Automated Driving Toolbox™

User's Guide

MATLAB®&SIMULINK®

R2020a

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™ User's Guide

Trademarks

Patents

Revision History

March 2017	Online only	New for Version 1.0 (Release 2017a)
September 2017	Online only	Revised for Version 1.1 (Release 2017b)
March 2018	Online only	Revised for Version 1.2 (Release 2018a)
September 2018	Online only	Revised for Version 1.3 (Release 2018b)
March 2019	Online only	Revised for Version 2.0 (Release 2019a)
September 2019	Online only	Revised for Version 3.0 (Release 2019b)
March 2020	Online only	Revised for Version 3.1 (Release 2020a)

Contents

1 Sensor Configuration and Coordinate System Transformations

2 Ground Truth Labeling and Verification

3 Tracking and Sensor Fusion

4 Planning, Mapping, and Control

5 Cuboid Driving Scenario Simulation

6 3D Simulation – User's Guide

1

Sensor Configuration and Coordinate System Transformations

Coordinate Systems in Automated Driving Toolbox

Automated Driving Toolbox uses these coordinate systems:

- **World**: A fixed universal coordinate system in which all vehicles and their sensors are placed.
- **Vehicle**: Anchored to the ego vehicle. Typically, the vehicle coordinate system is placed on the ground right below the midpoint of the rear axle.
- **Sensor**: Specific to a particular sensor, such as a camera or a radar.
- **Spatial**: Specific to an image captured by a camera. Locations in spatial coordinates are expressed in units of pixels.
- **Pattern**: A checkerboard pattern coordinate system, typically used to calibrate camera sensors.

These coordinate systems apply across Automated Driving Toolbox functionality, from perception to control to driving scenario simulation. For information on specific differences and implementation details in the 3D simulation environment using the Unreal Engine® from Epic Games®, see “Coordinate Systems for 3D Simulation in Automated Driving Toolbox” on page 6-10.

World Coordinate System

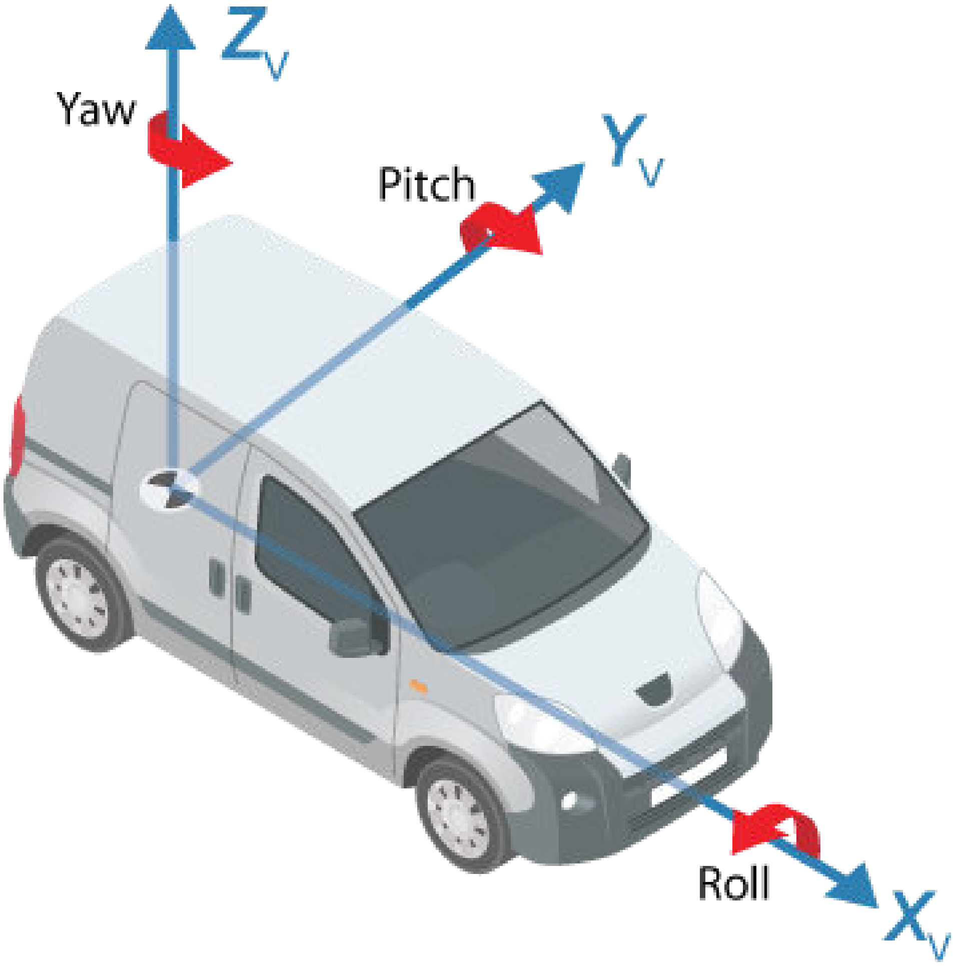
All vehicles, sensors, and their related coordinate systems are placed in the world coordinate system. A world coordinate system is important in global path planning, localization, mapping, and driving scenario simulation. Automated Driving Toolbox uses the right-handed Cartesian world coordinate system defined in ISO 8855, where the *Z*-axis points up from the ground. Units are in meters.

Vehicle Coordinate System

The vehicle coordinate system (X_V, Y_V, Z_V) used by Automated Driving Toolbox is anchored to the ego vehicle. The term ego vehicle refers to the vehicle that contains the sensors that perceive the environment around the vehicle.

- The X_V axis points forward from the vehicle.
- The Y_V axis points to the left, as viewed when facing forward.
- The Z_V axis points up from the ground to maintain the right-handed coordinate system.

The vehicle coordinate system follows the ISO 8855 convention for rotation. Each axis is positive in the clockwise direction, when looking in the positive direction of that axis.

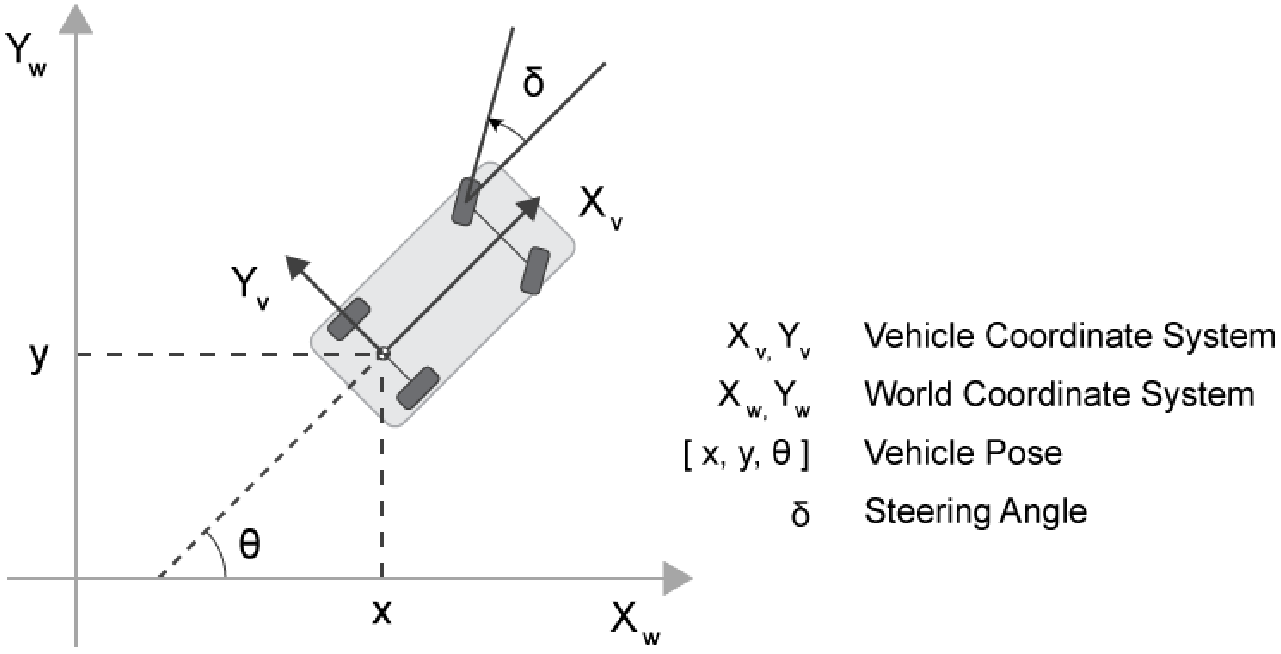


In most Automated Driving Toolbox functionality, such as cuboid driving scenario simulations and visual perception algorithms, the origin of the vehicle coordinate system is on the ground, below the midpoint of the rear axle. In 3D driving scenario simulations, the origin is on ground, below the longitudinal and lateral center of the vehicle. For more details, see “Coordinate Systems for 3D Simulation in Automated Driving Toolbox” on page 6-10.

Locations in the vehicle coordinate system are expressed in world units, typically meters.

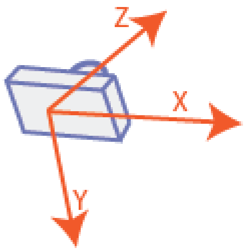
Values returned by individual sensors are transformed into the vehicle coordinate system so that they can be placed in a unified frame of reference.

For global path planning, localization, mapping, and driving scenario simulation, the state of the vehicle can be described using the pose of the vehicle. The steering angle of the vehicle is positive in the counterclockwise direction.

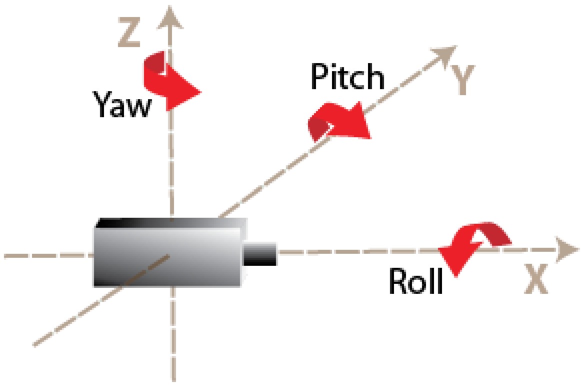

Y_w
δ
X_v
Y_v
y
θ
x
X_w
X_v, Y_v Vehicle Coordinate System
X_w, Y_w World Coordinate System
[x, y, θ] Vehicle Pose
δ Steering Angle

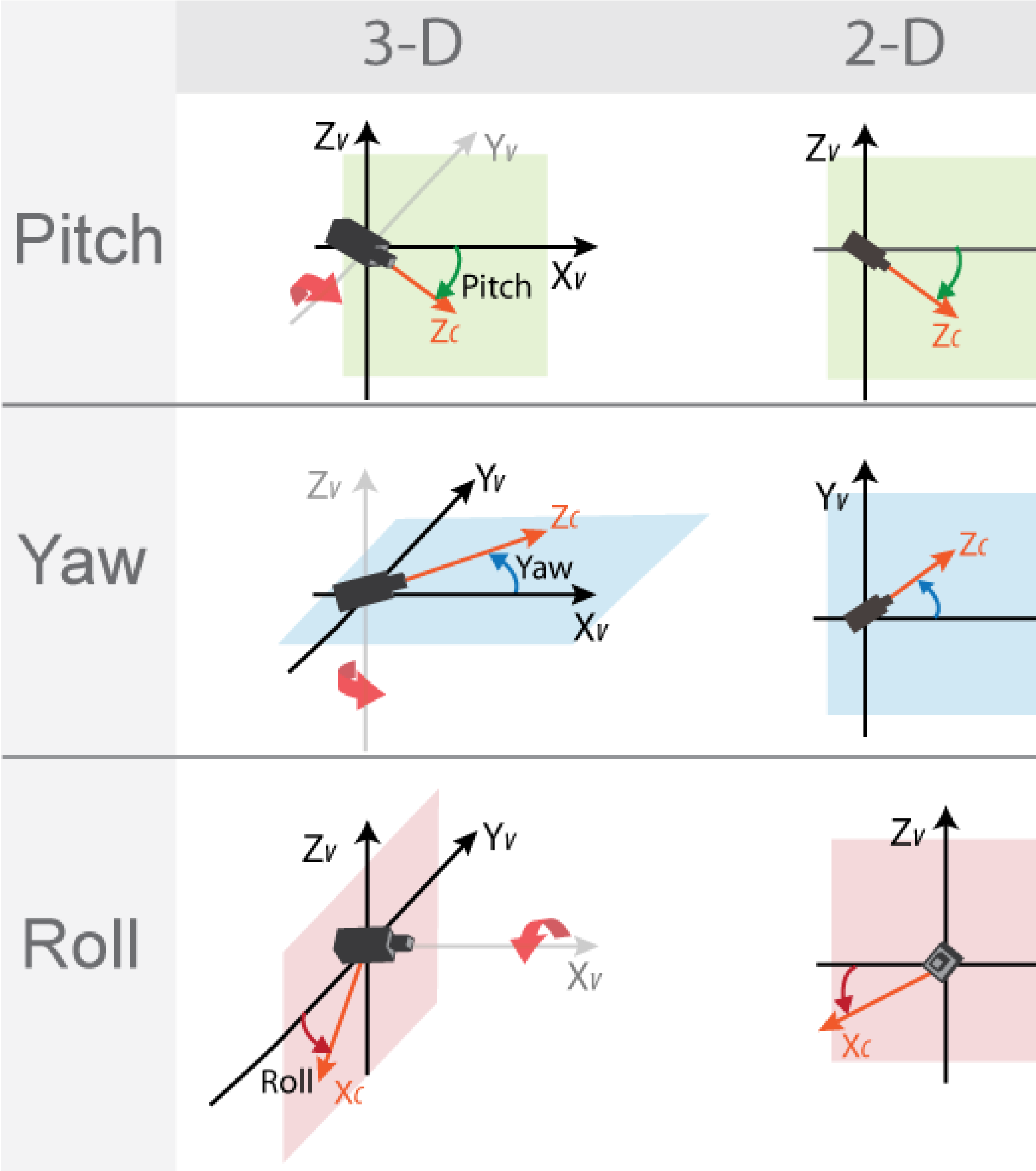
Sensor Coordinate System

An automated driving system can contain sensors located anywhere on or in the vehicle. The location of each sensor contains an origin of its coordinate system. A camera is one type of sensor used often in an automated driving system. Points represented in a camera coordinate system are described with the origin located at the optical center of the camera.



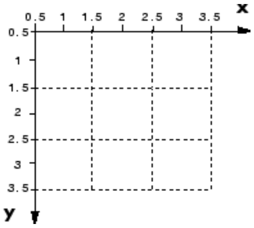
The yaw, pitch, and roll angles of sensors follow an ISO convention. These angles have positive clockwise directions when looking in the positive direction of the *Z*-, *Y*-, and *X*-axes, respectively.


Z
Yaw
Pitch
Y
Roll
X


3-D
2-D
Pitch
Zv
Yv
Pitch
Xv
Zc
Zv
Zc
Yaw
Zv
Yv
Zc
Yaw
Xv
Yv
Zc
Roll
Zv
Yv
Xv
Roll
Xc
Zv
Xc

Spatial Coordinate System

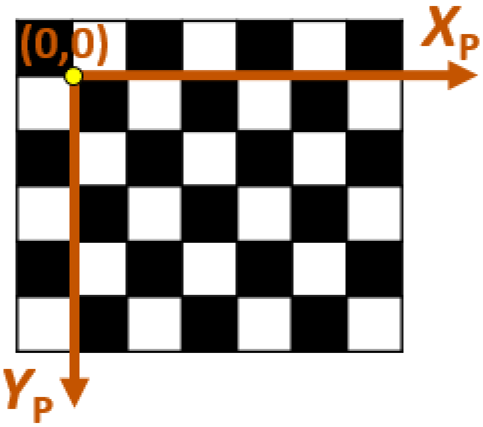
Spatial coordinates enable you to specify a location in an image with greater granularity than pixel coordinates. In the pixel coordinate system, a pixel is treated as a discrete unit, uniquely identified by an integer row and column pair, such as `(3,4)`. In the spatial coordinate system, locations in an image are represented in terms of partial pixels, such as `(3.3,4.7)`.



For more information on the spatial coordinate system, see “Spatial Coordinates” (Image Processing Toolbox).

Pattern Coordinate System

To estimate the parameters of a monocular camera sensor, a common technique is to calibrate the camera using multiple images of a calibration pattern, such as a checkerboard. In the pattern coordinate system, (X_P, Y_P), the X_P-axis points to the right and the Y_P-axis points down. The checkerboard origin is the bottom-right corner of the top-left square of the checkerboard.



Each checkerboard corner represents another point in the coordinate system. For example, the corner to the right of the origin is (1,0) and the corner below the origin is (0,1). For more information on calibrating a camera by using a checkerboard pattern, see “Calibrate a Monocular Camera” on page 1-9.

See Also

More About

- “Coordinate Systems for 3D Simulation in Automated Driving Toolbox” on page 6-10
- “Coordinate Systems in Vehicle Dynamics Blockset” (Vehicle Dynamics Blockset)

- “Coordinate Systems” (Computer Vision Toolbox)
- “Image Coordinate Systems” (Image Processing Toolbox)
- “Calibrate a Monocular Camera” on page 1-9

Calibrate a Monocular Camera

A monocular camera is a common type of vision sensor used in automated driving applications. When mounted on an ego vehicle, this camera can detect objects, detect lane boundaries, and track objects through a scene.

Before you can use the camera, you must calibrate it. Camera calibration is the process of estimating the intrinsic and extrinsic parameters of a camera using images of a calibration pattern, such as a checkerboard. After you estimate the intrinsic and extrinsic parameters, you can use them to configure a model of a monocular camera.

Estimate Intrinsic Parameters

The intrinsic parameters of a camera are the properties of the camera, such as its focal length and optical center. To estimate these parameters for a monocular camera, use Computer Vision Toolbox™ functions and images of a checkerboard pattern.

- If the camera has a standard lens, use the `estimateCameraParameters` function.
- If the camera has a fisheye lens, use the `estimateFisheyeParameters` function.

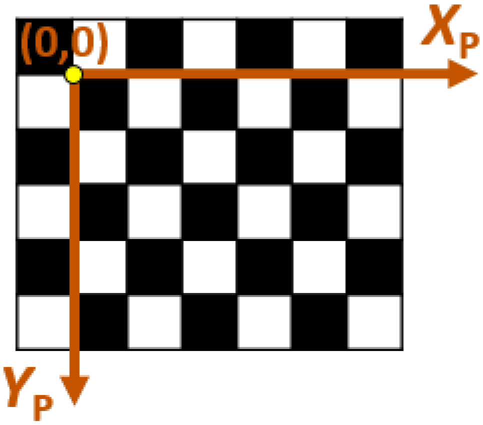
Alternatively, to better visualize the results, use the **Camera Calibrator** app. For information on setting up the camera, preparing the checkerboard pattern, and calibration techniques, see “Single Camera Calibrator App” (Computer Vision Toolbox).

Place Checkerboard for Extrinsic Parameter Estimation

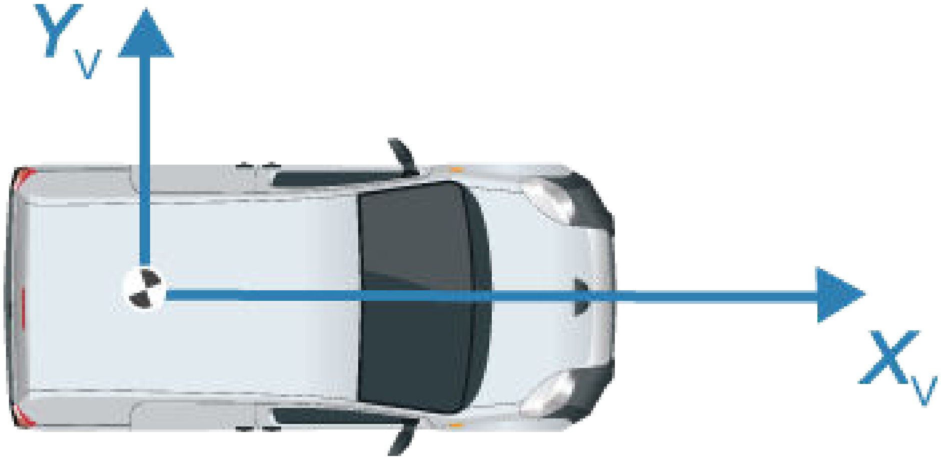
For a monocular camera mounted on a vehicle, the *extrinsic parameters* define the mounting position of that camera. These parameters include the rotation angles of the camera with respect to the vehicle coordinate system, and the height of the camera above the ground.

Before you can estimate the extrinsic parameters, you must capture an image of a checkerboard pattern from the camera. Use the same checkerboard pattern that you used to estimate the intrinsic parameters.

The checkerboard uses a pattern-centric coordinate system (X_P, Y_P), where the X_P-axis points to the right and the Y_P-axis points down. The checkerboard origin is the bottom-right corner of the top-left square of the checkerboard.



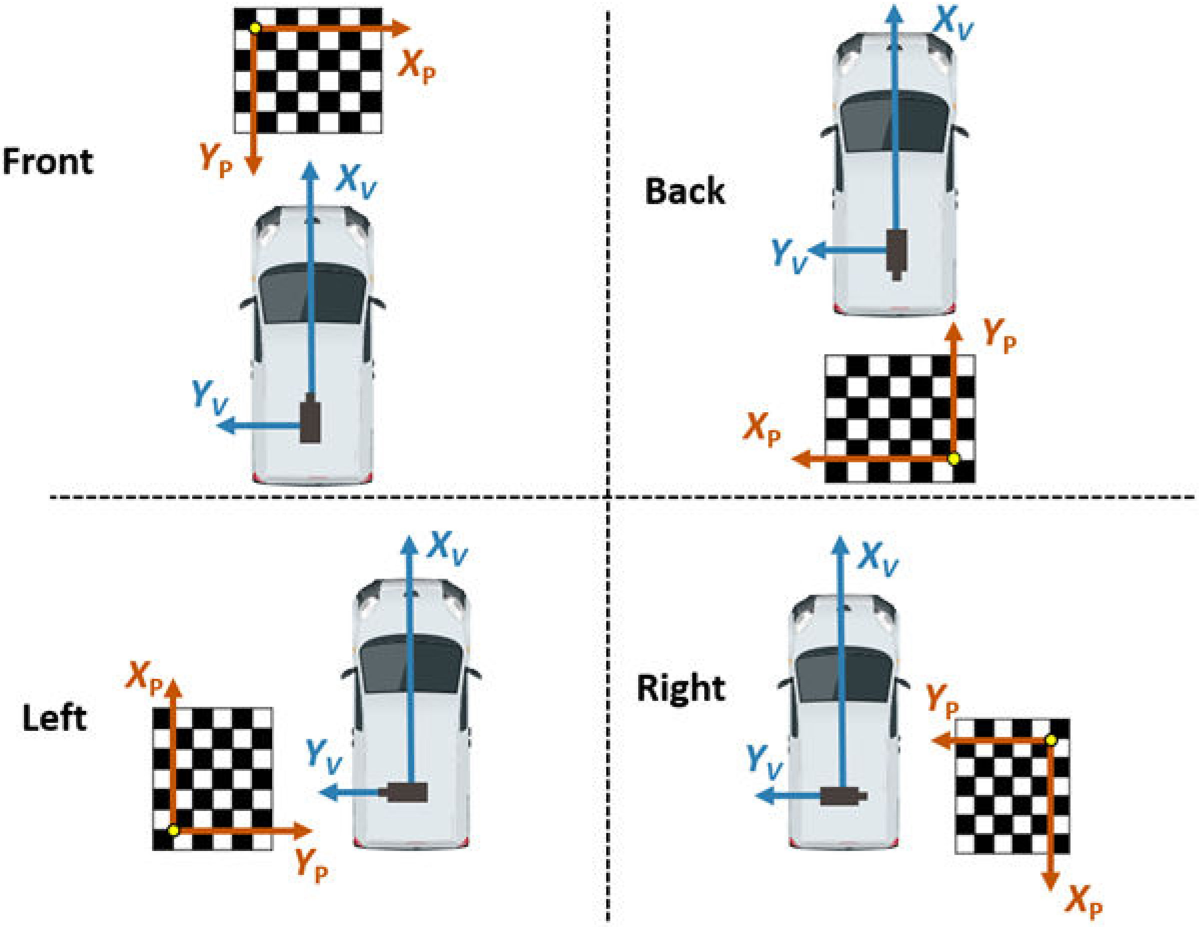
When placing the checkerboard pattern in relation to the vehicle, the X_P- and Y_P-axes must align with the X_V- and Y_V-axes of the vehicle. In the vehicle coordinate system, the X_V-axis points forward from the vehicle and the Y_V-axis points to the left, as viewed when facing forward. The origin is on the road surface, directly below the camera center (the focal point of the camera).



The orientation of the pattern can be either horizontal or vertical.

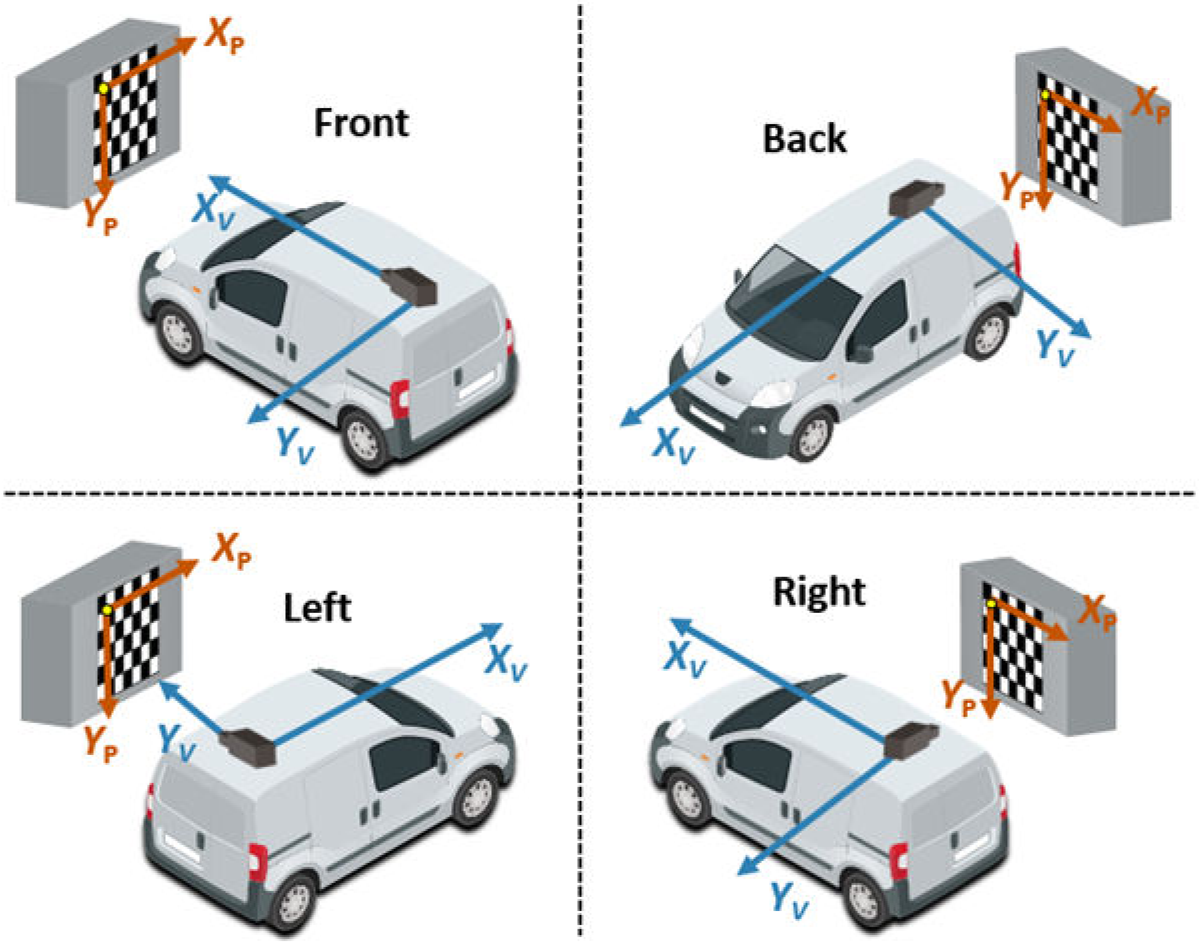
Horizontal Orientation

In the horizontal orientation, the checkerboard pattern is either on the ground or parallel to the ground. You can place the pattern in front of the vehicle, in back of the vehicle, or on the left or right side of the vehicle.



Vertical Orientation

In the vertical orientation, the checkerboard pattern is perpendicular to the ground. You can place the pattern in front of the vehicle, in back of the vehicle, or on the left of right side of the vehicle.



Estimate Extrinsic Parameters

After placing the checkerboard in the location you want, capture an image of it using the monocular camera. Then, use the `estimateMonoCameraParameters` function to estimate the extrinsic parameters. To use this function, you must specify the following:

- The intrinsic parameters of the camera
- The key points detected in the image, in this case the corners of the checkerboard squares
- The world points of the checkerboard
- The height of the checkerboard pattern's origin above the ground

For example, for image `I` and intrinsic parameters `intrinsics`, the following code estimates the extrinsic parameters. By default, `estimateMonoCameraParameters` assumes that the camera is facing forward and that the checkerboard pattern has a horizontal orientation.

```
[imagePoints,boardSize] = detectCheckerboardPoints(I);
squareSize = 0.029; % Square size in meters
worldPoints = generateCheckerboardPoints(boardSize,squareSize);
patternOriginHeight = 0; % Pattern is on ground
[pitch,yaw,roll,height] = estimateMonoCameraParameters(intrinsics, ...
                              imagePoints,worldPoints,patternOriginHeight);
```

To increase estimation accuracy of these parameters, capture multiple images and average the values of the image points.

Configure Camera Using Intrinsic and Extrinsic Parameters

Once you have the estimated intrinsic and extrinsic parameters, you can use the `monoCamera` object to configure a model of the camera. The following sample code shows how to configure the camera using parameters `intrinsics`, `height`, `pitch`, `yaw`, and `roll`:

```
monoCam = monoCamera(intrinsics,height,'Pitch',pitch,'Yaw',yaw,'Roll',roll);
```

See Also

Apps
Camera Calibrator

Functions
`detectCheckerboardPoints` | `estimateCameraParameters` | `estimateFisheyeParameters` | `estimateMonoCameraParameters` | `generateCheckerboardPoints`

Objects
`monoCamera`

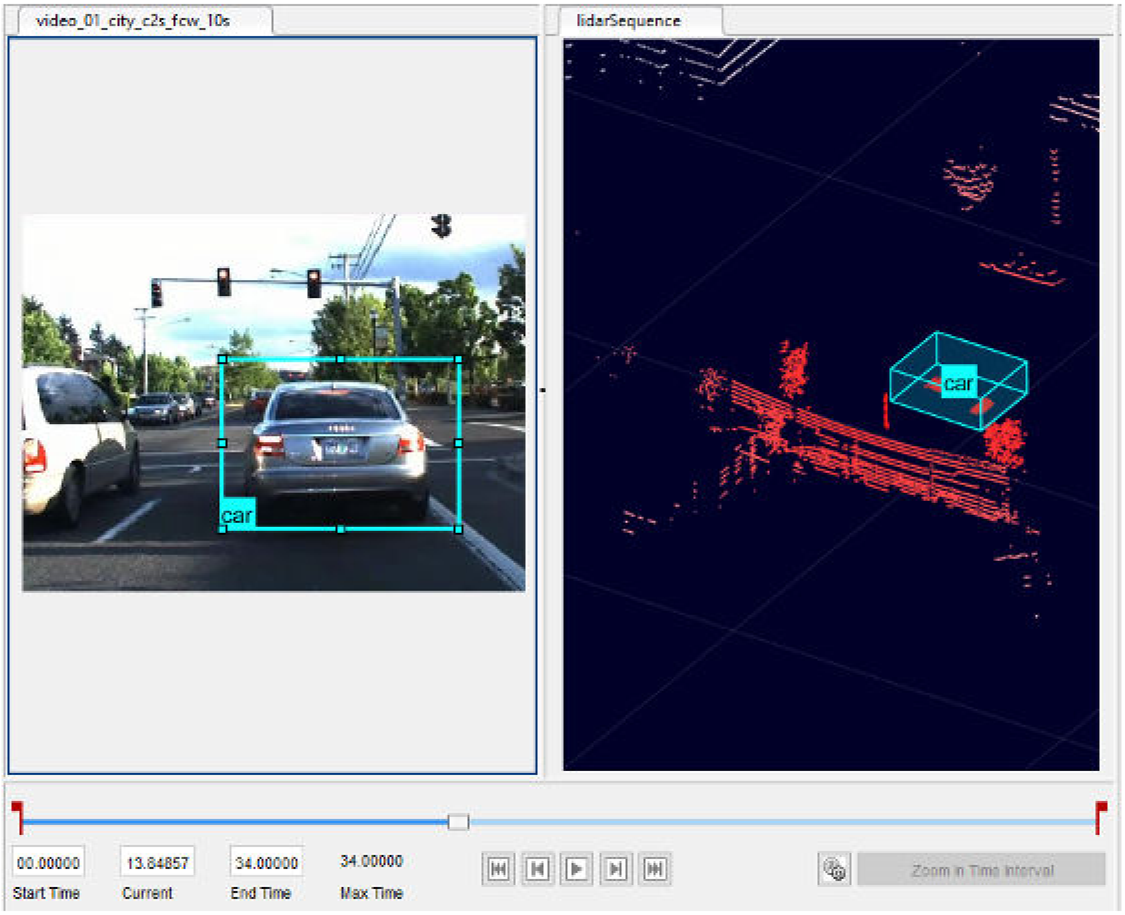
More About

- “Coordinate Systems in Automated Driving Toolbox” on page 1-2
- “Configure Monocular Fisheye Camera”
- “Single Camera Calibrator App” (Computer Vision Toolbox)
- “Fisheye Calibration Basics” (Computer Vision Toolbox)

Ground Truth Labeling and Verification

Get Started with the Ground Truth Labeler

The **Ground Truth Labeler** app enables you to interactively label ground truth data in a video, image sequence, or lidar point cloud. Using the app, you can simultaneously label multiple signals, such as data obtained from camera and lidar sensors mounted on a vehicle.



