Automated Driving Toolbox™ Release Notes

MATLAB®&SIMULINK®

MathWorks®

Automated Driving Toolbox™ Release Notes

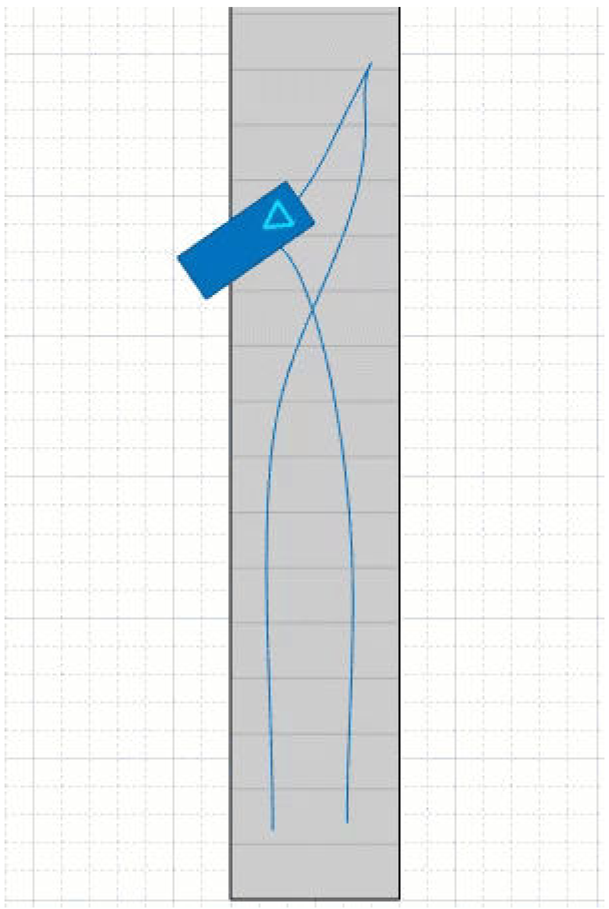
Contents

R2020b

R2020b

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.

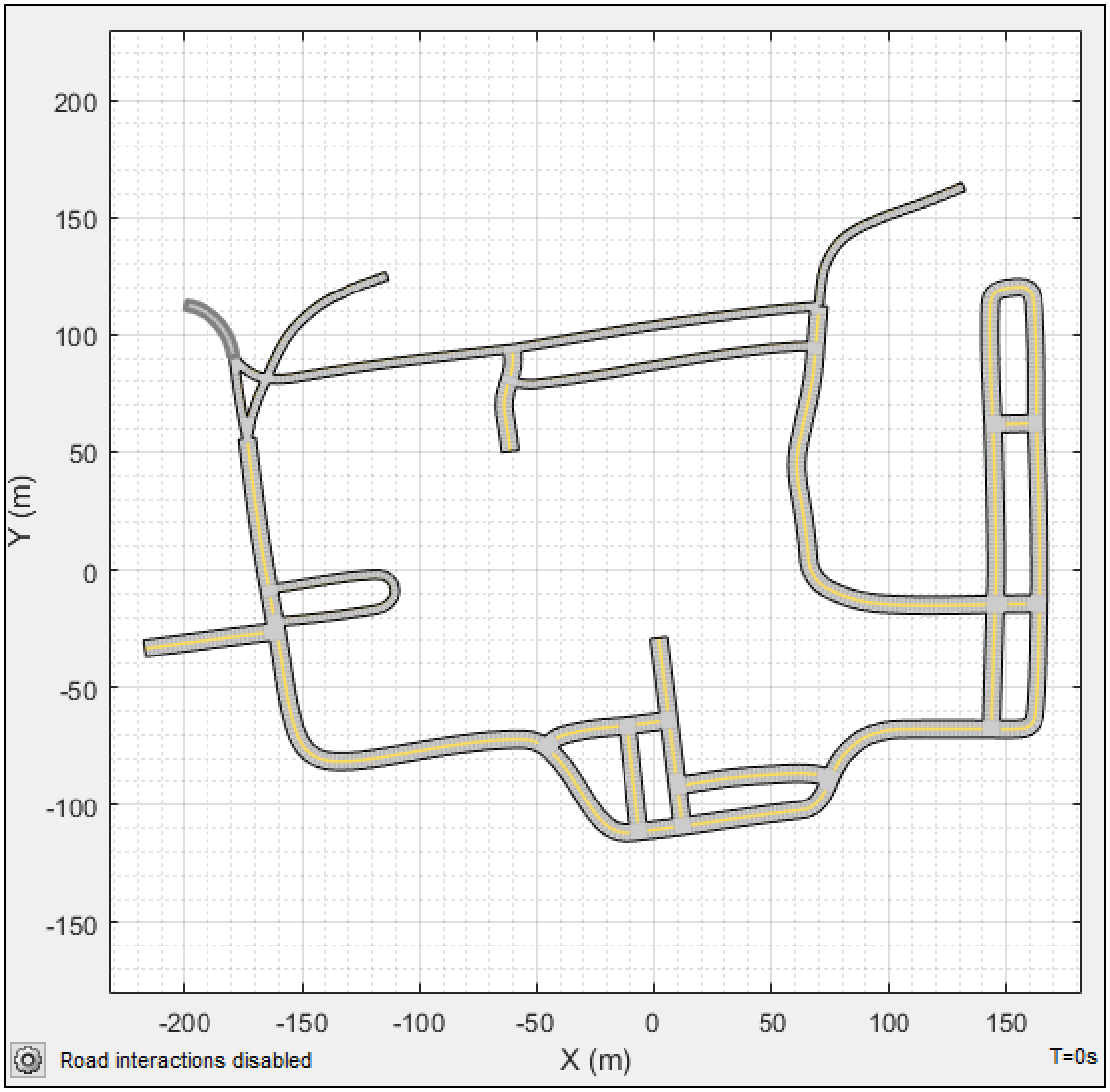


To test reverse motion algorithms, you can use the `Reverse_AEB` scenarios described in “Euro NCAP Driving Scenarios in Driving Scenario Designer”. To learn how to create your own reverse motion scenarios, see the “Create Reverse Motion Driving Scenarios Interactively” example.

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.

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.

