actors to spawn (appear) in the scenario during simulation
- Exit time for actors to despawn (disappear) from the scenario during simulation



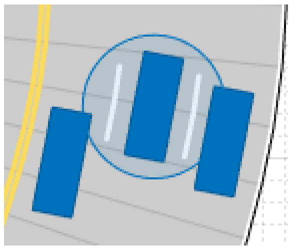
Mesh Plotter in Bird's-Eye Plot

In the `birdsEyePlot` object, you can now plot the meshes for actors in a driving scenario. To plot actor meshes:

1. Use the `targetMeshes` function to obtain the faces, vertices, and color of target actors that are relative to a specific actor.
2. Create a `meshPlotter` object to configure the display of the meshes.
3. Use this plotter with the `plotMesh` function to display the faces, vertices, and color of each actor mesh.

Ego Vehicle Indicator

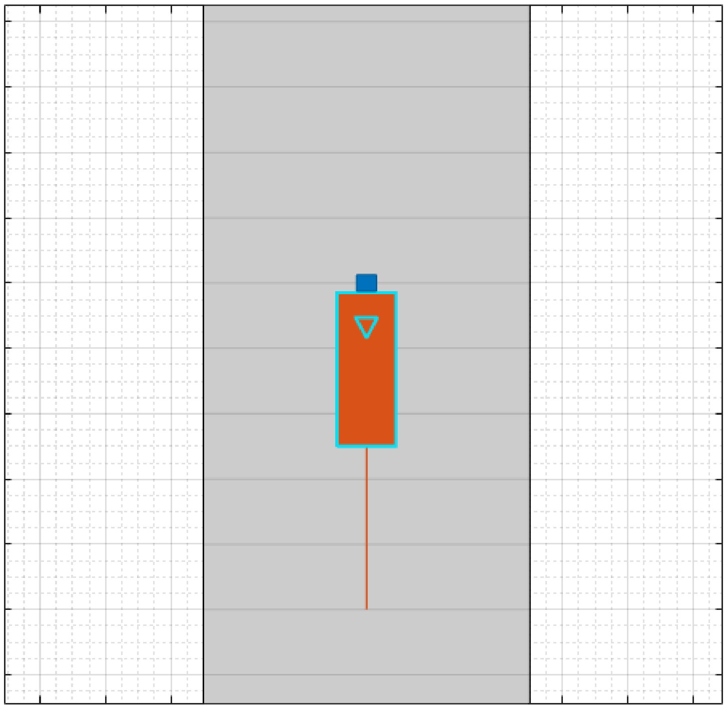
In the **Driving Scenario Designer** app, you can now add a visual indicator around the ego vehicle in a driving scenario. Use this option to identify the ego vehicle in simulations containing multiple actors.



You can also add this visual indicator to actors in driving scenarios created using a `drivingScenario` object. In the `plot` function used with this object, specify the `'ActorIndicators'` name-value pair with the `ActorID` values of the actors around which you want to draw the indicator.

Actor Pose Indicator

On the **Scenario Canvas** of the **Driving Scenario Designer** app, when you select an actor or pause your pointer on it, a triangle indicating the pose (position and orientation) of the actor is displayed at the actor origin.



You can optionally display this pose indicator during simulation, which is useful for visualizing a scenario with some vehicles moving forward and others moving in reverse.

Target Poses in Specified Range

The `targetPoses` function can optionally return poses that are within only a specified range of the ego vehicle. By generating poses that are only within the maximum detection range of the ego vehicle sensors, you can improve driving scenario performance. The generation of target poses in a specified range is not supported in the **Driving Scenario Designer** app.

Named Roads and Actors

In the `road`, `actor`, and `vehicle` functions, the `'Name'` name-value pair argument enables you to specify a name for created roads and actors. The `roadNetwork` function uses this name-value pair to import the names of OpenDRIVE, HERE HD Live Map, or OpenStreetMap roads.

Road Object

The `road` function can optionally return a `Road` object that contains the properties of the created road, such as its road centers and banking angle. These properties are read-only.

Scenario Generation Example: Automate scenario generation for driving applications

The Automatic Scenario Generation example shows how to automate scenario generation by using a set of start and goal positions specified for vehicles in a driving scenario. This example automatically generates random trajectories and adjusts the speed profile of each vehicle to synthesize a collision-free scenario. Use this example to create random driving scenarios for testing automated driving algorithms.

Driving Scenario Performance: Improved performance when simulating scenarios with large numbers of actors

The `drivingScenario` object and **Driving Scenario Designer** app have been redesigned for improved performance when simulating scenarios that contain a large number of actors. For example, this code generates a scenario with 100 vehicle actors by using the `vehicle` function.

```
scenario = drivingScenario;
numRoads = 50; % 2 vehicles per road

for i = 1:numRoads
    y = 10*i;
    roadCenters = [100 y 0; -100 y 0];
    road(scenario,roadCenters);

    v1 = vehicle(scenario);
    trajectory(v1,roadCenters,25);

    v2 = vehicle(scenario);
    trajectory(v2,flipud(roadCenters),25);
end
```

When simulating this scenario by using the `advance` function, the simulation is about 3x faster than in the previous release:

```
plot(scenario)
while advance(scenario)
end
```

For each call to the `advance` function, the approximate execution times are:

R2020a: 0.039s

R2020b: 0.014s

The simulation was timed on a *Windows 10, Intel(R) Xeon(R) CPU E5-1650 v4 @ 3.60 GHz* test system by using the `timeit` function:

```
timeit(@() advance(scenario))
```

Unreal Engine Scenario Simulation

Simulation 3D Vision Detection Generator Block: Generate synthetic object and lane boundary detections from the Unreal Engine simulation environment

The Simulation 3D Vision Detection Generator block models a synthetic vision sensor and generates object and lane boundary detections from a simulation environment. This environment is rendered using the Unreal Engine from Epic Games. The block includes parameters for modeling detection accuracy, measurement noise, and camera intrinsics.

Unreal Engine Camera Views: Visualize vehicle acceleration, pitch, and roll with improved camera controls and other usability improvements

The camera views in the Unreal Engine simulation environment include these usability improvements.

Smooth Transition Between Views

Press the keyboard keys **0**–**9** to transition smoothly between vehicle camera views.

Cycle Through Vehicles in Scene

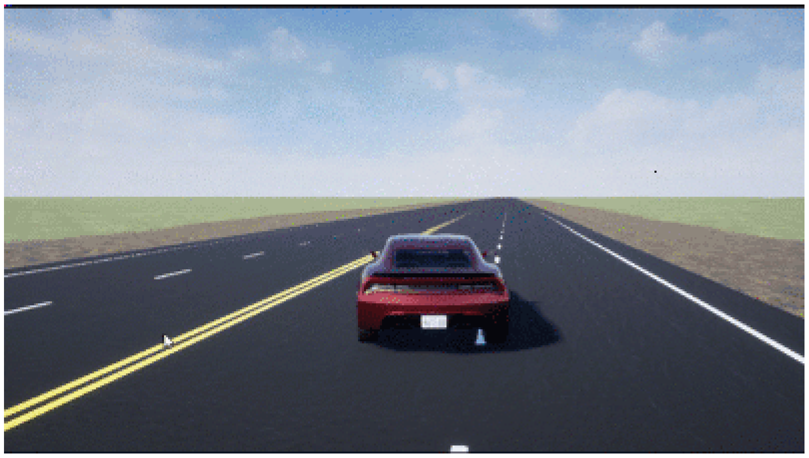
Press the **Tab** key to cycle the view between all vehicles in the scene.

Vehicle Acceleration and Rotation

Press the **L** key to toggle a camera lag effect on and off. When you enable the lag effect, the camera view includes:

- Position lag, based on the translational acceleration of the vehicle
- Rotation lag, based on the rotational velocity of the vehicle

This view provides for improved visualization of overall vehicle acceleration and rotation.

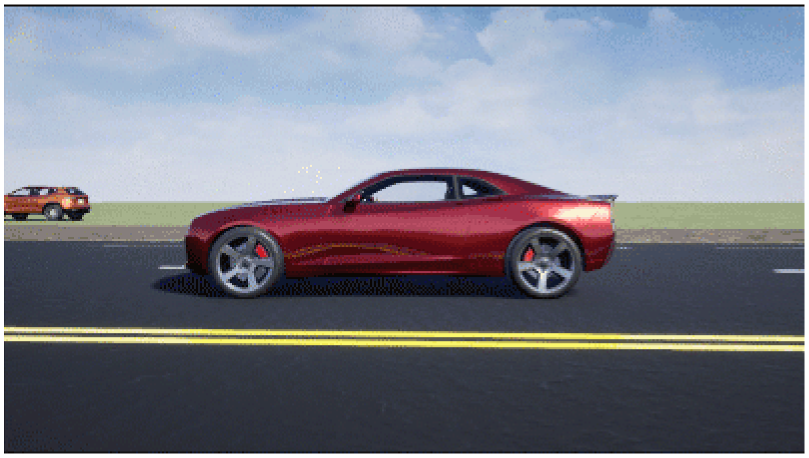


Vehicle Pitch and Roll

The views now lock the camera pitch and roll to the horizon, providing improved visualization of the vehicle pitch and roll.

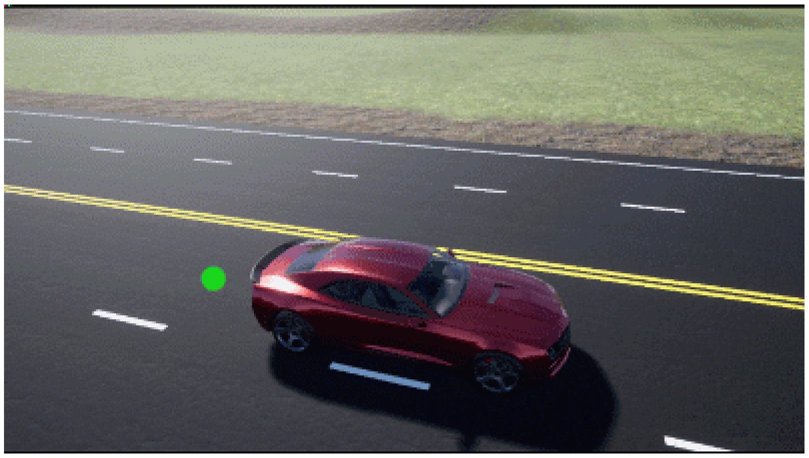
Camera Distance

Use the mouse scroll wheel to control the camera distance from the vehicle.



Free-Camera Views

Press the **F** key to toggle the free camera mode on and off. When you enable the free camera mode, you can use the mouse to change the pitch and yaw of the camera. This mode enables you to orbit the camera around the vehicle.



Detection and Tracking

Tracking Examples: Perform grid-based tracking, track multiple lane boundaries, and generate code for track-level fusion

The Grid-based Tracking in Urban Environments Using Multiple Lidars example shows how to track moving objects by using multiple lidar sensors and a grid-based tracker.

The Track Multiple Lane Boundaries with a Global Nearest Neighbor Tracker example shows how to design and test a multiple-lane tracking algorithm by using lane detections obtained by a probabilistic camera in a driving scenario.

The Generate Code for a Track Fuser with Heterogeneous Source Tracks example shows how to generate code for a track-level fusion algorithm where tracks originate from heterogeneous sources with different state definitions.

These examples require the Sensor Fusion and Tracking Toolbox software.

Functionality being removed or changed

vehicleDetectorFasterRCNN function now uses MobileNet-v2 network architecture and does not require type of vehicle detector model as input
Behavior change in future release

The `vehicleDetectorFasterRCNN` function now uses a modified version of the MobileNet-v2 convolutional neural network (CNN) as the base network for vehicle detector.

Previously, the `vehicleDetectorFasterRCNN` function enabled you to specify the type of vehicle detector model, `modelName`, as an input for vehicle detection. The valid `modelName` values were: `'full-view'` or `'front-rear-view'`, which specified models that were trained on different views of vehicle images.

The `vehicleDetectorFasterRCNN` function now uses a generic vehicle detector that works for test images containing any of these vehicle views: front, rear, left, or right.

Update Code

The table shows a typical usage of the `modelName` input argument of the `vehicleDetectorFasterRCNN` function. It also shows how to update your code by removing the input argument `modelName`.

Discouraged Usage	Recommended Replacement

`modelName = 'front-rear-view'`
`detector = vehicleDetectorFasterRCNN(modelName);`
	`detector = vehicleDetectorFasterRCNN;`

Localization and Mapping

Localization Examples: Develop lidar and visual SLAM algorithms for navigation using the Unreal Engine simulation environment

The Lidar Localization with Unreal Engine Simulation example shows how to develop and evaluate a lidar localization algorithm using synthetic lidar data.