00.00000	13.84857	34.00000	34.00000
Start Time	Current	End Time	Max Time

This example walks you through the multisignal ground truth labeling workflow in these steps.

1. “Load Ground Truth Signals to Label” on page 2-4 — Load multiple signals into the app and configure the display of those signals.
2. “Label Ground Truth for Multiple Signals” on page 2-9 — Create label definitions and label the signals by using automation algorithms.
3. “Export Ground Truth Labels for Multiple Signals” on page 2-21 — Export the labels from the app and explore the data.

You can use these exported labels, along with the associated signal frames, as training data for deep learning applications.

See Also

More About

- “Choose an App to Label Ground Truth Data” (Computer Vision Toolbox)

Load Ground Truth Signals to Label

The **Ground Truth Labeler** app provides options for labeling two types of signals.

- Image signals are image-based. You can load these signals from sources such as videos or image sequences.
- Point cloud signals are lidar-based. You can load these signals from sources such as a sequence of point cloud files.

In this example, you load a video and a point cloud sequence into the app. These signals are taken from a camera sensor and a lidar sensor mounted to a vehicle. The signals represent the same driving scene.

Load Timestamps

Load the timestamps for the point cloud sequence. The timestamps are a `duration` vector that is in the same folder as the sequence. To load the timestamps, you must temporarily add this folder to the MATLAB® search path.

```
pcSeqFolder = fullfile(toolboxdir('driving'),'drivingdata','lidarSequence');
addpath(pcSeqFolder)
load timestamps.mat
rmpath(pcSeqFolder)
```

The app also provides an option to specify timestamps for video sources. The video used in this example does not have a separate timestamps file, so when you load the video, you can read the timestamps directly from the video source.

Open Ground Truth Labeler App

To open the **Ground Truth Labeler** app, at the MATLAB command prompt, enter this command.

```
groundTruthLabeler
```

The app opens to an empty session.

Alternatively, you can open the app from the **Apps** tab, under **Automotive**.

Load Signals from Data Sources

The **Ground Truth Labeler** app enables you to load signals from multiple types of data sources. In the app, a data source is a file or folder containing one or more signals to label.

- For the video, the data source is an MP4 file that contains a single video.
- For the point cloud sequence, the data source is a folder containing a sequence of point cloud data (PCD) files. Together, these files represent a single point cloud sequence.

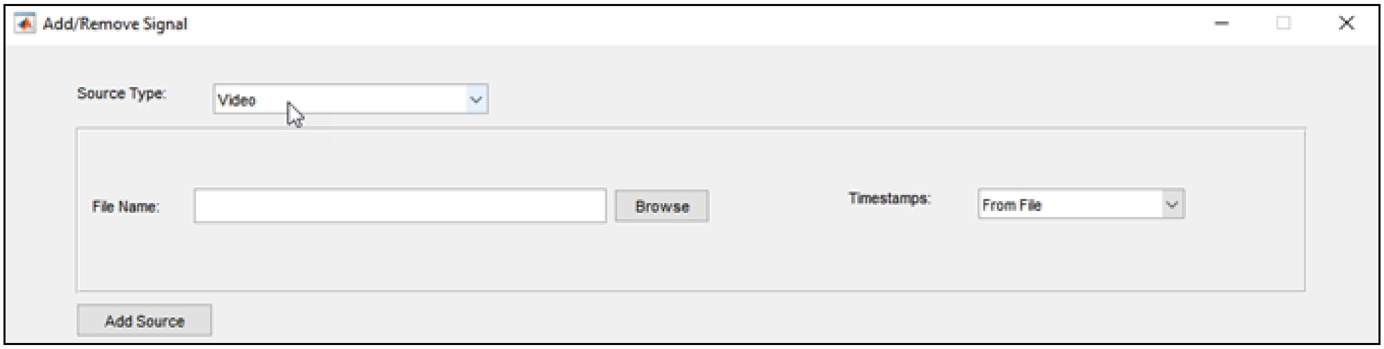
Other data sources, such as rosbags, can contain multiple signals that you can load. For more details on the relationship between sources and signals, see “Sources vs. Signals in Ground Truth Labeling” on page 2-28.

Load Video

Load the video into the app.

1 On the app toolstrip, click **Open** > **Add Signals**.

The Add/Remove Signal dialog box opens with the **Source Type** parameter set to `Video` and the **Timestamps** parameter set to `From File`.



Add/Remove Signal

Source Type: Video

File Name:

Timestamps: From File

2 In the **File Name** parameter, browse for this video file. *`matlabroot`* is the full path to your MATLAB installation folder, as returned by the `matlabroot` function.

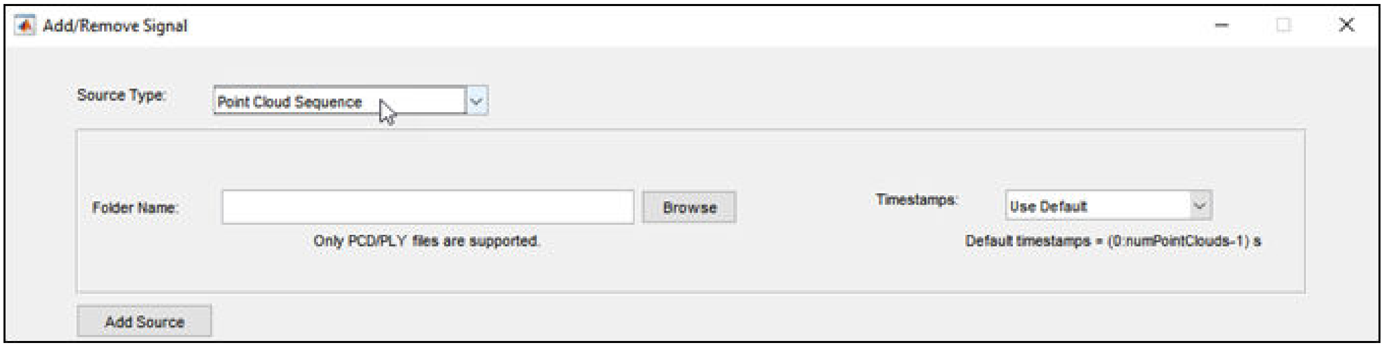
```
matlabroot\toolbox\driving\drivingdata\01_city_c2s_fcw_10s.mp4
```

3 Click **Add Source**. The video loads into the app, and the app reads the timestamps directly from the video. The source table displays the information about the video data source.

Load Point Cloud Sequence

Load the point cloud sequence into the app.

1 With the Add/Remove Signal dialog box still open and the video loaded, set the **Source Type** parameter to `Point Cloud Sequence`. The dialog box displays new options specific to loading point cloud sequences.



Add/Remove Signal

Source Type: Point Cloud Sequence

Folder Name:

Only PCD/PLY files are supported.

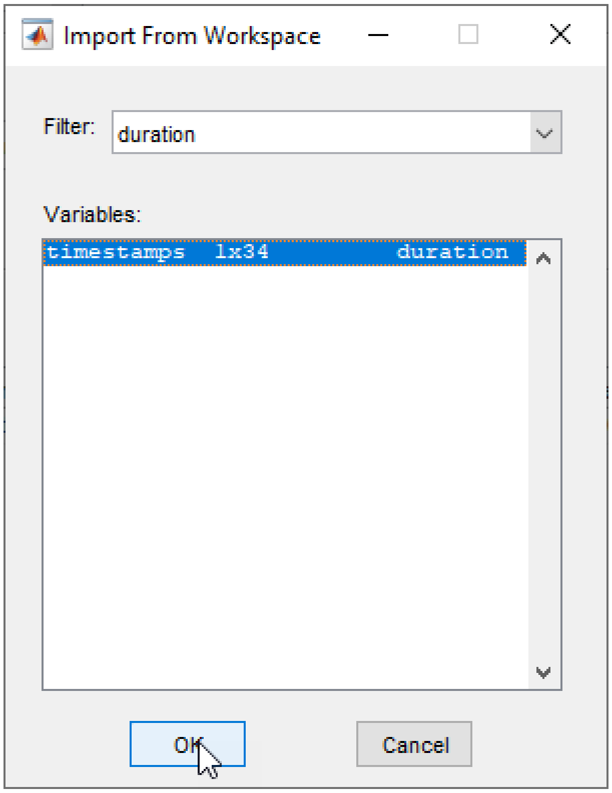
Timestamps: Use Default

Default timestamps = (0:numPointClouds-1) s

2 In the **Folder Name** parameter, browse for the `lidarSequence` folder, which contains the sequence of point cloud data (PCD) files to load.

```
matlabroot\toolbox\driving\drivingdata\lidarSequence
```

3 Set the **Timestamps** parameter to `From Workspace`. In the Import From Workspace dialog box, select the `timestamps` variable that you loaded for the point cloud sequence. Click **OK**.



Filter: duration

timestamps	1x34	duration

4 Click **Add Source**. The point cloud sequence loads into the app, and the app reads the timestamps from the `timestamps` variable. The source table displays the information about the data source for the point cloud sequence.

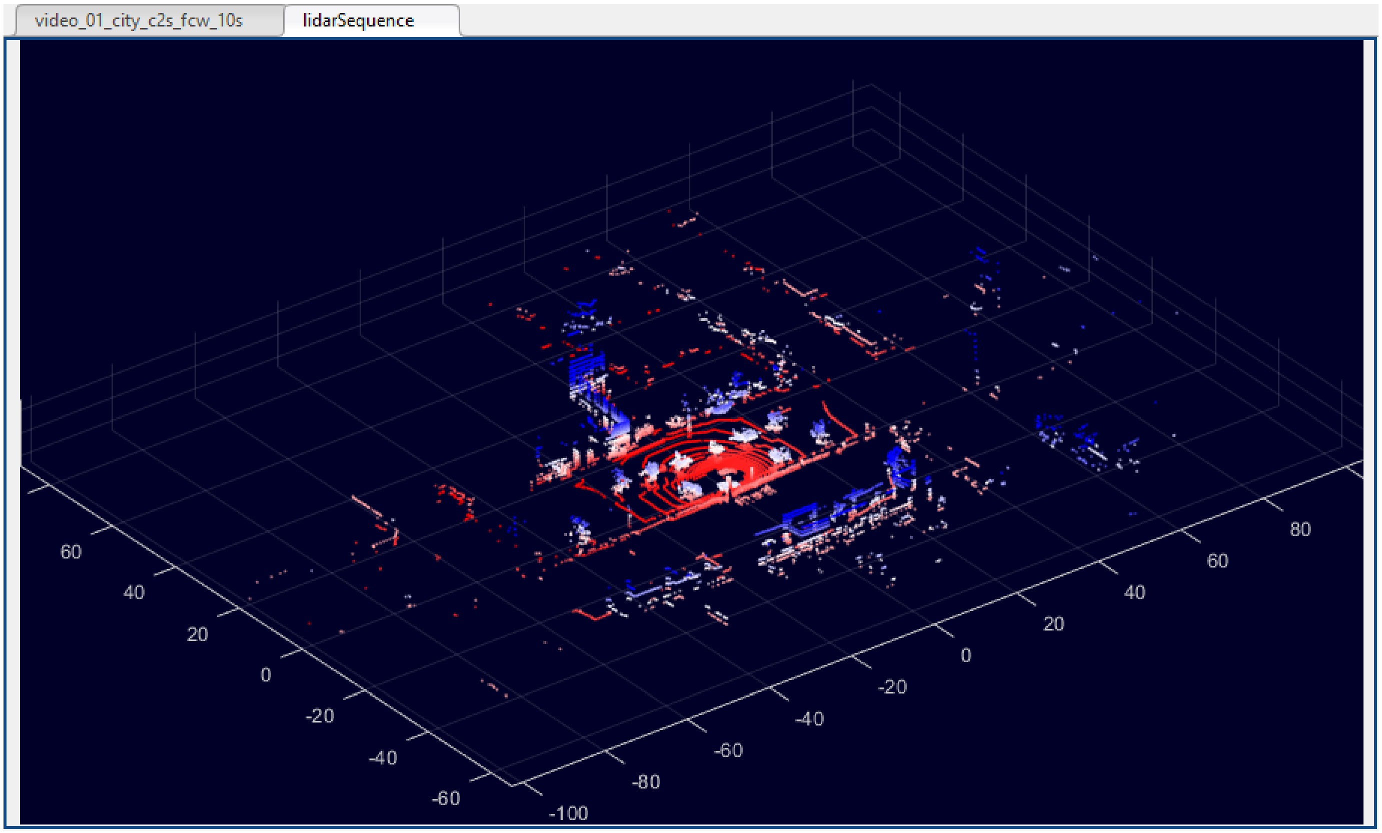
Verify Information About Loaded Signals

The table at the bottom of the Add/Remove Signal dialog box displays information about the loaded signals. Verify that the table displays this information for the loaded signals.

- The **Signal Name** column displays the signal names generated by the app. For the video, the signal name is the file name of the data source with the prefix `video_` and with no file extension. For the point cloud sequence, the signal name is the name of the source folder.
- The **Source** column displays the full file paths to the signal data sources.
- The **Signal Type** column displays the type of each signal. The video is of type `Image`. The point cloud sequence is of type `Point Cloud`.
- The **Time Range** column displays the duration of the signals based on the loaded timestamp data. Both signals are approximately 10 seconds long.

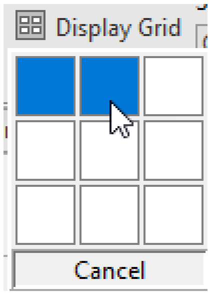
For the point cloud sequence, if you left **Timestamps** set to `Use Default`, then the **Time Range** value for the sequence ranges from 0 to 33 seconds. This range is based on the 34 PCD files in the folder. By default, the app sets the timestamps of a point cloud sequence to a `duration` vector from 0 to the number of valid point cloud files minus 1. Units are in seconds. If this issue occurs, in the table, select the check box for the point cloud sequence row. Then, click **Delete Selected**, load the signal again, and verify the signal information again.

After verifying that the signals loaded correctly, click **OK**. The app loads the signals and opens to the first frame of the last signal added, which for this example is the point cloud sequence.

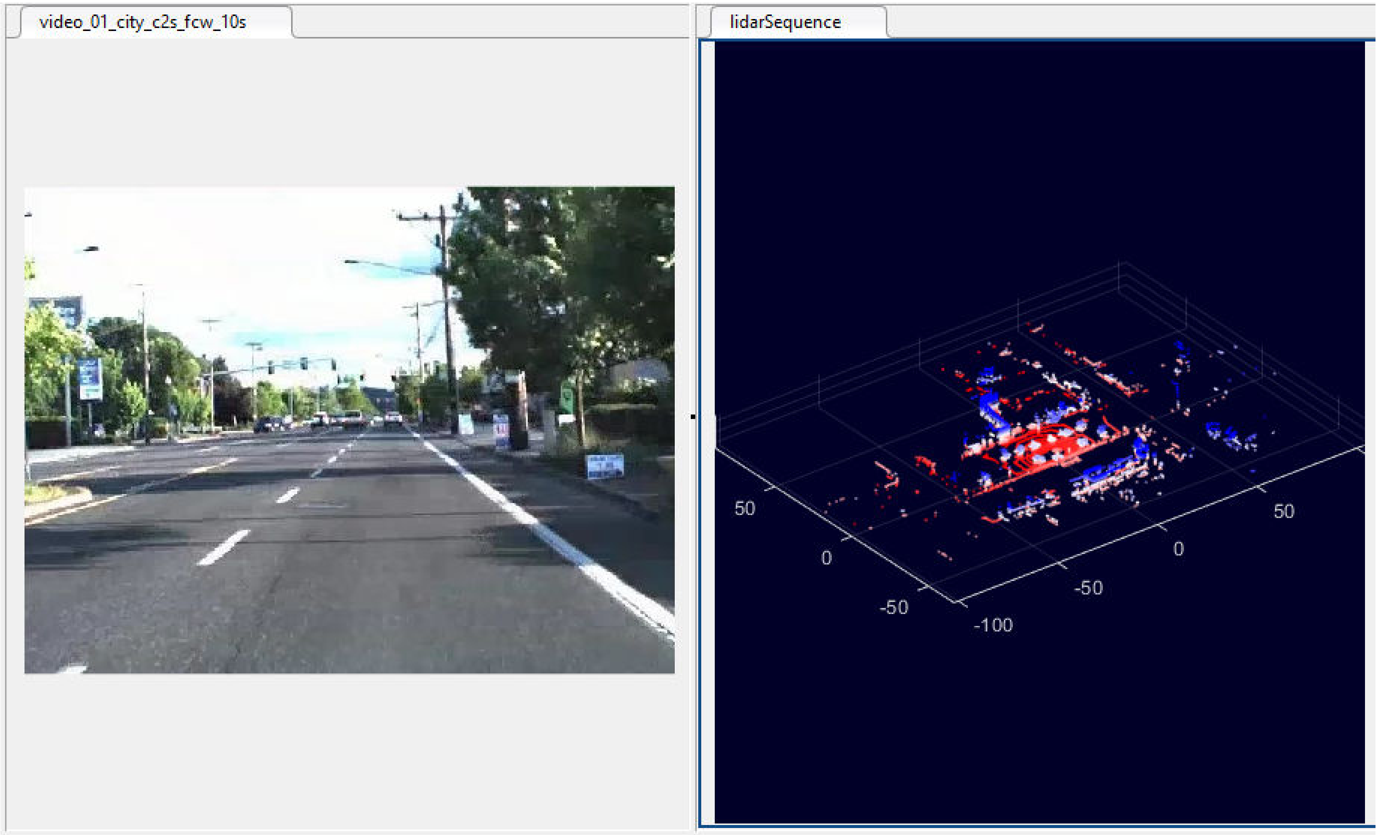


Configure Signal Display

When you first load the signals, the app displays only one signal at a time. To display the signals side-by-side, first, on the **Label** tab of the app toolstrip, click **Display Grid**. Then, move the pointer to select a 1-by-2 grid and click the grid.



The video and point cloud sequence display side-by-side.



To view the video and point cloud sequence together, in the slider below the signals, click the Play button ▶. The video plays more smoothly than the point cloud sequence because the video has more frames over approximately the same amount of time and therefore a higher frame rate.

By default, the app plays all frames from the signal with the highest frame rate. This signal is called the master signal. For all other signals, the app displays the frame that is time-aligned with the currently displaying frame of the master signal. To configure which signal is the master signal, use the options in the Playback Control Settings dialog box. To open this dialog box, below the slider, click the clock settings button ⚙. For more details about using these options to control the display of signal frames, see “Control Playback of Signal Frames for Labeling” on page 2-34.

After loading the signal and viewing the frames, you can now create label definitions and label the data, as described in “Label Ground Truth for Multiple Signals” on page 2-9.

See Also

More About

- “Sources vs. Signals in Ground Truth Labeling” on page 2-28
- “Control Playback of Signal Frames for Labeling” on page 2-34

Label Ground Truth for Multiple Signals

After loading the video and lidar point cloud sequence signals into the **Ground Truth Labeler** app, as described in the “Load Ground Truth Signals to Label” on page 2-4 procedure, create label definitions and label the signal frames. In this example, you label only a portion of the signals for illustrative purposes.

Create Label Definitions

Label definitions contain the information about the labels that you mark on the signals. You can create label definitions interactively within the app or programmatically by using a `labelDefinitionCreatorMultisignal` object. In this example, you create label definitions in the app.

Create ROI Label

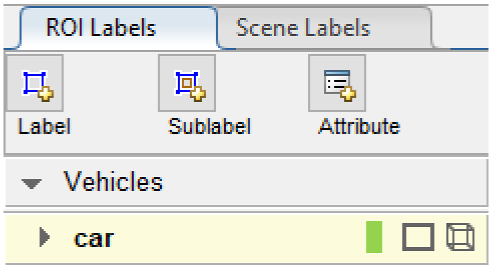
An ROI label is a label that corresponds to a region of interest (ROI) in a signal frame. You can define these ROI label types.

- `Rectangle/Cuboid` — Draw bounding box labels around objects, such as vehicles. In image signals, you draw labels of this type as 2-D rectangular bounding boxes. In point cloud signals, you draw labels of this type as 3-D cuboid bounding boxes.
- `Line` — Draw linear ROIs to label lines, such as lane boundaries.
- `Pixel label` — Draw pixels to label various classes, such as road or sky, for semantic segmentation.

For more details about these ROI label definitions, see “ROI Labels, Sublabels, and Attributes”.

Create an ROI label definition for labeling cars in the signal frames.

1. On the **ROI Labels** pane in the left pane, click **Label**.
2. Create a `Rectangle/Cuboid` label named `car`.
3. From the `Group` list, select `New Group` and name the group `Vehicles`. Adding labels to groups is optional.
4. Click **OK**. The **Vehicles** group name appears on the **ROI Labels** tab with the label **car** under it.



The **car** label is drawn differently on each signal. On the video, **car** is drawn as a 2-D rectangular bounding box of type `Rectangle`. On the point cloud sequence, **car** is drawn as a 3-D cuboid bounding box of type `Cuboid`.

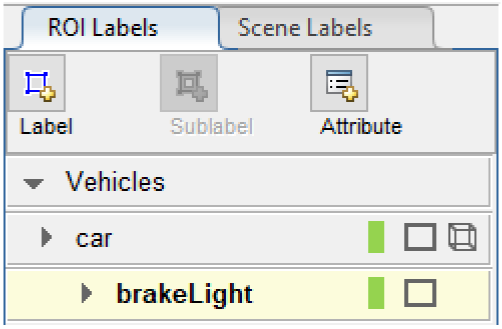
Create ROI Sublabel

A sublabel is a type of ROI label that corresponds to a parent ROI label. Each sublabel must belong to, or be a child of, a label definition that is in the **ROI Labels** tab. For example, in a driving scene, a vehicle label can have sublabels for headlights, license plates, or wheels. For more details about sublabels, see “ROI Labels, Sublabels, and Attributes”.

Create an ROI sublabel definition for labeling the brake lights of the labeled cars.

1. Select the parent label of the sublabel. On the **ROI Labels** tab in the left pane, click the **car** label to select it.
2. Click **Sublabel**.
3. Create a `Rectangle` sublabel named `brakeLight`. Cuboid sublabels are not supported, so this sublabel applies only for the video signal. Click **OK**.

   The **brakeLight** sublabel appears in the **ROI Labels** tab under the **car** label. The sublabel and parent label have the same color.



Create ROI Attribute

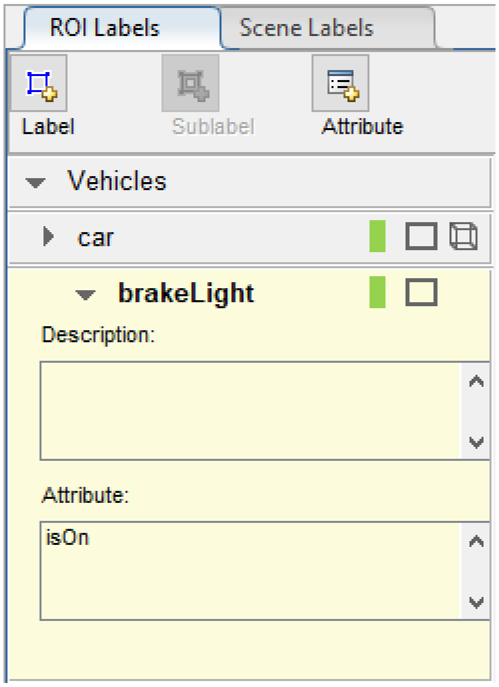
An ROI attribute specifies additional information about an ROI label or sublabel. For example, in a driving scene, attributes can include the type or color of a vehicle. You can define ROI attributes of these types.

- `Numeric Value` — Specify a numeric scalar attribute, such as the number of doors on a labeled vehicle.
- `String` — Specify a string scalar attribute, such as the color of a vehicle.
- `Logical` — Specify a logical true or false attribute, such as whether a vehicle is in motion.
- `List` — Specify a drop-down list attribute of predefined strings, such as make or model of a vehicle.

For more details about these attribute types, see “ROI Labels, Sublabels, and Attributes”.

Create an attribute to describe whether a labeled brake light is on or off.

1. On the **ROI Labels** tab in the left pane, select the **brakeLight** sublabel and click **Attribute**.
2. In the **Attribute Name** box, type `isOn`. Set the attribute type to `Logical`. Leave **Default Value** set to `Empty` and click **OK**.
3. In the **ROI Labels** tab, expand the **brakeLight** sublabel definition. The **Attribute** box for this sublabel now contains the **isOn** attribute.

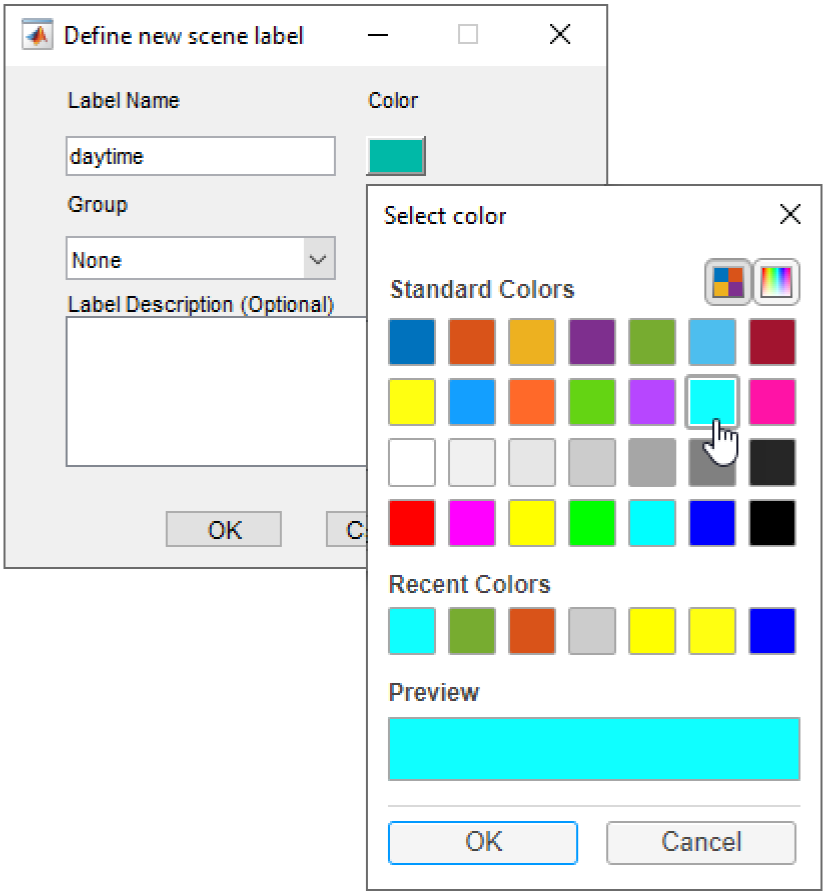


Create Scene Label

A scene label defines additional information across all signals in a scene. Use scene labels to describe conditions, such as lighting and weather, or events, such as lane changes.

Create a scene label to apply to the signal frames.

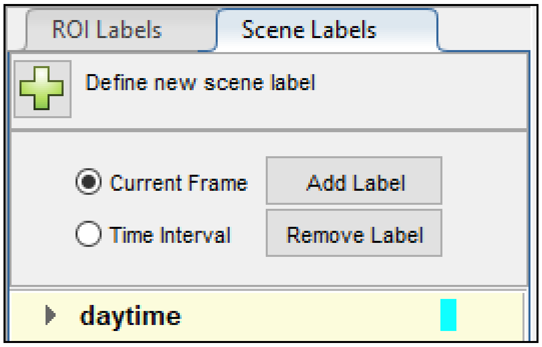
1. In the left pane of the app, select the **Scene Labels** tab.
2. Click **Define new scene label**, and in the **Label Name** box, enter a scene label named `daytime`.
3. Change the color of the label definition to light blue to reflect the nature of the scene label. Under the **Color** parameter, click the color preview and select the standard light blue colors. Then, click **OK** to close the color selection window.



Label Name: daytime

Group: None

4 Leave the **Group** parameter set to the default of None and click **OK**. The **Scene Labels** pane shows the scene label definition.



Verify Label Definitions

Verify that your label definitions have this setup.

1 The **ROI Labels** tab contains a **Vehicles** group with a **car** label of type `Rectangle/Cuboid`.

2 The **car** label contains a sublabel named **brakeLight**.

3 The **brakeLight** sublabel contains an attribute named **isOn**.

4 The **Scene Labels** tab contains a light blue scene label named **daytime**.

To edit or delete a label definition, right-click that label definition and select the appropriate edit or delete option. To save these label definitions to a MAT-file for use in future labeling sessions, on the **Label** tab of the app toolstrip, select **Save > Label Definitions**.

In future labeling sessions, if you need to reorder label definitions or move them to different groups, you can drag and drop them in the label definition panes.

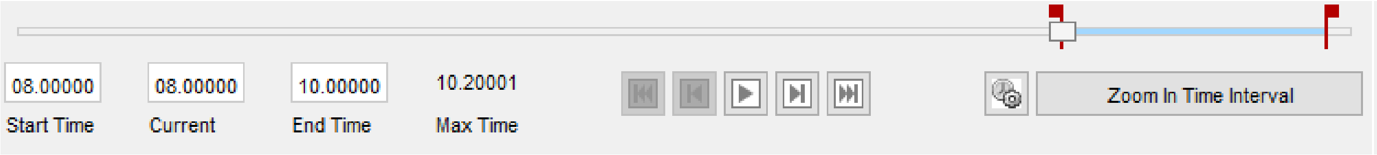
Label Video Using Automation

Use the **car** label to label one of the cars in a portion of the video. To assist with the labeling process, use one of the built-in label automation algorithms.

1. Select the time interval to label. Specify an interval from 8 to 10 seconds, during which the car in front is close to the ego vehicle. In the text boxes below the video, enter these times in this order:

   a. In the **Current** box, type `8` and press **Enter**.
   b. In the **Start Time** box, type `8` so that the slider is at the start of the time interval.
   c. In the **End Time** box, type `10`.

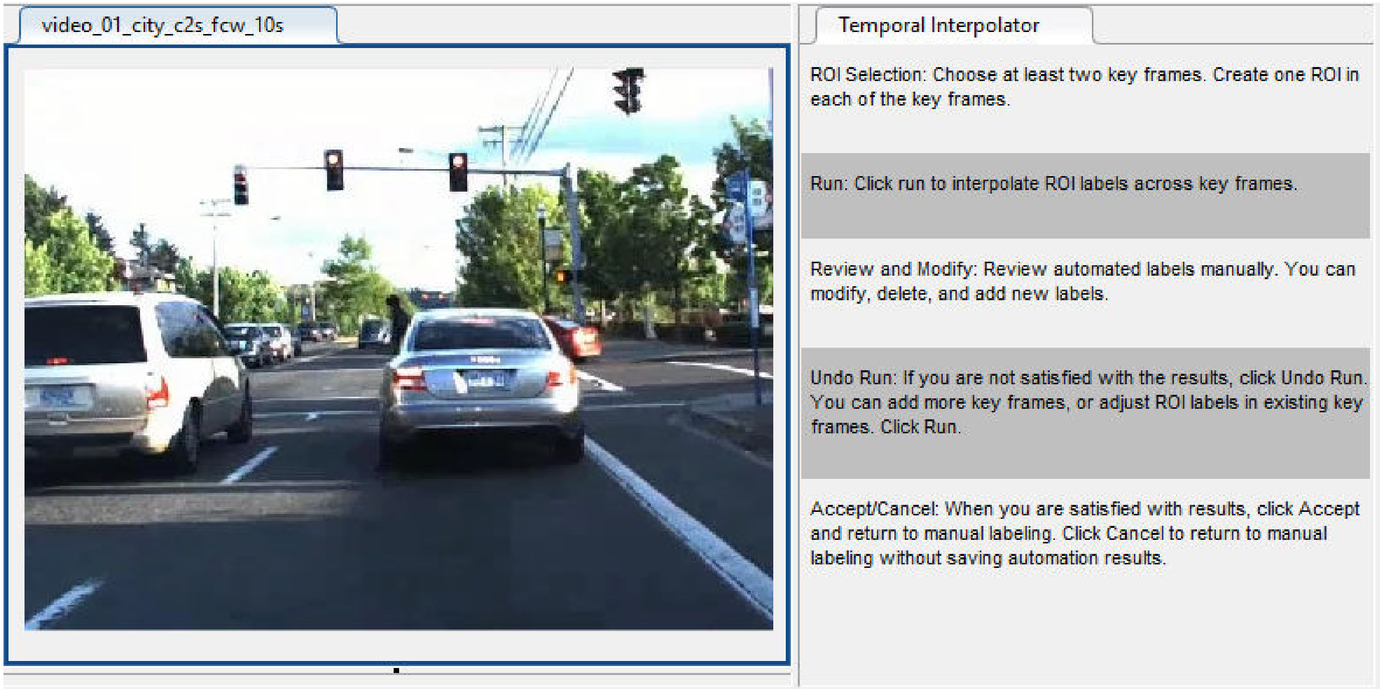
   The range slider and text boxes are set to this 8–10 second interval. The red flags indicate the start and end of the interval.