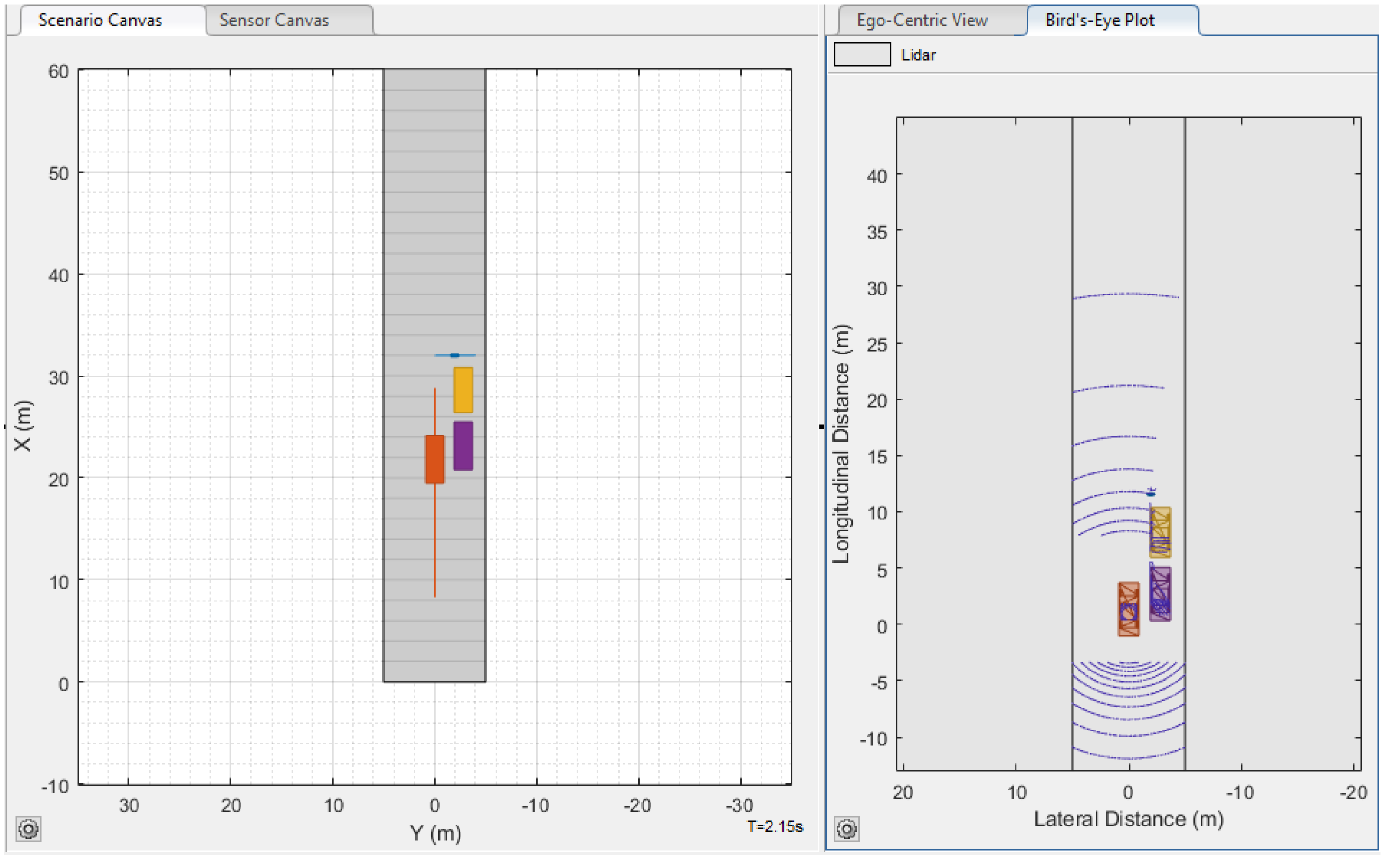
Both examples generate synthetic data from the Unreal Engine® simulation environment.

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

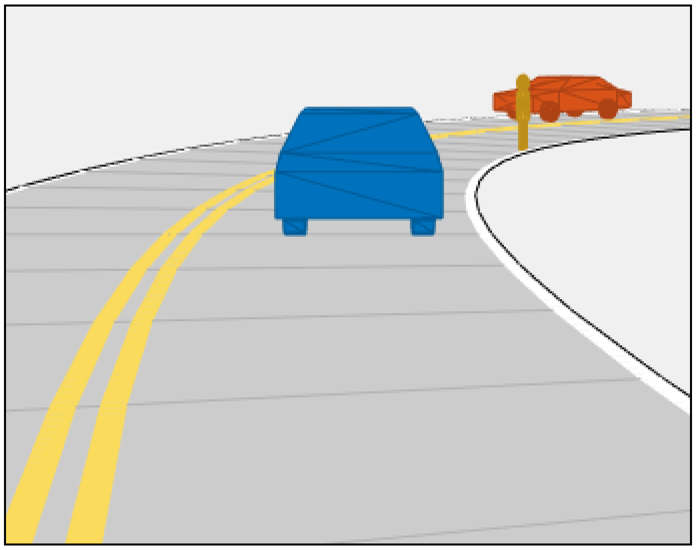
The Simulation 3D Vision Detection Generator block models a synthetic vision sensor and generates object and lane boundary detections from a simulation environment. This environment is rendered using the Unreal Engine from Epic Games®. The block includes parameters for modeling detection accuracy, measurement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

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).

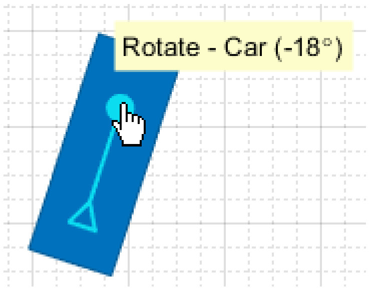
When you export a scenario containing a lidar sensor to Simulink®, the sensor in represented as a Lidar Point Cloud Generator block (introduced in R2020b).

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.

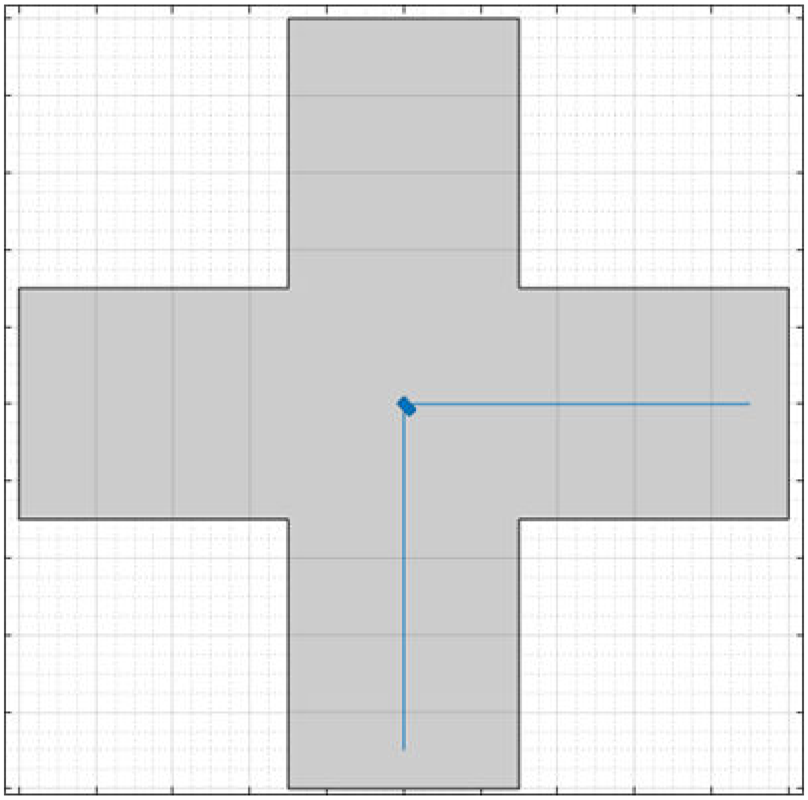
Interactive Actor Rotation

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.