The Develop Visual SLAM Algorithm Using Unreal Engine Simulation example shows how to develop a visual simultaneous localization and mapping (SLAM) algorithm using synthetic image data.

Both examples generate synthetic data from the Unreal Engine simulation environment.

HERE HD Live Map Marketplace Support: Read and visualize high-definition map data from the HERE HD Live Map Marketplace service

The HERE HD Live Map features—the `hereHDLMReader` object and map import in driving scenarios —now obtain map data from the Marketplace service provided by HERE Technologies. Previously, these features obtained map data from the DataStore service and required you to enter an App ID and App Code as credentials. To access HERE HD Live Map data from the Marketplace service, you must enter your Marketplace credentials, which consist of an Access Key ID and Access Key Secret.

Version History

HERE HD Live Map features no longer support DataStore credentials (App ID and App Code). In addition, the data obtained from the Marketplace catalogs might differ from the data in the DataStore catalogs. The `hereHDLMConfiguration` object has been updated to configure `hereHDLMReader` objects to read data from Marketplace catalogs only.

HERE HD Live Map Localization Layers: Read localization data such as barriers, signs, and poles from a road network

The `hereHDLMReader` object now supports reading localization map layers from the HD Localization Model of the HERE HD Live Map (HDLM) service. Use these layers to obtain information about objects along the road, such as roadside barriers, traffic signs, and poles alongside and over the road. Previously, the object supported reading data from only road and lane layers. Localization data for obstacles along the road is not supported.

Functionality being removed or changed

hereHDLMConfiguration(region) syntax will be removed
Warns

In `hereHDLMConfiguration` objects, the syntax for configuring a `hereHDLMReader` object to search catalogs from a specific region, `hereHDLMConfiguration(region)`, will be removed in a future release. Instead, specify the catalog name that corresponds to that region by using the `hereHDLMConfiguration(catalog)` syntax.

Previously, the catalog names for regions such as North America were not available to customers. HERE Technologies now makes these catalog names available through the HERE HD Live Map Marketplace, making the region syntax unnecessary.

Planning and Control

Trajectory Planning Example: Plan a vehicle trajectory through highway traffic

The Highway Trajectory Planning Using Frenet Reference Path example shows how to plan a local trajectory in a highway driving scenario. This example uses a reference path and dynamic list of obstacles to generate alternative trajectories for an ego vehicle.

This example requires the Navigation Toolbox software.

Functionality being removed or changed

InflationRadius and VehicleDimensions properties of vehicleCostmap object have been removed

Errors

The `InflationRadius` and `VehicleDimensions` properties of the `vehicleCostmap` object have been removed. Follow this process instead:

1. Use the `inflationCollisionChecker` function to create an `InflationCollisionChecker` object, which has the `InflationRadius` and `VehicleDimensions` properties.
2. Specify this object as the value of the `CollisionChecker` property of the `vehicleCostmap` object.

If you do specify these properties for `vehicleCostmap`, the object returns an error.

When the `vehicleCostmap` object was introduced in R2018a, this object inflated obstacles based on the specified inflation radius and vehicle dimensions only. The `InflationCollisionChecker` object, which is specified in the `CollisionChecker` property of `vehicleCostmap`, provides additional configuration options for inflating obstacles. For example, you can specify the number of circles used to compute the inflation radius, enabling more precise collision checking.

Update Code

The table shows a typical usage of the `InflationRadius` and `VehicleDimensions` properties of `vehicleCostmap`. It also shows how to update your code by using the corresponding properties of an `InflationCollisionChecker` object.

Invalid Usage	Recommended Replacement
`vehicleDims = vehicleDimensions(5,2);`	`vehicleDims = vehicleDimensions(5,2);`
`inflationRadius = 1.2;`	`inflationRadius = 1.2;`
`costmap = vehicleCostmap(C, ...`	`ccConfig = inflationCollisionChecker(vehicleDims, ...`
`    'VehicleDimensions',vehicleDims, ...`	`    'InflationRadius',inflationRadius);`
`    'InflationRadius',inflationRadius);`	`costmap = vehicleCostmap(C, ...`
	`    'CollisionChecker',ccConfig);`

Applications

Automated Driving Reference Applications: Lane following with intelligent vehicles, lane following with RoadRunner scenes, traffic light negotiation with Unreal Engine, and code generation for lane marker detection

The Highway Lane Following with Intelligent Vehicles example shows how to test highway lane following in a scenario with intelligent target vehicles. The example configures the non-ego vehicles as intelligent target vehicles such that they perform velocity keeping, lane change, or lane following. Then, it tests the lane following application for an ego vehicle with respect to the changing behaviour of non-ego vehicles.

The Highway Lane Following with RoadRunner Scene shows how to test lane following application on a scene created using the RoadRunner scene editing software.

The Traffic Light Negotiation with Unreal Engine Visualization example shows how to design a decision logic for negotiating a traffic light at an intersection and test on prebuilt scenarios in 3D simulation environments that uses Unreal Engine.

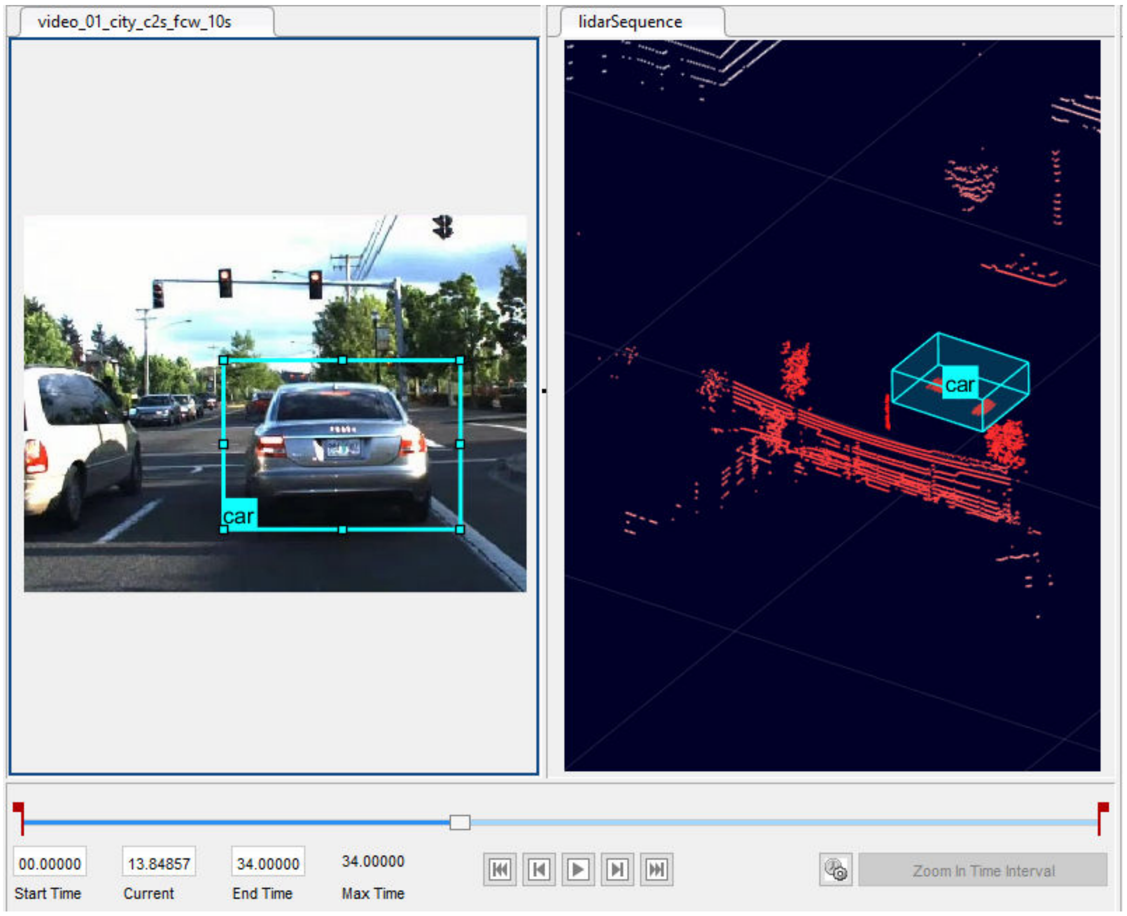
The Generate Code for Lane Marker Detector example show hows to test a lane marker detector algorithm on prebuilt scenarios in a 3D simulation environment and generate C++ code of the detector model for real-time application. This 3D simulation environment is rendered using the Unreal Engine from Epic Games

R2020a

Ground Truth Labeling

Multisignal Ground Truth Labeling: Label multiple lidar and video signals simultaneously

In the **Ground Truth Labeler** app, you can now label multiple signals representing the same scene within one app session.



Previously, you had to label each signal in separate sessions. With multisignal labeling, you can:

- Load multiple signal types, including lidar point cloud signals. Previously the app supported only image-based signals, which include videos and image sequences. You can now load signals individually or load a collection of signals from a single source, such as a rosbag. You can also create a custom reader for your own data source by using the `vision.labeler.loading.MultiSignalSource` API.
- Label signals that display a scene at the same timestamp within a single frame. You can also now label lidar signals by using the cuboid ROI label type. Cuboids are boxes that you draw around regions of interest within a lidar point cloud.
- Export labeled ground truth data across all signals within a `groundTruthMultisignal` object. Using this object, you can select labels by group name, signal name, signal type, label name, or label type. In addition, by using the `gatherLabelData` function, you can gather relevant data across multiple signals to train object detectors or semantic segmentation networks.

You can also create label definitions programmatically by using a `labelDefinitionCreatorMultisignal` object. You can then import these label definitions into the app.

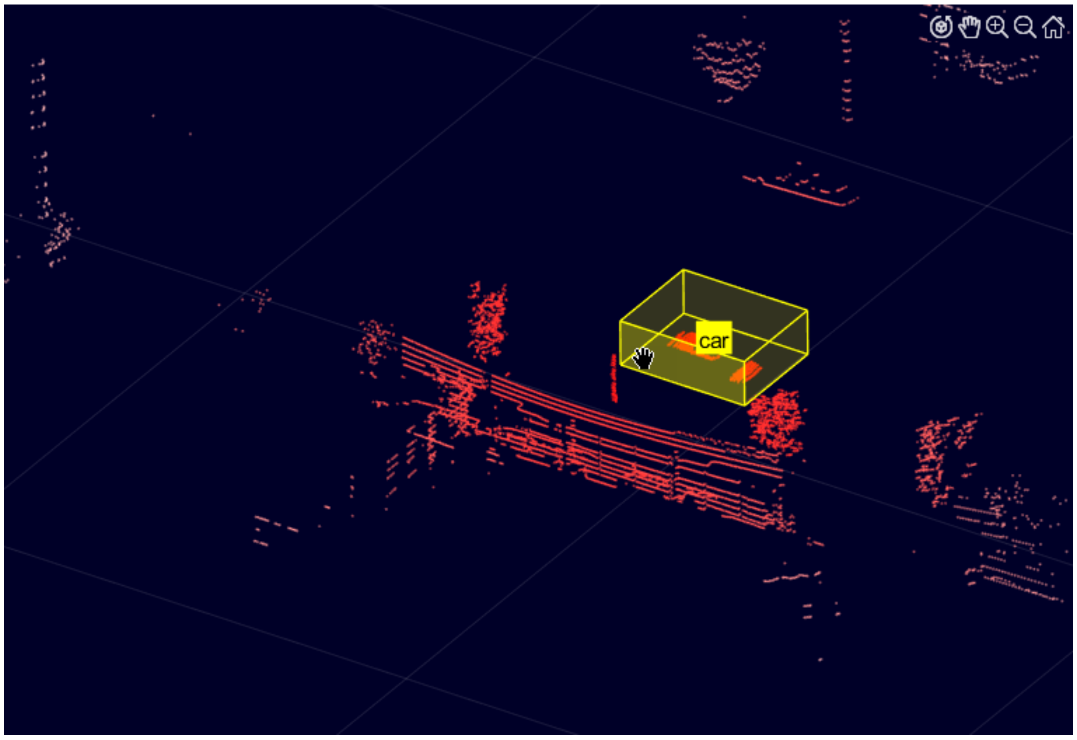
To get started labeling multiple signals, see Get Started with the Ground Truth Labeler.