Start Time: 08.00000
Current: 08.00000
End Time: 10.00000
Max Time: 10.20001

   The app displays signal frames from only this interval, and automation algorithms apply to only this interval. To expand the time interval to fill the entire playback section, click **Zoom In Time Interval**.

2. In the labeling window, click the video signal to select it. You can automate only one signal at a time, so you must select the signal that you want to automate.
3. Select the label that you want to automate. In the **ROI Labels** tab, click the **car** label.
4. From the app toolstrip, select **Select Algorithm > Temporal Interpolator**. This algorithm estimates rectangle ROIs between image frames by interpolating the ROI locations across the time interval.
5. Click **Automate**. The app prompts you to confirm that you want to label only a portion of the video. Click **Yes**. An automation session for the video opens. The right pane of the automation session displays the algorithm instructions.



ROI Selection: Choose at least two key frames. Create one ROI in each of the key frames.

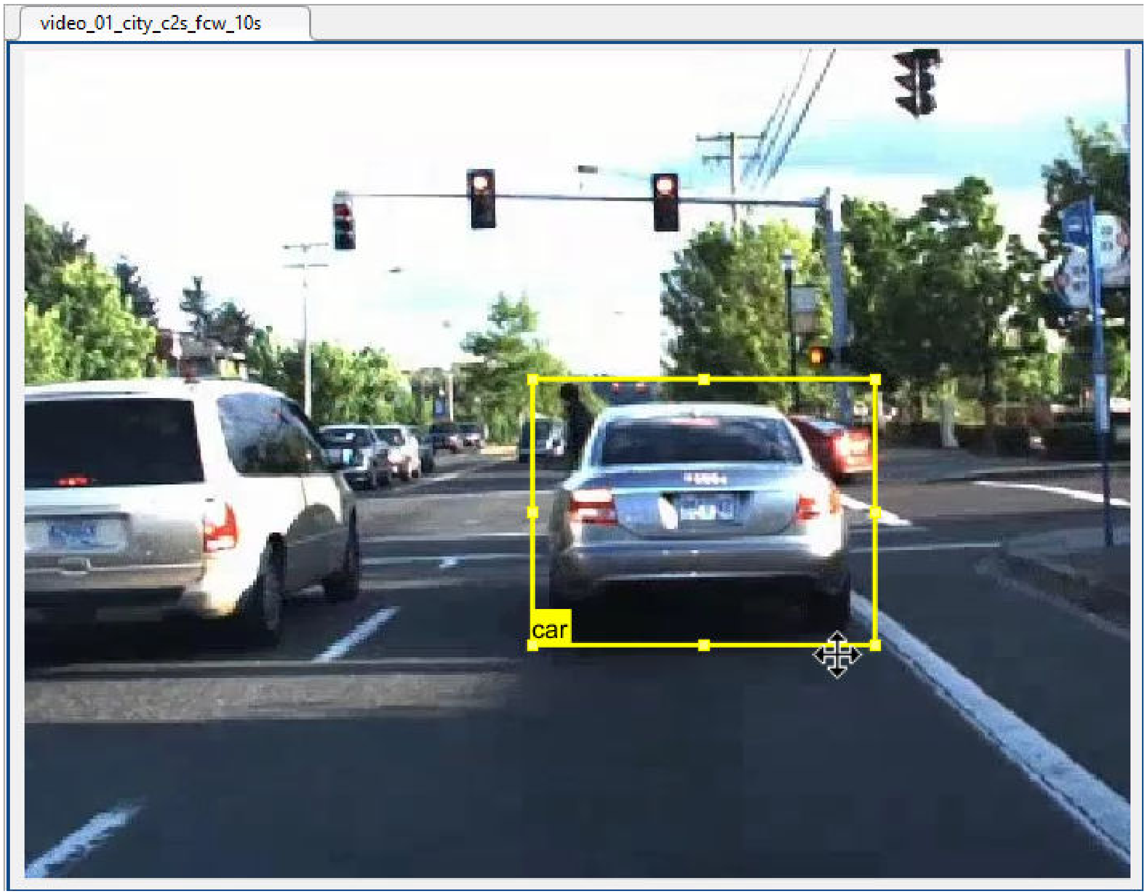
Run: Click run to interpolate ROI labels across key frames.

Review and Modify: Review automated labels manually. You can modify, delete, and add new labels.

Undo Run: If you are not satisfied with the results, click Undo Run. You can add more key frames, or adjust ROI labels in existing key frames. Click Run.

Accept/Cancel: When you are satisfied with results, click Accept and return to manual labeling. Click Cancel to return to manual labeling without saving automation results.

6 At the start of the time interval, click and drag to draw a **car** label around the car in the center of the frame. For this algorithm, you can draw only one label per frame. Labeling the other car would require a separate automation session.



By default, the **car** label appears only when you move your pointer over it. Optionally, to always display labels, on the app toolstrip, set **Show ROI Labels** to `Always`.

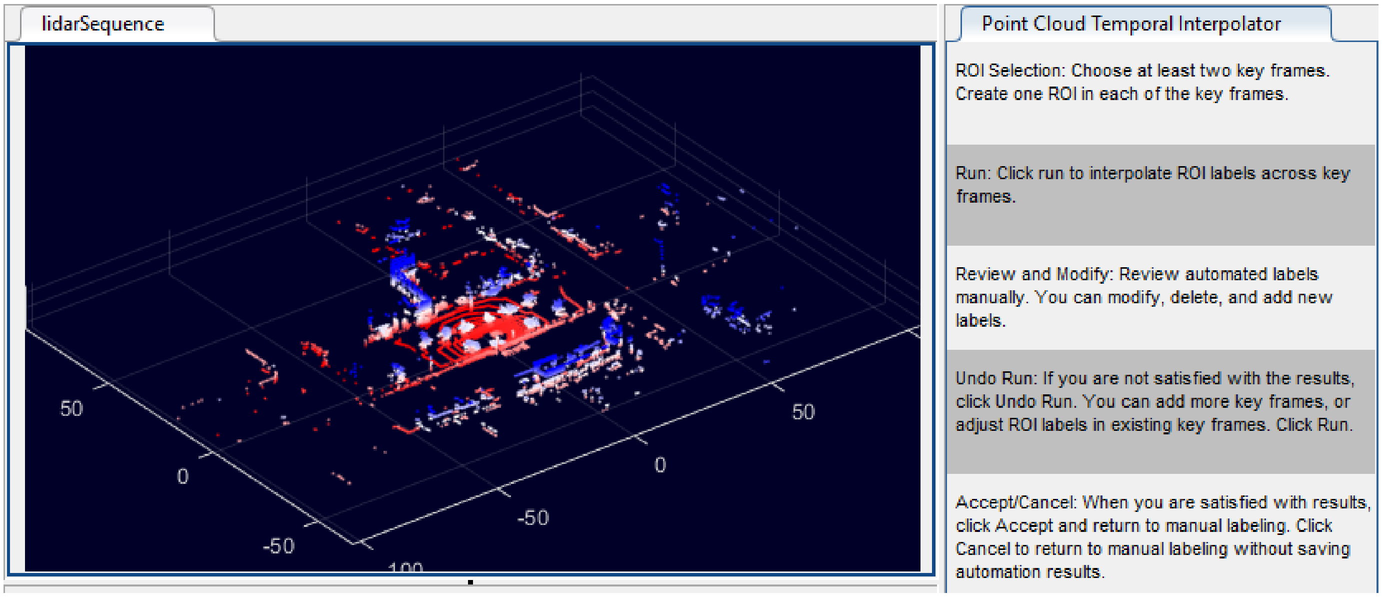
7 Drag the slider to the last frame and draw a **car** label around the same car in this frame. Optionally, to improve automation results, label the car in intermediate frames.
8 Click **Run**. The automation algorithm applies the **car** label to the intermediate frames. Drag the slider to view the results. If necessary, manually adjust the labels to improve their accuracy.
9 When you are satisfied with the results, click **Accept** to close the session and apply the labels to this portion of the video.

Label Point Cloud Sequence Using Automation

Use the same **car** label definition from the previous procedure to label a car in the point cloud sequence. To assist with the labeling process, use a built-in label automation algorithm designed for point cloud labeling. In this example, you label the ego vehicle, which is easier to see in the lidar point cloud sequence than the front car.

1 At the bottom of the app, verify that the time range is still set to 8 to 10 seconds.
2 In the labeling window, click the point cloud sequence to select it.
3 In the **ROI Labels** tab, click the **car** label definition.
4 On the **Label** tab of the app toolstrip, select **Select Algorithm > Point Cloud Temporal Interpolator**. This algorithm estimates cuboid ROIs between point cloud frames by interpolating the ROI locations across the time interval.

5 Click **Automate**. The app prompts you to confirm that you want to label only a portion of the point cloud sequence. Click **Yes**. An automation session for the point cloud sequence opens. The right pane of the automation session displays the algorithm instructions.



ROI Selection: Choose at least two key frames. Create one ROI in each of the key frames.

Run: Click run to interpolate ROI labels across key frames.

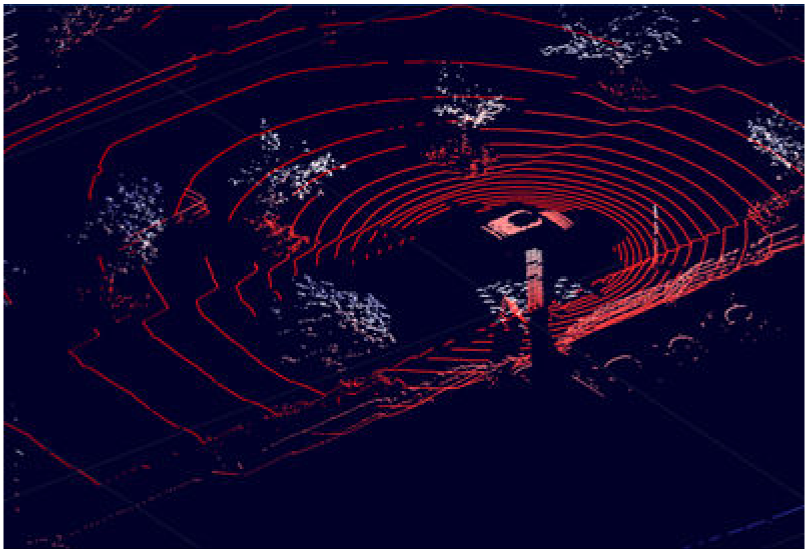
Review and Modify: Review automated labels manually. You can modify, delete, and add new labels.

Undo Run: If you are not satisfied with the results, click Undo Run. You can add more key frames, or adjust ROI labels in existing key frames. Click Run.

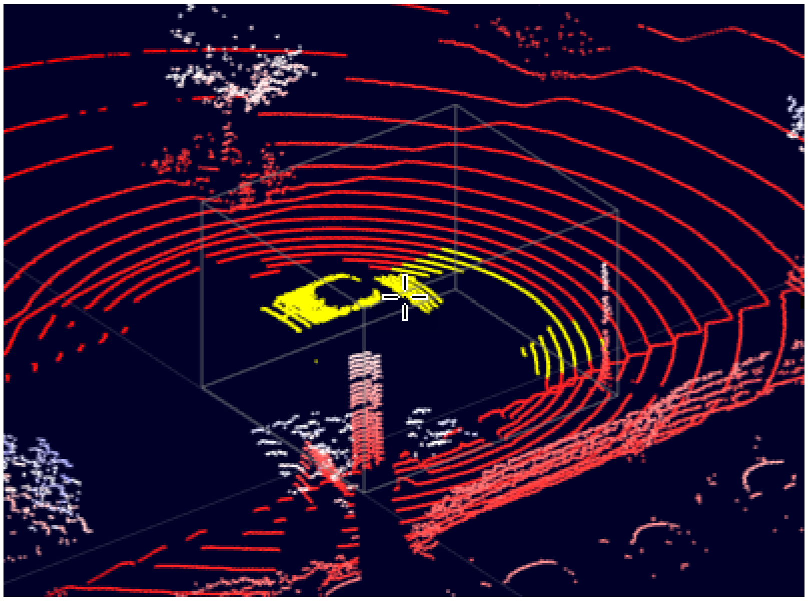
Accept/Cancel: When you are satisfied with results, click Accept and return to manual labeling. Click Cancel to return to manual labeling without saving automation results.

6 At the start of the time interval, draw a **car** label around the ego vehicle.

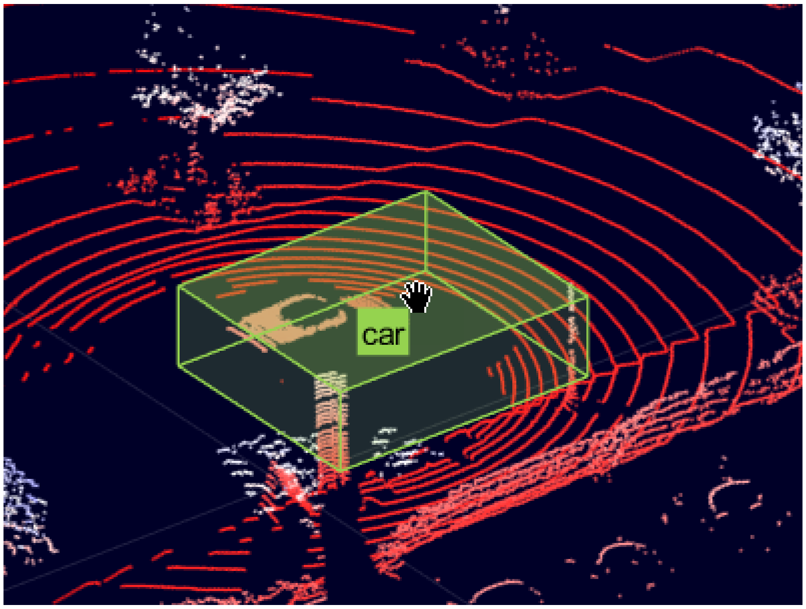
a Zoom in on the car, using either the scroll wheel or the Zoom In button at the top-right corner of the frame. You can also use the Pan button to center the car in the frame.



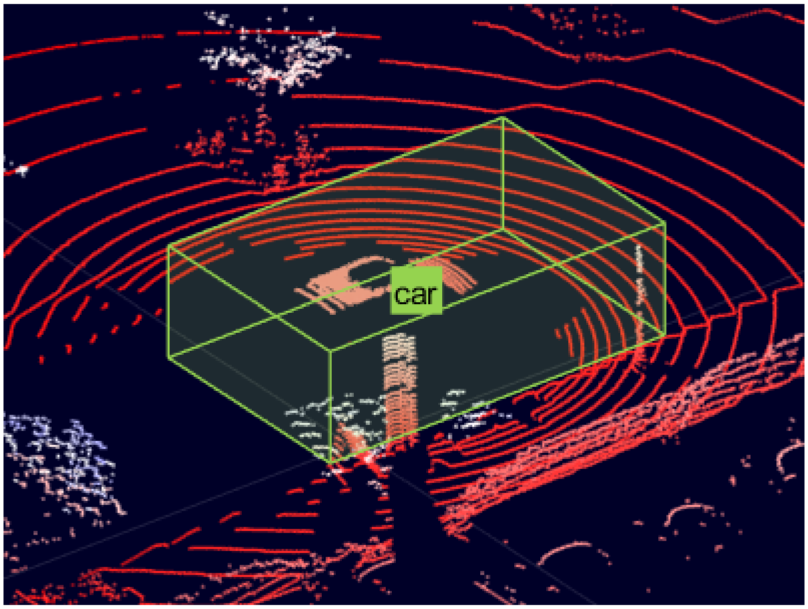
b On the **ROI Labels** tab in the left pane, click the **car** label. Drag the gray preview cuboid until it highlights the ego vehicle.



c Click the signal frame to create the label. The label snaps to the highlighted portion of the point cloud.



d Adjust the cuboid label until it fully encloses the car. To resize the cuboid, click and drag one of the cuboid faces. To move the cuboid, hold **Shift** and click and drag one of the cuboid faces.



For additional tips and techniques for labeling point clouds, see “Label Lidar Point Clouds for Object Detection” on page 2-38.

7 Click the cuboid and press **Ctrl+C** to copy it. Then, drag the slider to the last frame and paste (**Ctrl+V**) the cuboid into the new frame at the same position. Optionally, to improve automation results, manually adjust the position of the copied label.

8 Click **Run**. The automation algorithm applies the **car** label to the intermediate frames. Drag the slider to view the results. If necessary, manually adjust the labels to improve their accuracy.

9 When you are satisfied with the results, click **Accept** to close the session and apply the labels to this portion of the point cloud sequence.

Label with Sublabels and Attributes Manually

Manually label one frame of the video with the **brakeLight** sublabel and its **isOn** attribute. Lidar point cloud signals do not support sublabels and attributes, so you cannot label the point cloud sequence.

1 At the bottom of the app, verify that the time range is still set to 8 to 10 seconds. If necessary, drag the slider to the first frame of the time range.

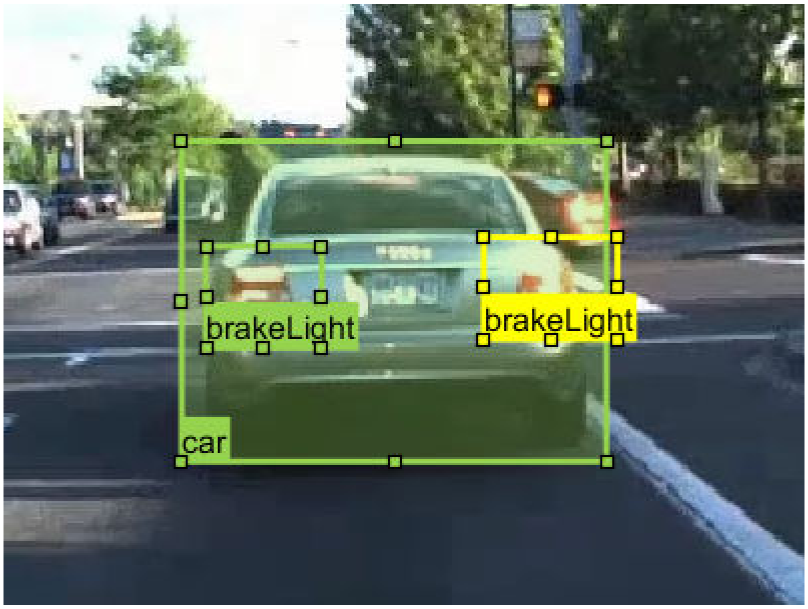
2 In the **ROI Labels** tab, click the **brakeLight** sublabel definition to select it.

3 Hide the point cloud sequence. On the **Label** tab of the app toolstrip, under **Show/Hide Signals**, clear the check mark for the lidar point cloud sequence. Hiding a signal only hides the display. The app maintains the labels for hidden signals, and you can still export them.

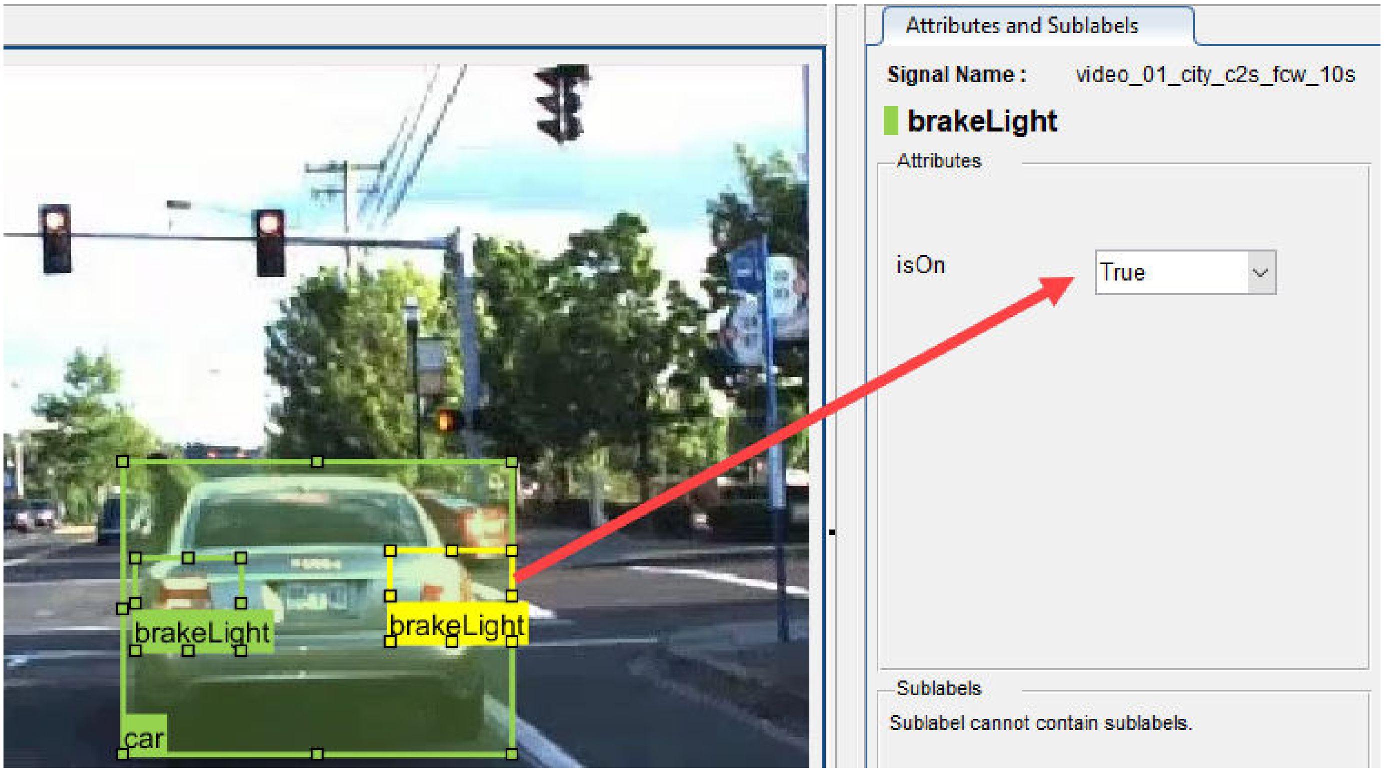
4 Expand the video signal to fill the entire labeling window.

5 In the video frame, select the drawn **car** label. The label turns yellow. You must select the **car** label (parent ROI) before you can add a sublabel to it.

6 Draw **brakeLight** sublabels for the car. Optionally, set **Show ROI Labels** to `Always` so that you can confirm the association between the **car** label and its sublabels.



7 On the video frame, select one of the **brakeLight** sublabels. Then, on the **Attributes and Sublabels** pane in the right pane, set the **isOn** attribute to `True`. Repeat this step for the other sublabel.



Signal Name : video_01_city_c2s_fcw_10s

brakeLight

Attributes

isOn True

Sublabels

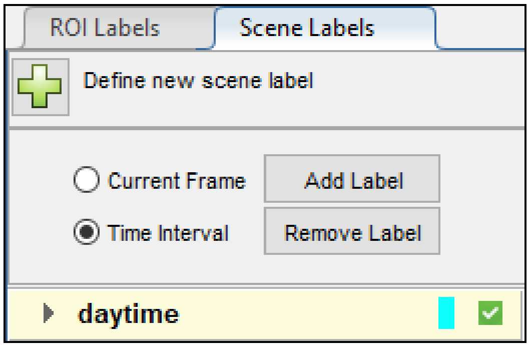
Sublabel cannot contain sublabels.

For more details about working with sublabels and attributes, see “Use Sublabels and Attributes to Label Ground Truth Data” (Computer Vision Toolbox).

Label Scene Manually

Apply the **daytime** scene label to the entire scene.

1. Expand the time interval back to the entire duration of all signals. If you zoomed in on the time interval, first click **Zoom Out Time Interval**. Then, drag the red flags to the start and end of the range slider.
2. In the left pane of the app, select the **Scene Labels** tab.
3. Select the **daytime** scene label definition.
4. Above the label definition, click **Time Interval**. Then, click **Add Label**. A check mark appears for the **daytime** scene label indicating that the label now applies to all frames in the time interval.



○ Current Frame
● Time Interval

daytime ☑

View Label Summary

With all labels, sublabels, and attributes applied to at least one frame of a signal, you can now optionally view a visual summary of the ground truth labels. On the app toolstrip, click **View Label Summary**. For more details, see “View Summary of Ground Truth Labels” (Computer Vision Toolbox).

Save App Session

On the app toolstrip, select **Save** and save a MAT-file of the app session. The saved session includes the data source, label definitions, and labeled ground truth. It also includes your session preferences, such as the layout of the app.

You can now either close the app session or continue to the “Export Ground Truth Labels for Multiple Signals” on page 2-21 step, where you export the labels.

See Also

More About

- “Label Lidar Point Clouds for Object Detection” on page 2-38
- “Label Pixels for Semantic Segmentation” (Computer Vision Toolbox)
- “Keyboard Shortcuts and Mouse Actions for Ground Truth Labeler” on page 2-30
- “View Summary of Ground Truth Labels” (Computer Vision Toolbox)

Export Ground Truth Labels for Multiple Signals

After labeling the signals by following the “Label Ground Truth for Multiple Signals” on page 2-9 procedure, export the labels and examine how they are stored.

Setup

Open the **Ground Truth Labeler** app session containing the labeled signals. You can open the session from the MATLAB® command line. For example, if you saved the session to a MAT-file named groundTruthLabelingSession, enter this command.

```
groundTruthLabeler groundTruthLabelingSession.mat
```

On the app toolstrip, select **Export Labels > To Workspace**. In the export to workspace window, use the default export variable name, gTruth, and click **OK**. The app exports a groundTruthMultisignal object, gTruth, to the MATLAB® workspace. This object contains the ground truth labels captured from the app session.

If you did not export a groundTruthMultisignal object to the workspace, load a predefined object from the variable gTruth. The function used to load this object is attached to this example as a supporting file. If you are using your own object, data such as label positions can differ from the data shown in this example.

```
if (~exist('gTruth','var'))
    gTruth = helperLoadGTruthGetStarted;
end
```

Display the properties of the groundTruthMultisignal object, gTruth. The object contains information about the signal data sources, label definitions, and ROI and scene labels. This information is stored in separate properties of the object.

```
gTruth
```

```
gTruth = 

  groundTruthMultisignal with properties:

          DataSource: [1×2 vision.labeler.loading.MultiSignalSource]
    LabelDefinitions: [3×7 table]
        ROILabelData: [1×1 vision.labeler.labeldata.ROILabelData]
      SceneLabelData: [1×1 vision.labeler.labeldata.SceneLabelData]
```

In this example, you examine the contents of each property to learn how the object stores ground truth labels.

Data Sources

The DataSource property contains information about the data sources. This property contains two MultiSignalSource objects: one for the video source and one for the point cloud sequence source. Display the contents of the DataSource property.

```
gTruth.DataSource
```

```
ans = 
  1×2 heterogeneous MultiSignalSource (VideoSource, PointCloudSequenceSource) array with properti

    SourceName
    SourceParams
    SignalName
    SignalType
    Timestamp
    NumSignals
```

The information stored in these objects includes the paths to the data sources, the names of the signals that they contain, and the timestamps for those signals. Display the signal names for the data sources.

```
gTruth.DataSource.SignalName
```

```
ans = 

    "video_01_city_c2s_fcw_10s"


ans = 

    "lidarSequence"
```

Label Definitions

The `LabelDefinitions` property contains a table of information about the label definitions. Display the label definitions table. Each row contains information about an ROI or scene label definition. The `car` label definition has two rows: one for when the label is drawn as a rectangle on `Image` signals and one for when the label is drawn as a cuboid on `PointCloud` signals.

```
gTruth.LabelDefinitions
```

```
ans =

  3×7 table

       Name        SignalType    LabelType       Group        Description     LabelColor      Hie
    ___________    __________    _________    ____________    ___________    ____________    ____

    {'car'    }    Image         Rectangle    {'Vehicles'}    {0×0 char}     {1×3 double}    {1×1
    {'car'    }    PointCloud    Cuboid       {'Vehicles'}    {0×0 char}     {1×3 double}    {1×1
    {'daytime'}    Time          Scene        {'None'    }    {0×0 char}     {1×3 double}    {0×0
```

The `Hierarchy` column stores information about the sublabel and attribute definitions of a parent ROI label. Display the sublabel and attribute information for the `car` label when it is drawn as a rectangle. This label contains one sublabel, `brakeLight`, and no attributes.

```
gTruth.LabelDefinitions.Hierarchy{1}
```

```
ans = 
  struct with fields:

     brakeLight: [1×1 struct]
           Type: Rectangle
    Description: ''
```

Display information about the `brakeLight` sublabel for the parent `car` label. The sublabel contains one attribute, `isOn`. Sublabels cannot have their own sublabels.

```
gTruth.LabelDefinitions.Hierarchy{1}.brakeLight
```

```
ans = 

  struct with fields:

           Type: Rectangle
    Description: ''
     LabelColor: [0.5862 0.8276 0.3103]
           isOn: [1×1 struct]
```

Display information about the `isOn` attribute for the `brakeLight` sublabel. This attribute has no default value, so the `DefaultValue` field is empty.

```
gTruth.LabelDefinitions.Hierarchy{1}.brakeLight.isOn
```

```
ans = 

  struct with fields:

    DefaultValue: []
     Description: ''
```

ROI Label Data

The `ROILlabelData` property contains an `ROILabelData` object with properties that contain ROI label data for each signal. The names of the properties match the names of the signals. Display the object property names.

```
gTruth.ROILabelData
```

```
ans = 

  ROILabelData with properties:

    video_01_city_c2s_fcw_10s: [204×1 timetable]
                lidarSequence: [34×1 timetable]
```

Each property contains a timetable of ROI labels drawn at each signal timestamp, with one column per label. View a portion the video and the lidar point cloud sequence timetables. Set a time interval from 8 to 8.5 seconds. This time interval corresponds to the start of the time interval labeled in the

“Label Ground Truth for Multiple Signals” on page 2-9 procedure. The video timetable contains more rows than the point cloud sequence timetable because the video contains more label frames.

```
timeInterval = timerange(seconds(8),seconds(8.5));
videoLabels = gTruth.ROILabelData.video_01_city_c2s_fcw_10s(timeInterval,:)
lidarLabels = gTruth.ROILabelData.lidarSequence(timeInterval,:)
```

```
videoLabels =

  10×1 timetable

      Time          car
    ________    ____________

    8 sec       {1×1 struct}
    8.05 sec    {1×1 struct}
    8.1 sec     {1×1 struct}
    8.15 sec    {1×1 struct}
    8.2 sec     {1×1 struct}
    8.25 sec    {1×1 struct}
    8.3 sec     {1×1 struct}
    8.35 sec    {1×1 struct}
    8.4 sec     {1×1 struct}
    8.45 sec    {1×1 struct}


lidarLabels =

  2×1 timetable

       Time           car
    __________    ____________

    8.0495 sec    {1×9 double}
    8.3497 sec    {1×9 double}
```

View the rectangle `car` labels for the first video frame in the time interval. The label data is stored in a structure.

```
videoLabels.car{1}
```

```
ans =

  struct with fields:

      Position: [296 203 203 144]
    brakeLight: [1×2 struct]
```

The `Position` field stores the positions of the `car` labels. This frame contains only one `car` label, so in this case, `Position` contains only one rectangle bounding box. The bounding box position is of the form `[x y w h]`, where:

- `x` and `y` specify the upper-left corner of the rectangle.

- `w` specifies the width of the rectangle, which is the length of the rectangle along the *x*-axis.
- `h` specifies the height of the rectangle, which is the length of the rectangle along the *y*-axis.

The `car` label also contains two `brakeLight` sublabels at this frame. View the `brakeLight` sublabels. The sublabels are stored in a structure array, with one structure per sublabel drawn on the frame.

```
videoLabels.car{1}.brakeLight
```

```
ans = 

  1×2 struct array with fields:

    Position
    isOn
```

View the bounding box positions for the sublabels.

```
videoLabels.car{1}.brakeLight.Position
```

```
ans =

   304   245    50    46


ans =

   435   243    54    51
```

View the values for the `isOn` attribute in each sublabel. For both sublabels, this attribute is set to logical `1` (`true`).

```
videoLabels.car{1}.brakeLight.isOn
```

```
ans =

  logical

   1


ans =

  logical

   1
```

Now view the cuboid `car` labels for the first point cloud sequence frame in the time interval. Point cloud sequences do not support sublabels or attributes. Instead of storing cuboid labels in the `Position` field of a structure, cuboid bounding box positions are stored in an `M`-by-9 matrix, where `M` is the number of cuboid labels. Because this frame contains only one cuboid label, in this case `M` is 1.