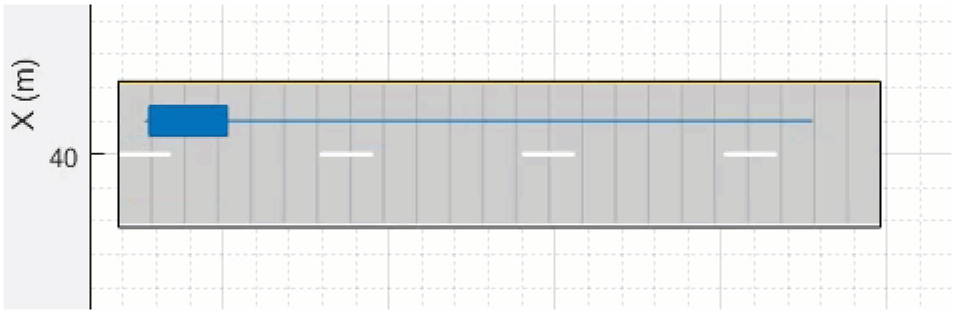
For sample scenarios with specified yaw constraints, see the `AEB_PedestrianTurning` scenarios described in “Euro NCAP Driving Scenarios in Driving Scenario Designer”.

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



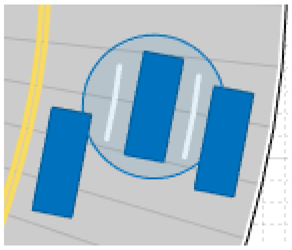
Mesh Plotter in Bird's-Eye Plot

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

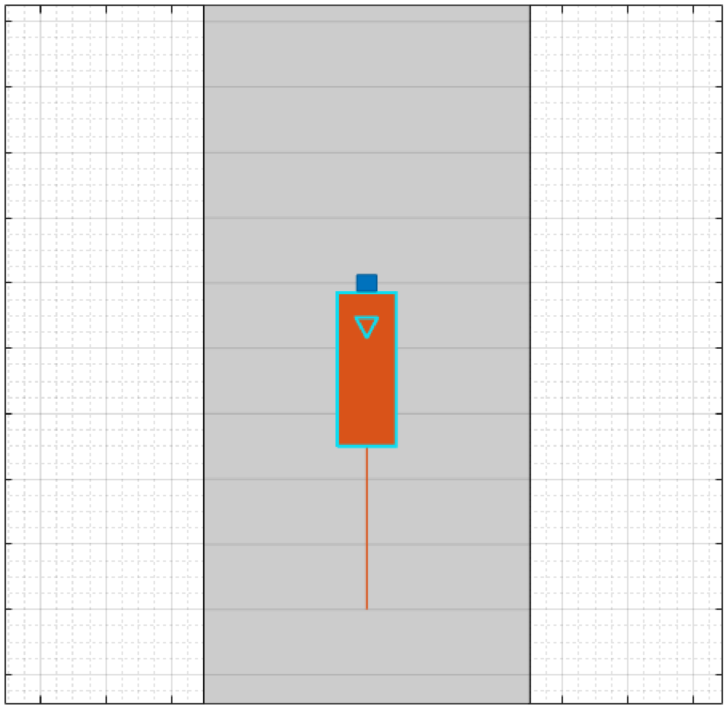
In the **Driving Scenario Designer** app, you can now add a visual indicator around the ego vehicle in a driving scenario. Use this option to identify the ego vehicle in simulations containing multiple actors.



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.

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.

Compatibility Considerations

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.

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

Enhancement	Image Labeler	Video Labeler	Ground Truth Labeler	Lidar Labeler
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
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

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

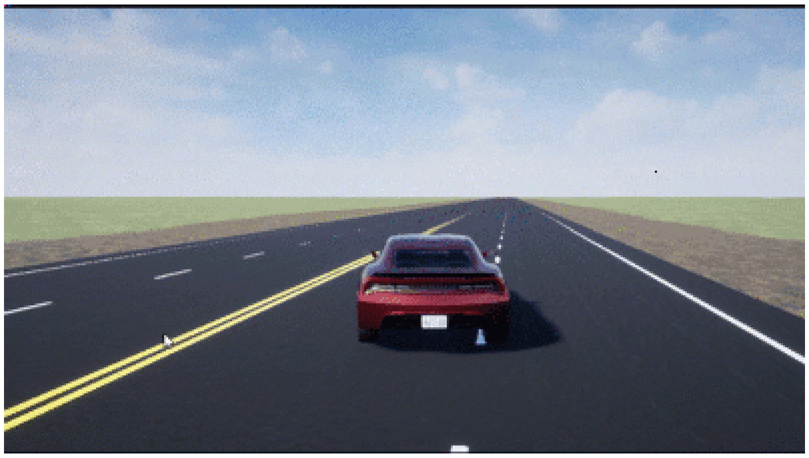
Press the **Tab** key to cycle the view between all vehicles in the scene.