Version History

If you open an app session that was created in a previous release, the session continues to run. However, the app now exports data as a `groundTruthMultisignal` object instead of a `groundTruth` object. If you do not need to label multiple signals simultaneously and do not require lidar labeling, use the **Video Labeler** app in Computer Vision Toolbox instead. The **Video Labeler** app continues to export `groundTruth` objects that were saved from the **Ground Truth Labeler** app in a previous release.

Lidar Labeling: Label lidar point clouds to train deep learning models

In the **Ground Truth Labeler** app, you can now label lidar point clouds. Previously, the app supported labeling of videos and image sequences only. To label lidar data, use the cuboid ROI label type. Cuboids are boxes that you draw around regions of interest within a lidar point cloud.



You can label lidar point clouds from these data sources:

- Point cloud sequences that are stored as point cloud data (PCD) or polygon (PLY) files
- Velodyne packet capture (PCAP) files
- Rosbags (requires ROS Toolbox)

You can use the labeled lidar data as training data for deep learning models, such as object detectors.

For more details on lidar labeling, see Label Lidar Point Clouds for Object Detection.

Ground Truth Labeler Enhancements: Rename scene labels, select ROI color, and configure ROI label name display

In the **Ground Truth Labeler** app, you can now:

- Rename scene labels.
- Set custom colors for ROI labels.
- Configure ROI label names to always display, never display, or display only when you pause your cursor over them.

Cuboid Scenario Simulation

Lidar Sensor Model: Generate synthetic point clouds from programmatic driving scenarios

Use the `lidarPointCloudGenerator` System object to model a lidar sensor and generate synthetic point cloud data for actors in a `drivingScenario` object.

The `lidarPointCloudGenerator` object obtains data from mesh representations of the roads and actors within the scenario.

- To obtain the road mesh for the road on which the ego vehicle travels, use the `roadMesh` function.
- To obtain the meshes of actors within the scenario, use the `actorProfiles` function. This function now additionally returns mesh properties. `Actor` and `Vehicle` objects also now contain mesh properties.

To define your own actor meshes, use the `extendedObjectMeshextendedObjectMesh` function or use one of these prebuilt meshes as a starting point:

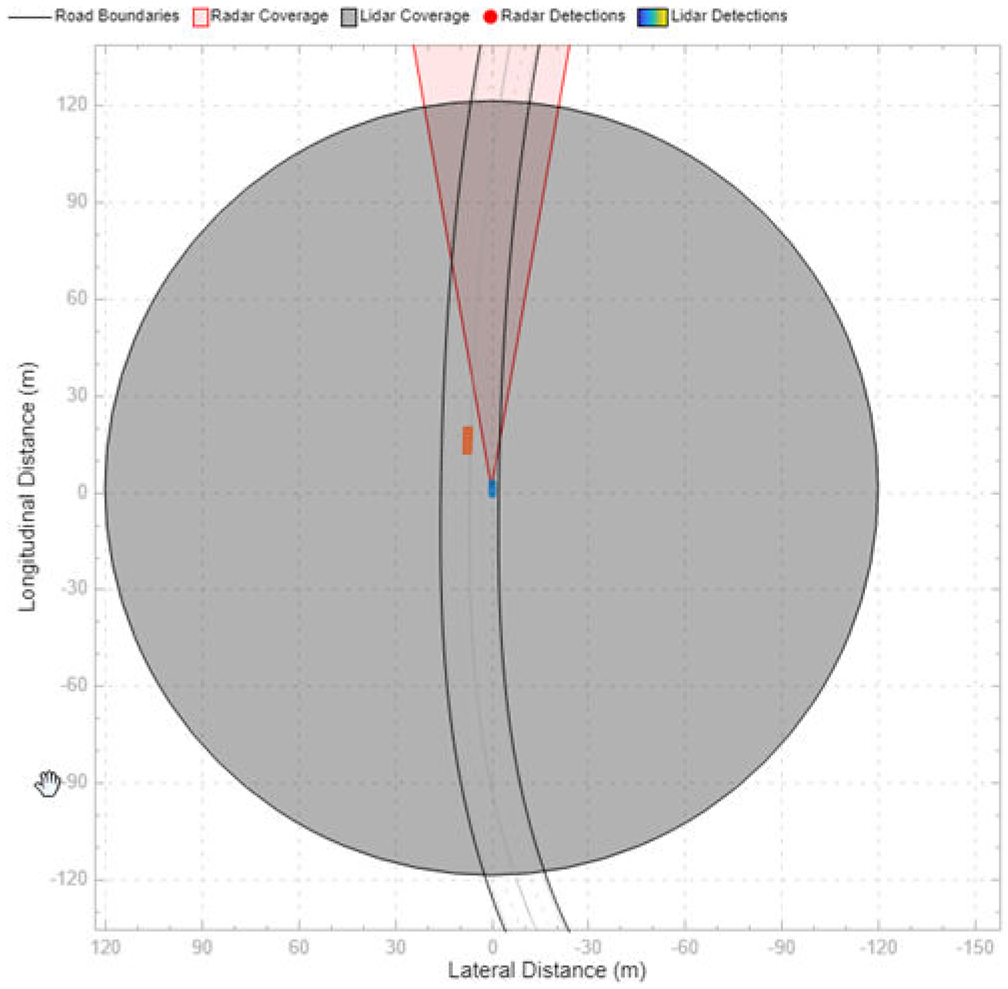
- `driving.scenario.carMesh`
- `driving.scenario.truckMesh`
- `driving.scenario.bicycleMesh`
- `driving.scenario.pedestrianMesh`

To visualize actor meshes on a bird's-eye plot, create a `pointCloudPlotter` object, and then plot the point cloud by using the `plotPointCloud` function.

For an example that shows how to fuse these synthetic point clouds with synthetic radar detections obtained from a `radarDetectionGenerator` System object, see the Track-Level Fusion of Radar and Lidar Data example

Bird's-Eye Scope Enhancements: Visualize radar and lidar data from 3D simulation sensors, and visualize actors from custom blocks

In the **Bird's-Eye Scope**, you can now visualize sensor data obtained from the 3D simulation environment, which is rendered using the Unreal Engine from Epic Games. You can visualize sensor coverage areas and detections from Simulation 3D Probabilistic Radar and Simulation 3D Lidar blocks. For more details about visualizing data from these sensors, see Visualize 3D Simulation Sensor Coverages and Detections

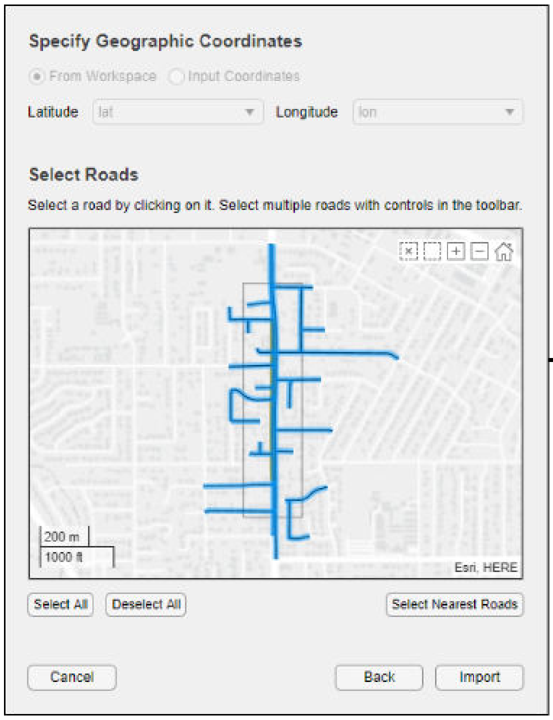


The scope also now visualizes actors from any blocks that create buses containing actor poses. Previously, the scope visualized actors output by the Scenario Reader block only. For details on the actor pose information required when creating these buses, see the **Actors** output port of the Scenario Reader block.

HERE HD Live Map Roads in Scenarios: Create driving scenarios using imported road data from high-definition geographic maps

In the **Driving Scenario Designer** app, you can now generate a road network with data obtained from the HERE HD Live Map [2] web service, provided by HERE Technologies.

2 You need to enter into a separate agreement with HERE in order to gain access to the HDLM services and to get the required credentials (access_key_id and access_key_secret) for using the HERE Service.

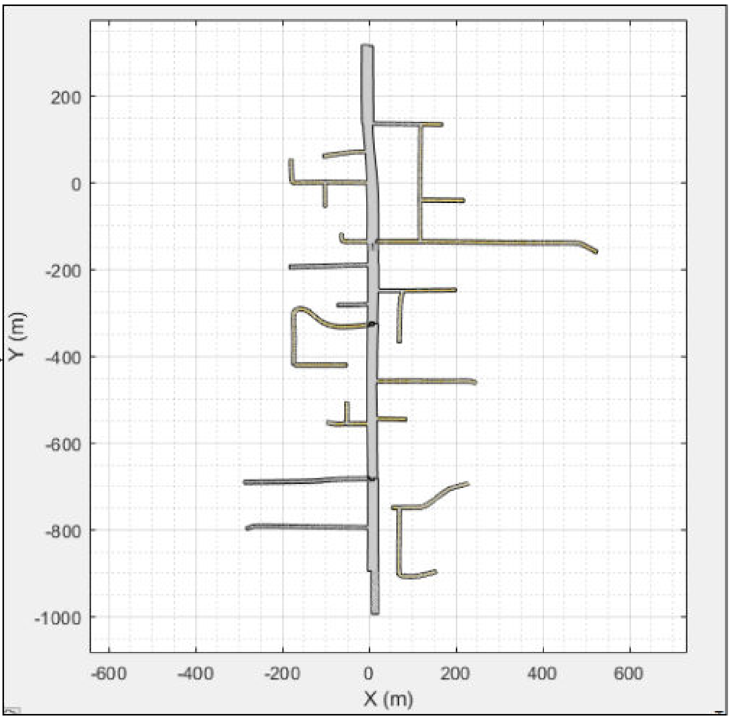

Specify Geographic Coordinates

From Workspace   Input Coordinates

Latitude: lat   Longitude: lon

Select Roads

Select a road by clicking on it. Select multiple roads with controls in the toolbar.



For more details, see Import HERE HD Live Map Roads into Driving Scenario.

You can also import these roads into a `drivingScenario` object by using the `'HEREHDLiveMap'` syntaxes in the `roadNetwork` function.

Scenario Coordinate Transformation Blocks: Convert between vehicle and world coordinates in driving scenarios, and convert between cuboid and 3D simulation coordinates

The Vehicle To World block converts non-ego actor poses from the coordinate system relative to the ego vehicle to the world coordinates of a driving scenario.

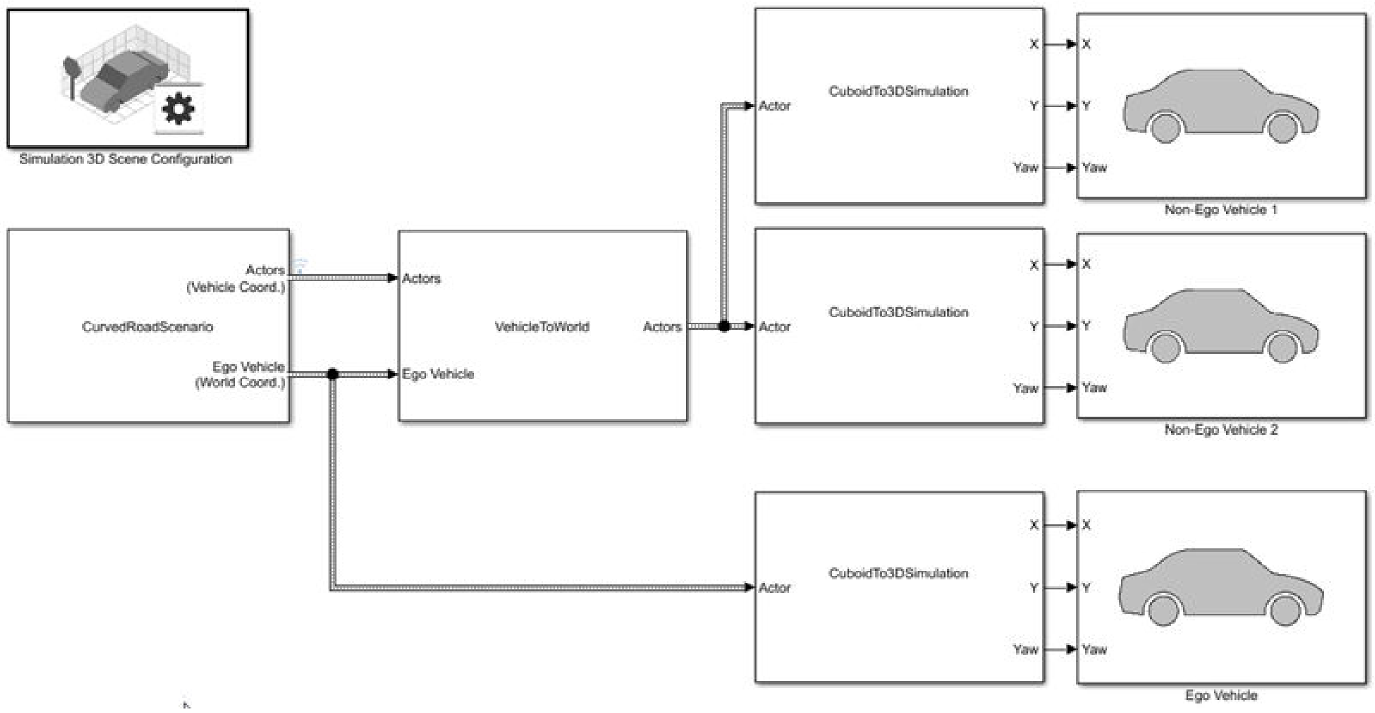
The Cuboid To 3D Simulation block converts vehicles authored in the cuboid environment into the coordinate system of the 3D simulation environment.