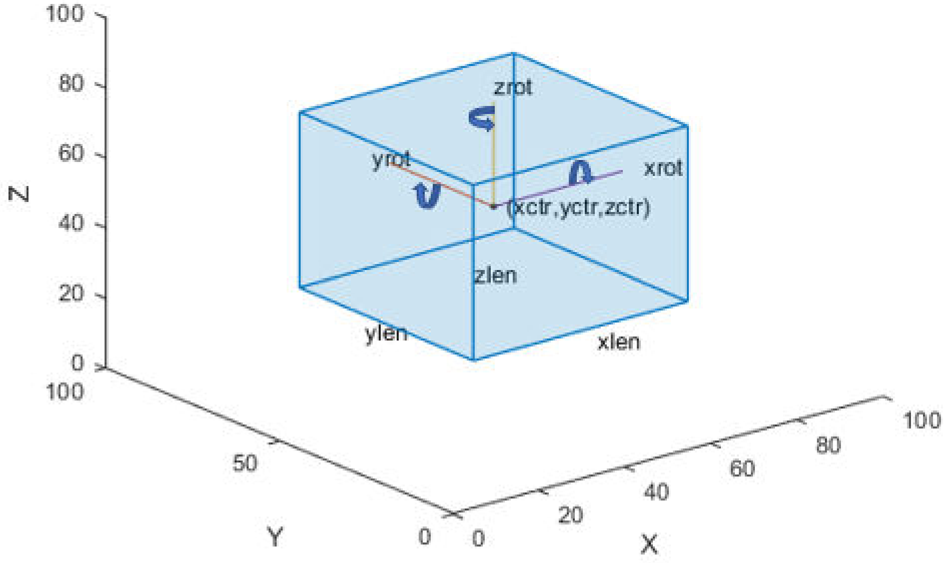
```
lidarLabels.car{1}
```

```
ans =

  Columns 1 through 7

   -1.1559   -0.7944    1.2012   12.6196    5.9278    3.0010         0

  Columns 8 through 9

         0         0
```

The 1-by-9 bounding box position is of the form `[xctr, yctr, zctr, xlen, ylen, zlen, xrot, yrot, zrot]`, where:

- `xctr`, `yctr`, and `zctr` specify the center of the cuboid.
- `xlen`, `ylen`, and `zlen` specify the length of the cuboid along the *x*-, *y*-, and *z*-axis, respectively.
- `xrot`, `yrot`, and `zrot` specify the rotation angles for the cuboid along the *x*-, *y*-, and *z*-axis, respectively. These angles are clockwise-positive when looking in the forward direction of their corresponding axes.

This figure shows how these values specify the position of a cuboid.



Scene Label Data

The `SceneLabelData` property contains a `SceneLabelData` object with properties that contain scene label data across all signals. The names of the properties match the names of the scene labels. Display the object property names.

```
gTruth.SceneLabelData
```

```
ans =
  SceneLabelData with properties:

    daytime: [0 sec    10.15 sec]
```

The `daytime` label applies to the entire time interval, which is approximately 10 seconds.

Use Ground Truth Labels

The labels shown in this example are for illustrative purposes only. For your own labeling, after you export the labels, you can use them as training data for object detectors. To gather label data from the `groundTruthMultisignal` object for training, use the `gatherLabelData` function.

To share labeled ground truth data, share the ground truth MAT-file containing the `groundTruthMultisignal` object, not the MAT-file containing the app session. For more details, see “Share and Store Labeled Ground Truth Data” (Computer Vision Toolbox).

See Also

`ROILabelData` | `SceneLabelData` | `gatherLabelData` | `groundTruthMultisignal`

More About

- “Share and Store Labeled Ground Truth Data” (Computer Vision Toolbox)
- “How Labeler Apps Store Exported Pixel Labels” (Computer Vision Toolbox)

Sources vs. Signals in Ground Truth Labeling

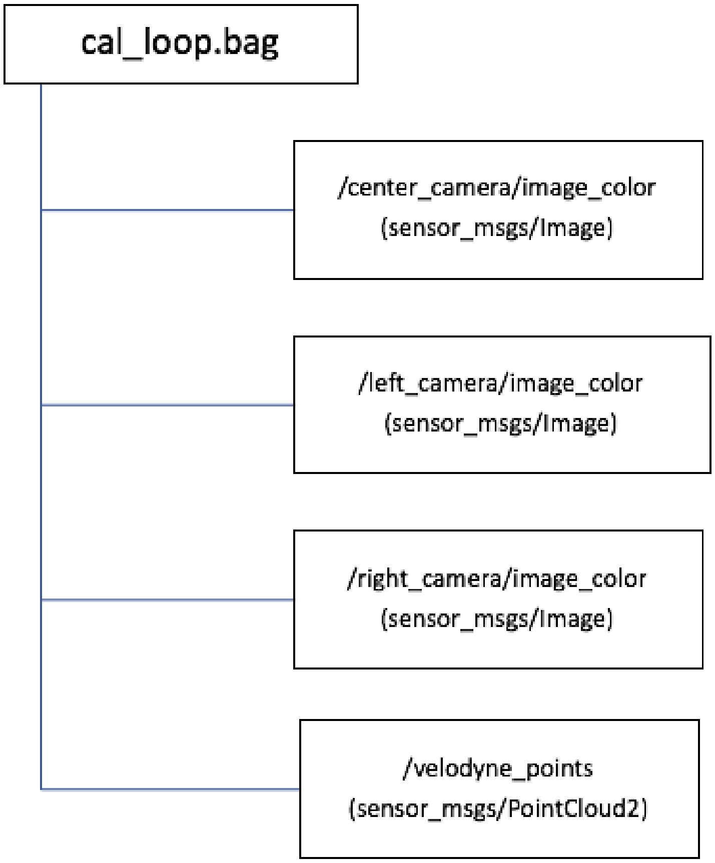
In the **Ground Truth Labeler** app, a source is the file or folder containing the data that you want to load. A signal is the data from that source that you want to label. A source can contain one or more signals.

In many cases, a source contains only one signal. Consider an AVI video file. The source is the AVI file and the signal is the video that you load from that file. Other sources that have only one signal include Velodyne® packet capture (PCAP) files and folders that contain image or point cloud sequences.

Sources such as rosbags can contain multiple signals. Consider a rosbag named `cal_loop.bag`. The rosbag contains data obtained from four sensors mounted on a vehicle. The source is the rosbag file. The signals in the rosbag are `sensor_msgs` topics that correspond to the data from the four sensors. The topics have these names.

- `/center_camera/image_color` — Image sequence obtained from the center camera
- `/left_camera/image_color` — Image sequence obtained from the left camera
- `/right_camera/image_color` — Image sequence obtained from the right camera
- `/velodyne_points` — Point cloud sequence obtained from a Velodyne lidar sensor

This diagram depicts the relationship between the source and each of its four signals.



See Also

`groundTruthMultisignal` | `vision.labeler.loading.MultiSignalSource`

More About

- “Load Ground Truth Signals to Label” on page 2-4

Keyboard Shortcuts and Mouse Actions for Ground Truth Labeler

Note On Macintosh platforms, use the **Command (⌘)** key instead of **Ctrl**.

Label Definitions

Task	Action
In the **ROI Label Definition** pane, navigate through ROI labels and their groups	Up arrow or down arrow
In the **Scene Label Definition** pane, navigate through scene labels and their groups	Hold **Alt** and press the up arrow or down arrow
Reorder labels within a group or move labels between groups	Click and drag labels
Reorder groups	Click and drag groups

Frame Navigation and Time Interval Settings

Navigate between frames in a video or image sequence, and change the time interval of the video or image sequence. These controls are located in the bottom pane of the app.

Task	Action
Go to the next frame	Right arrow
Go to the previous frame	Left arrow
Go to the last frame	• PC: **End** • Mac: Hold **Fn** and press the right arrow
Go to the first frame	• PC: **Home** • Mac: Hold **Fn** and press the left arrow
Navigate through time interval boxes and frame navigation buttons	**Tab**
Commit time interval settings	Press **Enter** within the active time interval box (**Start Time**, **Current**, or **End Time**)

Labeling Window

Perform labeling actions, such as adding, moving, and deleting regions of interest (ROIs).

Task	Action
Undo labeling action	**Ctrl+Z**
Redo labeling action	**Ctrl+Y**
Select all rectangle and line ROIs	**Ctrl+A**
Select specific rectangle and line ROIs	Hold **Ctrl** and click the ROIs you want to select

Task	Action
Cut selected rectangle and line ROIs	**Ctrl+X**
Copy selected rectangle and line ROIs to clipboard	**Ctrl+C**
Paste copied rectangle and line ROIs • If a sublabel was copied, both the sublabel and its parent label are pasted. • If a parent label was copied, only the parent label is pasted, not its sublabels. For more details, see “Use Sublabels and Attributes to Label Ground Truth Data” (Computer Vision Toolbox).	**Ctrl+V**
Delete selected rectangle and line ROIs	**Delete**
Copy all pixel ROIs	**Ctrl+Shift+C**
Paste copied pixel ROIs	**Ctrl+Shift+V**
Fill all or all remaining pixels	**Shift+click**

Cuboid Drawing

Draw cuboids to label lidar point clouds. For examples on how to use these shortcuts to label lidar point clouds efficiently, see “Label Lidar Point Clouds for Object Detection” on page 2-38.

Note To enable these shortcuts, you must first click within the point cloud frame to select it.

Task	Action
Resize a cuboid uniformly across all dimensions before applying it to the point cloud	Hold **A** and move the scroll wheel up to increase size or down to decrease size
Resize a cuboid along only the *x*-dimension before applying it to the point cloud	Hold **X** and move the scroll wheel up to increase size or down to decrease size
Resize a cuboid along only the *y*-dimension before applying it to the point cloud	Hold **Y** and move the scroll wheel up to increase size or down to decrease size
Resize a cuboid along only the *z*-dimension before applying it to the point cloud	Hold **Z** and move the scroll wheel up to increase size or down to decrease size
Resize a cuboid after applying it to the point cloud	Click and drag one of the cuboid faces
Move a cuboid	Hold **Shift** and click and drag one of the cuboid faces The cuboid is translated along the dimension of the selected face.
Rotate the point cloud display	Hold **R** and click and drag the point cloud display

Polyline Drawing

Draw ROI line labels on a frame. ROI line labels are polylines, meaning that they are composed of one or more line segments.

Task	Action
Commit a polyline to the frame, excluding the currently active line segment	Press **Enter** or right-click while drawing the polyline
Commit a polyline to the frame, including the currently active line segment	Double-click while drawing the polyline A new line segment is committed at the point where you double-click.
Delete the previously created line segment in a polyline	**Backspace**
Cancel drawing and delete the entire polyline	**Escape**

Polygon Drawing

Draw polygons to label pixels on a frame.

Task	Action
Commit a polygon to the frame, excluding the currently active line segment	Press **Enter** or right-click while drawing the polygon The polygon closes up by forming a line between the previously committed point and the first point in the polygon.
Commit a polygon to the frame, including the currently active line segment	Double-click while drawing polygon The polygon closes up by forming a line between the point where you double-clicked and the first point in the polygon.
Remove the previously created line segment from a polygon	**Backspace**
Cancel drawing and delete the entire polygon	**Escape**

Zooming

Zooming in and out operates differently on the frames of an image signal than it does for the point clouds of a lidar signal.

Task	Action
Zoom in or out of an image frame	Move the scroll wheel up to zoom in or down to zoom out If the frame is in pan mode, then zooming is not supported. To enable zooming, in the upper-right corner of the frame, either click the Pan button to disable panning or click one of the zoom buttons.
Zoom in on specific section of an image frame	In the upper-right corner of the frame, click the Zoom In button and then draw a box around the section of the frame that you want to zoom in on Zooming in on a specific section of a point cloud is not supported.
Zoom in on a point cloud	Move the scroll wheel up, or click within the point cloud and move the cursor up or left Zooming in is supported in all modes (pan, zoom, and rotate).
Zoom out of a point cloud	Move the scroll wheel down, or click within the point cloud and move the cursor down or right Zooming out is supported in all modes (pan, zoom, and rotate).

App Sessions

Task	Action
Save current session	**Ctrl+S**

See Also

Ground Truth Labeler

More About

- “Get Started with the Ground Truth Labeler” on page 2-2

Control Playback of Signal Frames for Labeling

The **Ground Truth Labeler** app enables you to label multiple image or lidar point cloud signals simultaneously. When playing the signals or navigating between frames, you can control which frames display for each signal by changing the frame rate at which the signals display.

Signal Frames

The signals that you label are composed of frames. Each frame has a discrete timestamp associated with it, but the app treats each frame as a duration of [t_0, t_1), where:

- t_0 is the timestamp of the current frame.
- t_1 is the timestamp of the next frame.

When you label a frame that displays in the app, the label applies to the duration of that frame.

The intervals between frames are units of time, such as seconds. This time interval is the frame rate of the signal. Specify the timestamps for a signal as a `duration` vector. Each timestamp corresponds to the start of a frame.

Master Signal

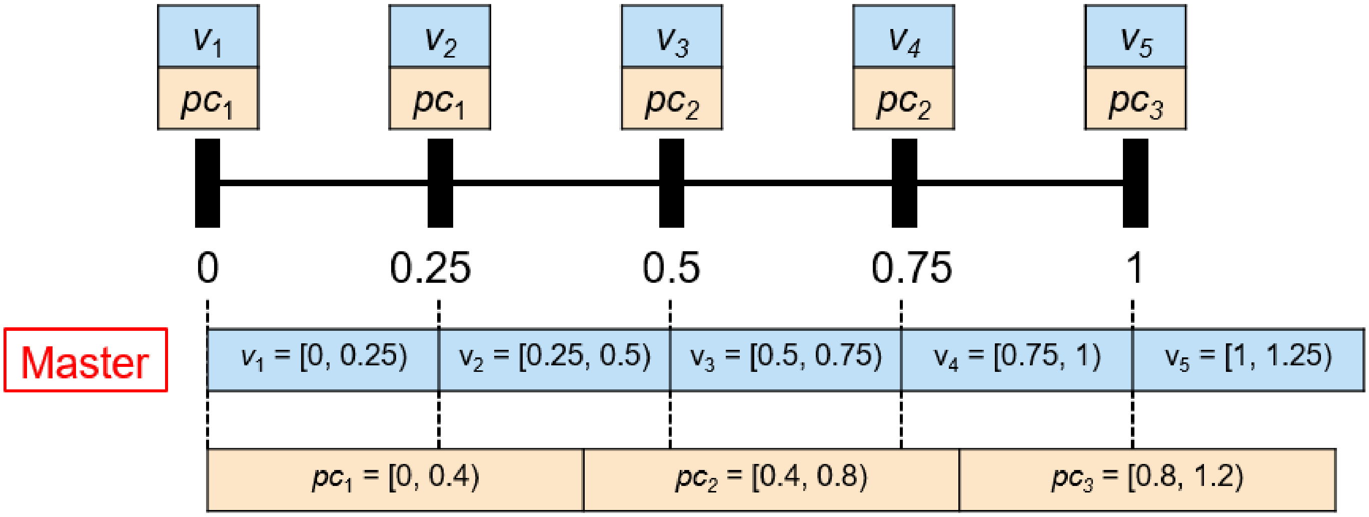
When you load multiple signals into a new app session, by default, the app designates the signal with the highest frame rate as the master signal. When you play back signals or navigate between frames, the app displays all frames from the master signal.

In the app, you can label signals only from within the time range of the master signal. When you view a frame from the master signal, the app displays the frames from all other signals that are at that timestamp. In this scenario, when navigating between frames, frames from signals with lower frame rates are sometimes repeated.

Consider an app session containing two signals: a video, *v*, and a lidar point cloud sequence, *pc*.

- The video has a frame rate of 4 frames per second, with a 0.25-second duration per frame. This signal is the master signal.
- The point cloud sequence has a frame rate of 2.5 frames per second, with a 0.4-second duration per frame.

This figure shows the frames that display over the first second in this scenario.



At time 0, the app displays the initial frame for each signal: v_1 for the video and pc_1 for the point cloud sequence. When you click the Next Frame button, the time skips to 0.25 seconds.

- For the video, the app displays the next frame, v_2.
- For the point cloud sequence, the app displays pc_1 again.

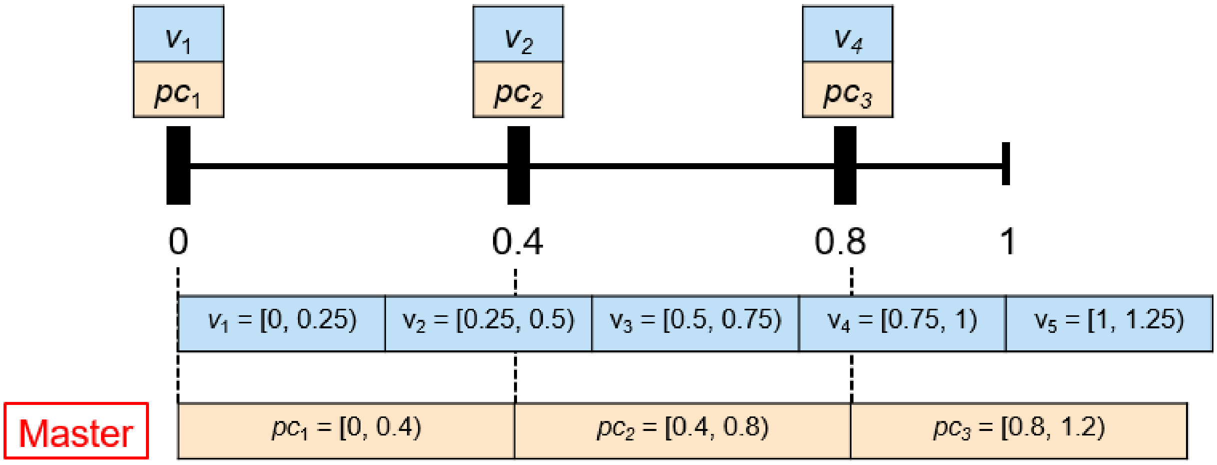
The app repeats the point cloud frame because the next point cloud frame, pc_2, does not start until 0.4 seconds. To display this frame, you must either set the **Current Time** parameter to 0.4 seconds or click the Next Frame button again to navigate to a time of 0.5 seconds.

Keep the signal with the highest frame rate as the master signal when you want to display and label all frames for all signals.

Change Master Signal

After loading signals, you can change the master signal from the Playback Control Settings dialog box. To open this dialog box, below the slider, click the clock settings button ⚙. Then, select **Master signal** and change the master signal to a different signal loaded into the app. When you change the master signal to a signal with a lower frame rate, frames from signals with higher frame rates are sometimes skipped.

Consider the app session described in the previous section, except with the point cloud sequence as the master signal.



When you skip from pc_2 to pc_3, the app skips over v_3 entirely. You can see v_3 only if you set **Current Time** to a time in the range [0.5, 0.75).

Designate the signal with the lowest frame rate as the master signal when you want to label signals only at synchronized times.

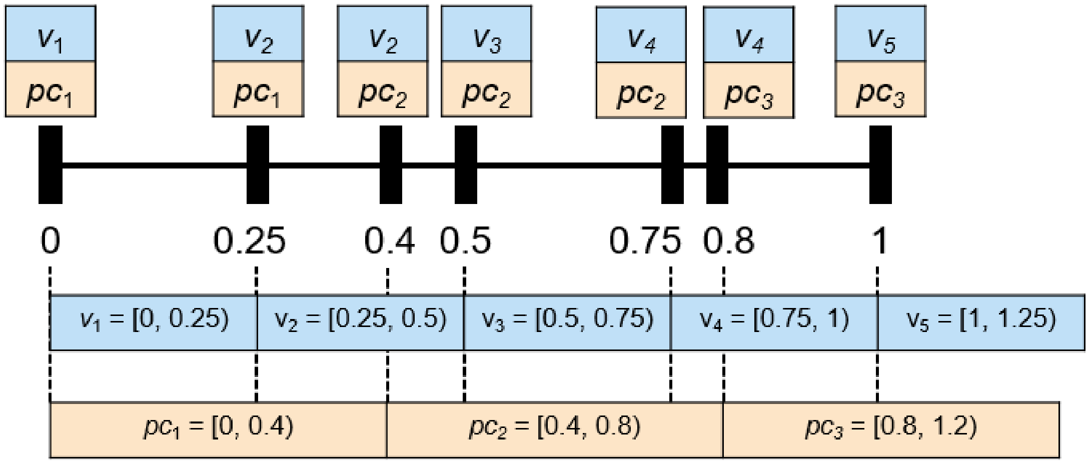
Changing the master signal after you begin labeling can affect existing scene labels. For example, suppose you apply a scene label to the entire time range of the master signal. If you change the master signal, the time range changes. If the new master signal has a longer duration, then the scene label no longer applies to the entire time range.

If you load a new signal into an app session that has a higher frame rate than the existing signals, the app does not automatically designate the new signal as the master signal. The app chooses a master signal only the first time you load signals into a session. To designate the new signal as the master signal, select that signal from the **Master signal** list in the Playback Control Settings dialog box.

Display All Timestamps

In the Playback Control Settings dialog box, you can select **All timestamps** to display all signals. Choose this option to verify and visualize the loaded frames. Do not select this option for labeling. When you display all timestamps, the navigation between frames is uneven and the frames of multiple signals are repeated.

Consider the app session described in the previous sections, except with all timestamps displaying. This figure shows the frames that display.



Specify Timestamps

You can specify your own timestamp vector and use those timestamps as the ones that the app uses to navigate between frames. In the Playback Control Settings dialog box, select **Timestamps from workspace**, click the **From Workspace** button, and specify a `duration` vector from the MATLAB workspace.

Frame Display and Automation

When you select a signal for automation, in the automation session, the app displays all frames of the selected signal for the specified time interval. Because you can automate only one signal at a time, the app plays back the signal frames at the frame rate for that signal.

See Also

`duration` | `groundTruthMultisignal`

More About

- "Load Ground Truth Signals to Label" on page 2-4

Label Lidar Point Clouds for Object Detection

The **Ground Truth Labeler** app enables you to label point cloud data obtained from lidar sensors. To label point clouds, you use cuboids, which are 3-D bounding boxes that you draw around the points in a point cloud. You can use cuboid labels to create ground truth data for training object detectors.

This example walks you through labeling lidar point cloud data by using cuboids.

Set Up Lidar Point Cloud Labeling

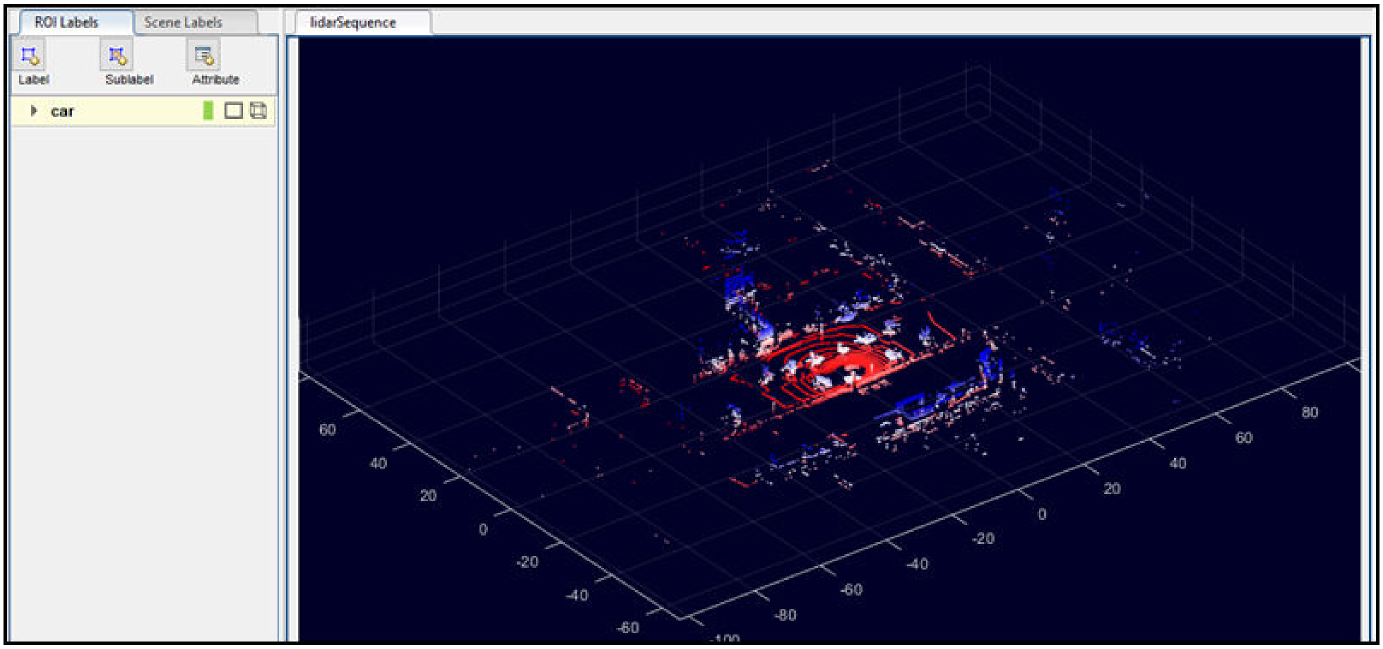
Load a point cloud sequence into the app and define a cuboid label.

1. Open the **Ground Truth Labeler** app. At the MATLAB command prompt, enter this command.

   ```
   groundTruthLabeler
   ```
2. On the app toolstrip, select **Open** > **Add Signals**.
3. In the Add/Remove Signal dialog box, set **Source Type** to `Point Cloud Sequence`.
4. In the **Folder Name** parameter, browse for the `lidarSequence` folder, which contains the point cloud sequence. *`matlabroot`* is the full path to your MATLAB installation folder, as returned by the `matlabroot` function.

   ```
   matlabroot\toolbox\driving\drivingdata\lidarSequence
   ```
5. Click **Add Source** to load the point cloud sequence, using the default timestamps. Then, click **OK** to close the Add/Remove Signal dialog box. The app displays the first point cloud in the sequence.
6. In the **ROI Labels** pane on the left side of the app, click **Label**.
7. Create a `Rectangle/Cuboid` label named `car`. Click **OK**.

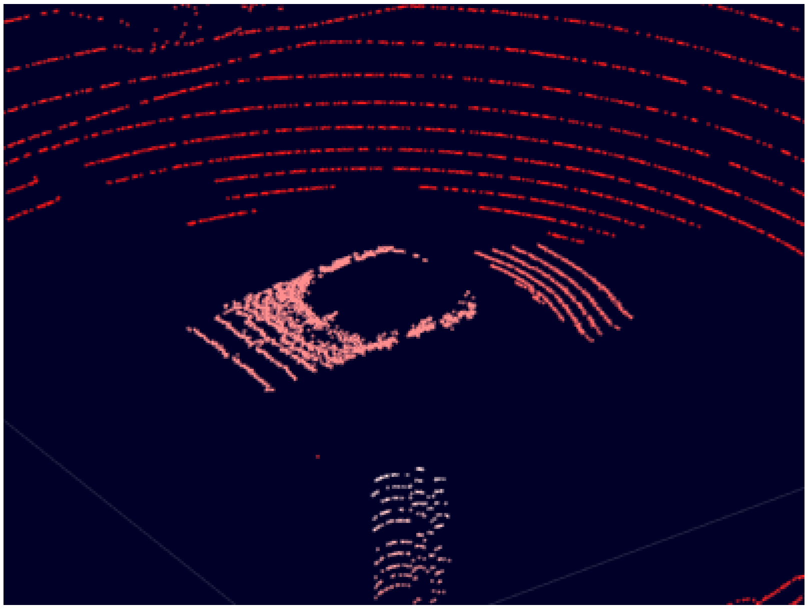
This figure shows the **Ground Truth Labeler** app setup after following these steps.



Zoom, Pan, and Rotate Frame

The zoom, pan, and 3-D rotation options help you locate and label objects of interest in a point cloud. Use these tools to zoom in and center on the ego vehicle in the first point cloud frame. The ego vehicle is located at the origin of the point cloud.

1. In the upper-right corner of the frame, click the Zoom In button 🔍.
2. Click the ego vehicle until you are zoomed in enough to see the points that make it up.

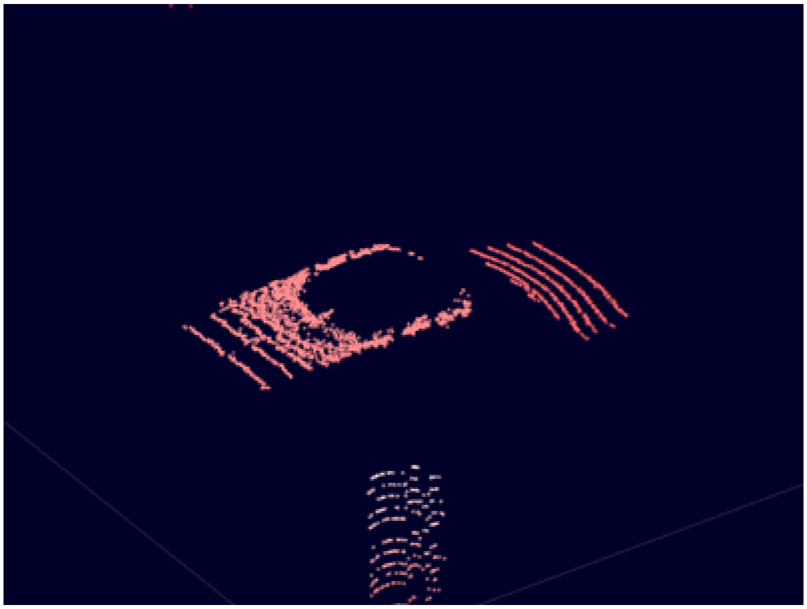


Optionally, you can use the Pan button ✋ or Rotate 3D button ⟲ to help you view more of the ego vehicle points.

Hide Ground

The point cloud data includes points from the ground, which can make it more difficult to isolate the ego vehicle points. The app provides an option to hide the ground by using the `segmentGroundFromLidarData` function.

Hide the ground points from the point cloud. On the app toolstrip, on the **Lidar** tab, click **Hide Ground**. This setting applies to all frames in the point cloud.



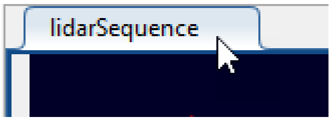
This option only hides the ground from the display. It does not remove ground data from the point cloud. If you label a section of the point cloud containing hidden ground points, when you export the ground truth labels, those ground points are a part of that label.

To configure the ground hiding algorithm, click **Ground Settings** and adjust the options in the Hide Ground dialog box.

Label Cuboid

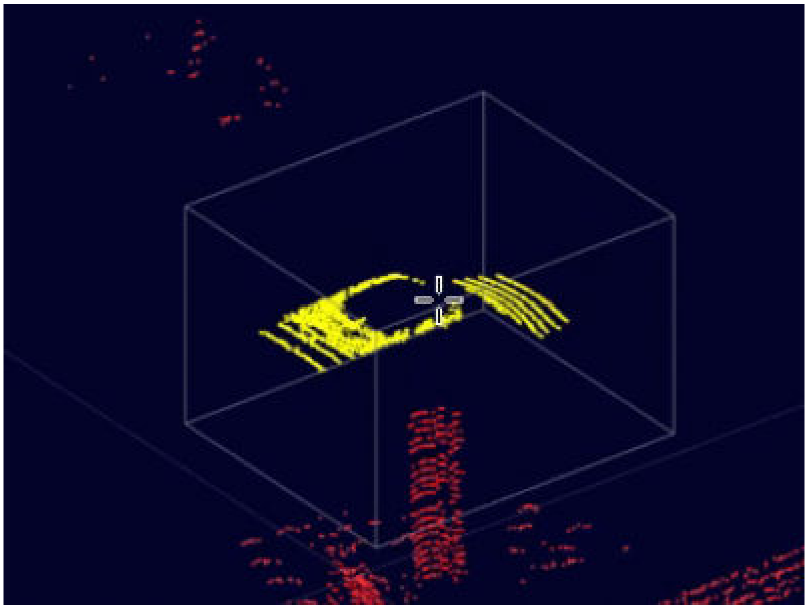
Label the ego vehicle by using a cuboid label.

1. In the **ROI Labels** pane on the left, click the **car** label.
2. Select the lidar point sequence frame by clicking the **lidarSequence** tab.

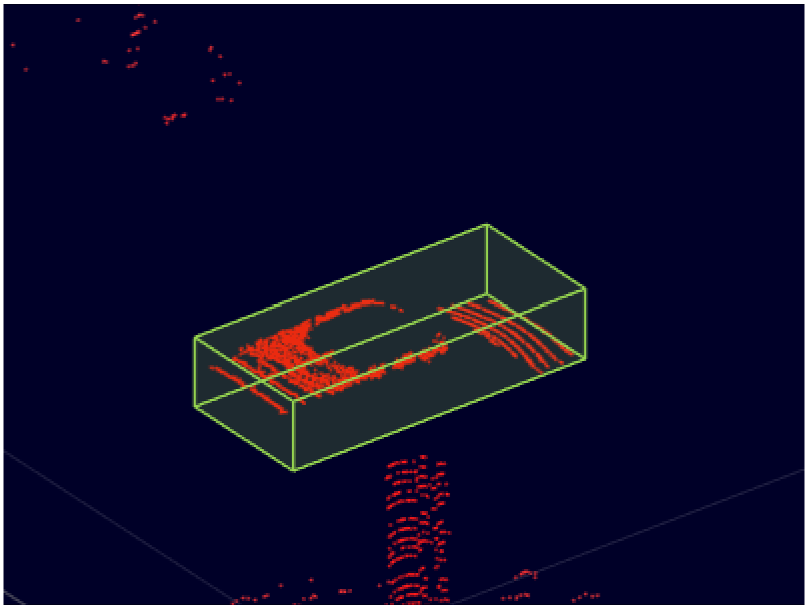
   Note To enable the labeling keyboard shortcuts, you must first select the signal frame.
3. Move the pointer over the ego vehicle until the gray preview cuboid encloses the ego vehicle points. The points enclosed in the preview cuboid highlight in yellow.

   To resize the preview cuboid, hold the **A** key and move the mouse scroll wheel up or down.



Optionally, to resize the preview cuboid in only the *x*-, *y*-, or *z*-direction, move the scroll wheel up and down while holding the **X**, **Y**, or **Z** key, respectively.