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:

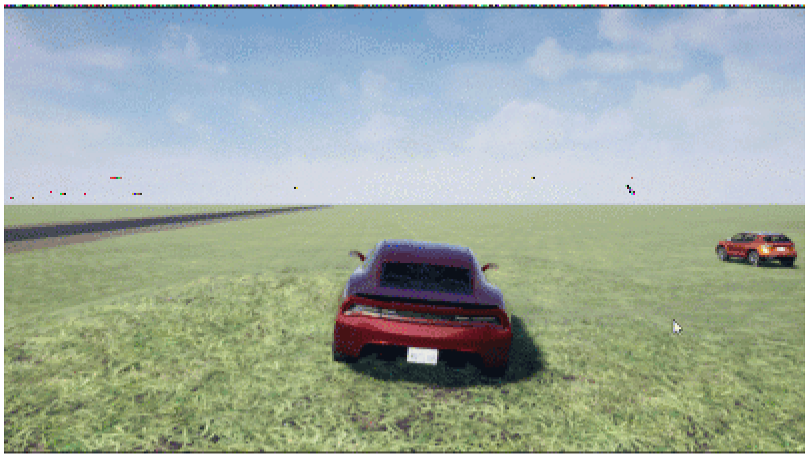
- Position lag, based on the translational acceleration of the vehicle
- Rotation lag, based on the rotational velocity of the vehicle

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



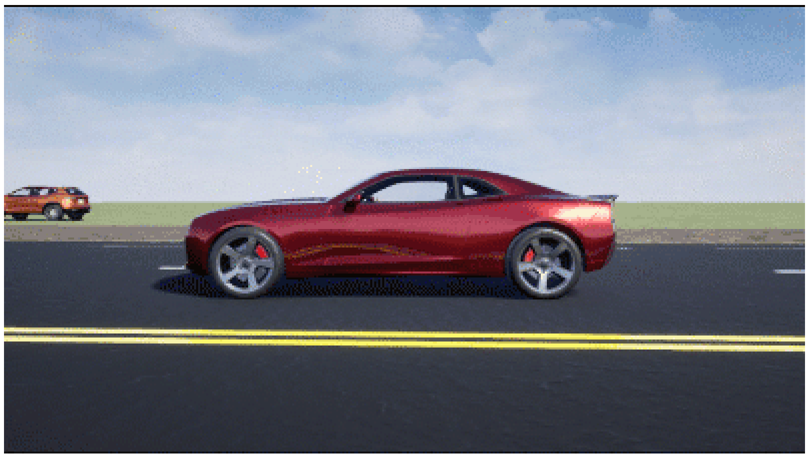
Vehicle Pitch and Roll

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



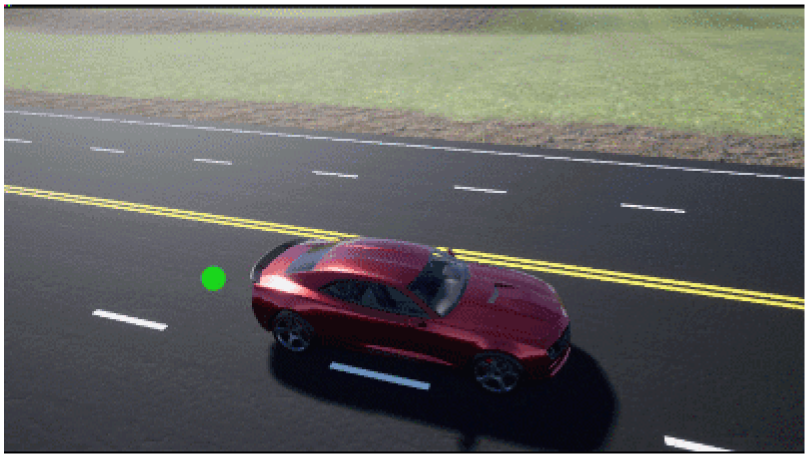
Camera Distance

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.



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.

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.

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.