By using these two blocks together, you can take scenarios created in the **Driving Scenario Designer** app and recreate them within the 3D simulation environment. To recreate these scenarios, use this workflow.

1. In the **Driving Scenario Designer** app, create a driving scenario. As a starting point, use one of the prebuilt cuboid versions of 3D simulation environment scenes. For details, see Cuboid Versions of 3D Simulation Scenes in Driving Scenario Designer.
2. In a Simulink model, read the ground truth data from the app scenario file by using a Scenario Reader block. Configure the block to output the poses of both the ego vehicle and non-ego vehicles.
3. Configure a Simulation 3D Scene Configuration block to display the 3D simulation scene that is equivalent to the one you used in the app.

4. Convert the non-ego vehicle poses into world coordinates by using a Vehicle To World block.
5. Convert the ego and non-ego vehicle poses into the coordinate system of the 3D environment by using Cuboid To 3D Simulation blocks. These blocks offset the positions of the vehicles to account for the difference between origins in the two environments. In the cuboid environment, the origin is underneath the center of the rear axle. In the 3D simulation environment, the origin is at the approximate geometric center of the vehicle.
6. Specify the converted **X**, **Y**, and **Yaw** positions of all vehicles as the inputs to Simulation 3D Vehicle with Ground Following blocks. Configure the blocks to recreate the cuboid scene, and then simulate the model.

This block diagram shows a sample model of this workflow.



For an example that follows this workflow, see Visualize 3D Simulation Sensor Coverages and Detections.

You can also convert non-ego vehicle poses from world coordinates to the coordinate system relative to an ego vehicle by using the World To Vehicle block.

In addition, if you are using `drivingScenario` objects to create scenarios, you can now perform the programmatic equivalent of the Vehicle To World block conversion by using the `driving.scenario.targetsToScenario` function.

Programmatic Sensor Import: Read programmatically created radar and vision sensors into the Driving Scenario Designer app

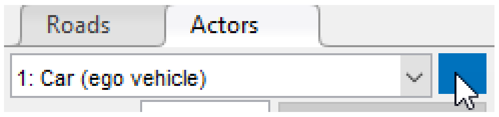
You can now import programmatically created radar and vision sensors into the **Driving Scenario Designer** app. The programmatic sensors must be created using `radarDetectionGenerator` and `visionDetectionGenerator` objects. You can also generate these programmatic sensors by using MATLAB code exported from the app.

The import of a `lidarPointCloudGenerator` System object into the app is not supported.

Custom Actor Colors: Specify the colors of actors in a driving scenario

In the **Driving Scenario Designer** app, you can now change the colors of actors in the scenario.

To change the color of an actor, next to the actor selection list, click the color patch for that actor.



Then, use the color picker to select one of the standard colors commonly used in MATLAB graphics or specify a custom color. You can also set a single default color for all newly created actors of a specific class.

To set the plot display colors of actors in programmatic driving scenarios, use the `PlotColor` property of `Actor` and `Vehicle` objects in a `drivingScenario` object. For details on setting this property, see the `'PlotColor'` name-value pair of the `actor` and `vehicle` functions.

Ego Vehicle Ground Following: Orient the ego vehicle to follow the road surface elevation in closed-loop simulations

In the Scenario Reader block, select the **Ego vehicle follows ground** parameter to orient the ego vehicle to follow the elevation of the road surface. The block updates the elevation, roll, pitch, and yaw of the ego vehicle and outputs actors and lane boundaries relative to the updated ego vehicle coordinates. Use this parameter in closed-loop simulations where the elevation of the road network varies.

Rear-Facing Lane Detections: Detect lane boundaries from rear-facing cameras in driving scenarios

In the Scenario Reader block, you can now output lane boundaries that are behind the ego vehicle. By specifying these lane boundaries to a Vision Detection Generator block, you can generate synthetic detections from rear-facing cameras mounted to the ego vehicle. For an example, see Test Open-Loop ADAS Algorithm Using Driving Scenario.

To output these rear-facing lane boundaries, in the Scenario Reader block, specify negative distances in the **Distances from ego vehicle for computing boundaries (m)** parameter. Previously, the block computed only positive distances, which correspond to lane boundaries in front of the ego vehicle.

You can also output lane boundaries in programmatic scenarios for use with `visionDetectionGenerator` objects. In the `laneBoundaries` function, specify negative distances in the `'XDistance'` name-value pair.

Road Interactions in Scenarios: Control the ability to modify roads in driving scenarios

In the **Driving Scenario Designer** app, when you import OpenDRIVE road networks or road data from the HERE HD Live Map web service, the ability to modify roads is disabled by default. Disabling these road interactions prevents you from accidentally modifying roads that are meant to match real-world scenarios. The prebuilt scenarios that simulate the 3D simulation scenes also have road interactions disabled. All other prebuilt scenarios and any scenarios that you create yourself have road interactions enabled by default.

To turn on or off road interactions in the app, in the bottom-left corner of the **Scenario Canvas** pane, first click the Configure the Scenario Canvas button ⚙. Then, select **Enable road interactions** or **Disable road interactions**, respectively.

Cuboid Versions of 3D Simulation Scenes: Build scenarios in the Driving Scenario Designer app for use in a 3D simulation environment

The **Driving Scenario Designer** app now provides prebuilt scenarios that recreate scenes from a 3D simulation environment. In these cuboid versions of the scenes, you can add vehicles, which are represented as simple box shapes, and specify their trajectories. Then, you can simulate these vehicles and trajectories in your Simulink model by using the higher fidelity 3D simulation versions of the scenes. The 3D environment renders the scenes using the Unreal Engine from Epic Games.

For details on opening these scenes and on the scenes that are available, see Cuboid Versions of 3D Simulation Scenes in Driving Scenario Designer.

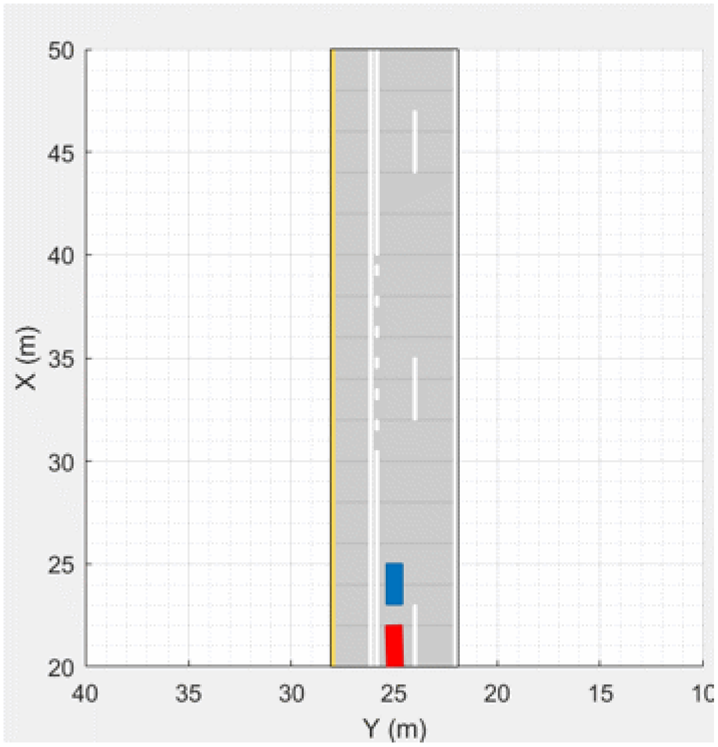
For an example that shows how to use these scenes with a 3D simulation Simulink model, see Visualize 3D Simulation Sensor Coverages and Detections.

laneMarking Function Enhancements: Define lane marking with multiple marker styles

You can now use the `laneMarking` function to define multiple marker styles along a lane by following these steps.

1. Create an array of lane marking objects with different marker types. Use the name-value pair `'SegmentRange'` to specify the range for each marker type. For example, this code specifies a lane marking with two marker types.

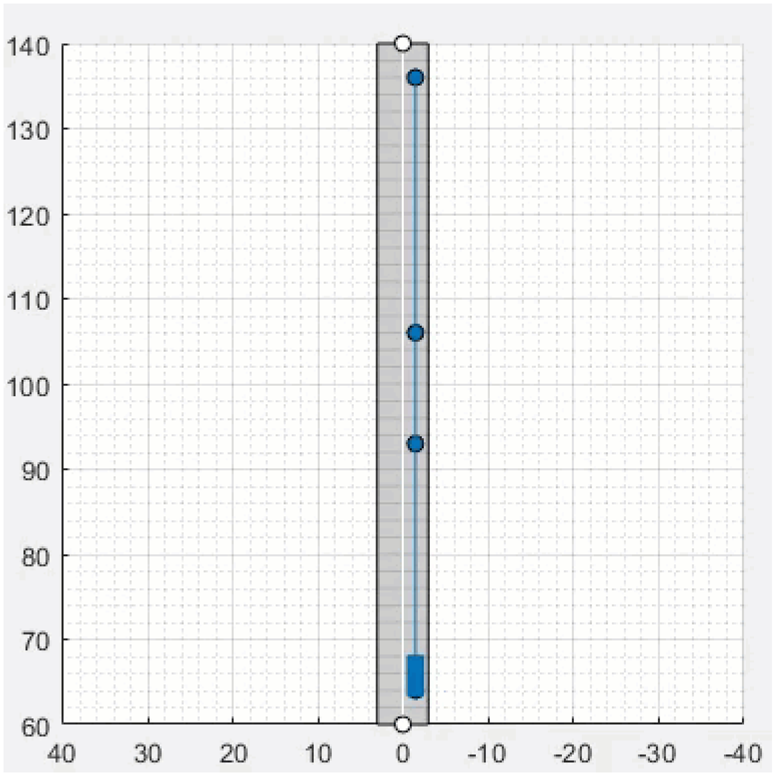
   ```
   ([laneMarking('Solid') laneMarking('Dashed')],'SegmentRange',[0.5 0.5]);
   ```
2. Pass the array as input to the `laneMarking` function. The function outputs a composite lane marking object that contains the properties of different markers along the lane.



Example of Driving Scenario Using Composite Lane Marking for Passing Zones

trajectory Function Enhancements: Pause actors at a waypoint

The `trajectory` function now takes wait times as an input to pause actors at specific waypoints along a trajectory. Use the `waittime` input argument of the `trajectory` function to generate stop-and-go driving scenarios.



Example of Stop-and-Go Driving Scenario

Driving Scenario Designer App Enhancements: Add composite lane markings and wait times

In the **Driving Scenario Designer** app, you can now:

- Add composite lane markings to a lane by specifying different markers along a lane.
- Add wait times to pause an actor at desired waypoints along its trajectory.

Driving Scenarios: Improved performance when creating road networks and actor trajectories

The `drivingScenario` object and **Driving Scenario Designer** app show improved performance when creating roads or trajectories of more than 40 km and when creating road networks containing approximately 500 roads or more. The table shows speed-ups of up to 75% when creating road networks and up to 95% when creating actor trajectories.

Scenario	R2019b	R2020a
Single long road (~44 km)	24.1 s	5.73 s
Large road network (489 roads)	21.48 s	12.49 s
Single long actor trajectory (~44 km)	10.08 s	0.43 s

Unreal Engine Scenario Simulation

3D Scene Customization: Simulate driving scenarios in a 3D environment using scenes created in the Unreal Editor

The Simulation 3D Scene Configuration block now provides options for simulating driving scenarios and sensors within your own customized scenes. Previously, the block enabled you to simulate only within a set of prebuilt scenes. The customized scenes must have been created using the Unreal Editor and must be compatible with Version 4.23. Using custom scenes, you can:

- Simulate vehicles and sensors from your Simulink model directly in the Unreal® Editor. Use this option to quickly modify your scene based on simulation results.
- Package scenes into an executable file and simulate from them by using the Simulation 3D Scene Configuration block. Use this option to speed up performance and to simulate in custom scenes without having to open the Unreal Editor.