4 Click the signal frame to draw the cuboid. Because the **Shrink to Fit** option is selected by default on the app toolstrip, the cuboid shrinks to fit the points within it.



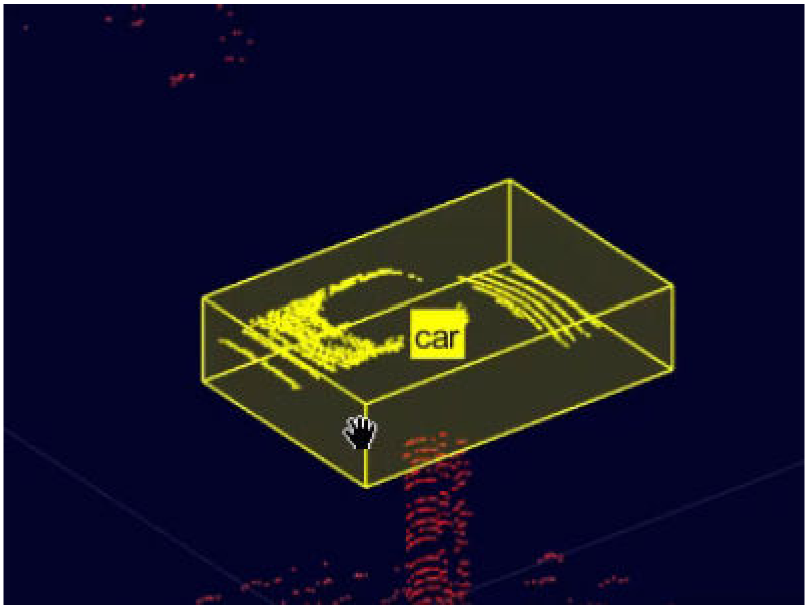
For more control over the labeling of point clouds, on the app toolstrip, click **Snap to Cluster**. When you label with this option selected, the cuboid snaps to the nearest point cloud cluster by using the `segmentLidarData` function. To configure point cloud clustering, click **Cluster Settings** and adjust the options in the dialog box.

Modify Cuboid Label

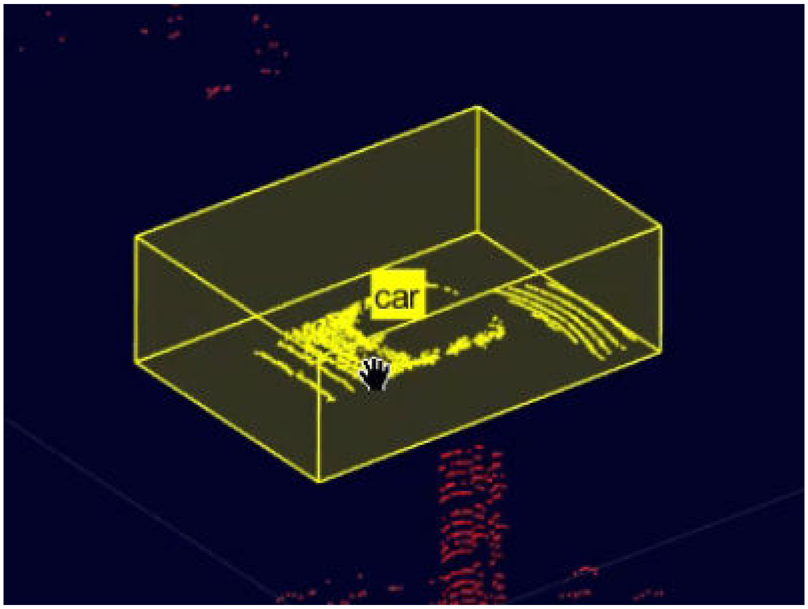
After drawing a cuboid label, you can resize or move the cuboid to make the label more accurate. For example, in the previous procedure, the **Shrink to Fit** option shrinks the cuboid label to fit the detected ego vehicle points. The actual ego vehicle is slightly larger than this cuboid. Expand the size of this cuboid until it more accurately reflects the size of the ego vehicle.

1 To enable the point cloud labeling keyboard shortcuts, verify that the **lidarSequence** tab is selected.

2 In the signal frame, click the drawn cuboid label. Drag the faces to expand the cuboid.



3 Move the cuboid until it is centered on the ego vehicle. Hold **Shift** and drag the faces of the cuboid.



Apply Cuboids to Multiple Frames

When labeling objects between frames, you can copy cuboid labels and paste them to other frames.

1. Select the cuboid for the ego vehicle and press **Ctrl+C** to copy it.
2. At the bottom of the app, click the Next Frame button ⏭ to navigate to the next frame.
3. Press **Ctrl+V** to paste the cuboid onto the frame.

You can also use an automation algorithm to apply a label to multiple frames. The app provides a built-in temporal interpolation algorithm for labeling point clouds in intermediate frames. For an example that shows that how to apply this automation algorithm, see “Label Ground Truth for Multiple Signals” on page 2-9.

Configure Display

The app provides additional options for configuring the display of signal frames.

Change Colormap

For additional control over the point cloud display, on the **Lidar** tab, you can change the colormap options. You can also change the colormap values by changing the **Colormap Value** parameter, which has these options:

- `Z Height` — Colormap values increase along the *z*-axis. Select this option when finding objects that are above the ground, such as traffic signs.
- `Radial Distance` — Colormap values increase away from the point cloud origin. Select this option when finding objects that are far from the origin.

Change Views

On the **Lidar** tab of the app toolstrip, the **Camera View** section contains options for changing the perspective from which you view the point cloud. These views are centered at the point cloud origin, which is the assumed position of the ego vehicle.

You can select from these views:

- **Bird's Eye View** — View the point cloud from directly above the ego vehicle.
- **Chase View** — View the point cloud from a few meters behind the ego vehicle.
- **Ego View** — View the point cloud from inside the ego vehicle.

These views assume that the vehicle is traveling along the positive *x*-direction of the point cloud. If the vehicle is traveling in a different direction, set the appropriate option in the **Ego Direction** parameter.

Use these views when verifying your point cloud labels. Avoid using these views while labeling. Instead, use the default view and locate objects to label by using the pan, zoom, and rotation options.

See Also

More About

- “Get Started with the Ground Truth Labeler” on page 2-2
- “Keyboard Shortcuts and Mouse Actions for Ground Truth Labeler” on page 2-30

Create Class for Loading Custom Ground Truth Data Sources

In the **Ground Truth Labeler** app, you can label signals from image and point cloud data sources. These sources include videos, image sequences, point cloud sequences, Velodyne packet capture (PCAP) files, and rosbags. To load data sources that the app does not natively support, you can create a class to load that source into the app.

This example shows how to use one of the predefined data source classes that load signals from data sources into the **Ground Truth Labeler** app: the `vision.labeler.loading.PointCloudSequenceSource` class. The app uses this specific class to load sequences of point cloud data (PCD) or polygon (PLY) files from a folder.

To get started, open the `vision.labeler.loading.PointCloudSequenceSource` class. Use the properties and methods described for this class to help you write your own custom class.

```
edit vision.labeler.loading.PointCloudSequenceSource
```

Custom Class Folder

The **Ground Truth Labeler** app recognizes data source classes only if those files are in a `+vision/+labeler/+loading` folder that is on the MATLAB search path.

The `vision.labeler.loading.PointCloudSequenceSource` class and other predefined data source classes are stored in this folder.

```
matlabroot\toolbox\vision\vision\+vision\+labeler\+loading
```

In this path, *`matlabroot`* is the root of your MATLAB folder.

Save the data source classes that you create to this folder. Alternatively, create your own `+vision/+labeler/+loading` folder, add it to the MATLAB search path, and save your class to this folder.

Class Definition

Data source classes must inherit from the `vision.labeler.loading.MultiSignalSource` class. View the class definition for the `vision.labeler.loading.PointCloudSequenceSource` class.

```
classdef PointCloudSequenceSource < vision.labeler.loading.MultiSignalSource
```

When you load a point cloud sequence signal into the **Ground Truth Labeler** app, the app creates an instance of the class, that is, a `PointCloudSequenceSource` object. After labeling this signal in the app, when you export the labels, the exported `groundTruthMultisignal` object stores this `PointCloudSequenceSource` object in its `DataSource` property.

When defining your data source class, replace `PointCloudSequenceSource` with the name of your custom data source class.

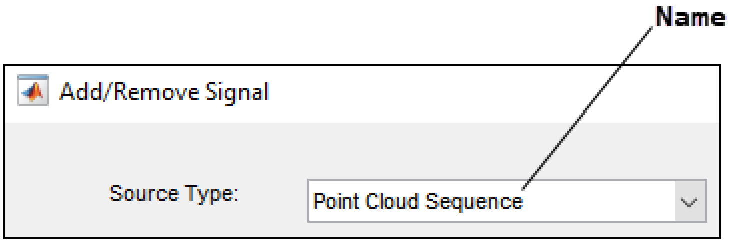
Class Properties

Data source classes must define these abstract, constant properties.

- `Name` — A string scalar specifying the type of the data source

- Description — A string scalar describing the class

In the **Ground Truth Labeler** app, when you load signals from the Add/Remove Signal dialog box, the Name string appears as an option in the **Source Type** parameter. This figure shows the Name string for the vision.labeler.loading.PointCloudSequenceSource class.



Source Type: Point Cloud Sequence

The Description string does not appear in the dialog box. However, both the Name and Description strings are stored as read-only properties in instances of this class.

This code shows the Name and Property strings for the vision.labeler.loading.PointCloudSequenceSource class.

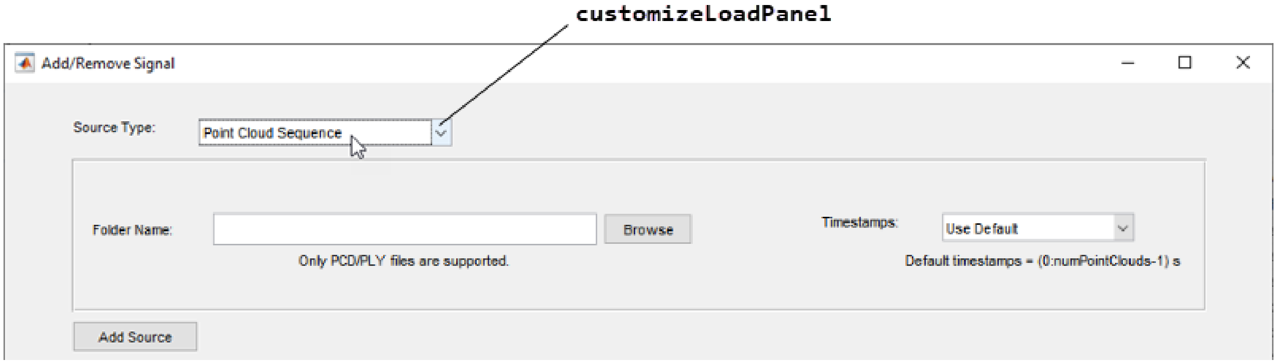
```
    properties (Constant)
        Name = "Point Cloud Sequence"

        Description = "A PointCloud sequence reader"
    end
```

When defining your data source class, define the Name and Description property values to match the name and description of your custom data source. You can also define any additional private properties that are specific to loading your data source. The source-specific properties for the vision.labeler.loading.PointCloudSequenceSource class are not shown in this example, but you can view them in the class file.

Method to Customize Load Panel

In data source classes, the customizeLoadPanel method controls the display of the panel for loading signals in the Add/Remove Signal dialog box of the app. This panel is a Panel object created by using the uipanel function. The panel contains the parameters and controls needed to load signals from data sources.

This figure shows the loading panel for the vision.labeler.loading.PointCloudSequenceSource class. In the **Source Type** list, when you select Point Cloud Sequence, the app calls the customizeLoadPanel method and loads the panel for point cloud sequences.



customizeLoadPanel

Source Type: Point Cloud Sequence

Timestamps: Use Default

Add/Remove Signal
Folder Name:
Browse
Only PCD/PLY files are supported.
Default timestamps = (0:numPointClouds-1) s
Add Source

This code shows the `customizeLoadPanel` method for the `vision.labeler.loading.PointCloudSequenceSource` class. This method uses the `uicontrol` function to define the text, buttons, and parameters in the panel.

```
        function customizeLoadPanel(this, panel)
            this.Panel = panel;

            computePositions(this);

            this.FolderPathText = uicontrol('Parent', this.Panel,...
                'Style', 'text',...
                'String', 'Folder Name: ',...
                'Position', this.FolderPathTextPos,...
                'HorizontalAlignment', 'left',...
                'Tag', 'fileText');

            this.FolderPathBox = uicontrol('Parent', this.Panel,...
                'Style', 'edit',...
                'String', '',...
                'Position', this.FolderPathBoxPos,...
                'Tag', 'fileEditBox');

            this.FolderTextBox = uicontrol('Parent', this.Panel,...
                'Style', 'Text',...
                'String', 'Only PCD/PLY files are supported.',...
                'Position', this.FolderTextPos,...
                'Tag', 'fileText');

            this.FolderBrowseButton = uicontrol('Parent', this.Panel,...
                'Style', 'togglebutton',...
                'String', 'Browse',...
                'Position', this.FolderBrowseButtonPos,...
                'Callback', @this.browseButtonCallback,...
                'Tag', 'browseBtn');

            this.TimeStampsText = uicontrol('Parent', this.Panel,...
                'Style', 'text',...
                'String', 'Timestamps: ',...
                'Position', this.TimeStampsTxtPos,...
                'HorizontalAlignment', 'left',...
                'Tag', 'timeStampTxt');

            this.TimeStampsDropDown = uicontrol('Parent', this.Panel,...
                'Style', 'popupmenu',...
                'String', ["Use Default", "From Workspace"],...
                'Position', this.TimeStampsDropDownPos,...
                'Callback', @this.timeStampsDropDownCallback,...
                'Tag', 'timeStampSourceSelectList');

            this.TimeStampsNote = uicontrol('Parent', this.Panel,...
                'Style', 'text',...
                'String', 'Default timestamps = (0:numPointClouds-1) s',...
                'Position', this.TimeStampsNotePos,...
                'HorizontalAlignment', 'left',...
                'Tag', 'timeStampNote');
        end
```

When developing this method or other data source methods, you can use the static method `loadPanelChecker` to preview the display and functionality of the loading dialog box for your custom data source. This method does not require you to have an app session open to use it. For example, use the `loadPanelChecker` method with the `vision.labeler.loading.PointCloudSequence` class.

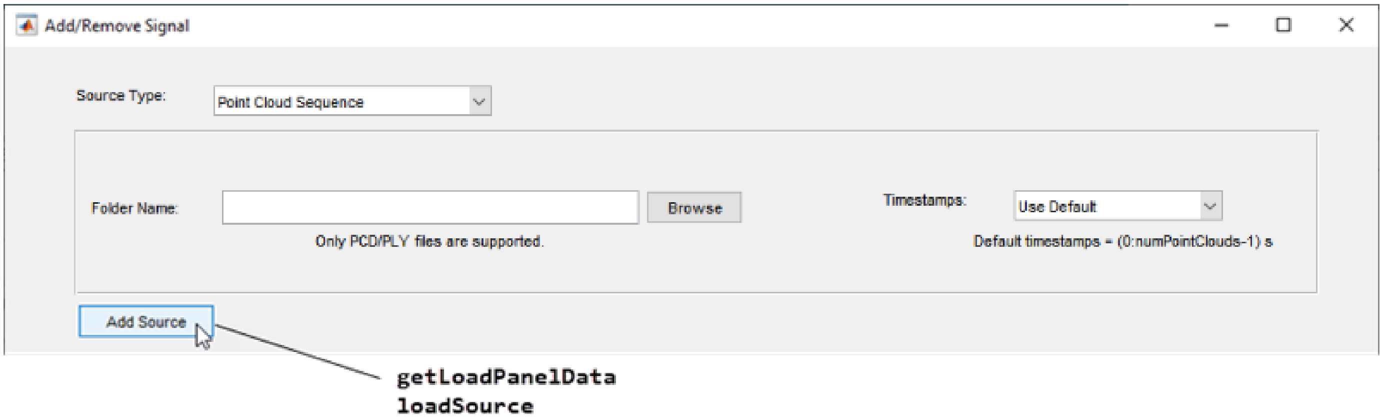
```
vision.labeler.loading.PointCloudSequenceSource.loadPanelChecker
```

Methods to Get Load Panel Data and Load Data Source

In the Add/Remove Signal dialog box, after you browse for a signal, set the necessary parameters, and click **Add Source**, the app calls these two methods in succession.

- `getLoadPanelData` — Get the data entered into the panel.
- `loadSource` — Load the data into the app.

This figure shows the relationship between these methods and the **Add Source** button when loading a point cloud sequence signal by using the `vision.labeler.loading.PointCloudSequenceSource` class.



When defining a custom data source, you must define the `getLoadPanelData` method, which returns these outputs.

- `sourceName` — The name of the data source
- `sourceParams` — A structure containing fields with information required to load the data source

This code shows the `getLoadPanelData` method for the `vision.labeler.loading.PointCloudSequenceSource` class. This method sets `sourceName` to the name entered in the **Folder Name** parameter of the dialog box and `sourceParams` to an empty structure. If the **Timestamps** parameter is set to `From Workspace` and has timestamps loaded, then the app populates this structure with those timestamps.

```
function [sourceName, sourceParams] = getLoadPanelData(this)
    sourceName = string(this.FolderPathBox.String);
    sourceParams = struct();
end
```

You must also define the `loadSource` method in your custom data class. This method must take the `sourceName` and `sourceParams` returned from the `getLoadPanelData` method as inputs. This method must also populate these properties, which are stored in the instance of the data source object that the app creates.

- `SignalName` — String identifiers for each signal in a data source
- `SignalType` — An array of `vision.labeler.loading.SignalType` enumerations defining the type of each signal in the data source
- `Timestamp` — A vector or cell array of timestamps for each signal in the data source
- `SourceName` — The name of the data source
- `SourceParams` — A structure containing fields with information required to load the data source

This code shows the `loadSource` method for the `vision.labeler.loading.PointCloudSequenceSource` class. This method performs these actions.

1. Check that the point cloud sequence has the correct extension and save the information required for reading the point clouds into a `fileDatastore` object.
2. Set the `Timestamp` property of the data source object.
   - If timestamps are loaded from a workspace variable (**Timestamps** = `From workspace`), then the method sets `Timestamp` to the timestamps stored in the `sourceParams` input.
   - If timestamps are derived from the point cloud sequence itself (**Timestamps** = `Use Default`), then the method sets `Timestamp` to a `duration` vector of seconds, with one second per point cloud.
3. Validate the loaded point cloud sequence.
4. Set the `SignalName` property to the name of the data source folder.
5. Set the `SignalType` property to the `PointCloud` signal type.
6. Set the `SourceName` and `SourceParams` properties to the `sourceName` and `sourceParams` outputs, respectively.

```
function loadSource(this, sourceName, sourceParams)

    % Load file
    ext = {'.pcd', '.ply'};
    this.Pcds = fileDatastore(sourceName,'ReadFcn', @pcread, 'FileExtensions', ext);

    % Populate timestamps

    if isempty(this.Timestamp)
        if isfield(sourceParams, 'Timestamps')
```

```
                    setTimestamps(this, sourceParams.Timestamps);
                else
                    this.Timestamp = {seconds(0:1:numel(this.Pcds.Files)-1)'};
                end
            else
                if ~iscell(this.Timestamp)
                    this.Timestamp = {this.Timestamp};
                end
            end

            import vision.internal.labeler.validation.*
            checkPointCloudSequenceAndTimestampsAgreement(this.Pcds,this.Timestamp{1});

            % Populate signal names and types
            [~, folderName, ~] = fileparts(sourceName);

            this.SignalName = makeValidName(this, string(folderName), "pointcloudSequence_");
            this.SignalType = vision.labeler.loading.SignalType.PointCloud;

            this.SourceName = sourceName;
            this.SourceParams = sourceParams;
        end
```

Method to Read Frames

The last required method that you must define is the `readFrame` method. This method reads a frame from a signal stored in the data source. The app calls this method each time you navigate to a new frame. The index to a particular timestamp in the `Timestamp` property is passed to this method.

This code shows the `readFrame` method for the `vision.labeler.loading.PointCloudSequenceSource` class. The method reads frames from the point cloud sequence by using the `pcread` function.

```
        function frame = readFrame(this, signalName, index)
            if ~strcmpi(signalName, this.SignalName)
                frame = [];
            else
                frame = pcread(this.Pcds.Files{index});
            end
        end
```

You can also define any additional private properties that are specific to loading your data source. The source-specific methods for the `vision.labeler.loading.PointCloudSequenceSource` class are not shown in this example but you can view them in the class file.

Use Predefined Data Source Classes

This example showed how to use the `vision.labeler.loading.PointCloudSequenceSource` class to help you create your own custom class. This table shows the complete list of data source classes that you can use as starting points for your own class.

Class	Data Source Loaded by Class	Command to View Class Source Code
`vision.labeler.loading.VideoSource`	Video file	`edit vision.labeler.loading.VideoSource`
`vision.labeler.loading.ImageSequenceSource`	Image sequence folder	`edit vision.labeler.loading.ImageSequen`
`vision.labeler.loading.VelodyneLidarSource`	Velodyne packet capture (PCAP) file	`edit vision.labeler.loading.VelodyneLid`
`vision.labeler.loading.RosbagSource`	Rosbag file	`edit vision.labeler.loading.RosbagSourc`
`vision.labeler.loading.PointCloudSequenceSource`	Point cloud sequence folder	`edit vision.labeler.loading.PointCloudS`
`vision.labeler.loading.CustomImageSource`	Custom image format	`edit vision.labeler.loading.CustomImage`

See Also

Apps
Ground Truth Labeler

Classes
`vision.labeler.loading.MultiSignalSource`

Objects
`groundTruthMultisignal`

Tracking and Sensor Fusion

Visualize Sensor Data and Tracks in Bird's-Eye Scope

The **Bird's-Eye Scope** visualizes signals from your Simulink model that represent aspects of a driving scenario. Using the scope, you can analyze:

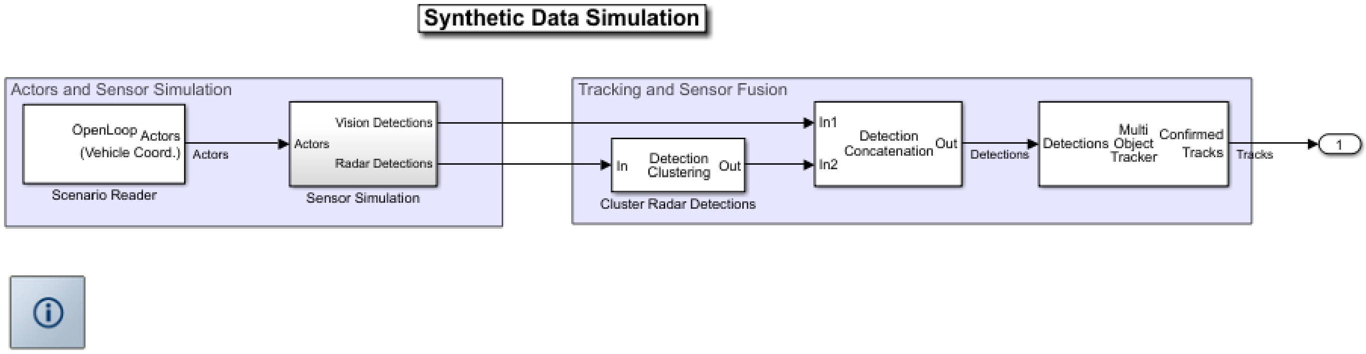
- Sensor coverages of vision, radar, and lidar sensors
- Sensor detections of actors and lane boundaries
- Tracks of moving objects in the scenario

This example shows you how to display these signals on the scope and analyze the signals during simulation.

Open Model and Scope

Open a model containing signals for sensor detections and tracks. This model is used in the “Sensor Fusion Using Synthetic Radar and Vision Data in Simulink” example. Also add the file folder of the model to the MATLAB search path.

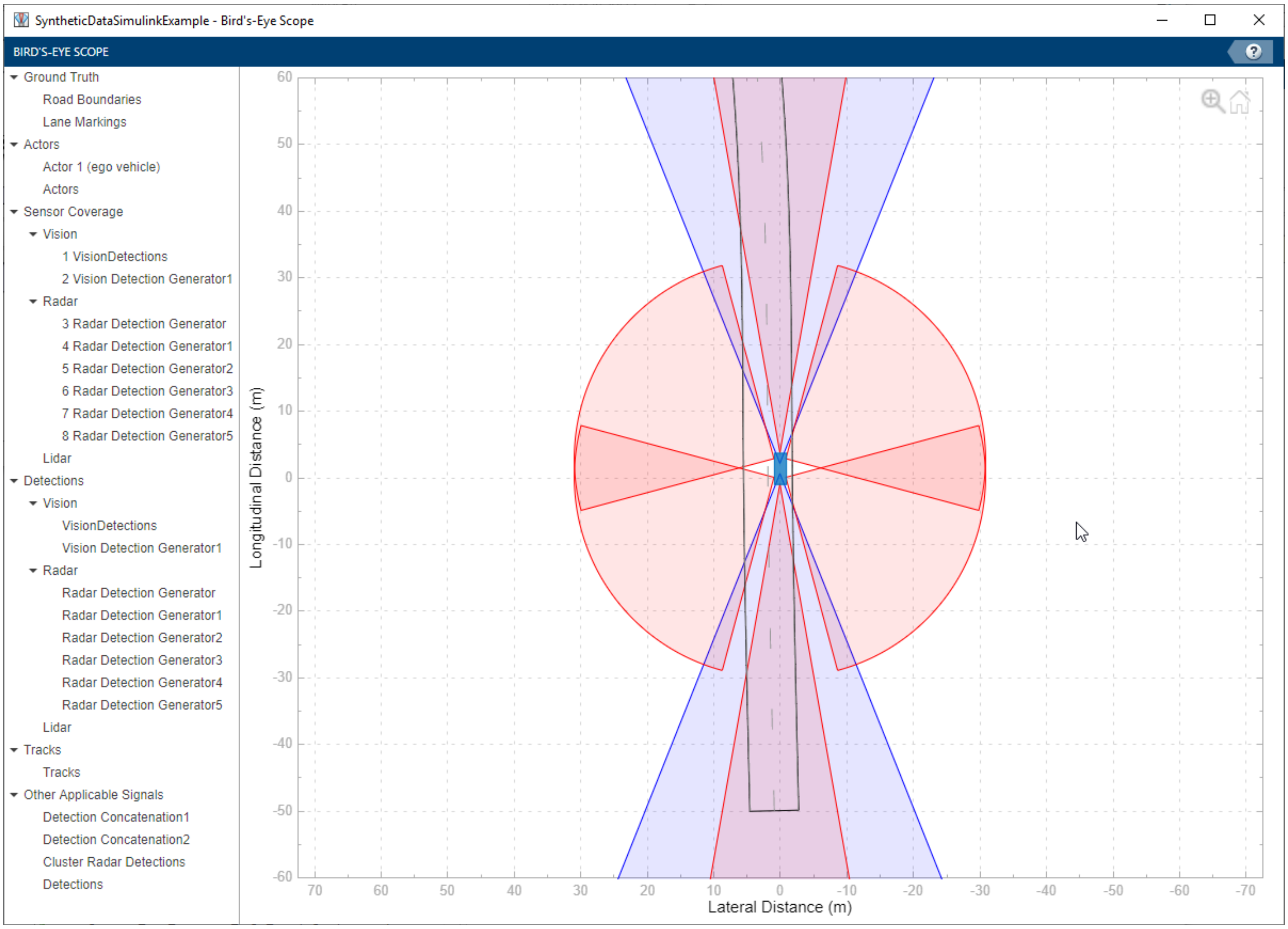
```
addpath(genpath(fullfile(matlabroot,'examples','driving')))
open_system('SyntheticDataSimulinkExample')
```



Open the scope from the Simulink toolstrip. Under **Review Results**, click **Bird's-Eye Scope**.

Find Signals

When you first open the **Bird's-Eye Scope**, the scope canvas is blank and displays no signals. To find signals from the opened model that the scope can display, on the scope toolstrip, click **Find Signals**. The scope updates the block diagram and automatically finds the signals in the model.


SyntheticDataSimulinkExample - Bird's-Eye Scope
BIRD'S-EYE SCOPE
Ground Truth
Road Boundaries
Lane Markings
Actors
Actor 1 (ego vehicle)
Actors
Sensor Coverage
Vision
1 VisionDetections
2 Vision Detection Generator1
Radar
3 Radar Detection Generator
4 Radar Detection Generator1
5 Radar Detection Generator2
6 Radar Detection Generator3
7 Radar Detection Generator4
8 Radar Detection Generator5
Lidar
Detections
Vision
VisionDetections
Vision Detection Generator1
Radar
Radar Detection Generator
Radar Detection Generator1
Radar Detection Generator2
Radar Detection Generator3
Radar Detection Generator4
Radar Detection Generator5
Lidar
Tracks
Tracks
Other Applicable Signals
Detection Concatenation1
Detection Concatenation2
Cluster Radar Detections
Detections
Longitudinal Distance (m)
60
50
40
30
20
10
0
-10
-20
-30
-40
-50
-60
70
60
50
40
30
20
10
0
-10
-20
-30
-40
-50
-60
-70
Lateral Distance (m)

The left pane lists all the signals that the scope found. These signals are grouped based on their sources within the model.

Signal Group	Description	Signal Sources
Ground Truth	Road boundaries and lane markings in the scenario You cannot modify this group or any of its signals. To inspect large road networks, use the **World Coordinates View** window. See “Vehicle and World Coordinate Views”.	• Scenario Reader block

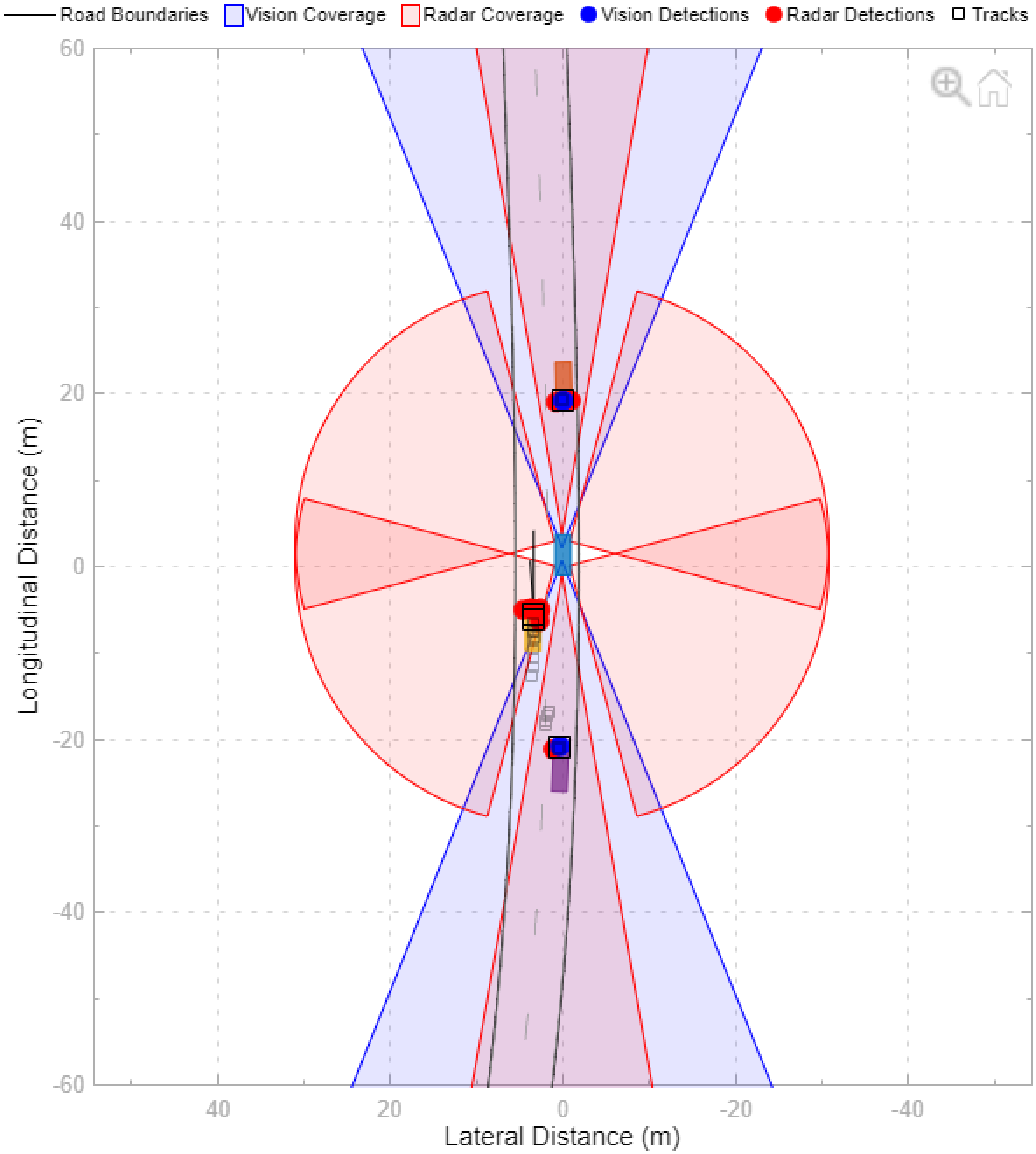
Signal Group	Description	Signal Sources
Actors	Actors in the scenario, including the ego vehicle You cannot modify this group or any of its signals or subgroups. To view actors that are located away from the ego vehicle, use the **World Coordinates View** window. See "Vehicle and World Coordinate Views".	• Scenario Reader block • Vision Detection Generator and Radar Detection Generator blocks (for actor profile information only, such as the length, width, and height of actors) • If actor profile information is not set or is inconsistent between blocks, the scope sets the actor profiles to the default actor profile values for each block. • The profile of the ego vehicle is always set to the default profile for each block.
Sensor Coverage	Coverage areas of vision, radar, and lidar sensors, sorted into **Vision**, **Radar**, and **Lidar** subgroups You can modify signals in this group. You can rename or delete subgroups but not the top-level **Sensor Coverage** group. You can also add subgroups and move signals between subgroups. If you delete a subgroup, its signals move to the top-level **Sensor Coverage** group.	• Vision Detection Generator block • Radar Detection Generator block • Simulation 3D Probabilistic Radar block • Simulation 3D Lidar block

Signal Group	Description	Signal Sources
Detections	Detections obtained from vision, radar, and lidar sensors, sorted into **Vision**, **Radar**, and **Lidar** subgroups You can modify signals in this group. You can rename or delete subgroups but not the top-level **Detections** group. You can also add subgroups and move signals between subgroups. If you delete a subgroup, its signals move to the top-level **Detections** group.	• Vision Detection Generator block • Radar Detection Generator block • Simulation 3D Probabilistic Radar block • Simulation 3D Lidar block
Tracks	Tracks of objects in the scenario You can modify signals in this group. You can rename or delete subgroups but not the top-level **Tracks** group. You can also add subgroups to this group and move signals into them. If you delete a subgroup, its signals move to the top-level **Tracks** group.	• Multi-Object Tracker block
Other Applicable Signals	Signals that the scope cannot automatically group, such as ones that combine information from multiple sensors You can modify signals in this group but you cannot add subgroups. Signals in this group do not display during simulation.	• Blocks that combine or cluster signals (such as the Detection Concatenation block) • Nonvirtual Simulink buses containing position and velocity information for detections and tracks • Vehicle To World and World To Vehicle blocks • Any blocks that create buses containing actor poses For details on the actor pose information required when creating these buses, see the **Actors** output port of the Scenario Reader block.