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

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))
```

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.

Update Code

The table shows a typical usage of the `hereHDLMConfiguration(region)` syntax. It also shows how to update your code using the `hereHDLMConfiguration(catalog)` syntax.

Discouraged Usage	Recommended Replacement
`catalog = hereHDLMConfiguration('North America');`	`catalog = hereHDLMConfiguration('hrn:here:data::olp-h`

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);`
`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);`

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`.

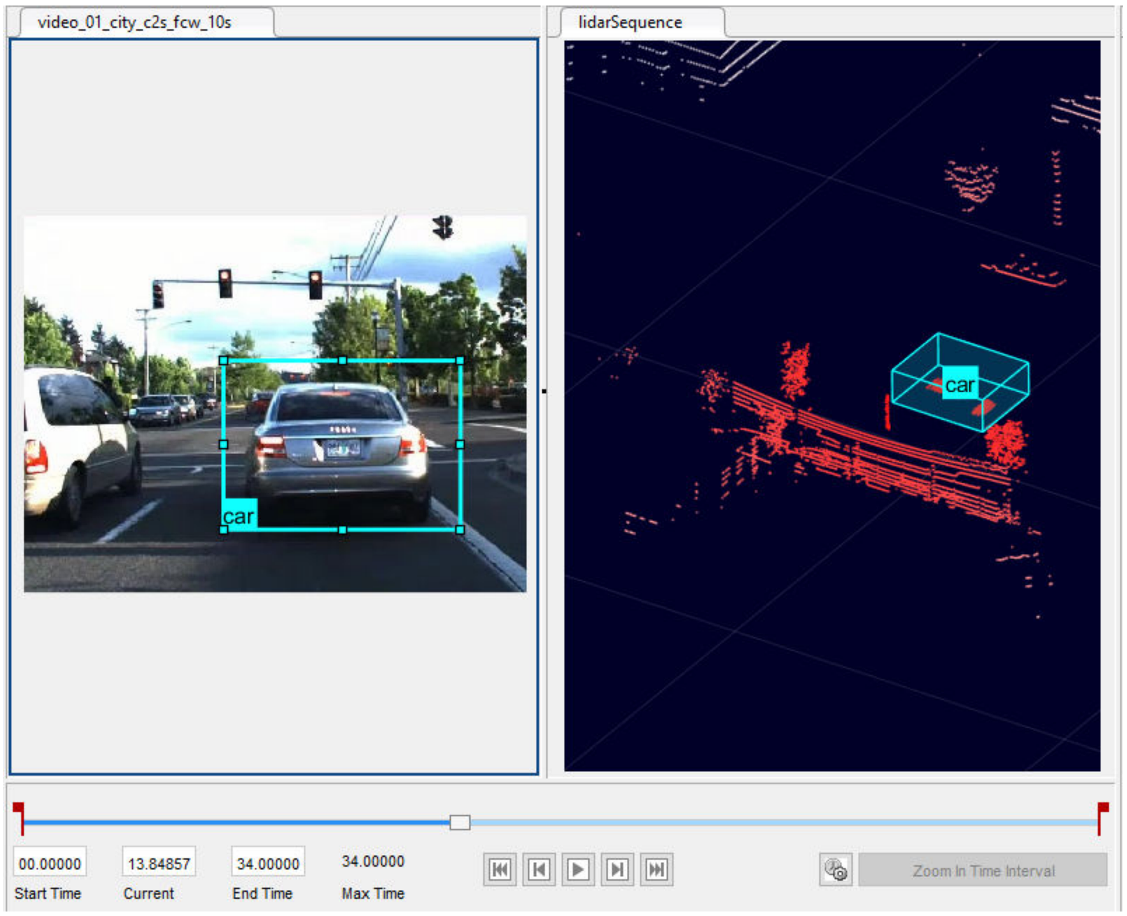
Discouraged Usage	Recommended Replacement

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

R2020a

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.

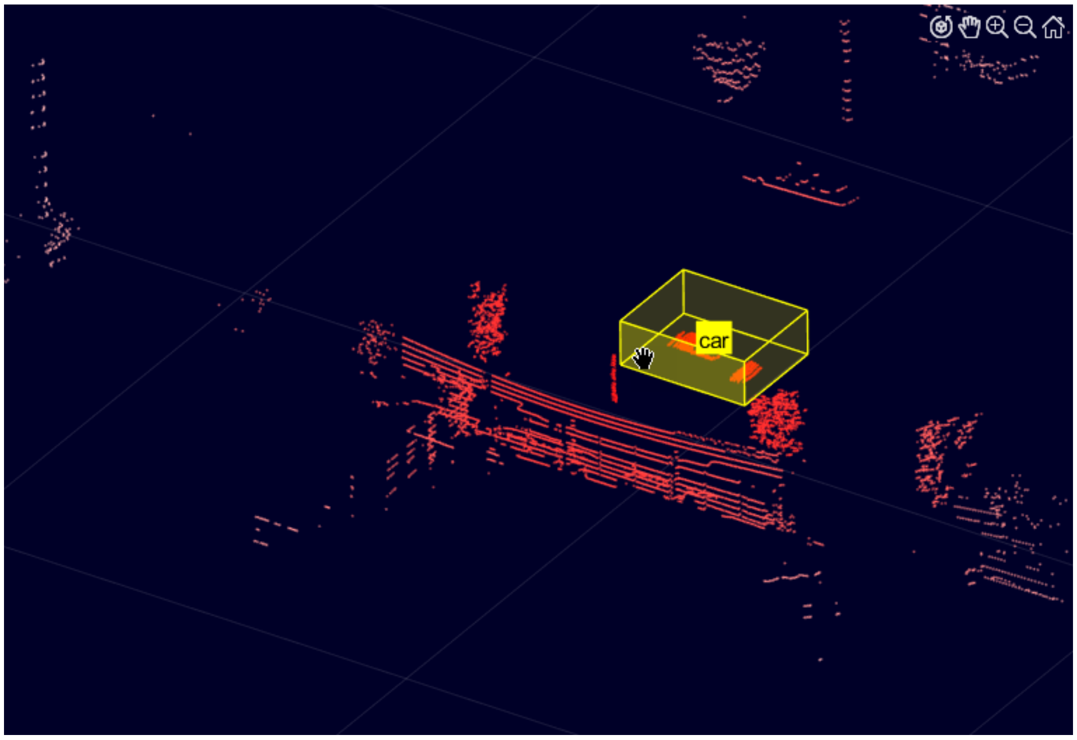
To get started labeling multiple signals, see Get Started with the Ground Truth Labeler.

Compatibility Considerations

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.

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.

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:

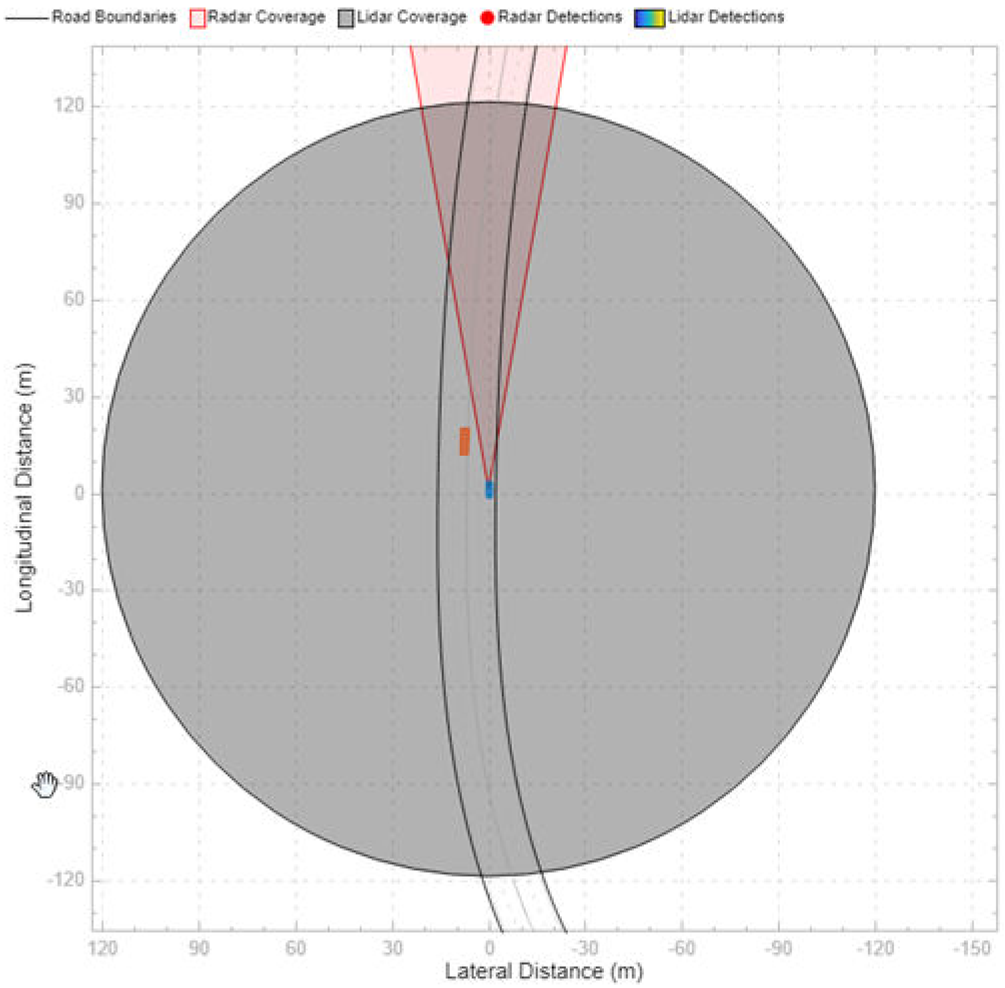
- `driving.scenario.carMesh`
- `driving.scenario.truckMesh`
- `driving.scenario.bicycleMesh`
- `driving.scenario.pedestrianMesh`

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

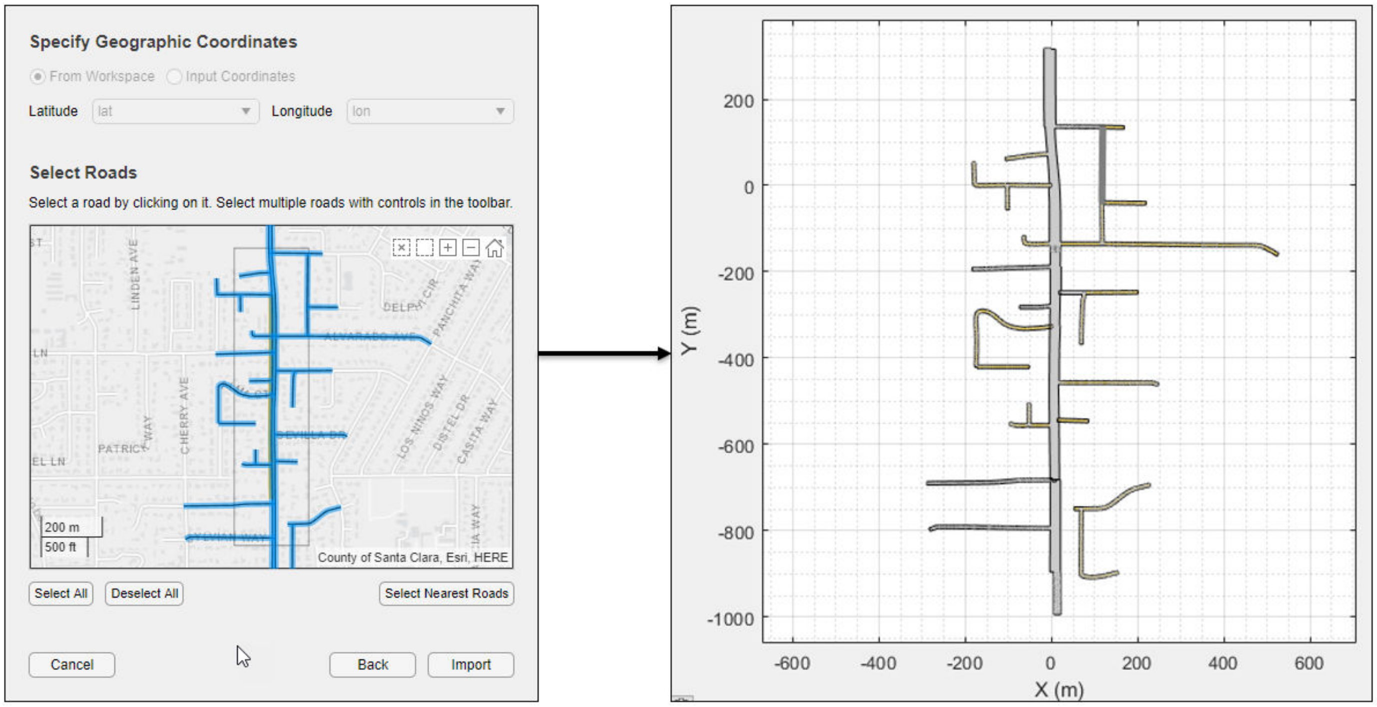
In the **Bird's-Eye Scope**, you can now visualize sensor data obtained from the 3D simulation environment, which is rendered using the Unreal Engine from Epic Games. You can visualize sensor coverage areas and detections from Simulation 3D Probabilistic Radar and Simulation 3D Lidar blocks. For more 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[1] web service, provided by HERE Technologies.


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.
County of Santa Clara, Esri, HERE
Select All
Deselect All
Select Nearest Roads
Cancel
Back
Import
Y (m)
X (m)

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.

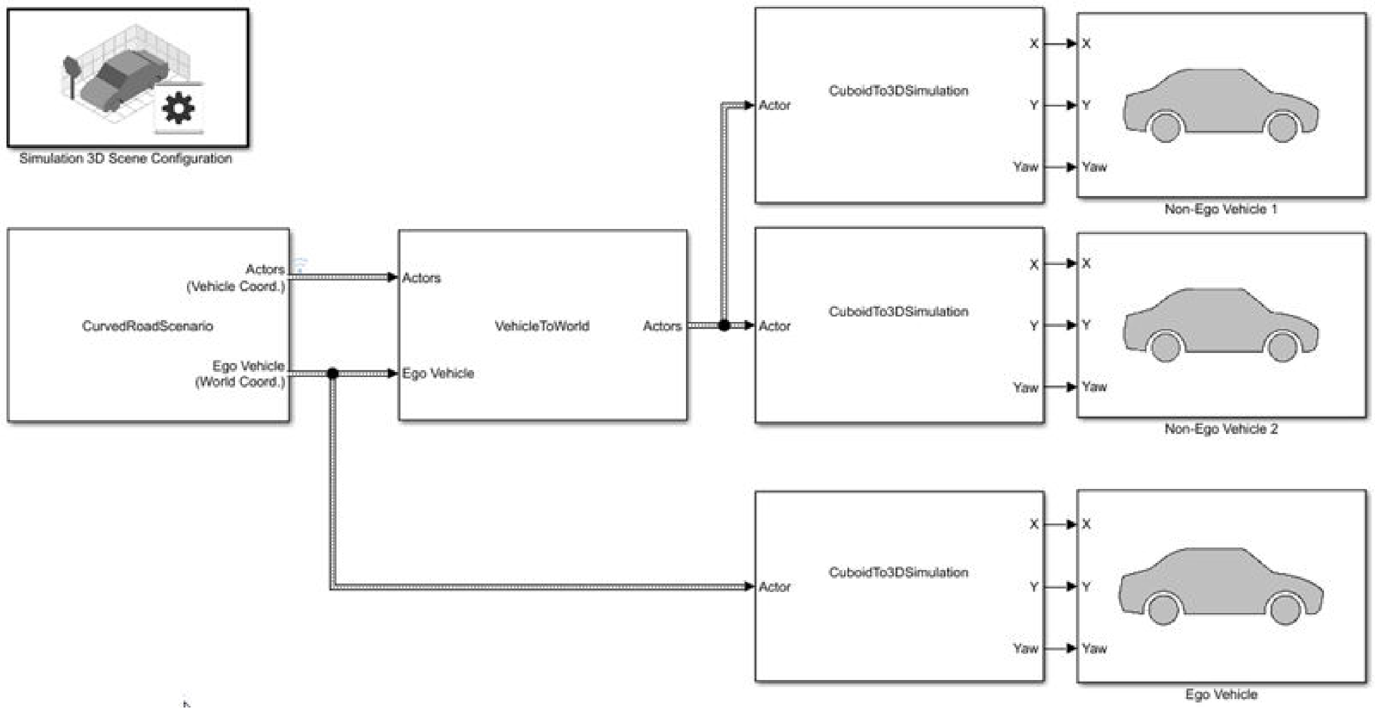