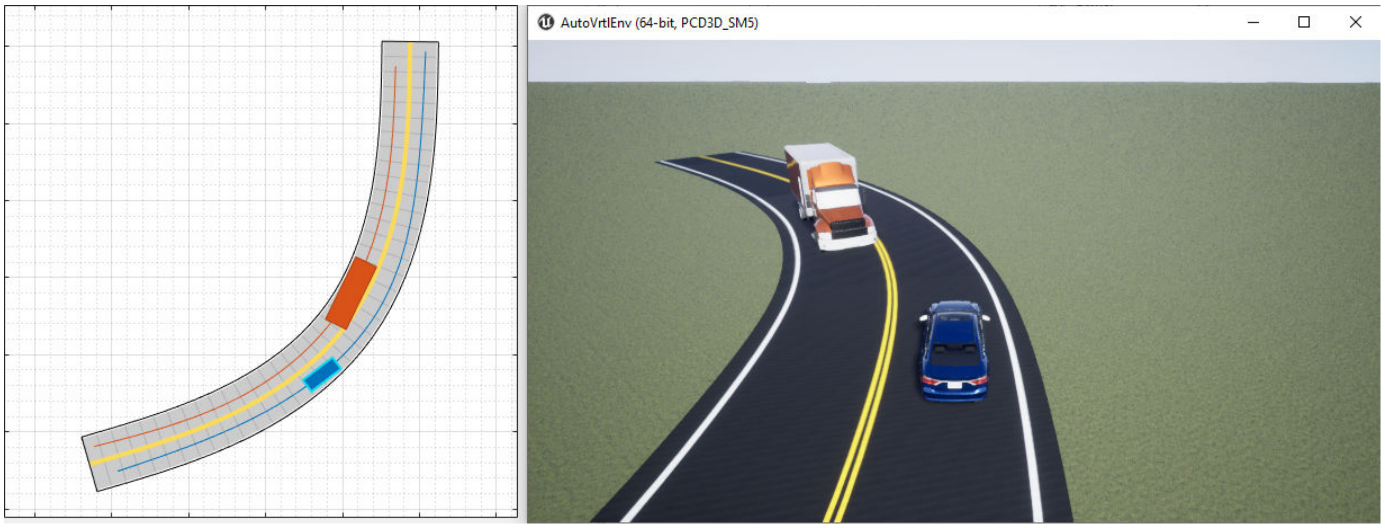
To use custom scenes, you must install the Automated Driving Toolbox Interface for Unreal Engine 4 Projects. This support package includes a plugin that establishes a connection between the Unreal Editor and MATLAB. It also includes customizable versions of the prebuilt 3D scenes that you can select from the Simulation 3D Scene Configuration block, with the exception of the **Virtual Mcity** scene.

Scene customization is available on Windows® 64-bit platforms only and requires Visual Studio® 2017 or higher.

For more details on scene customization, see Customize 3D Scenes for Automated Driving.

3D Display for Cuboid Simulations: Visualize scenarios in a 3D environment from the Driving Scenario Designer app

In the **Driving Scenario Designer** app, click **3D Display** to visualize your cuboid scenario in a 3D environment. The app renders this environment using the Unreal Engine from Epic Games.



You can also use this display as a preview of a scenario that you recreate for the 3D simulation environment in Simulink. For an example, see Visualize 3D Simulation Sensor Coverages and Detections.

Headless Mode: Run 3D simulations more quickly by not opening the Unreal Engine visualization window

In the Simulation 3D Scene Configuration block, use the **Display 3D window** parameter to select whether to display the 3D visualization window during simulation.

Consider running simulations without visualization, that is, in headless mode, in these cases.

- You want to run multiple 3D simulations in parallel to test models in different Unreal Engine scenarios.
- You want to capture sensor data to analyze in MATLAB but do not need to watch the visualization.

3D Simulation Version Upgrade: Run 3D simulations using Unreal Engine, Version 4.23

The 3D visualization engine that comes installed with Automated Driving Toolbox has been updated to Unreal Engine, Version 4.23. Previously, the toolbox used Unreal Engine, Version 4.19.

Version History

If your Simulink model uses a custom executable or project developed in a previous Unreal Engine version, you must migrate that project or executable to version 4.23. For more details on migrating projects or executables to newer Unreal Engine versions, see the Unreal Engine 4 documentation.

Box Truck Vehicle Type: Simulate vehicles with the dimensions of a box truck in the 3D simulation environment

You can configure the Simulation 3D Vehicle with Ground Following block to implement a box truck in 3D simulations. To create vehicles of this type, set the **Type** parameter of the vehicle block to `Box truck`. For box truck dimensions, see the Box Truck reference page.

Functionality being removed or changed

Renamed parameter in Simulation 3D Scene Configuration block

Behavior change

In the Simulation 3D Scene Configuration block, the **Scene description** parameter has been renamed to **Scene name**. Use this parameter to simulate in one of the default, prebuilt scenes provided with Automated Driving Toolbox. Starting in R2020a, to simulate in one of these scenes, you must first set the **Scene source** parameter to `Default Scenes`, which is the default selection for this parameter.

Detection and Tracking

YOLO v2 Vehicle Detection: Detect vehicles using a vehicle detector pretrained by a you-only-look-once (YOLO) v2 network

Use the `vehicleDetectorYOLOv2` function to detect vehicles by using a pretrained YOLO v2 vehicle detector.

SSD Object Detection: Detect objects in monocular camera images using the single shot multibox detector (SSD) algorithm

The `configureDetectorMonoCamera` function can now configure a monocular camera to use the SSD algorithm, returning an `ssdObjectDetectorMonoCamera` object.

Multiple-Object Tracking Enhancements: Initialize, confirm, and delete tracks, and predict track states at specified times

In a multi-object tracker created using a `multiObjectTracker` System object, you can now perform these actions.

- Manually initialize tracks in the tracker by using the `initializeTrack` function.
- Manually delete existing tracks from the tracker by using the `deleteTrack` function.
- Confirm or delete tracks based on recent track history by using the `ConfirmationThreshold` and `DeletionThreshold` properties of the tracker. The tracker now uses the `trackHistoryLogic` object to confirm or delete tracks.
- Predict tracks to specified times by using the `predictTracksToTime` function.

In addition, in MATLAB, the tracker now returns tracks as an array of `objectTrack` objects. When generating C or C++ code using MATLAB Coder™, the tracker still returns tracks as an array of structures, which was previously the only returned track format. However, the `Time` field of these structures has been renamed to `UpdateTime`. This field corresponds to the `UpdateTime` property of `objectTrack` objects.

These enhancements make the `multiObjectTracker` System object more closely aligned with the trackers in Sensor Fusion and Tracking Toolbox, making it easier to switch between trackers in your code.

Version History

As a result of these enhancements, the `ConfirmationParameters` and `NumCoastingUpdates` properties are no longer recommended. Instead, use `ConfirmationThreshold` and `DeletionThreshold`, respectively. For details about updating your code to use the recommended properties, `ConfirmationParameters` and `NumCoastingUpdates` properties of the `multiObjectTracker` System object are not recommended.

If you are using a previous version of MATLAB, then the change in output track format has additional compatibility considerations. For more details, see Track output format of `multiObjectTracker` changed.

Track History Logic: Confirm and delete tracks based on recent track history

The `trackHistoryLogic` object confirms or deletes tracks based on the recent track history. Configure this object to manage the tracks of a `multiObjectTracker` System object.

Alpha-Beta Estimation Filter: Track objects using a linear motion and measurement models

The `trackingABF` object is an alpha-beta tracking filter that follows a linear motion model and has a linear measurement model. Linear motion is defined by constant velocity or constant acceleration. Use this filter to predict the future location of an object, reduce noise for a detected location, and help associate multiple objects with their tracks.

Code Generation: Generate C/C++ code using MATLAB Coder

These objects and functions now support code generation.

- `parabolicLaneBoundary`
- `findParabolicLaneBoundaries`
- `cubicLaneBoundary`
- `findCubicLaneBoundaries`
- `insertLaneBoundary`
- `computeBoundaryModel`

Tracking Examples: Fuse radar and lidar tracks, perform track-to-track fusion in Simulink, and track vehicles using lidar in Simulink

The Track-Level Fusion of Radar and Lidar Data example shows how to fuse tracks obtained by radar and lidar sensor measurements.

The Track-to-Track Fusion for Automotive Safety Applications in Simulink example shows how to perform track-to-track level fusion by building a decentralized tracking architecture in Simulink.

The Track Vehicles Using Lidar Data in Simulink example shows the Simulink workflow for processing lidar point cloud data and using that data to track vehicles.

These examples require the Sensor Fusion and Tracking Toolbox software.

Functionality being removed or changed

ConfirmationParameters and NumCoastingUpdates properties of the multiObjectTracker System object are not recommended

Still runs

The `ConfirmationParameters` and `NumCoastingUpdates` properties of the `multiObjectTracker` System object are not recommended. Instead, use their corresponding properties: `ConfirmationThreshold` and `DeletionThreshold`, respectively. These properties are the same ones used in Sensor Fusion and Tracking Toolbox trackers, making it easier to switch between trackers in your code.

There are no current plans to remove `ConfirmationParameters` and `NumCoastingUpdates`. If you do specify these properties, the values in the corresponding `ConfirmationThreshold` and `DeletionThreshold` properties are updated to match.

Update Code

The table shows a typical usage of the `ConfirmationParameters` and `NumCoastingUpdates` properties, where you set the properties during creation by using name-value pairs. The table also shows how to update your code by using the corresponding new properties.

Recommended	Not Recommended

`tracker = multiObjectTracker( ...`
`  'ConfirmationParameters',[4 5], ...`
`  'NumCoastingUpdates',10);`
	`tracker = multiObjectTracker( ...`
	`  'ConfirmationThreshold',[4 5], ...`
	`  'Deletionthreshold',10);`

Track output format of multiObjectTracker changed

Behavior change

Starting from R2020a, the track output format of `multiObjectTracker` changes from track structure to `objectTrack`. As a result, when you load a `multiObjectTracker` created in an earlier version of MATLAB, you need to release the tracker first so that it can allow `objectTrack` as the track output format.

Localization and Mapping

Geographic Coordinate Transformations: Convert between geographic and local coordinates

Use the `latlon2local` function to convert geographic latitude-longitude coordinates to local (*x*, *y*) coordinates. To convert local coordinates to geographic coordinates, use the `local2latlon` function.

Multiroute Geographic Map Display: Simultaneously stream geographic coordinates from multiple driving routes

The `geoplayer` object now supports the display of multiple driving routes. To control which route remains visible in the plot, use the `CenterOnID` property.

Lidar SLAM Examples: Build a map from lidar data using a simultaneous localization and mapping algorithm

The Build a Map from Lidar Data Using SLAM example shows how to process recorded lidar data to build a map and estimate the trajectory of a vehicle by using a SLAM algorithm.

The Design Lidar SLAM Algorithm Using 3D Simulation Environment shows how to build a map using synthetic lidar data recorded from a 3D simulation environment.

Planning and Control

Quaternions: Represent orientation and rotations efficiently for localization

The `quaternion` data type enables efficient representation of orientation and rotations. In automated driving, sensors such as inertial measurement units (IMUs) report orientation readings as quaternions. To use this data for localization, you can capture it in a `quaternion` object and convert it to other rotation formats, such as Euler angles and rotation matrices. For more details on quaternions, see Rotations, Orientations, and Quaternions for Automated Driving.

Applications

Automated Driving Reference Applications: Simulate highway lane following, highway lane change, and traffic light negotiation systems

The Highway Lane Following example shows how to simulate a highway lane-following application that has controller, sensor fusion, and vision processing components. These components are tested in a 3D simulation environment that includes camera and radar sensor models. To automate the testing of these components and their generated code using Simulink Test software, see the Automate Testing for Highway Lane Following example.

The Highway Lane Change example shows how to simulate an automated lane change maneuver system for a highway driving scenario.

The Traffic Light Negotiation example shows how to design and test decision logic for negotiating a traffic light at an intersection.