Before simulation but after clicking **Find Signals**, the scope canvas displays all **Ground Truth** signals except for non-ego actors and all **Sensor Coverage** signals. The non-ego actors and the signals under **Detections** and **Tracks** do not display until you simulate the model. The signals in **Other Applicable Signals** do not display during simulation. If you want the scope to display specific signals, move them into the appropriate group before simulation. If an appropriate group does not exist, create one.

Run Simulation

Simulate the model from within the **Bird's-Eye Scope** by clicking **Run**. The scope canvas displays the detections and tracks. To display the legend, on the scope toolstrip, click **Legend**.



During simulation, you can perform these actions:

- Inspect detections, tracks, sensor coverage areas, and ego vehicle behavior. The default view displays the simulation in vehicle coordinates and is centered on the ego vehicle. To view the wider area around the ego vehicle, or to view other parts of the scenario, on the scope toolstrip, click **World Coordinates**. The **World Coordinates View** window displays the entire scenario, with the ego vehicle circled. This circle is not a sensor coverage area. To return to the default display of either window, move your pointer over the window, and in the upper-right corner, click the home button ⌂ that appears. For more details on these views, see “Vehicle and World Coordinate Views”.

- Update signal properties. To access the properties of a signal, first select the signal from the left pane. Then, on the scope toolstrip, click **Properties**. Using these properties, you can, for example, show or hide sensor coverage areas or detections. In addition, to highlight certain sensor coverage areas, you can change their color or transparency.
- Update **Bird's-Eye Scope** settings, such as changing the axes limits of the **Vehicle Coordinates View** window or changing the display of signal names. On the scope toolstrip, click **Settings**. You cannot change the **Track position selector** and **Track velocity selector** settings during simulation. For more details, see the “Settings” section of the **Bird's-Eye Scope** reference page.

After simulation, you can hide certain detections or tracks for the next simulation. In the left pane, under **Detections** or **Tracks**, right-click the signal you want to hide. Then, select **Move to Other Applicable** to move that signal into the **Other Applicable Signals** group. To hide sensor coverage areas, select the corresponding signal in the left pane, and on the **Properties** tab, clear the **Show Sensor Coverage** parameter. You cannot hide **Ground Truth** signals during simulation.

Organize Signal Groups (Optional)

To further organize the signals, you can rename signal groups or move signals into new groups. For example, you can rename the **Vision** and **Radar** subgroups to **Front of Car** and **Back of Car**. Then you can drag the signals as needed to move them into the appropriate groups based on the new group names. When you drag a signal to a new group, the color of the signal changes to match the color assigned to its group.

You cannot rename or delete the top-level groups in the left pane, but you can rename or delete any subgroup. If you delete a subgroup, its signals move to the top-level group.

Update Model and Rerun Simulation

After you run the simulation, modify the model and inspect how the changes affect the visualization on the **Bird's-Eye Scope**. For example, in the Sensor Simulation subsystem of the model, open the two Vision Detection Generator blocks. These blocks have sensor indices of `1` and `2`, respectively. On the **Measurements** tab of each block, reduce the **Maximum detection range (m)** parameter to `50`. To see how the sensor coverage changes, rerun the simulation.

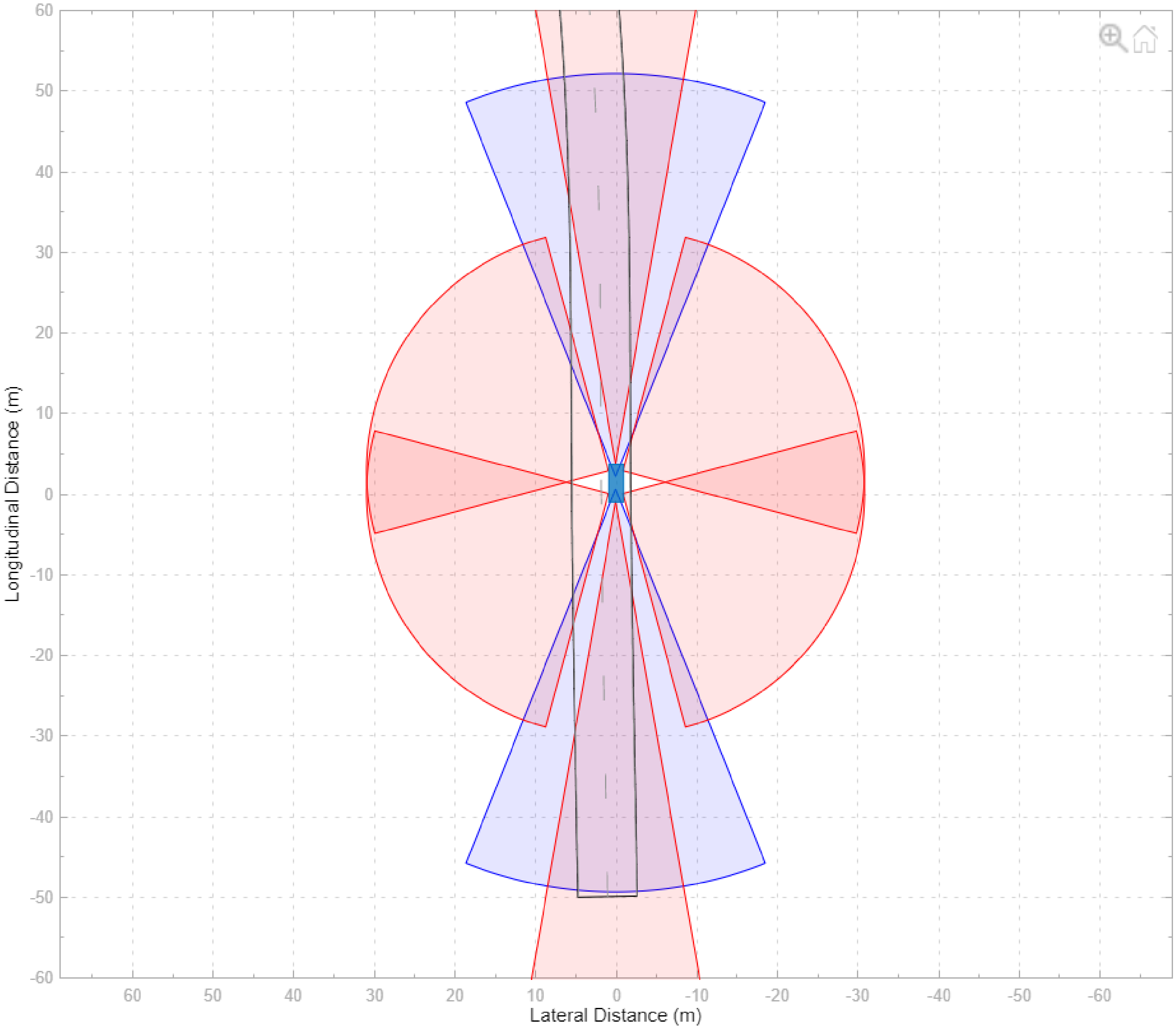
When you modify block parameters, you can rerun the simulation without having to find signals again. If you add or remove blocks, ports, or signal lines, then you must click **Find Signals** again before rerunning the simulation.

Save and Close Model

Save and close the model. The settings for the **Bird's-Eye Scope** are also saved.

If you reopen the model and the **Bird's-Eye Scope**, the scope canvas is initially blank.

Click **Find Signals** to find the signals again and view the saved signal properties. For example, if you reduced the detection range in the previous step, the scope canvas displays this reduced range.



When you are done simulating the model, remove the model file folder from the MATLAB search path.

```
rmpath(genpath(fullfile(matlabroot,'examples','driving')))
```

See Also

Apps
Bird's-Eye Scope

Blocks
Detection Concatenation | Multi Object Tracker | Radar Detection Generator | Scenario Reader |
Simulation 3D Lidar | Simulation 3D Probabilistic Radar | Vision Detection Generator

Related Examples

- “Visualize 3D Simulation Sensor Coverages and Detections” on page 6-35
- “Sensor Fusion Using Synthetic Radar and Vision Data in Simulink”
- “Lateral Control Tutorial”
- “Autonomous Emergency Braking with Sensor Fusion”
- “Test Open-Loop ADAS Algorithm Using Driving Scenario” on page 5-94
- “Test Closed-Loop ADAS Algorithm Using Driving Scenario” on page 5-100

Linear Kalman Filters

In this section...

When you use a Kalman filter to track objects, you use a sequence of detections or measurements to construct a model of the object motion. Object motion is defined by the evolution of the state of the object. The Kalman filter is an optimal, recursive algorithm for estimating the track of an object. The filter is recursive because it updates the current state using the previous state, using measurements that may have been made in the interval. A Kalman filter incorporates these new measurements to keep the state estimate as accurate as possible. The filter is optimal because it minimizes the mean-square error of the state. You can use the filter to predict future states or estimate the current state or past state.

State Equations

For most types of objects tracked in Automated Driving Toolbox, the state vector consists of one-, two- or three-dimensional positions and velocities.

Start with Newton equations for an object moving in the *x*-direction at constant acceleration and convert these equations to space-state form.

$$m\ddot{x} = f$$
$$\ddot{x} = \frac{f}{m} = a$$

If you define the state as

$$x_1 = x$$
$$x_2 = \dot{x},$$

you can write Newton’s law in state-space form.

$$\frac{d}{dt}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} a$$

You use a linear dynamic model when you have confidence that the object follows this type of motion. Sometimes the model includes process noise to reflect uncertainty in the motion model. In this case, Newton’s equations have an additional term.

$$\frac{d}{dt}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} a + \begin{bmatrix} 0 \\ 1 \end{bmatrix} v_k$$

v_k is the unknown noise perturbations of the acceleration. Only the statistics of the noise are known. It is assumed to be zero-mean Gaussian white noise.

You can extend this type of equation to more than one dimension. In two dimensions, the equation has the form

$$\frac{d}{dt}\begin{bmatrix} x_1 \\ x_2 \\ y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \\ y_1 \\ y_2 \end{bmatrix} + \begin{bmatrix} 0 \\ a_x \\ 0 \\ a_y \end{bmatrix} + \begin{bmatrix} 0 \\ v_x \\ 0 \\ v_y \end{bmatrix}$$

The 4-by-4 matrix on the right side is the state transition model matrix. For independent *x*- and *y*-motions, this matrix is block diagonal.

When you transition to discrete time, you integrate the equations of motion over the length of the time interval. In discrete form, for a sample interval of *T*, the state-representation becomes

$$\begin{bmatrix} x_{1,\,k+1} \\ x_{2,\,k+1} \end{bmatrix} = \begin{bmatrix} 1 & T \\ 0 & 1 \end{bmatrix}\begin{bmatrix} x_{1,\,k} \\ x_{2,\,k} \end{bmatrix} + \begin{bmatrix} 0 \\ T \end{bmatrix}a + \begin{bmatrix} 0 \\ 1 \end{bmatrix}\tilde{v}$$

The quantity x_{k+1} is the state at discrete time *k+1*, and x_k is the state at the earlier discrete time, *k*. If you include noise, the equation becomes more complicated, because the integration of noise is not straightforward.

The state equation can be generalized to

$$x_{k+1} = F_k x_k + G_k u_k + v_k$$

F_k is the state transition matrix and G_k is the control matrix. The control matrix takes into account any known forces acting on the object. Both of these matrices are given. The last term represents noise-like random perturbations of the dynamic model. The noise is assumed to be zero-mean Gaussian white noise.

Continuous-time systems with input noise are described by linear stochastic differential equations. Discrete-time systems with input noise are described by linear stochastic differential equations. A state-space representation is a mathematical model of a physical system where the inputs, outputs, and state variables are related by first-order coupled equations.

Measurement Models

Measurements are what you observe about your system. Measurements depend on the state vector but are not always the same as the state vector. For instance, in a radar system, the measurements can be spherical coordinates such as range, azimuth, and elevation, while the state vector is the Cartesian position and velocity. For the linear Kalman filter, the measurements are always linear functions of the state vector, ruling out spherical coordinates. To use spherical coordinates, use the extended Kalman filter.

The measurement model assumes that the actual measurement at any time is related to the current state by

$$z_k = H_k x_k + w_k$$

w_k represents measurement noise at the current time step. The measurement noise is also zero-mean white Gaussian noise with covariance matrix *Q* described by $Q_k = E[n_k n_k^T]$.

Linear Kalman Filter Equations

Without noise, the dynamic equations are

$$x_{k+1} = F_k x_k + G_k u_k .$$

Likewise, the measurement model has no measurement noise contribution. At each instance, the process and measurement noises are not known. Only the noise statistics are known. The

$$z_k = H_k x_k$$

You can put these equations into a recursive loop to estimate how the state evolves and also how the uncertainties in the state components evolve.

Filter Loop

Start with a best estimate of the state, $x_{0/0}$, and the state covariance, $P_{0/0}$. The filter performs these steps in a continual loop.

1. Propagate the state to the next step using the motion equations.

   $$x_{k+1|k} = F_k x_{k|k} + G_k u_k .$$

   Propagate the covariance matrix as well.

   $$P_{k+1|k} = F_k P_{k|k} F_k^T + Q_k .$$

   The subscript notation *k+1|k* indicates that the quantity is the optimum estimate at the *k+1* step propagated from step *k*. This estimate is often called the *a priori* estimate.

   Then predict the measurement at the updated time.

   $$z_{k+1|k} = H_{k+1} x_{k+1|k}$$

2. Use the difference between the actual measurement and predicted measurement to correct the state at the updated time. The correction requires computing the Kalman gain. To do this, first compute the measurement prediction covariance (innovation)

   $$S_{k+1} = H_{k+1} P_{k+1|k} H_{k+1}^T + R_{k+1}$$

   Then the Kalman gain is

   $$K_{k+1} = P_{k+1|k} H_{k+1}^T S_{k+1}^{-1}$$

   and is derived from using an optimality condition.

3. Correct the predicted estimate with the measurement. Assume that the estimate is a linear combination of the predicted state and the measurement. The estimate after correction uses the subscript notation, *k+1|k+1*. is computed from

   $$x_{k+1|k+1} = x_{k+1|k} + K_{k+1}(z_{k+1} - z_{k+1|k})$$

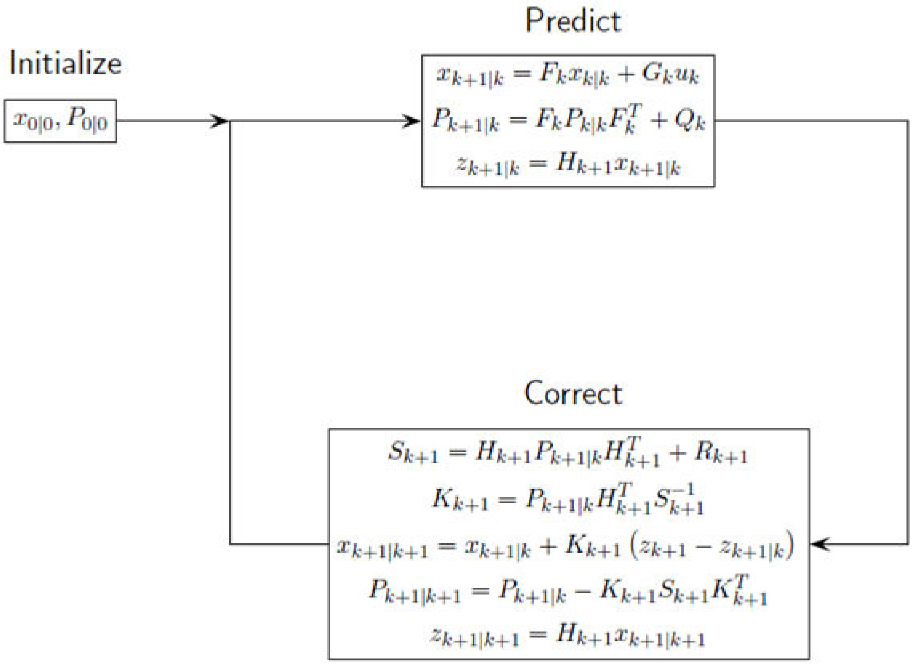
   where K_{k+1} is the Kalman gain. The corrected state is often called the *a posteriori* estimate of the state because it is derived after the measurement is included.

   Correct the state covariance matrix

$$P_{k+1|k+1} = P_{k+1|k} - K_{k+1}S_{k+1}K'_{k+1}$$

Finally, you can compute a measurement based upon the corrected state. This is not a correction to the measurement but is a best estimate of what the measurement would be based upon the best estimate of the state. Comparing this to the actual measurement gives you an indication of the performance of the filter.

This figure summarizes the Kalman loop operations.


Initialize
$x_{0|0}, P_{0|0}$
Predict
$x_{k+1|k} = F_k x_{k|k} + G_k u_k$
$P_{k+1|k} = F_k P_{k|k} F_k^T + Q_k$
$z_{k+1|k} = H_{k+1} x_{k+1|k}$
Correct
$S_{k+1} = H_{k+1} P_{k+1|k} H_{k+1}^T + R_{k+1}$
$K_{k+1} = P_{k+1|k} H_{k+1}^T S_{k+1}^{-1}$
$x_{k+1|k+1} = x_{k+1|k} + K_{k+1}\left(z_{k+1} - z_{k+1|k}\right)$
$P_{k+1|k+1} = P_{k+1|k} - K_{k+1} S_{k+1} K_{k+1}^T$
$z_{k+1|k+1} = H_{k+1} x_{k+1|k+1}$

Constant Velocity Model

The linear Kalman filter contains a built-in linear constant-velocity motion model. Alternatively, you can specify the transition matrix for linear motion. The state update at the next time step is a linear function of the state at the present time. In this filter, the measurements are also linear functions of the state described by a measurement matrix. For an object moving in 3-D space, the state is described by position and velocity in the *x*-, *y*-, and *z*-coordinates. The state transition model for the constant-velocity motion is

$$\begin{bmatrix} x_{k+1} \\ v_{x,k+1} \\ y_{k+1} \\ v_{y,k+1} \\ z_{k+1} \\ v_{z,k+1} \end{bmatrix} = \begin{bmatrix} 1 & T & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & T & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & T \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_k \\ v_{x,k} \\ y_k \\ v_{y,k} \\ z_k \\ v_{z,k} \end{bmatrix}$$

The measurement model is a linear function of the state vector. The simplest case is one where the measurements are the position components of the state.

$$\begin{bmatrix} m_{x,k} \\ m_{y,k} \\ m_{z,k} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x_k \\ v_{x,k} \\ y_k \\ v_{y,k} \\ z_k \\ v_{z,k} \end{bmatrix}$$

Constant Acceleration Model

The linear Kalman filter contains a built-in linear constant-acceleration motion model. Alternatively, you can specify the transition matrix for constant-acceleration linear motion. The transition model for linear acceleration is

$$\begin{bmatrix} x_{k+1} \\ v_{x,k+1} \\ a_{x,k+1} \\ y_{k+1} \\ v_{y,k+1} \\ a_{y,k+1} \\ z_{k+1} \\ v_{z,k+1} \\ a_{z,k+1} \end{bmatrix} = \begin{bmatrix} 1 & T & \frac{1}{2}T^2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & T & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & T & \frac{1}{2}T^2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & T & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & T & \frac{1}{2}T^2 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & T \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_k \\ v_{x,k} \\ a_{x,k} \\ y_k \\ v_{y,k} \\ a_{y,k} \\ z_k \\ v_{z,k} \\ a_{z,k} \end{bmatrix}$$

The simplest case is one where the measurements are the position components of the state.

$$\begin{bmatrix} m_{x,k} \\ m_{y,k} \\ m_{z,k} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_k \\ v_{x,k} \\ a_{x,k} \\ y_k \\ v_{y,k} \\ a_{y,k} \\ z_k \\ v_{z,k} \\ a_{y,k} \end{bmatrix}$$

See Also

Objects
`trackingKF`

Extended Kalman Filters

In this section...

Use an extended Kalman filter when object motion follows a nonlinear state equation or when the measurements are nonlinear functions of the state. A simple example is when the state or measurements of the object are calculated in spherical coordinates, such as azimuth, elevation, and range.

State Update Model

The extended Kalman filter formulation linearizes the state equations. The updated state and covariance matrix remain linear functions of the previous state and covariance matrix. However, the state transition matrix in the linear Kalman filter is replaced by the Jacobian of the state equations. The Jacobian matrix is not constant but can depend on the state itself and time. To use the extended Kalman filter, you must specify both a state transition function and the Jacobian of the state transition function.

Assume there is a closed-form expression for the predicted state as a function of the previous state, controls, noise, and time.

$$x_{k+1} = f(x_k, u_k, w_k, t)$$

The Jacobian of the predicted state with respect to the previous state is

$$F^{(x)} = \frac{\partial f}{\partial x}.$$

The Jacobian of the predicted state with respect to the noise is

$$F^{(w)} = \frac{\partial f}{\partial w_i}.$$

These functions take simpler forms when the noise enters linearly into the state update equation:

$$x_{k+1} = f(x_k, u_k, t) + w_k$$

In this case, $F^{(w)} = 1_M$.

Measurement Model

In the extended Kalman filter, the measurement can be a nonlinear function of the state and the measurement noise.

$$z_k = h(x_k, v_k, t)$$

The Jacobian of the measurement with respect to the state is

$$H^{(x)} = \frac{\partial h}{\partial x}\,.$$

The Jacobian of the measurement with respect to the measurement noise is

$$H^{(v)} = \frac{\partial h}{\partial v}\,.$$

These functions take simpler forms when the noise enters linearly into the measurement equation:

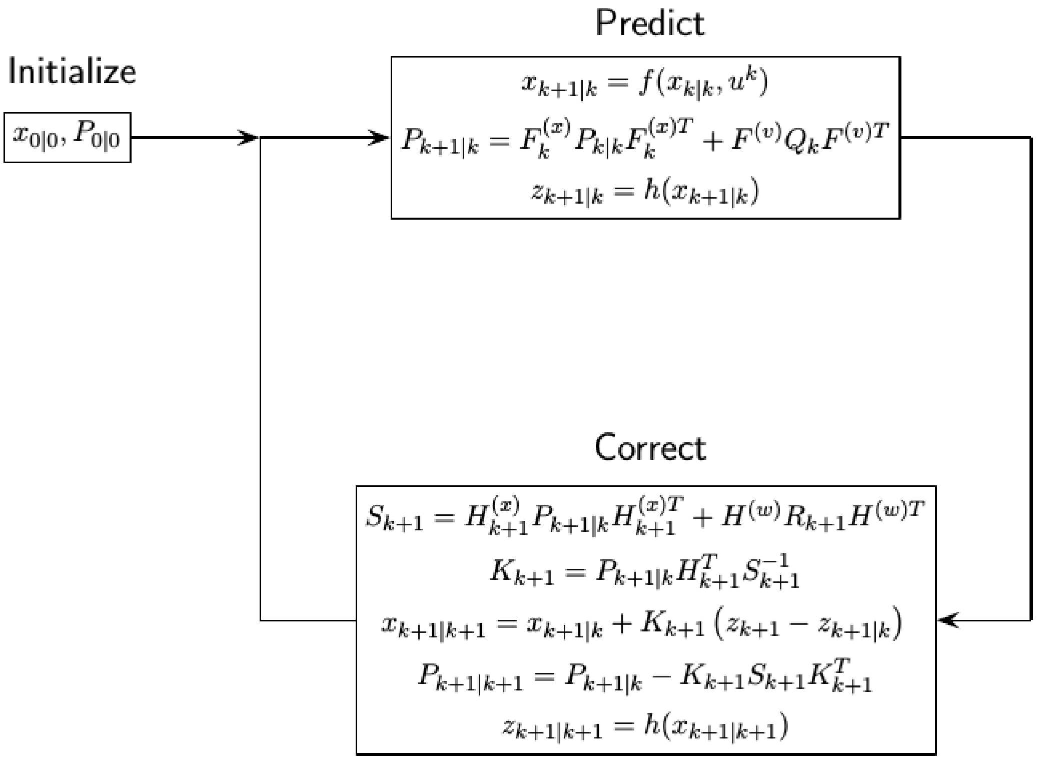
$$z_k = h(x_k, t) + v_k$$

In this case, $H^{(v)} = 1_N$.

Extended Kalman Filter Loop

This extended kalman filter loop is almost identical to the linear Kalman filter loop except that:

- The exact nonlinear state update and measurement functions are used whenever possible and the state transition matrix is replaced by the state Jacobian
- The measurement matrices are replaced by the appropriate Jacobians.


Predict
$x_{k+1|k} = f(x_{k|k}, u^k)$
$P_{k+1|k} = F_k^{(x)} P_{k|k} F_k^{(x)T} + F^{(v)} Q_k F^{(v)T}$
$z_{k+1|k} = h(x_{k+1|k})$
Initialize
$x_{0|0}, P_{0|0}$
Correct
$S_{k+1} = H_{k+1}^{(x)} P_{k+1|k} H_{k+1}^{(x)T} + H^{(w)} R_{k+1} H^{(w)T}$
$K_{k+1} = P_{k+1|k} H_{k+1}^T S_{k+1}^{-1}$
$x_{k+1|k+1} = x_{k+1|k} + K_{k+1} \left(z_{k+1} - z_{k+1|k}\right)$
$P_{k+1|k+1} = P_{k+1|k} - K_{k+1} S_{k+1} K_{k+1}^T$
$z_{k+1|k+1} = h(x_{k+1|k+1})$

Predefined Extended Kalman Filter Functions

Automated Driving Toolbox provides predefined state update and measurement functions to use in the extended Kalman filter.

Motion Model	Function Name	Function Purpose
Constant velocity	`constvel`	Constant-velocity state update model
	`constveljac`	Constant-velocity state update Jacobian
	`cvmeas`	Constant-velocity measurement model
	`cvmeasjac`	Constant-velocity measurement Jacobian
Constant acceleration	`constacc`	Constant-acceleration state update model
	`constaccjac`	Constant-acceleration state update Jacobian
	`cameas`	Constant-acceleration measurement model
	`cameasjac`	Constant-acceleration measurement Jacobian
Constant turn rate	`constturn`	Constant turn-rate state update model
	`constturnjac`	Constant turn-rate state update Jacobian
	`ctmeas`	Constant turn-rate measurement model
	`ctmeasjac`	Constant-turnrate measurement Jacobian

See Also

Objects
`trackingEKF`

Planning, Mapping, and Control

Display Data on OpenStreetMap Basemap

This example shows how to display a driving route and vehicle positions on an OpenStreetMap® basemap.

Add the OpenStreetMap basemap to the list of basemaps available for use with the `geoplayer` object. After you add the basemap, you do not need to add it again in future sessions.