The Cuboid To 3D Simulation block converts vehicles authored in the cuboid environment into the coordinate system of the 3D simulation environment.

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. 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.

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.



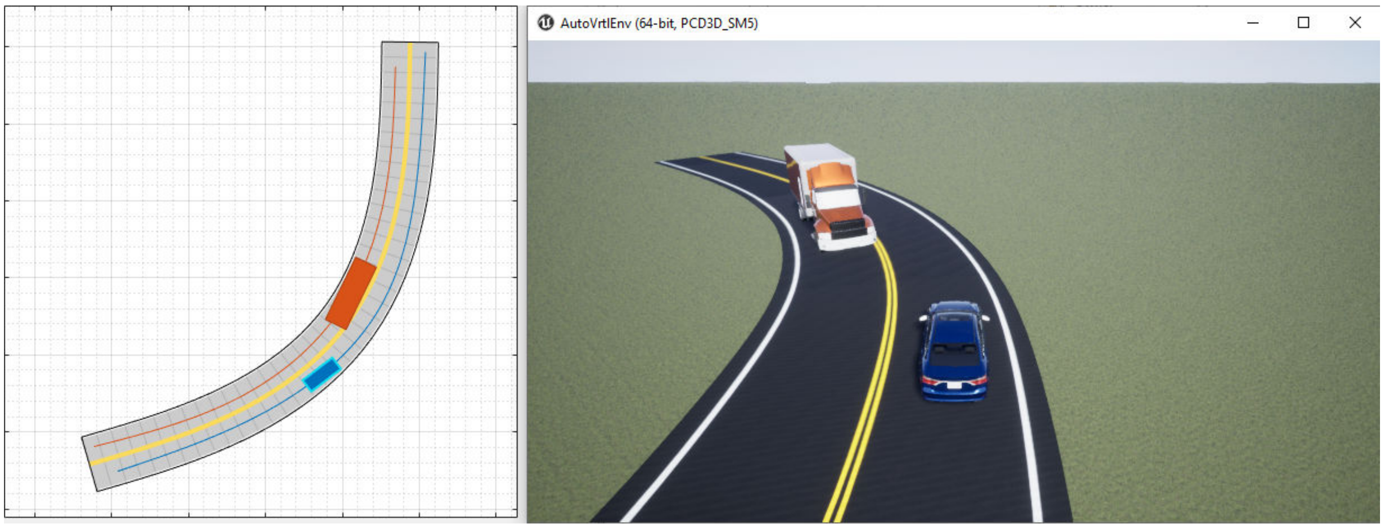
For an example that follows this workflow, see Visualize 3D Simulation Sensor Coverages and Detections.

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.

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.

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

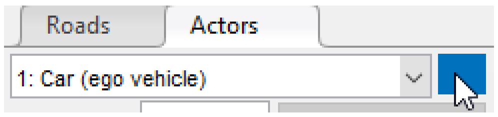
You can now import programmatically created radar and vision sensors into the **Driving Scenario Designer** app. The programmatic sensors must be created using `radarDetectionGenerator` and `visionDetectionGenerator` objects. You can also generate these programmatic sensors by using MATLAB code exported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

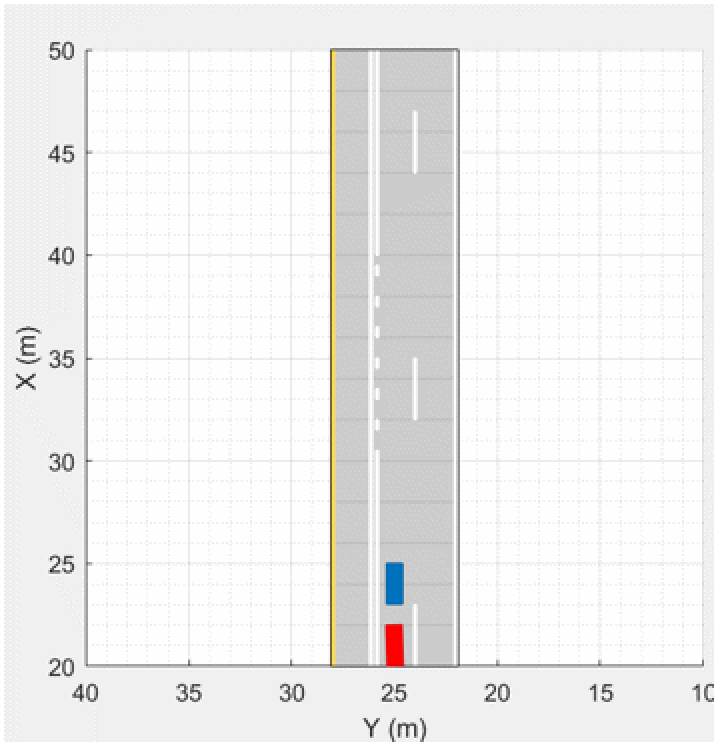
For an example that shows how to use these scenes with a 3D simulation Simulink model, see Visualize 3D Simulation Sensor Coverages and Detections.

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.

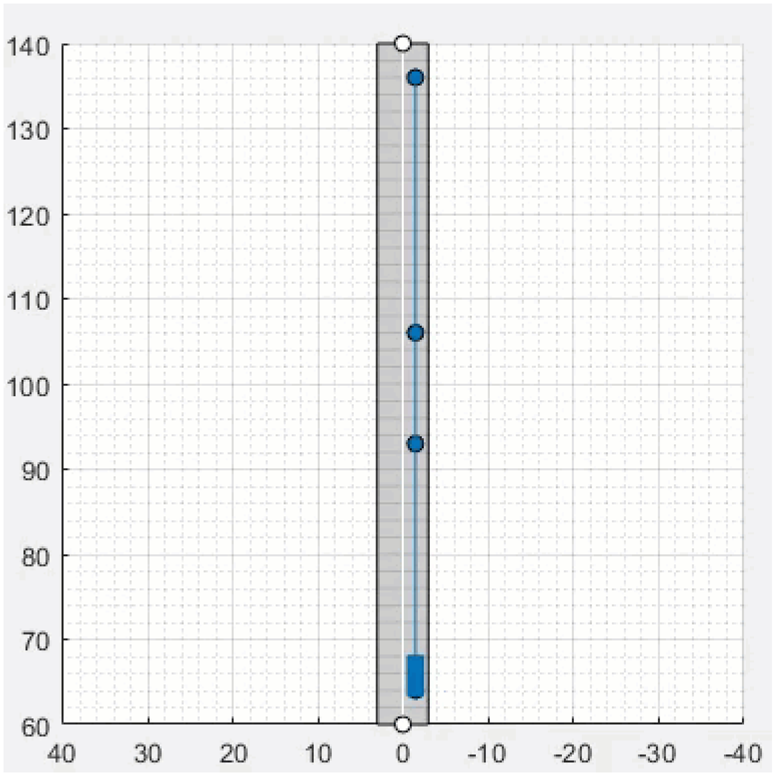
   ```
   ([laneMarking('Solid') laneMarking('Dashed')],'SegmentRange',[0.5 0.5]);
   ```
2. Pass the array as input to the `laneMarking` function. The function outputs a composite lane marking object that contains the properties of different markers along the lane.



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.

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.

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.

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.

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.

Compatibility Considerations

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” on page 2-16.

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” on page 2-17.

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.

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.