R2019b

Ground Truth Labeling

Ground Truth Labeling Enhancements: Copy and paste pixel labels, improved pan and zoom, and improved frame navigation

With the **Ground Truth Labeler** app, you can now:

- Copy and paste pixel labels
- Pan and zoom more easily within the labeling window.
- Navigate to a specific frame by clicking on the scrubber or visual summary timeline

Lane Boundary Detection Algorithm: Automate the labeling of lane boundaries using the Ground Truth Labeler

The **Ground Truth Labeler** app now includes a built-in algorithm for automating the labeling of lane boundaries in a video or image sequence. Select this algorithm from the **Automate Labeling** section of the app toolstrip.

Cuboid Scenario Simulation

drivingScenario Import: Read programmatically created driving scenarios into the Driving Scenario Designer app and Simulink

You can now import programmatically created driving scenarios into the **Driving Scenario Designer** app or Simulink by using the Scenario Reader block. You can create programmatic driving scenarios by generating a `drivingScenario` object from the app or specifying a `drivingScenario` object at the MATLAB command line. These objects enable you to create multiple variations of scenarios. You can then import these scenarios into the app or into Simulink and test your driving algorithm on these variations. For more details, see Create Driving Scenario Variations Programmatically.

Driving Scenario Designer Export to Simulink: Generate Simulink models of driving scenarios and sensors

You can now generate a Simulink model from a scenario developed using the **Driving Scenario Designer** app. The generated models contain a Scenario Reader that reads roads and actors from the scenario and sensor detections blocks that recreate the sensors defined in the app. For more details on generating these blocks, see Generate Sensor Detection Blocks Using Driving Scenario Designer.

drivingScenario Enhancements: Create roads with driving, parking, border, shoulder, and restricted lanes

Use the `laneType` function to define different lane types for roads in a driving scenario. You can define driving, parking, border, shoulder, and restricted lanes. To create a driving scenario containing roads with different types of lanes, follow these steps:

1. Define lane types by using the `laneType` function to create a lane type object.
2. Create lane specifications for a road by using the `lanespec` function. Add the lane type object to lane specifications by using the `'Type'` name-value pair of the `lanespec` function.
3. Add roads with specified lanes to the driving scenario by using the `road` function.

roadNetwork Enhancements: Import additional lane types of OpenDRIVE roads into a driving scenario

You can now read and import parking, border, shoulder, and restricted lane types in an OpenDRIVE road network into a driving scenario by using the `roadNetwork` function. Previously, only driving lanes were supported. To show lane types in the driving scenario plot, use the `'ShowLaneTypes'` name-value pair of the `roadNetwork` function.

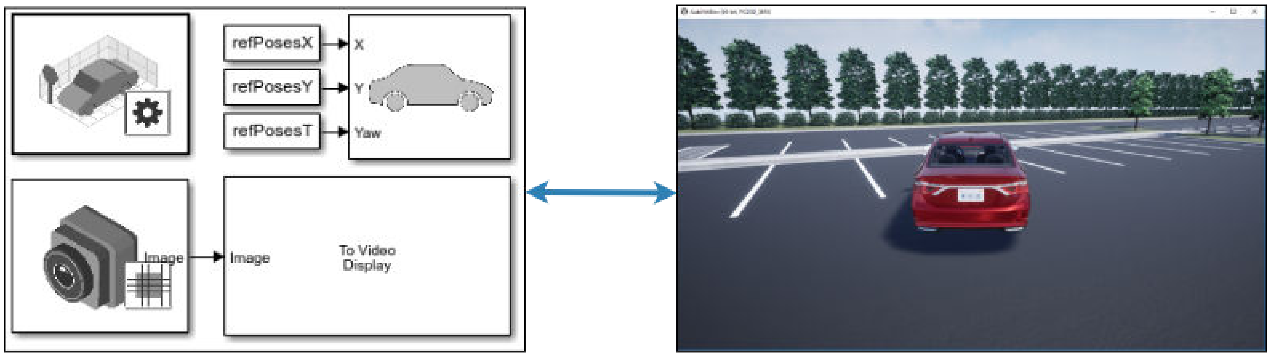
Bird's-Eye Scope World Coordinates View: Visualize scenarios in world coordinates

Using the **Bird's-Eye Scope**, you can now view the ground truth of a scenario in world coordinates. Previously, the scope displayed scenarios in vehicle coordinates only. You can simultaneously view scenarios in both vehicle coordinates and world coordinates.

Unreal Engine Scenario Simulation

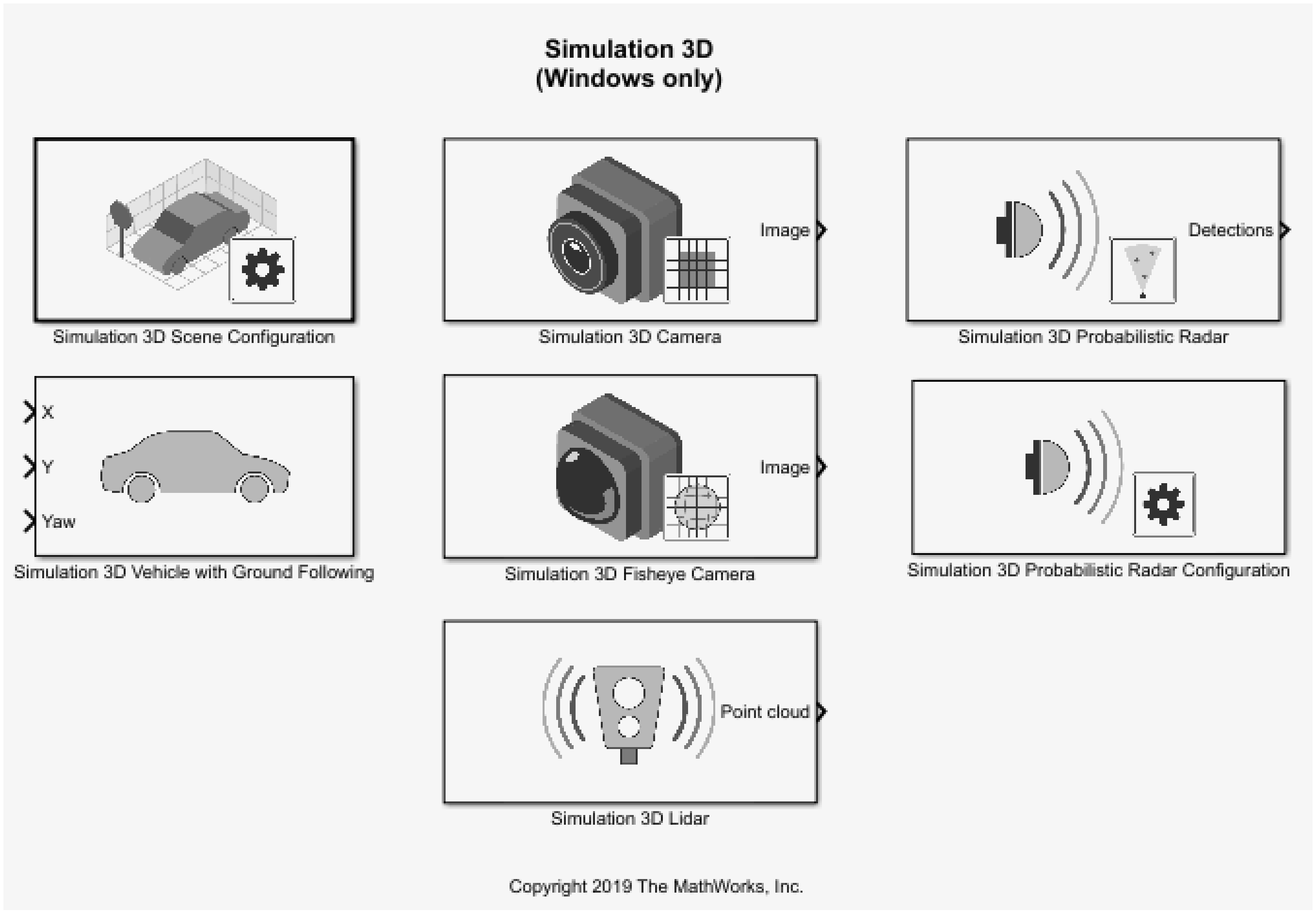
3D Simulation: Develop, test, and verify driving algorithms in a 3D simulation environment rendered using the Unreal Engine from Epic Games

Automated Driving Toolbox provides a cosimulation framework for modeling driving algorithms in Simulink and visualizing their performance in a 3D environment. This 3D simulation environment is rendered using the Unreal Engine from Epic Games.



To use the provided 3D simulation blocks, open the Simulation 3D block library.

```
drivingsim3d
```


Simulation 3D
(Windows only)
Simulation 3D Scene Configuration
Image
Simulation 3D Camera
Detections
Simulation 3D Probabilistic Radar
X
Y
Yaw
Simulation 3D Vehicle with Ground Following
Image
Simulation 3D Fisheye Camera
Simulation 3D Probabilistic Radar Configuration
Point cloud
Simulation 3D Lidar
Copyright 2019 The MathWorks, Inc.

Using these blocks, you can:

- Configure prebuilt scenes in the 3D simulation environment.
- Place and move vehicles within these scenes.
- Set up camera, radar, and lidar sensors on the vehicles.
- Simulate sensor outputs based on the environment around the vehicle.
- Obtain ground truth data for semantic segmentation and depth information.

Use 3D simulation to supplement real data when developing, testing, and verifying the performance of automated driving algorithms. If you have a vehicle model, you can use sensor blocks to perform realistic closed-loop simulations that encompass the entire automated driving stack, from perception to control.

To get started, see these examples:

- Select Waypoints for 3D Simulation
- Design of Lane Marker Detector in 3D Simulation Environment
- Visualize Automated Parking Valet Using 3D Simulation
- Simulate Lidar Sensor Perception Algorithm

- Simulate Radar Sensors in 3D Environment

To learn more, see Unreal Engine Driving Scenario Simulation.

Note 3D simulation is supported on Windows only.

Detection and Tracking

Track-to-Track Fusion Example: Fuse tracks from multiple vehicles to increase automotive safety (requires Sensor Fusion and Tracking Toolbox)

The Track-to-Track Fusion for Automotive Safety Applications example shows how to fuse tracks from multiple vehicles to provide a more comprehensive estimate of the environment than can be seen by either vehicle alone. This example requires a Sensor Fusion and Tracking Toolbox license.

YOLO v2 Acceleration: Acceleration support for YOLO v2 object detection

The `detect` function used with `yolov2ObjectDetectorMonoCamera` objects now supports performance optimization in both CPU and GPU execution environments. To set the performance optimization, use the `'Acceleration'` name-value pair of the `detect` function.

Code Generation: Generate C/C++ code using MATLAB Coder

These objects and functions now support code generation:

- `acfObjectDetectorMonoCamera`
- `birdsEyeView`
- `segmentLaneMarkerRidge`

Localization and Mapping

Lidar Example: Build a map from lidar data

The Build a Map from Lidar Data example shows how to process 3-D lidar sensor data to progressively build a map, with assistance from inertial measurement unit (IMU) readings. You can use these built maps to plan paths for vehicle navigation or to perform localization. The example also shows how to evaluate and improve the built maps using global positioning system (GPS) readings.

This example requires a Mapping Toolbox™ license.

HERE HD Live Map Linux Support: Read and visualize high-definition map data on Linux machines

`hereHDLMReader` objects are now supported on Linux machines. The HERE HD Live Map service is now supported on all platforms (Windows, Mac, and Linux®).

Planning and Control

Velocity Profiler: Generate the velocity profile of a driving path given kinematic constraints

The Velocity Profiler block generates a velocity profile of a driving path that satisfies a set of specified kinematic constraints. These constraints include the physical limitations of the vehicle and comfort criteria such as maximum allowable speed, maximum lateral acceleration, and maximum longitudinal jerk.

You can use the generated velocity profile as the input reference velocities of a longitudinal controller, as shown in the Automated Parking Valet in Simulink example.

For more details on using the Velocity Profiler block, see these examples:

- Velocity Profile of Straight Path
- Velocity Profile of Path with Curve and Direction Change

Functionality being removed or changed

InflationRadius and VehicleDimensions properties of vehicleCostmap object will be removed

Warns

The `InflationRadius` and `VehicleDimensions` properties of `vehicleCostmap` objects will be removed in a future release. Instead:

1. Use the `inflationCollisionChecker` function to create an `InflationCollisionChecker` object, which has the properties `InflationRadius` and `VehicleDimensions`.
2. Specify this object as the value of the `CollisionChecker` property of `vehicleCostmap`.

If you do specify these properties for `vehicleCostmap`, the values in the corresponding properties of `CollisionChecker` are updated to match.