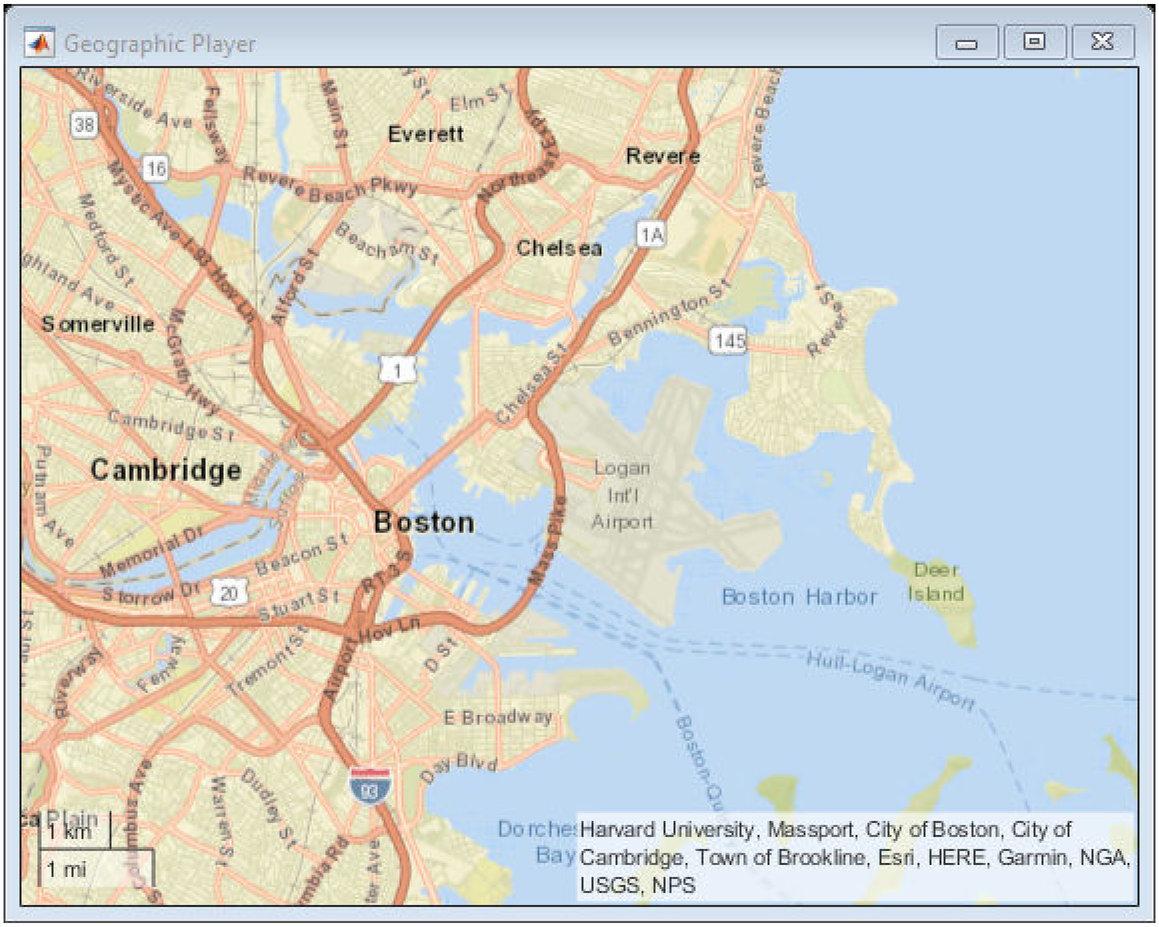
```
name = 'openstreetmap';
url = 'https://a.tile.openstreetmap.org/${z}/${x}/${y}.png';
copyright = char(uint8(169));
attribution = copyright + "OpenStreetMap contributors";
addCustomBasemap(name,url,'Attribution',attribution)
```

Load a sequence of latitude and longitude coordinates.

```
data = load('geoRoute.mat');
```

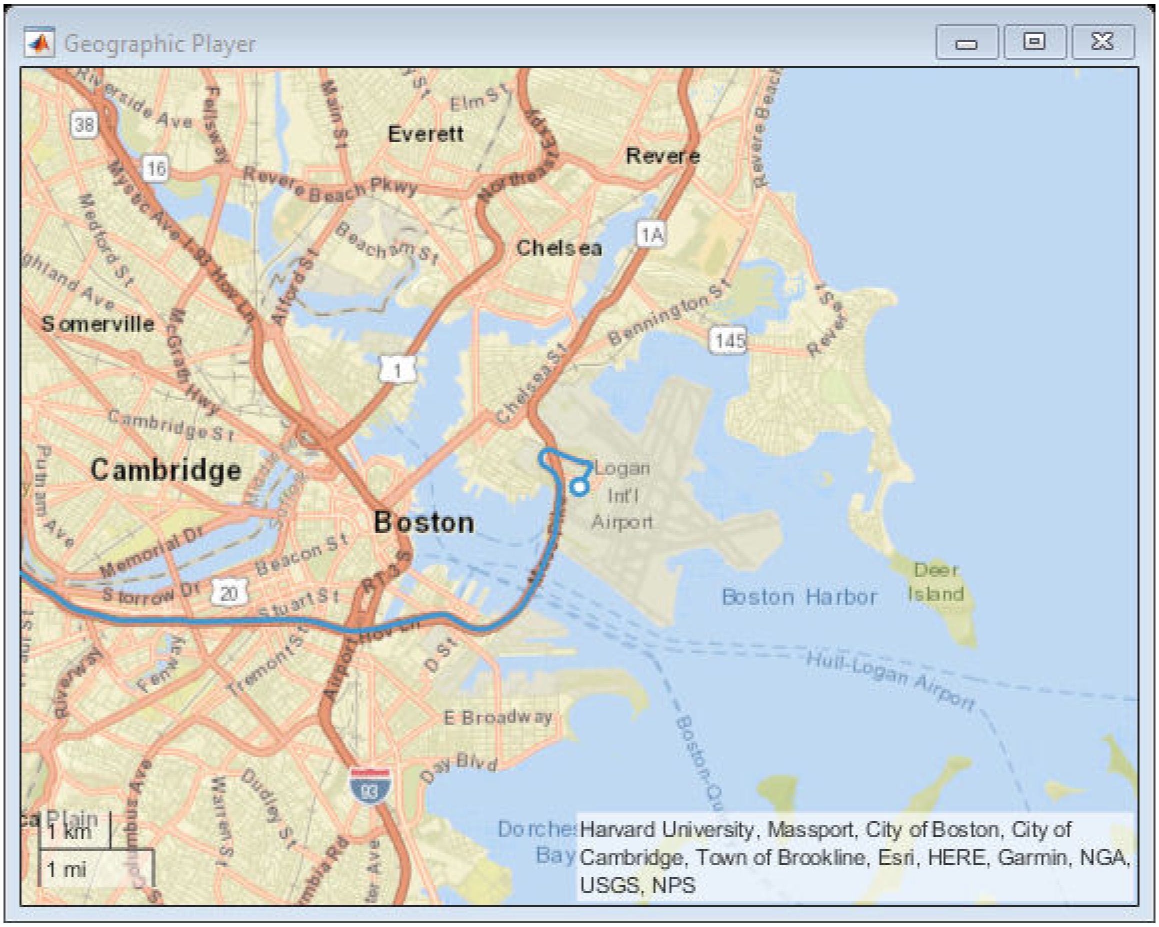
Create a geographic player. Center the geographic player on the first position of the driving route and set the zoom level to 12.

```
zoomLevel = 12;
player = geoplayer(data.latitude(1),data.longitude(1),zoomLevel);
```


Geographic Player
Everett
Revere
Chelsea
Somerville
Cambridge
Boston
Logan Int'l Airport
Boston Harbor
Deer Island
1 km
1 mi
Harvard University, Massport, City of Boston, City of Cambridge, Town of Brookline, Esri, HERE, Garmin, NGA, USGS, NPS

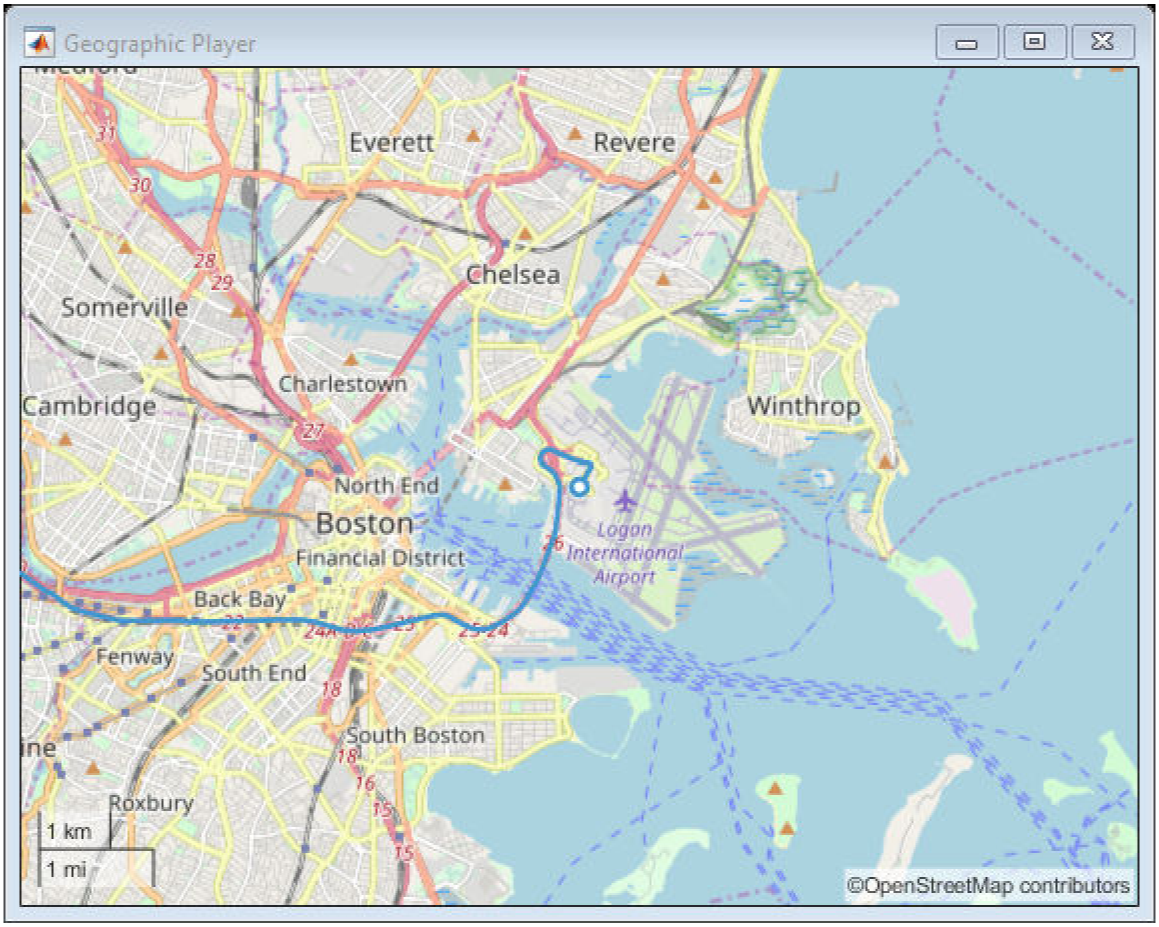
Display the full route.

```
plotRoute(player,data.latitude,data.longitude);
```



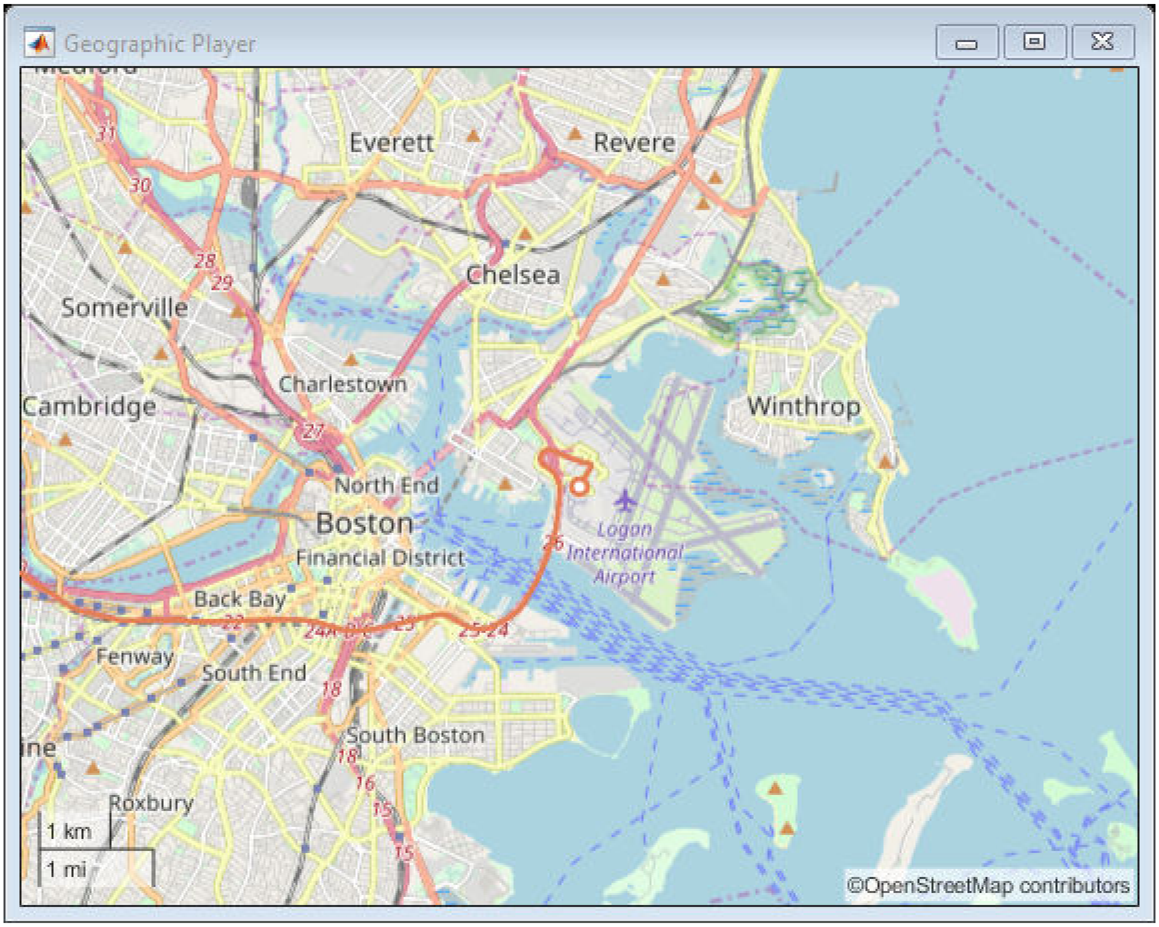
By default, the geographic player uses the World Street Map basemap (`'streets'`) provided by Esri®. Update the geographic player to use the added OpenStreetMap basemap instead.

```
player.Basemap = 'openstreetmap';
```



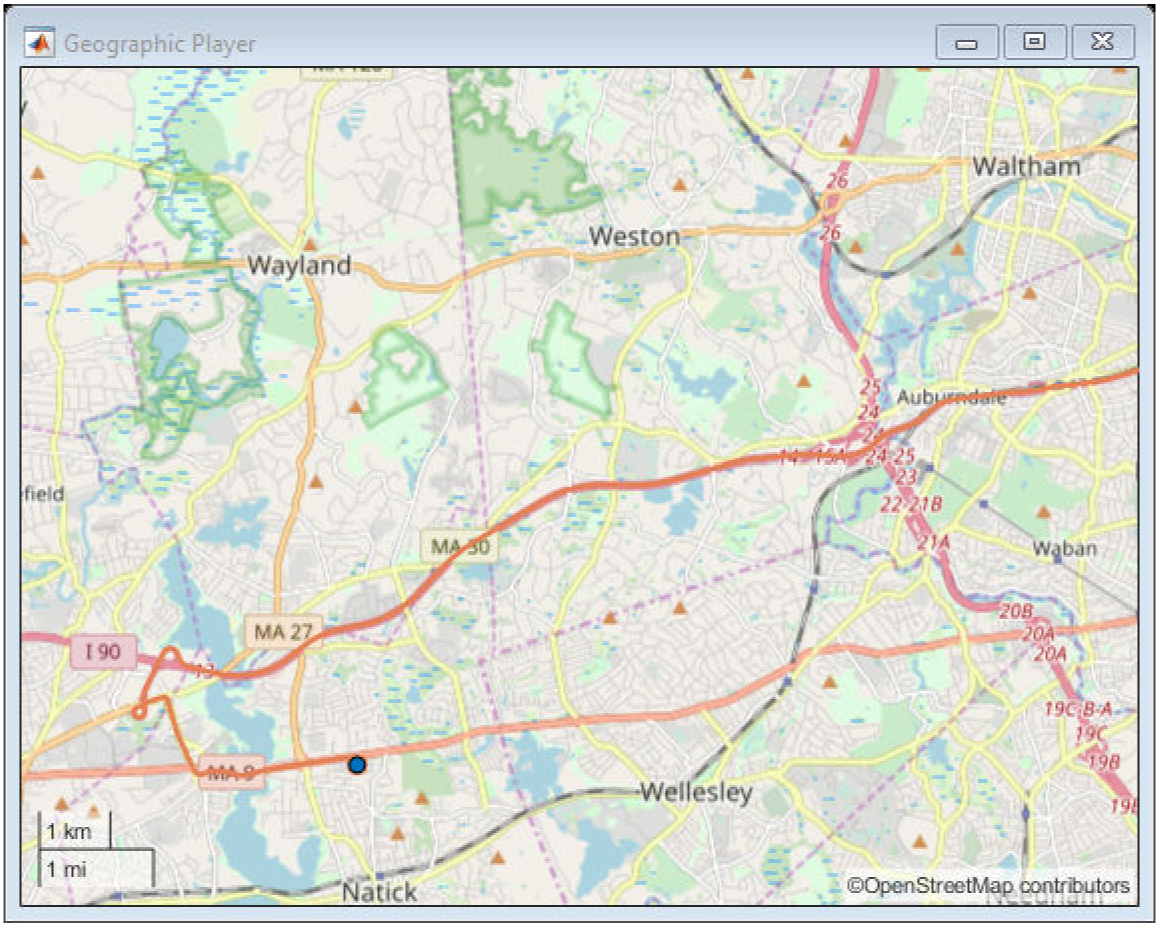
Display the route again.

```
plotRoute(player,data.latitude,data.longitude);
```



Display the positions of the vehicle in a sequence.

```
for i = 1:length(data.latitude)
   plotPosition(player,data.latitude(i),data.longitude(i))
end
```



See Also

addCustomBasemap | geoplayer | plotPosition | plotRoute | removeCustomBasemap

Access HERE HD Live Map Data

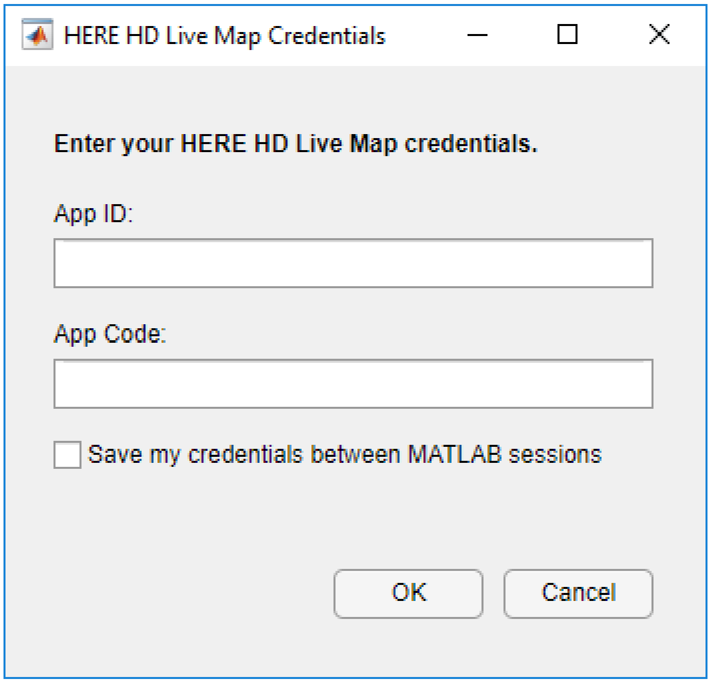
HERE HD Live Map[1] (HERE HDLM), developed by HERE Technologies, is a cloud-based web service that enables you to access highly accurate, continuously updated map data. The data is composed of tiled map layers containing information such as the topology and geometry of roads and lanes, road-level attributes, and lane-level attributes. This data is suitable for a variety of ADAS applications, including localization, scenario generation, navigation, and path planning.

Using Automated Driving Toolbox functions and objects, you can create a HERE HDLM reader, read map data from the HERE HDLM web service, and then visualize the data from certain layers.

Step 1: Enter Credentials

Before you can use the HERE HDLM web service, you must enter the credentials that you obtained from your agreement with HERE Technologies. To set up your credentials, use the `hereHDLMCredentials` function.

```
hereHDLMCredentials setup
```



HERE HD Live Map Credentials

Enter your HERE HD Live Map credentials.

App ID:

App Code:

□ Save my credentials between MATLAB sessions

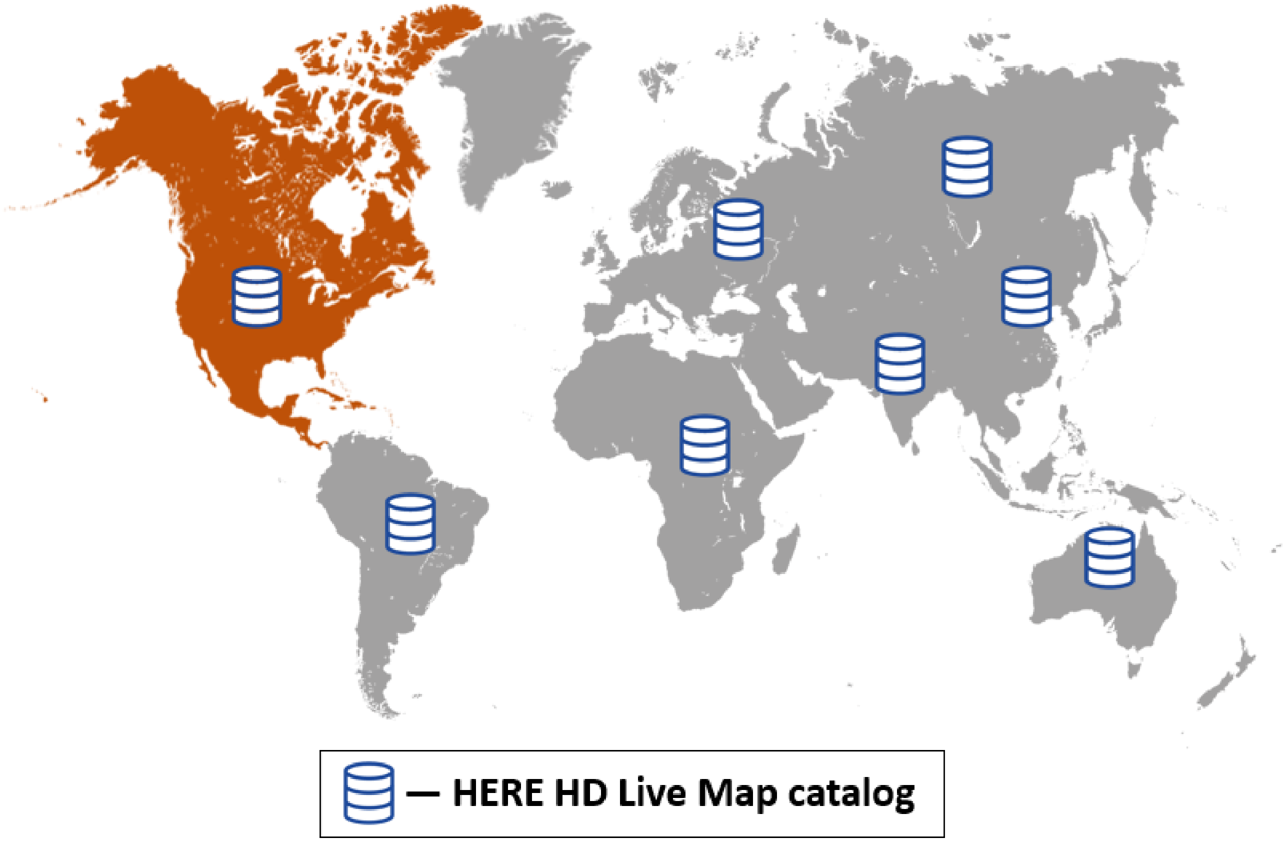
For more details, see “Enter HERE HD Live Map Credentials” on page 4-12.

Step 2: Create Reader Configuration

Optionally, to speed up performance, create a `hereHDLMConfiguration` object that configures the reader to search for map data in only a specific catalog. These catalogs correspond to various geographic regions. For example, create a configuration for the North America region.

```
config = hereHDLMConfiguration('North America');
```

[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 (app_id and app_code) for using the HERE Service.

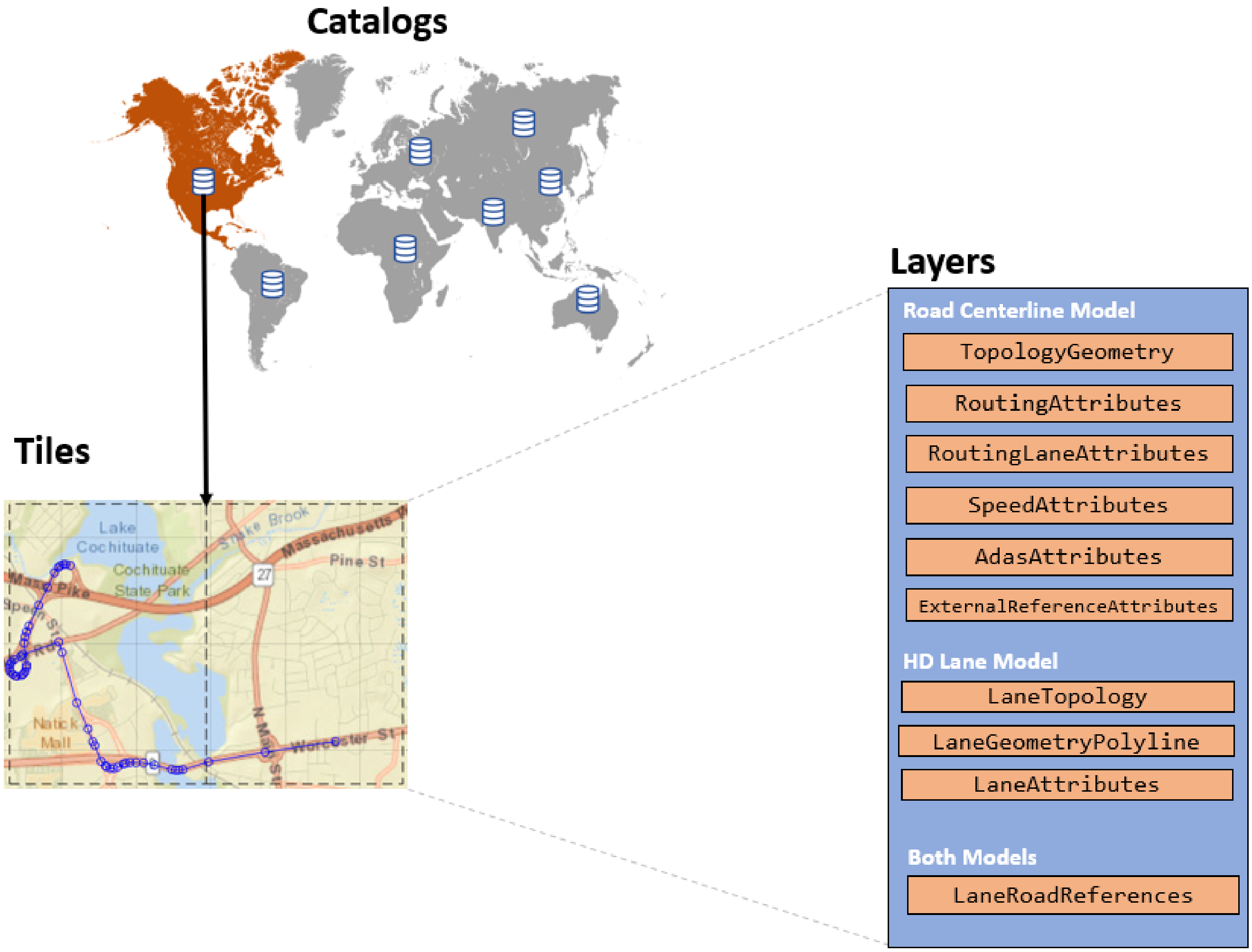


For more details, see “Create Configuration for HERE HD Live Map Reader” on page 4-14.

Step 3: Create Reader

Create a `hereHDLMReader` object and optionally specify the configuration. The reader enables you to read HERE HDLM map data, which is stored is a series of layers, for selected map tiles. You can select map tiles by map tile ID or by specifying the coordinates of a driving route. For example, create a reader that reads tiled map layer data for a driving route in North America.

```
route = load(fullfile(matlabroot,'examples','driving','geoSequenceNatickMA.mat'));
reader = hereHDLMReader(route.latitude,route.longitude,'Configuration',config);
```



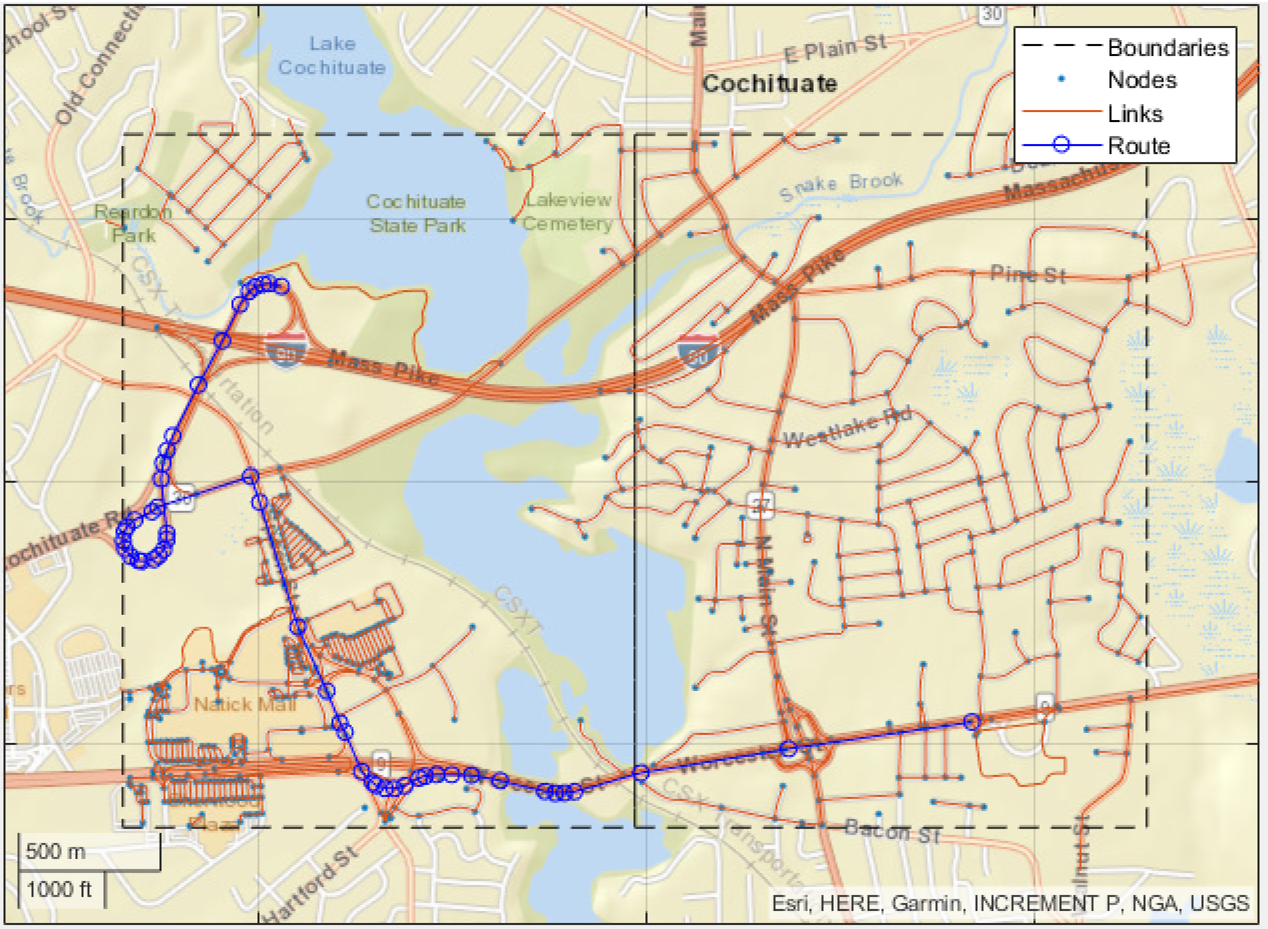
For more details, see “Create HERE HD Live Map Reader” on page 4-18.

Step 4: Read and Visualize Data

Use the `read` function to read data for the selected map tiles. The map data is returned as a series of layer objects. To plot map data for a selected layer, use the `plot` function. For example, read and plot the topology geometry layer for the selected map tiles, and overlay the driving route on the plot.

```
topology = read(reader,'TopologyGeometry');
```

```
topology = 

  2×1 TopologyGeometry array with properties:

   Data:
    HereTileId
    IntersectingLinkRefs
    LinksStartingInTile
    NodesInTile
    TileCenterHere2dCoordinate
    Metadata:
     Catalog
     CatalogVersion

plot(topology)
hold on
geoplot(lat,lon,'bo-','DisplayName','Route');
hold off
```



For more details, see “Read and Visualize Data Using HERE HD Live Map Reader” on page 4-22.

See Also

`hereHDLMConfiguration` | `hereHDLMCredentials` | `hereHDLMReader` | `plot` | `read`

More About

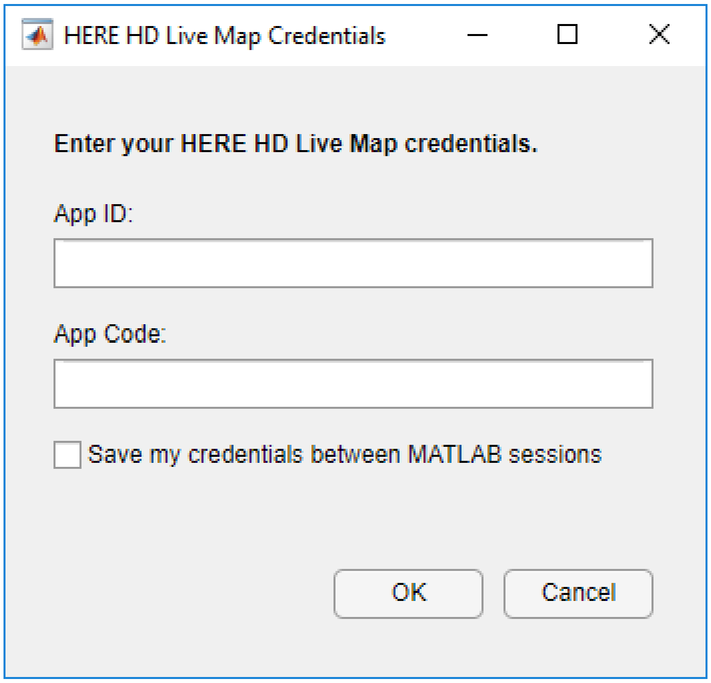
- “HERE HD Live Map Layers” on page 4-30
- “Use HERE HD Live Map Data to Verify Lane Configurations”
- “Import HERE HD Live Map Roads into Driving Scenario” on page 5-72

External Websites

- HD Live Map Data Specification

Enter HERE HD Live Map Credentials

To access the HERE HD Live Map[2] (HERE HDLM) web service, valid HERE credentials are required. You can obtain these credentials by entering into a separate agreement with HERE Technologies. The first time that you use a HERE HDLM function or object in a MATLAB session, a dialog box prompts you to enter these credentials.


HERE HD Live Map Credentials

Enter your HERE HD Live Map credentials.

App ID:

App Code:

□ Save my credentials between MATLAB sessions

Enter a valid **App ID** and **App Code**, and click **OK**. The credentials are now saved for the rest of your MATLAB session on your machine. To save your credentials for future MATLAB sessions on your machine, in the dialog box, select **Save my credentials between MATLAB sessions**. These credentials remain saved until you delete them.

To change your credentials, or to set up your credentials before using a HERE HDLM function or object such as `hereHDLMReader` or `hereHDLMConfiguration`, use the `hereHDLMCredentials` function.

```
hereHDLMCredentials setup
```

You can also use this function to later delete your saved credentials.

```
hereHDLMCredentials delete
```

After you enter your credentials, you can then configure your HERE HDLM reader to search for map data in only a specific geographic region. See “Create Configuration for HERE HD Live Map Reader” on page 4-14. Alternatively, you can create the reader without specifying a configuration. See “Create HERE HD Live Map Reader” on page 4-18.

[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 (app_id and app_code) for using the HERE Service.