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.

Compatibility Considerations

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.

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

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

These objects and functions now support code generation.

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

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.

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.

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.

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.

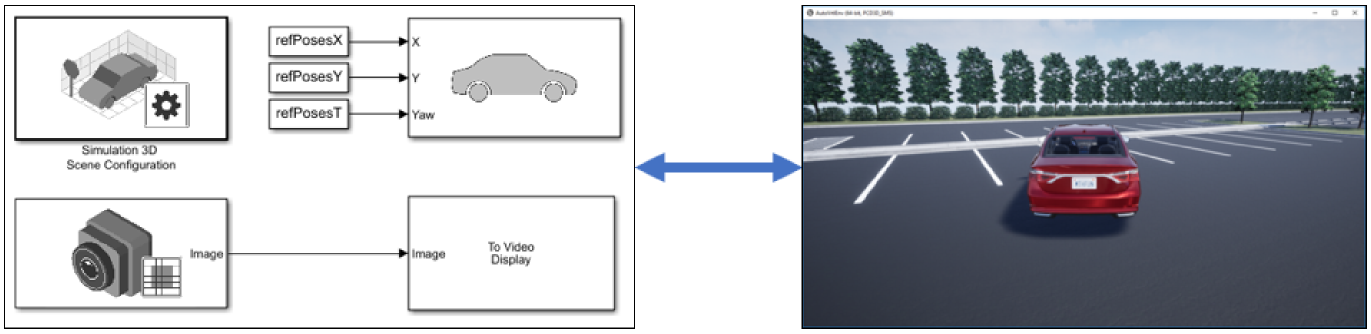
Renamed parameter in Simulation 3D Scene Configuration block
Behavior change

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.

R2019b

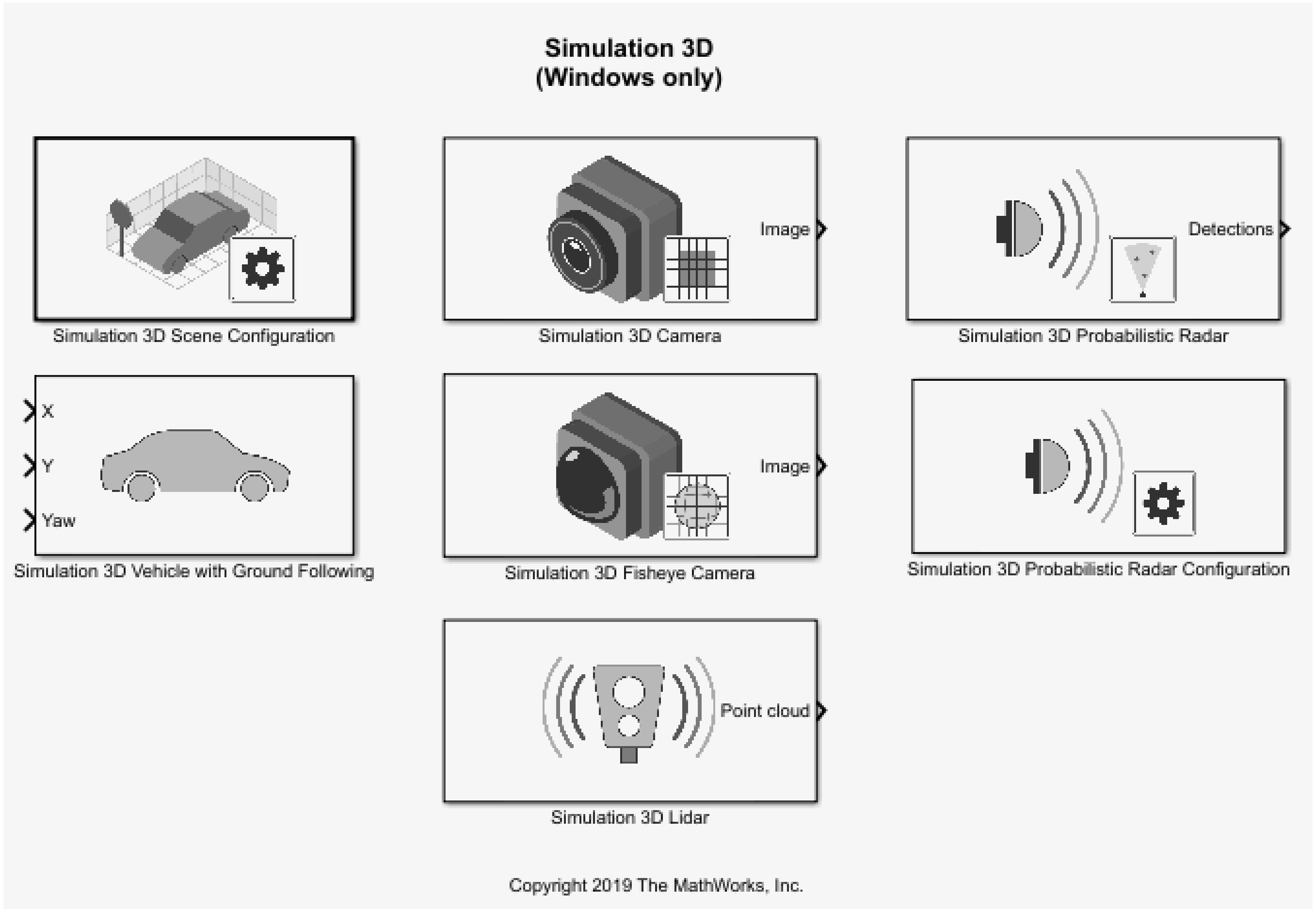
3D Simulation: Develop, test, and verify driving algorithms in a 3D simulation environment rendered using the Unreal Engine from Epic Games

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.

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.

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

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.

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.

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.

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®).

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`

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

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[2] (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.

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

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.

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.

- Move multiple selected ROIs in an image.
- Edit previously created label definitions.
- Add additional list items to a previously created attribute.

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.

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.

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.

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.

R2018b

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.

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” on page 5-4.

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.

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.

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.

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" on page 5-5.

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).

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.

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.

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);`	`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);`

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);`

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.

R2018a

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.

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.

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.

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.

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

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

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.

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.

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.

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.

Path method being removed

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

Compatibility Considerations

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.

Direction of Yaw Angle Rotation Adjusted

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

Compatibility Considerations

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

R2017b

Version: 1.1

New Features

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.

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

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.

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.

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

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

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`

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.

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