When the `vehicleCostmap` object was introduced in R2018a, this object inflated obstacles based on the specified inflation radius and vehicle dimensions only. The `InflationCollisionChecker` object, which is specified in the `CollisionChecker` property of `vehicleCostmap`, provides additional configuration options for inflating obstacles. For example, you can specify the number of circles used to compute the inflation radius, enabling more precise collision checking.

Update Code

The table shows a typical usage of the `InflationRadius` and `VehicleDimensions` properties of `vehicleCostmap`. It also shows how to update your code by using the corresponding properties of an `InflationCollisionChecker` object.

Discouraged Usage	Recommended Replacement

`vehicleDims = vehicleDimensions(5,2);`
`inflationRadius = 1.2;`
`costmap = vehicleCostmap(C, ...`
`    'VehicleDimensions',vehicleDims, ...`
`    'InflationRadius',inflationRadius);`
	`vehicleDims = vehicleDimensions(5,2);`
	`inflationRadius = 1.2;`
	`ccConfig = inflationCollisionChecker(vehicleDims, ...`
	`    'InflationRadius',inflationRadius);`
	`costmap = vehicleCostmap(C, ...`
	`    'CollisionChecker',ccConfig);`

R2019a

Ground Truth Labeling

Ground Truth Labeling: Organize labels by logical groups, use assisted freehand for pixel labeling, and other enhancements

With the **Ground Truth Labeler** app, you can now:

- Create groups for organizing label definitions. You can also move labels between groups by dragging them.
- Use the assisted freehand to create pixel regions of interest (ROIs) for semantic segmentation. This tool automatically find edges between selected points in an image.
- Move multiple selected ROIs in an image.
- Edit previously created label definitions.
- Add additional list items to a previously created attribute.

Cuboid Scenario Simulation

Scenario Reader: Read driving scenarios into Simulink to test vehicle controllers and sensor fusion algorithms

The Scenario Reader block reads the roads and actors from a scenario file created using the **Driving Scenario Designer** app. Use the output actor poses and lane boundaries to test your vehicle control and sensor fusion models. The block supports open-loop and closed-loop models and can return outputs in either vehicle coordinates or world coordinates.

For more details on using the Scenario Reader block, see these examples:

- Test Open-Loop ADAS Algorithm Using Driving Scenario
- Test Closed-Loop ADAS Algorithm Using Driving Scenario

Scenario Generation Example: Generate virtual driving scenarios from recorded vehicle data

The Scenario Generation from Recorded Vehicle Data example shows how to generate a virtual driving scenario from GPS and lidar data recorded from a vehicle.

Using virtual scenarios, you can:

- Visualize and study the real scenario being recreated from the recorded vehicle data.
- Synthesize scenario variations by programmatically modifying the virtual scenario. You can use these variations when designing and evaluating autonomous driving systems.

Detection and Tracking

YOLO v2 Object Detection: Detect objects in a monocular camera using a "you-only-look-once" v2 deep learning object detector

The `configureDetectorMonoCamera` function can now configure a YOLO v2 object detector, returning a `yolov2ObjectDetectorMonoCamera` object.

Tracking Examples: Track vehicles using lidar; evaluate the performance of extended object trackers

The Track Vehicles Using Lidar: From Point Cloud to Track List example shows how to use a joint probabilistic data association (JPDA) tracker to track vehicles with a lidar sensor.

In addition, the Extended Object Tracking example now shows how to track extended objects using a probability hypothesis density (PHD) tracker. The example also shows how to use error and assignment metrics to evaluate the results of different trackers.

These examples require a Sensor Fusion and Tracking Toolbox license.

Localization and Mapping

HERE HD Live Map Reader: Read and visualize data from high-definition maps designed for automated driving applications

Use the `hereHDLMReader` object to read road and lane network data from the HERE HD Live Map [3] (HDLM) web service, provided by HERE Technologies. HERE HDLM content provides highly detailed and accurate information about the vehicle environment and is suitable for applications such as localization, scenario generation, navigation, and path planning.

To configure the reader object to read in map data from a specific catalog or version, use a `hereHDLMConfiguration` object. To manage your HERE HDLM credentials, use the `hereHDLMCredentials` function.

For more details, see Access HERE HD Live Map Data. For an example, see Use HERE HD Live Map Data to Verify Lane Configurations.

Note HERE HDLM reader objects do not work on Linux machines.

Custom Basemaps: Choose geographic basemaps on which to visualize driving routes in geoplayer

The `geoplayer` object now supports the use of custom basemaps from providers such as HERE Technologies and OpenStreetMap. To specify a custom basemap, use the `addCustomBasemap` function. To remove a custom basemap, use the `removeCustomBasemap` function.

3 You need to enter into a separate agreement with HERE in order to gain access to the HDLM services and to get the required credentials (access_key_id and access_key_secret) for using the HERE Service.

Planning and Control

Longitudinal Controller: Control the velocity of autonomous vehicles

The Longitudinal Controller Stanley block computes the acceleration and deceleration commands needed to control the velocity of a vehicle. The block computes these commands using the discrete proportional-integral control law. Use this block in a closed-loop simulation to adjust the velocity of a vehicle as it follows a path.

Dynamic Lateral Controller: Control the steering angle of autonomous vehicles considering realistic vehicle dynamics

The Lateral Controller Stanley block now includes an option to specify a dynamic bicycle vehicle model. Use this model to compute the steering angle of vehicles in highway scenarios or other high-speed environments.

Path Smoother: Smooth a planned vehicle path

Use the Path Smoother Spline block and `smoothPathSpline` function to smooth paths that were planned using a `pathPlannerRRT` object or other path planner. To generate a smoothed path, the block and function fit a parametric cubic spline onto the original path. The generated paths are smooth enough for vehicle controllers to execute.

Code Generation for Path Planning: Generate C/C++ code for vehicle path planning using MATLAB Coder

These path planning functions and objects now support code generation:

- `vehicleDimensions`
- `inflationCollisionChecker`
- `vehicleCostmap`
- `checkFree`
- `checkOccupied`
- `getCosts`
- `setCosts`
- `pathPlannerRRT`
- `plan`
- `driving.Path`
- `interpolate`
- `driving.DubinsPathSegment`
- `driving.ReedsSheppPathSegment`
- `checkPathValidity`
- `smoothPathSpline`

For information on code generation limitations for any function or object, see its individual reference page. For a code generation example, see Code Generation for Path Planning and Vehicle Control.

You can also generate code from these functions and objects in Simulink by using the MATLAB Function block.

R2018b

Ground Truth Labeling

Define multiple custom labels in Ground Truth Labeler connector

You can now synchronize the **Ground Truth Labeler** app with external labeling tools containing multiple custom labels. Specify these labels and their descriptions using the `LabelName` and `LabelDescription` properties of the `driving.connector.Connector` class.

Ground Truth Labeler enhancements

The **Ground Truth Labeler** app now includes visuals indicating the relationship between the labels and sublabels of an image. For more details on the label-sublabel relationship, see Use Sublabels and Attributes to Label Ground Truth Data (Computer Vision System Toolbox).

In addition, in the Label Summary window, you can now navigate between unlabeled frames. For more details on the Label Summary window, see View Summary of Ground Truth Labels (Computer Vision System Toolbox).

Cuboid Scenario Simulation

Bird's-Eye Scope for Simulink: Analyze sensor coverages, detections, and tracks in your model

The **Bird's-Eye Scope** displays streaming detections and object tracks from your model on a bird's-eye plot. You can use the **Bird's-Eye Scope** to:

- Inspect the coverage areas of radar and vision sensors.
- Analyze the sensor detections of lanes and actors in a driving scenario.
- Analyze the tracks of moving objects.

To get started using the scope, see Visualize Sensor Data and Tracks in Bird's-Eye Scope.

Prebuilt Driving Scenarios: Test driving algorithms using Euro NCAP and other prebuilt scenarios

In the **Driving Scenario Designer** app, you can now test that your algorithms comply with ADAS industry standards by using prebuilt Euro NCAP driving scenarios. These scenarios model multiple variations of Euro NCAP test procedures for lane keeping assist, automatic emergency braking, and emergency lane keeping. For more details, see Generate Synthetic Detections from a Euro NCAP Scenario and the Automatic Emergency Braking with Sensor Fusion example.

In addition to Euro NCAP scenarios, the app includes prebuilt driving scenarios of common driving maneuvers at intersections. See Generate Synthetic Detections from a Prebuilt Driving Scenario

OpenDRIVE File Import Support: Load OpenDRIVE roads into a driving scenario

In the **Driving Scenario Designer** app, you can now include roads built using the OpenDRIVE format specification. For more details, see Add OpenDRIVE Roads to Driving Scenario.

You can also load these roads into a `drivingScenario` object by using the `roadNetwork` function.

Radar Sensor Model Enhancements: Model occlusions in radar sensors

In the `radarDetectionGenerator` System object, use the `HasOcclusion` property to generate detections only from objects for which the radar has a direct line of sight.

Actors follow road elevation and banking angles in Driving Scenario Designer

In the **Driving Scenario Designer** app, when you create an actor and specify waypoints for it to follow, the actor now travels along the elevation angle and banking angle of the road.

Functionality being removed or changed

Corrections to Image Width and Image Height camera parameters of Driving Scenario Designer

Behavior change

Starting in R2018b, in the **Camera Settings** group of the **Driving Scenario Designer** app, the **Image Width** and **Image Height** parameters set their expected values. Previously, **Image Width** set the height of images produced by the camera, and **Image Height** set the width of images produced by the camera.

If you are using R2018a, to produce the expected image sizes, transpose the values set in the **Image Width** and **Image Height** parameters.

Detection and Tracking

Monocular Camera Parameter Estimation: Configure a monocular camera by estimating its extrinsic parameters

The `estimateMonoCameraParameters` function estimates the extrinsic parameters of a monocular camera that has been calibrated using a checkerboard pattern. For more details, see Calibrate a Monocular Camera.

Monocular camera setup with fisheye lens example

The Configure Monocular Fisheye Camera example shows how to set up a monocular camera that has a fisheye lens.

Sensor fusion and tracking examples

The following examples require a Sensor Fusion and Tracking Toolbox license.

- The Extended Object Tracking example shows how to track objects whose dimensions span multiple sensor resolution cells.
- The Visual-Inertial Odometry Using Synthetic Data example shows how to estimate the pose (position and orientation) of a vehicle by using an inertial measurement unit (IMU) and a monocular camera.

Planning and Control

Improved Collision Checking in vehicleCostmap Object: Configure collision checking to plan paths through narrow passages

The `inflationCollisionChecker` function creates a configuration object that specifies how the `vehicleCostmap` object checks for collisions. You can use this collision-checking configuration object to reduce the amount of obstacle inflation in the costmap. By reducing this inflation amount, path planning algorithms can plan collision-free paths through narrow passages such as parking spots.

For compatibility considerations, see `InflationRadius` and `VehicleDimensions` properties of `vehicleCostmap` object are not recommended.

Kinematic Lateral Controller: Control the steering angle of an autonomous vehicle

The Lateral Controller Stanley block and `lateralControllerStanley` function compute the steering angle of a vehicle using the Stanley method, a kinematic control algorithm. Use this block or function in a closed-loop simulation to adjust the steering angle of a vehicle as it follows a path. To learn more, see Lateral Control Tutorial.

Obtain transition poses and direction changes from a planned path

The `driving.Path` object returned by `pathPlannerRRT` now contains more specific descriptions of path segments, including their motion lengths, motion directions, and motion types (Dubins or Reeds-Shepp). Use the `interpolate` function to sample poses along the path, including transition poses, and to return changes in direction. You can then use these sampled poses and direction changes to develop a path smoothing algorithm.

For compatibility considerations, see `connectingPoses` function and `driving.Path` object properties `KeyPoses` and `NumSegments` are not recommended.

Functionality being removed or changed

InflationRadius and VehicleDimensions properties of vehicleCostmap object are not recommended

Still runs

The `InflationRadius` and `VehicleDimensions` properties of `vehicleCostmap` are not recommended. Instead:

1. Use the `inflationCollisionChecker` function to create an `InflationCollisionChecker` object, which has the properties `InflationRadius` and `VehicleDimensions`.
2. Specify this object as the value of the `CollisionChecker` property of `vehicleCostmap`.

There are no current plans to remove the `InflationRadius` and `VehicleDimensions` properties of `vehicleCostmap`. If you do specify these properties, the values in the corresponding properties of `CollisionChecker` are updated to match.

When the `vehicleCostmap` object was introduced in R2018a, this object inflated obstacles based on the specified inflation radius and vehicle dimensions only. The `InflationCollisionChecker` object, which is specified in the `CollisionChecker` property of `vehicleCostmap`, provides additional configuration options for inflating obstacles. For example, you can specify the number of circles used to represent the vehicle shape, enabling more precise collision checking.

Update Code

The table shows a typical usage of the `InflationRadius` and `VehicleDimensions` properties of `vehicleCostmap`. It also shows how to update your code using the corresponding properties of an `InflationCollisionChecker` object.

Discouraged Usage	Recommended Replacement

`vehicleDims = vehicleDimensions(5,2);`
`inflationRadius = 1.2;`
`costmap = vehicleCostmap(C, ...`
`    'VehicleDimensions',vehicleDims, ...`
`    'InflationRadius',inflationRadius);`
	`vehicleDims = vehicleDimensions(5,2);`
	`inflationRadius = 1.2;`
	`ccConfig = inflationCollisionChecker(vehicleDims, ...`
	`    'InflationRadius',inflationRadius);`
	`costmap = vehicleCostmap(C, ...`
	`    'CollisionChecker',ccConfig);`

connectingPoses function and driving.Path object properties KeyPoses and NumSegments are not recommended
Still runs

The `connectingPoses` function and the `KeyPoses` and `NumSegments` properties of the `driving.Path` object are not recommended. Instead, use the `interpolate` function, which returns key poses, connecting poses, transition poses, and direction changes. The `KeyPoses` and `NumSegments` properties are no longer relevant. `KeyPoses`, `NumSegments`, and `connectingPoses` will be removed in a future release.

In R2018a, `connectingPoses` enabled you to obtain intermediate poses either along the entire path or along the path segments that are between key poses (as specified by `KeyPoses`). Using the `interpolate` function, you can now obtain intermediate poses at any specified point along the path. The `interpolate` function also provides transition poses at which changes in direction occur.

Update Code

Remove all instances of `KeyPoses` and `NumSegments` and replace all instances of `connectingPoses` with `interpolate`. The table shows typical usages of `connectingPoses` and how to update your code to use `interpolate` instead. Here, `path` is a `driving.Path` object returned by `pathPlannerRRT`.

Discouraged Usage	Recommended Replacement
`poses = connectingPoses(path);`	`poses = interpolate(path);`

`segID = 1;`
`posesSegment = connectingPoses(path,segID);`
	`interpolate` does not have a direct syntax for obtaining segment poses. However, you can sample poses of a segment using a specified step time. For example:
	
	`step = 0.1;`
	`samples = 0 : step : path.PathSegments(1).Length;`
	`segmentPoses = interpolate(path,samples);`

R2018a

Ground Truth Labeling

Ground Truth Pixel Labeling: Interactively label individual pixels in video data

In the **Ground Truth Labeler** app, you can now interactively label individual pixels in video data for training semantic segmentation algorithms. You can also automate the labeling. See Automate Ground Truth Labeling for Semantic Segmentation.

Ground Truth Label Attributes: Organize and classify ground truth labels using attributes and sublabels

In the **Ground Truth Labeler** app, you can now attach attributes to labels and create hierarchical sublabels. For more details, see Define Ground Truth Data for Video or Image Sequences.

Cuboid Scenario Simulation

Driving Scenario Designer: Interactively define actors and driving scenarios to test controllers and sensor fusion algorithms

Use the **Driving Scenario Designer** app to design a synthetic driving scenario composed of roads and actors (vehicles, pedestrians, and so on). You can generate visual and radar detections of actors in the scenario to test your sensor fusion and control algorithms. To learn how to generate detections, see Generate Synthetic Detections from an Interactive Driving Scenario.

Add and detect lanes in Driving Scenario

You can add lane markings to roads in a driving scenario simulation using the new lane marking function, `laneMarking`, and lane specification function, `lanespec`. The driving scenario `road` method accepts a lane specification as an input. To plot lane markings in `birdsEyePlot`, use `laneMarkingPlotter` and `plotLaneMarking`.

In addition, the vision detection generator System object, `visionDetectionGenerator`, can now detect lanes in a driving scenario simulation. The corresponding Simulink block, Vision Detection Generator, can also detect lanes.

Path method being removed

The `path` method of the `actor` and `vehicle` classes is being removed. Use the `trajectory` method instead.

Version History

Functionality	Result	Use Instead	Compatibility Considerations
`path` method	Still runs	`trajectory` method	Replace all instances of `path` with `trajectory`. The `path` syntax which assumes a default speed does not exist in `trajectory`. You must specify a speed input argument.

Detection and Tracking

Lidar Segmentation: Quickly segment 3-D point clouds from lidar

Use the `segmentLidarData` function to segment organized point clouds into clusters.

Point Cloud Reader for Velodyne PCAP Files: Import Velodyne lidar data into MATLAB

Use a `velodyneFileReader` object to read point cloud data from Velodyne packet capture (PCAP) files.

Detect lanes more precisely by using third-degree polynomial lane boundary models

Use the `cubicLaneBoundary` and `findCubicLaneBoundaries` functions to create and find lane boundaries using third-degree polynomial models. You can display detected lanes on a bird's-eye-view plot, and overlay the lane markings onto images, by using the `insertLaneBoundary` function.

Transform [x,y,z] locations in vehicle coordinates to image coordinates

The `vehicleToImage` method of `monoCamera` now accepts three-dimensional [*x*,*y*,*z*] point coordinates. Previously, `vehicleToImage` accepted only [*x*,*y*] coordinates. By transforming [*x*,*y*,*z*] locations in vehicle coordinates, you can display point locations above the road surface.

Direction of Yaw Angle Rotation Adjusted

The `monoCamera` function was updated to correct the direction of rotation for the yaw angle.

Version History

Functionality	Compatibility Considerations
`monoCamera` function	If you are using R2017b version of this function, you must multiply the yaw angle by `-1`.

Localization and Mapping

Streaming Geographic Map Display: Visualize a geographic route on a map

Use the `geoplayer` function to create an interactive map that displays the streaming geographic coordinates of a driving route.

Planning and Control

Path Planning: Plan driving paths using an RRT* path planner and costmap

Use the `pathPlannerRRT`, `vehicleCostmap`, and `checkPathValidity` functions to plan a driving path by using an optimal rapidly exploring random tree (RRT*) motion-planning algorithm. To learn how to use these functions to plan a path, see the Automated Parking Valet example.

Applications

ACC Reference Application: Use a reference model to simulate and test adaptive cruise controller (ACC) systems

The ACC reference application is a model of an ACC system implemented using sensor fusion. Use this model to design your own ACC system, test it in Simulink using synthetic radar and vision data generated by Automated Driving System Toolbox™ blocks, and automatically generate C code. To learn more, see Adaptive Cruise Control with Sensor Fusion.

R2017b

Version: 1.1

New Features

Ground Truth Labeling

Ground Truth Labeling App: Reverse playback capability while processing algorithms

In the **Ground Truth Labeler** app, you can now process the video in reverse using the automation algorithm. You can also now dock and undock the Visual Summary display.

Cuboid Scenario Simulation

Sensor Simulation Using Simulink Blocks: Generate synthetic object lists from camera and radar sensor models

Use the Radar Detection Generator and the Vision Detection Generator blocks to generate synthetic detections for testing and design of your sensor fusion and tracking algorithms

Code Generation for Sensor Models: Generate C code for camera and radar sensor models

Use the `radarDetectionGenerator` and `visionDetectionGenerator` System objects to generate C code to generate synthetic sensor detection object lists.

Detection and Tracking

Sensor Fusion Simulink Blocks: Track multiple objects and fuse detections from multiple sensors

Use the Detection Concatenation block and the Multi Object Tracker block to fuse and track objects detected by multiple sensors.

Applications

Autonomous Driving Examples

- Sensor Fusion Using Synthetic Radar and Vision Data
- Adaptive Cruise Control with Sensor Fusion
- Evaluate and Visualize Lane Boundary Detections Against Ground Truth
- Radar Signal Simulation and Processing for Automated Driving

R2017a

Version: 1.0

New Features

Ground Truth Labeling

Ground Truth Labeling

The **Ground Truth Labeler** app enables you to label ground truth data in a video or in a sequence of images. Use the app to interactively specify rectangular and polyline regions of interest (ROIs), and scene labels. You can export marked labels from the app and use them to train an object detector or to compare against ground truth data. The app includes computer vision algorithms to automate the labeling of ground truth by using detection and tracking algorithms. It also provides an API and workflow that enables you to import your own algorithms to automate the labeling of ground truth. You can also use the `driving.connector.Connector` API to display additional time-synchronized signals, such as lidar or CAN bus data.

Ground Truth Labeling Utilities	Description
Ground Truth Labeler	App for labeling ground truth data in a video or sequence of images
`groundTruth`	Object for storing ground truth labels
`groundTruthDataSource`	Create a ground truth data source
`objectDetectorTrainingData`	Create training data from ground truth data for an object detector
`driving.automation.AutomationAlgorithm`	Define automated labeling algorithm in the Ground Truth Labeler app
`driving.connector.Connector`	Interface to connect an external tool to the Ground Truth Labeler app
`evaluateLaneBoundaries`	Evaluate lane boundary models against ground truth

Cuboid Scenario Simulation

Bird’s-Eye Plot

Use `birdsEyePlot` to display a bird's-eye plot of a 2-D scene in the immediate vicinity of a vehicle. You can use bird's-eye plots with sensors capable of detecting objects and lanes.

Driving Scenario Generation and Sensor Models

The `drivingScenario` class defines road networks, vehicles, and traffic scenarios. A driving scenario is a 3-D arena containing roads and actors. Actors can represent anything that moves, such as cars, pedestrians, and bicycles. Actors can also include stationary obstacles that can influence the motion of other actors. You can use `radarDetectionGenerator` and the `visionDetectionGenerator` to create statistical models for generating synthetic radar and camera sensor detections.

Detection and Tracking

Monocular Camera Sensor Configuration

Use the `monoCamera` object to define your monocular camera configuration. You can use this object to convert locations between vehicle and image coordinate systems. You can also use `birdsEyeView` with the `monoCamera` object to create a bird’s-eye-view image.

Object and Lane Boundary Detection

Detect objects using machine learning techniques, including deep learning. You can also segment, detect, and model parabolic lane boundaries using RANSAC. Configure object detectors to detect objects of a known physical size using the `configureDetectorMonoCamera` function.

Object Detection

- `vehicleDetectorACF`
- `vehicleDetectorFasterRCNN`
- `peopleDetectorACF`
- `configureDetectorMonoCamera`
- `acfObjectDetectorMonoCamera`
- `objectDetectorTrainingData`
- `fastRCNNObjectDetectorMonoCamera`
- `fasterRCNNObjectDetectorMonoCamera`

Lane Boundary Detection

- `segmentLaneMarkerRidge`
- `findParabolicLaneBoundaries`
- `parabolicLaneBoundary`
- `insertLaneBoundary`
- `evaluateLaneBoundaries`
- `fitPolynomialRANSAC`
- `ransac`

Multi-object Tracking

You can create a multi-object tracker for sensor fusion. The tracker uses Kalman filters for estimating the state of motion of an object. Measurements made on the object let you continuously solve for the object's position and velocity. You can use constant-velocity or constant-acceleration motion models, or define your own models.

- `multiObjectTracker`
- `objectDetection`
- `getTrackPositions`
- `getTrackVelocities`

- `trackingKF`
- `trackingEKF`
- `trackingUKF`

Applications

Automated Driving Examples

The release of Automated Driving System Toolbox includes the following examples.

Reference Applications
Visual Perception Using Monocular Camera
Forward Collision Warning Using Sensor Fusion
Sensor Fusion Using Synthetic Radar and Vision Data

Tracking and Sensor Fusion
Forward Collision Warning Using Sensor Fusion
Track Multiple Vehicles Using a Camera
Track Pedestrians from a Moving Car
Multiple Object Tracking Tutorial
Code Generation for Tracking and Sensor Fusion

Perception with Computer Vision
Visual Perception Using Monocular Camera
Ground Plane and Obstacle Detection Using Lidar
Train a Deep Learning Vehicle Detector

Algorithm Validation and Visualization
Automate Ground Truth Labeling of Lane Boundaries
Annotate Video Using Detections in Vehicle Coordinates
Visualize Sensor Coverage, Detections, and Tracks
Evaluate Lane Boundary Detections Against Ground Truth Data

Scenario Generation
Sensor Fusion Using Synthetic Radar and Vision Data
Driving Scenario Tutorial
Define Road Layouts
Create Actor and Vehicle Paths
Model Radar Sensor Detections
Model Vision Sensor Detections