See Also

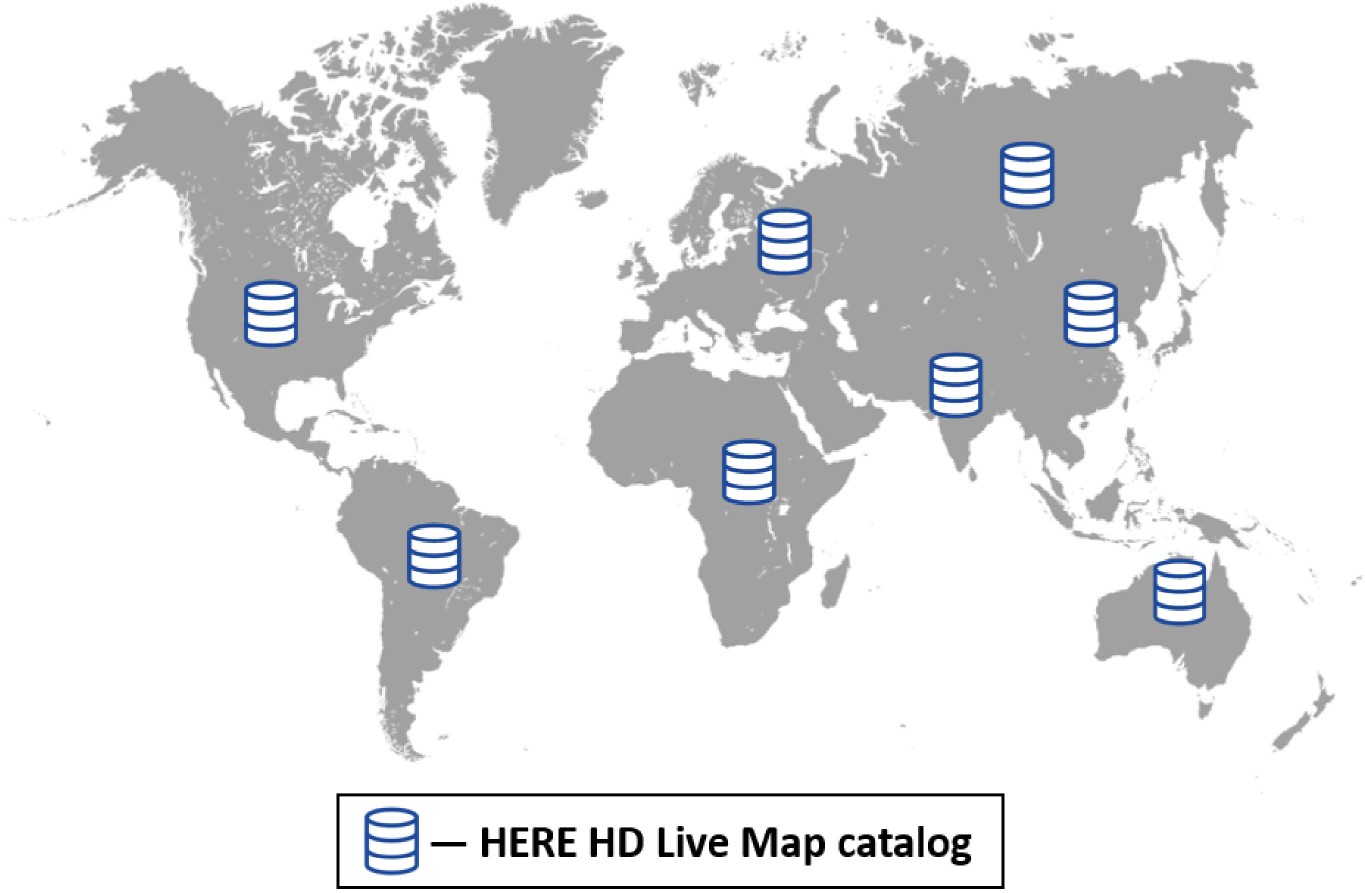
`hereHDLMConfiguration` | `hereHDLMCredentials` | `hereHDLMReader`

More About

- “Create Configuration for HERE HD Live Map Reader” on page 4-14
- “Create HERE HD Live Map Reader” on page 4-18

Create Configuration for HERE HD Live Map Reader

In the HERE HD Live Map[3] (HERE HDLM) web service, map data is stored in a set of databases called catalogs. Each catalog corresponds to a different geographic region (North America, India, Western Europe, and so on). Previous versions of each catalog are also available from the service.



By creating a `hereHDLMConfiguration` object, you can configure a HERE HDLM reader to search for map data from only a specific catalog. These configurations speed up performance of the reader, because the reader does not search unnecessary catalogs for map data. You can also configure a reader to search from only a specific version of a catalog.

Configuring a HERE HDLM reader using a `hereHDLMConfiguration` object is optional. If you do not specify a configuration, by default, the reader searches for map tiles across all catalogs and returns map data from the latest version of that catalog.

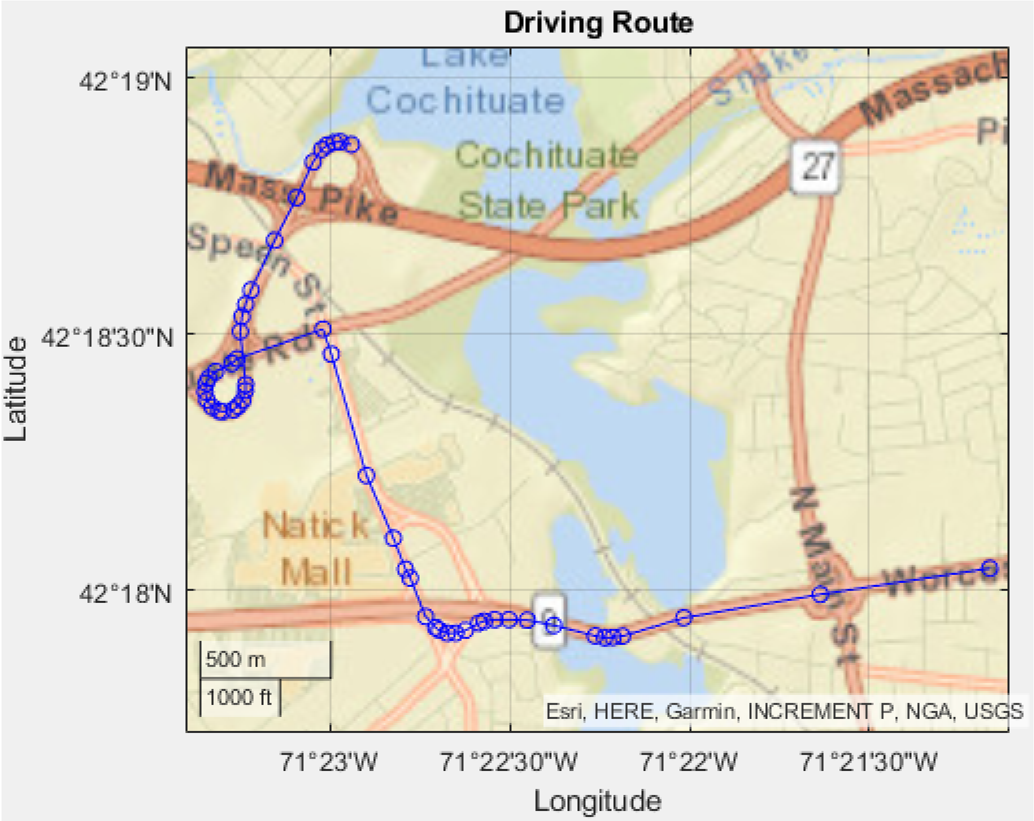
Create Configuration for Specific Catalog

Configuring a HERE HDLM reader to search only a specific catalog can speed up performance.

Consider a driving route located in North America.

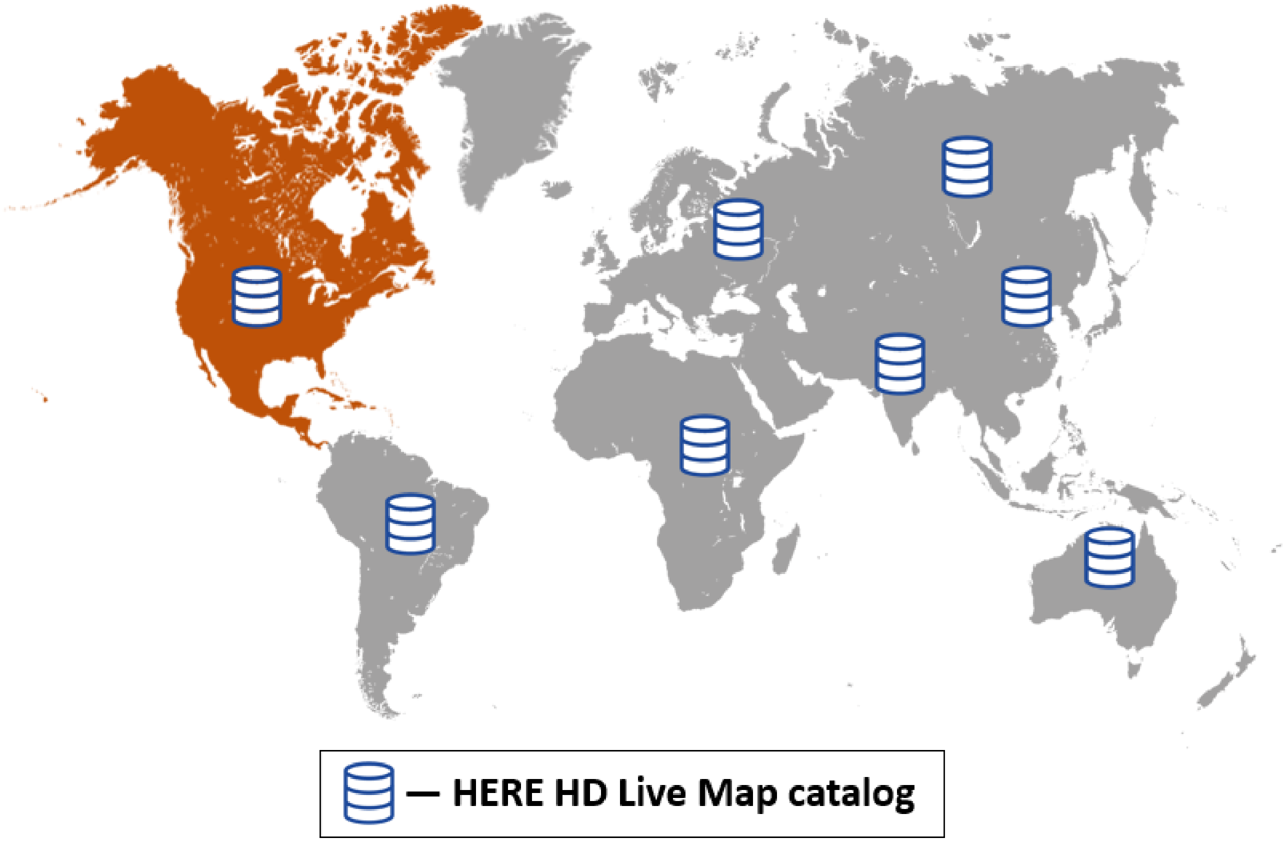
[3] You need to enter into a separate agreement with HERE in order to gain access to the HDLM services and to get the required credentials (app_id and app_code) for using the HERE Service.

```
route = load(fullfile(matlabroot,'examples','driving','geoSequenceNatickMA.mat'));
lat = route.latitude;
lon = route.longitude;
geoplot(lat,lon,'bo-');
geobasemap('streets')
title('Driving Route')
```



Suppose you want to read map data for that route from the HERE HDLM service. You can create a `hereHDLMConfiguration` object that configures a HERE HDLM reader to search for that map data within only the North America catalog.

```
config = hereHDLMConfiguration('North America');
```



If you do not specify such a configuration, by default, the reader searches all available catalogs for this map data.

To configure a HERE HDLM reader for a specific catalog, you can specify either the region name or catalog name. This table shows the HERE HDLM region names and corresponding production catalog names.

Region	Catalog
'Asia Pacific'	'here-hdmap-ext-apac-1'
'Eastern Europe'	'here-hdmap-ext-eeu-1'
'India'	'here-hdmap-ext-rn-1'
'Middle East And Africa'	'here-hdmap-ext-mea-1'
'North America'	'here-hdmap-ext-na-1'
'Oceania'	'here-hdmap-ext-au-1'
'South America'	'here-hdmap-ext-sam-1'
'Western Europe'	'here-hdmap-ext-weu-1'

Create Configuration for Specific Version

The HERE HDLM service also contains map data for previous versions of each catalog. You can configure a reader to access map data from a specific catalog version.

For example, create a configuration object for the previous version of the Western Europe catalog.

```
configLatest = hereHDLMConfiguration('Western Europe');
previousVersion = configLatest.CatalogVersion - 1;
configPrevious = hereHDLMConfiguration('WesternEurope',previousVersion);
```

The HERE HDLM service determines the availability of previous versions of the catalog. If you specify a version of the catalog that is not available, then the `hereHDLMConfiguration` object returns an error.

Configure Reader

To configure a HERE HDLM reader, specify the configuration object when you create the `hereHDLMReader` object. This configuration is stored in the `Configuration` property of the reader.

For example, create a HERE HDLM reader using the configuration and latitude-longitude coordinates that you created in the “Create Configuration for Specific Catalog” on page 4-14 section. Your catalog version might differ from the one shown here. This reader is configured for the latest catalog version, but the HERE HDLM service is continually updated and frequently produces new map versions.

```
reader = hereHDLMReader(lat,lon,'Configuration',config);
reader.Configuration
```

```
  hereHDLMConfiguration with properties:

           Catalog: 'here-hdmap-ext-na-1'
    CatalogVersion: 2054
```

For details about creating HERE HDLM readers, see “Create HERE HD Live Map Reader” on page 4-18.

See Also

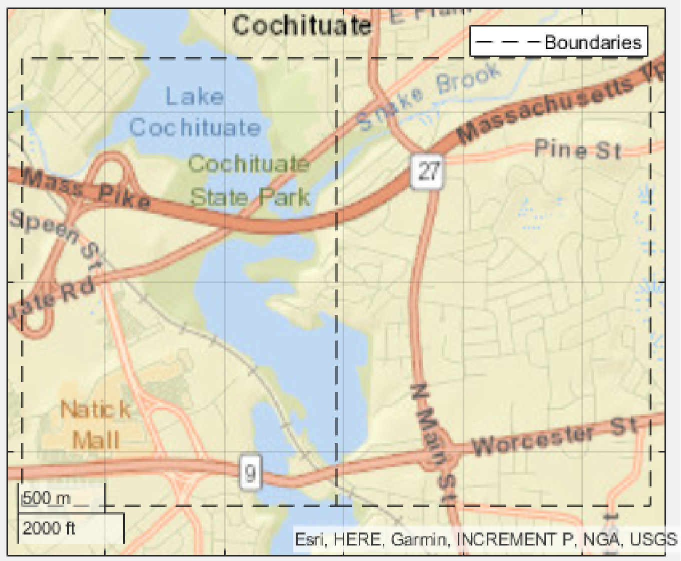
`hereHDLMConfiguration` | `hereHDLMReader`

More About

- “Create HERE HD Live Map Reader” on page 4-18

Create HERE HD Live Map Reader

A `hereHDLMReader` object reads HERE HD Live Map[4] (HERE HDLM) data from a selection of map tiles. By default, these map tiles are set to a zoom level of 14, which corresponds to a rectangular area of about 5–10 square kilometers.



You select the map tiles from which to read data when you create a `hereHDLMReader` object. You can specify the map tile IDs directly, or you can specify a driving route and read data from the map tiles of that route.

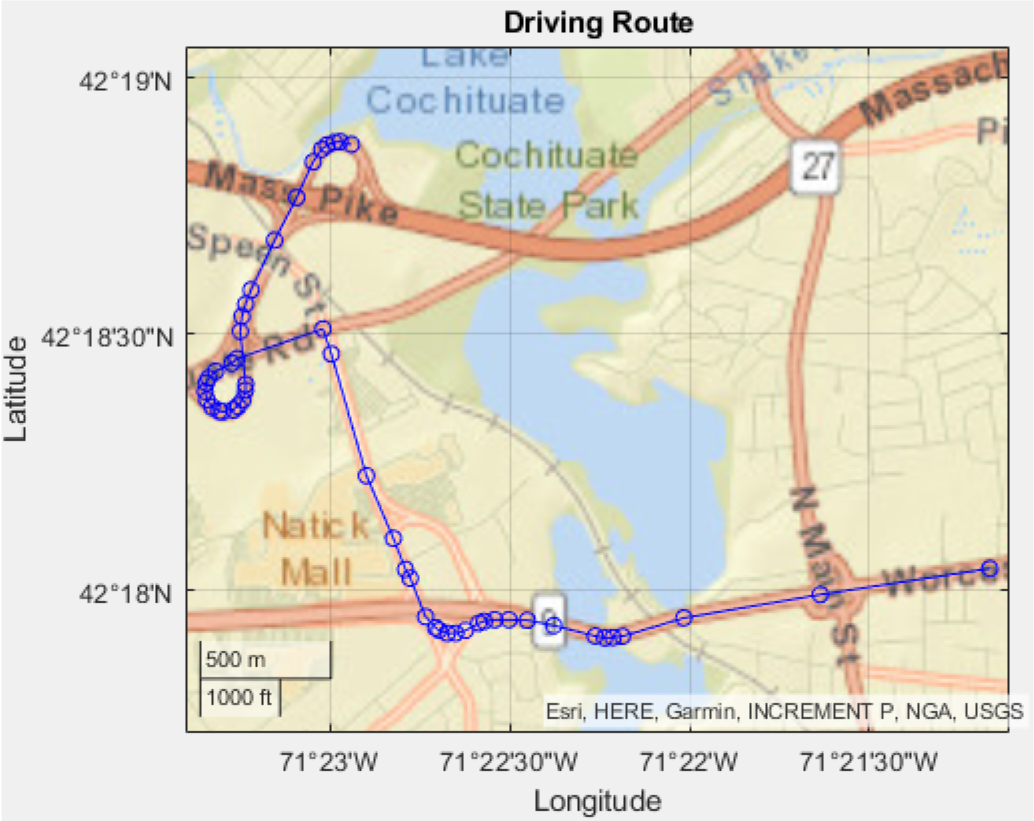
Create Reader from Specified Driving Route

If you have a driving route stored as a vector of latitude-longitude coordinates, you can use these coordinates to select map tiles from which to read data.

Load the latitude-longitude coordinates for a driving route in North America. For reference, display the route on a geographic axes.

```
route = load(fullfile(matlabroot,'examples','driving','geoSequenceNatickMA.mat'));
lat = route.latitude;
lon = route.longitude;

geoplot(lat,lon,'bo-');
geobasemap('streets')
title('Driving Route')
```

[4] 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 (app_id and app_code) for using the HERE Service.



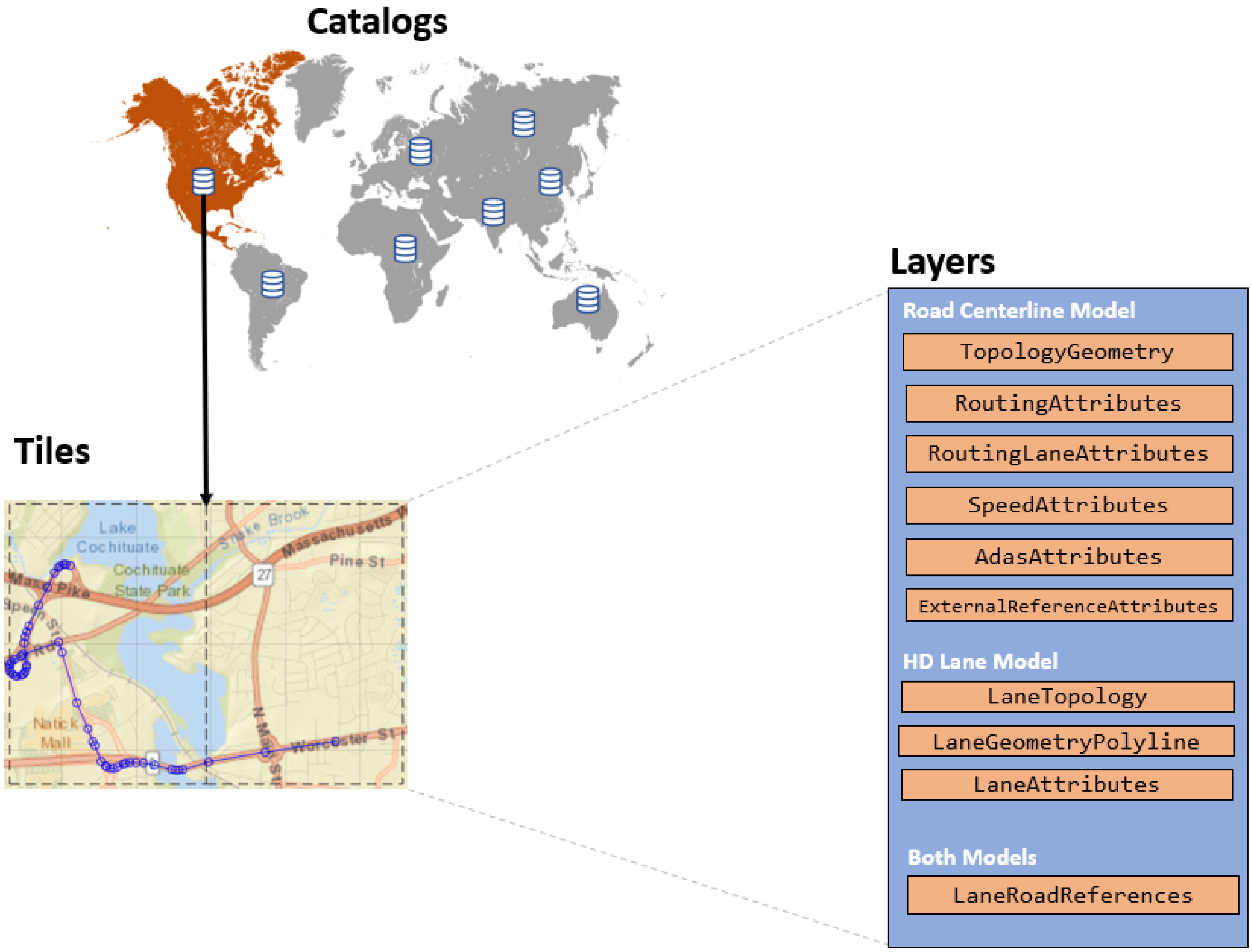
Create a `hereHDLMConfiguration` object for reading data from only the North America catalog. For more details about configuring HERE HDLM readers, see “Create Configuration for HERE HD Live Map Reader” on page 4-14. If you have not previously set up HERE HDLM credentials, a dialog box prompts you to enter them.

```
config = hereHDLMConfiguration('North America');
```

Create a `hereHDLMReader` object using the specified driving route and configuration.

```
reader = hereHDLMReader(lat,lon,'Configuration',config);
```

This HERE HDLM reader enables you to read map data for the tiles that the driving route is on. The map data is stored in a set of layers containing detailed information about various aspects of the map. The reader supports reading data from the map layers for the Road Centerline Model and HD Lane Model. For more details on the layers in these models, see “HERE HD Live Map Layers” on page 4-30.



If you call the `read` function with the HERE HDLM reader, you can read the map tile data for a specific layer. If the layer supports visualization, you can also plot the layer. For more details, see "Read and Visualize Data Using HERE HD Live Map Reader" on page 4-22.

Create Reader from Specified Map Tile IDs

If you know the IDs of the map tiles from which you want to read data, when you create a `hereHDLMReader` object, you can specify the map tile IDs directly. Specify the map tile IDs as an array of unsigned 32-bit integers.

Create a `hereHDLMReader` object using the map tile IDs and configuration from the previous section.

```
tileIds = uint32([321884279 321884450]);
reader = hereHDLMReader(tileIds);
```

This reader is equivalent to the reader created in the previous section. The only difference between these two readers is the method for selecting the map tiles from which to read data.

To learn more about reading and plotting data from map tiles, see "Read and Visualize Data Using HERE HD Live Map Reader" on page 4-22.

See Also

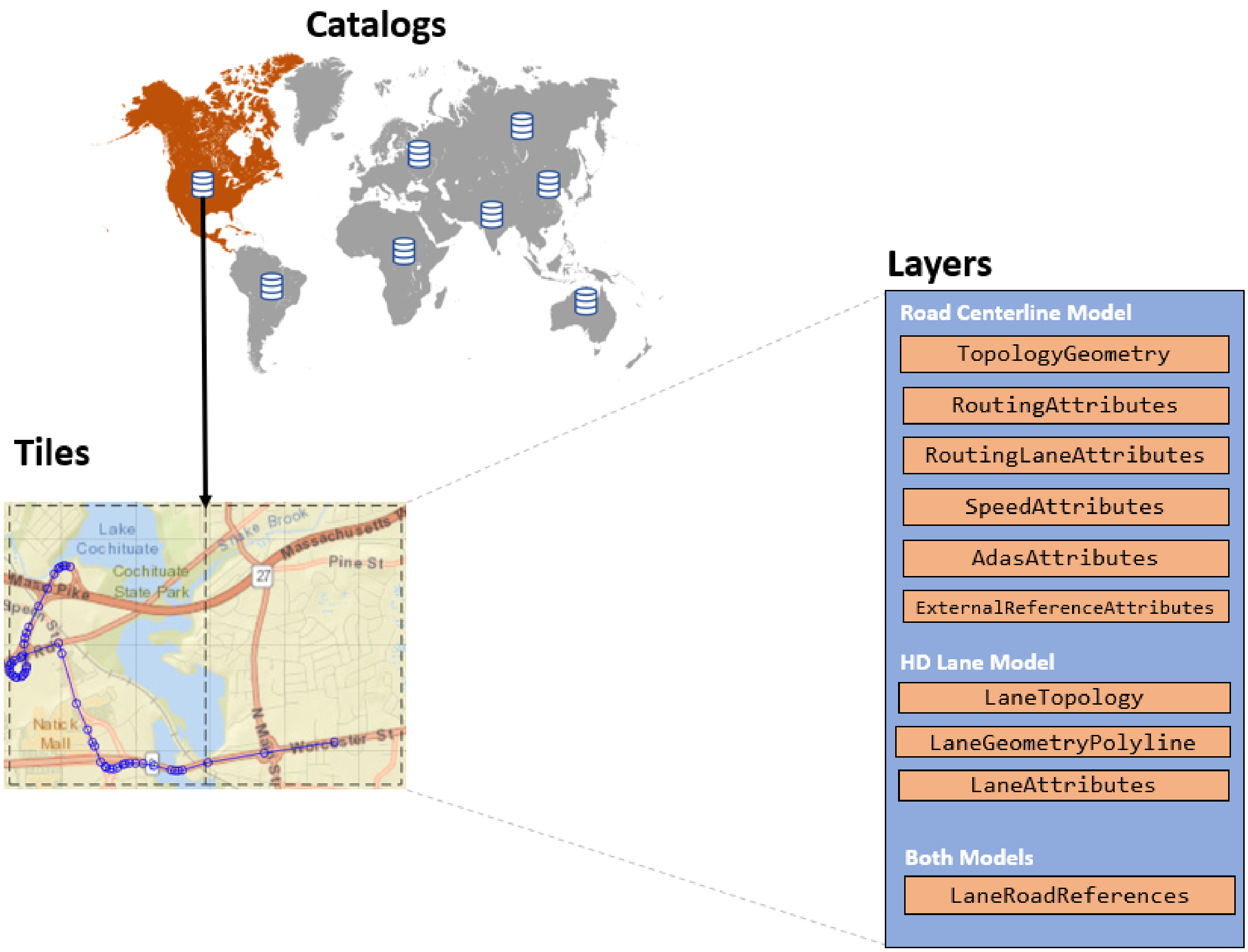
`hereHDLMConfiguration` | `hereHDLMReader` | `read`

More About

- “Read and Visualize Data Using HERE HD Live Map Reader” on page 4-22
- “HERE HD Live Map Layers” on page 4-30

Read and Visualize Data Using HERE HD Live Map Reader

You can read map tile data from the HERE HD Live Map[5] (HERE HDLM) web service by using a `hereHDLMReader` object and the `read` function. This data is composed of a series of map layer objects. The diagram shows the layers available for map tiles corresponding to a driving route in North America.



You can use this map layer data for a variety of automated driving applications. You can also visualize certain layers by using the `plot` function.

Create Reader

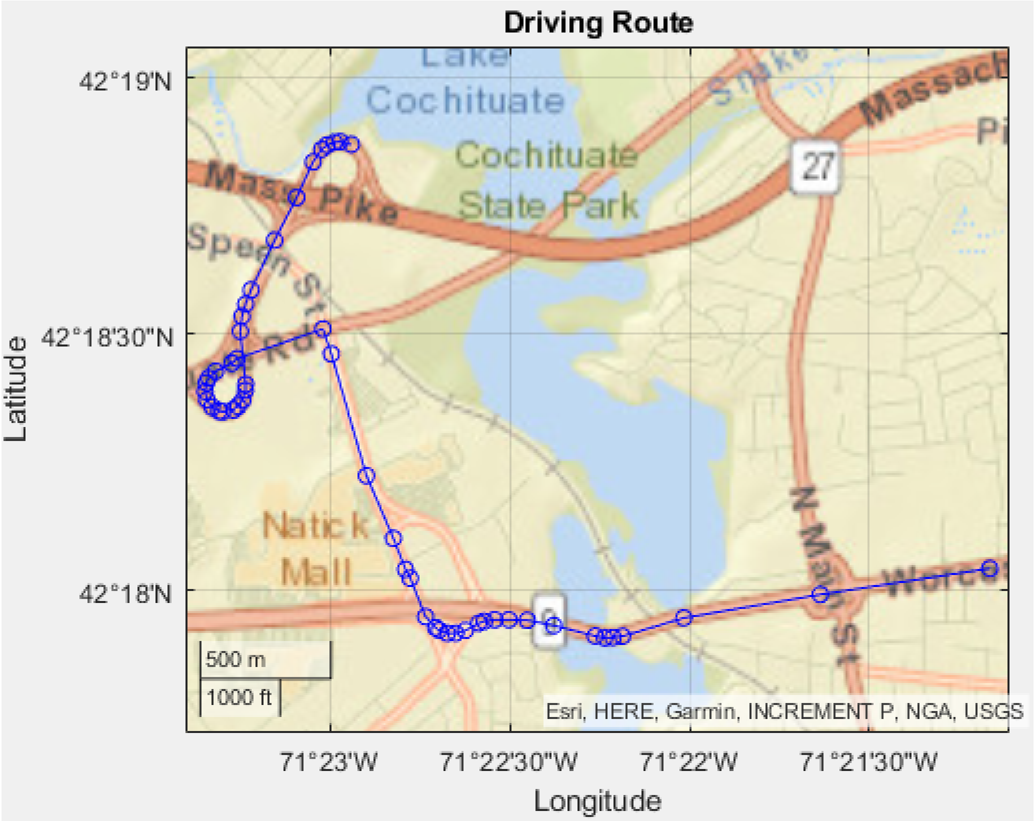
To read map data using the `read` function, you must specify a `hereHDLMReader` object as an input argument. This object specifies the map tiles from which you want to read data.

Create a `hereHDLMReader` object that can read data from the map tiles of a driving route in North America. Configure the reader to read data from only the North America catalog by specifying a

[5] 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 (app_id and app_code) for using the HERE Service.

hereHDLMConfiguration object for the Configuration property of the reader. If you have not previously entered HERE HDLM credentials, a dialog box prompts you to enter them. For reference, display the driving route on a geographic axes.

```
route = load(fullfile(matlabroot,'examples','driving','geoSequenceNatickMA.mat'));
lat = route.latitude;
lon = route.longitude;
config = hereHDLMConfiguration('North America');
reader = hereHDLMReader(lat,lon,'Configuration',config);


geoplot(lat,lon,'bo-');
geobasemap('streets')
title('Driving Route')
```



For more details about configuring a HERE HDLM reader, see “Create Configuration for HERE HD Live Map Reader” on page 4-14. For more details about creating a reader, see “Create HERE HD Live Map Reader” on page 4-18.

Read Map Layer Data

To read map layer data from the HERE HDLM web service, call the read function with the reader you created in the previous section and the name of the map layer you want to read. For example, read data from the layer containing the topology geometry of the road. The data is returned as an array of map layer objects.

```
topology = read(reader,'TopologyGeometry')
topology = 

  2×1 TopologyGeometry array with properties:

   Data:
    HereTileId
    IntersectingLinkRefs
    LinksStartingInTile
    NodesInTile
    TileCenterHere2dCoordinate

   Metadata:
    Catalog
    CatalogVersion
```

Each map layer object corresponds to a map tiles that you selected using the input `hereHDLMReader` object. The IDs of these map tiles are stored in the `TileIds` property of the HERE HDLM reader.

Inspect the properties of the map layer object for the first map tile. Your catalog version might differ from the one shown here.

```
topology(1)
```

```
ans = 

  TopologyGeometry with properties:

   Data:
                 HereTileId: 321884279
       IntersectingLinkRefs: [38×1 struct]
        LinksStartingInTile: [490×1 struct]
                NodesInTile: [336×1 struct]
    TileCenterHere2dCoordinate: [42.3083 -71.3782]

   Metadata:
                    Catalog: 'here-hdmap-ext-na-1'
             CatalogVersion: 2066
```

The properties of the `TopologyGeometry` layer object correspond to valid HERE HDLM fields for that layer. In these layer objects, the names of the layer fields are modified to fit the MATLAB naming convention for object properties. For each layer field name, the first letter and first letter after each underscore are capitalized and the underscores are removed. This table shows sample name changes.

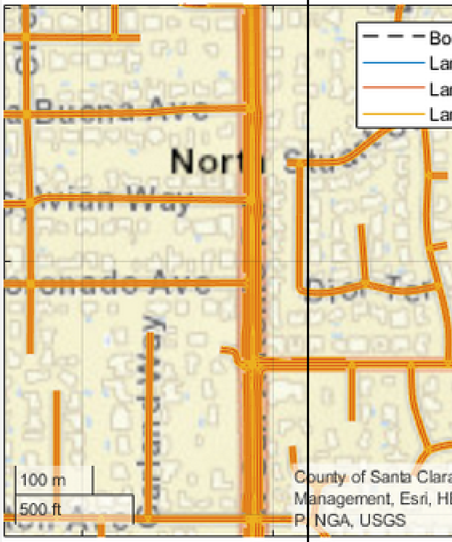
HERE HDLM Layer Fields	MATLAB Layer Object Property
here_tile_id	HereTileId
tile_center_here_2d_coordinate	TileCenterHere2dCoordinate
nodes_in_tile	NodesInTile

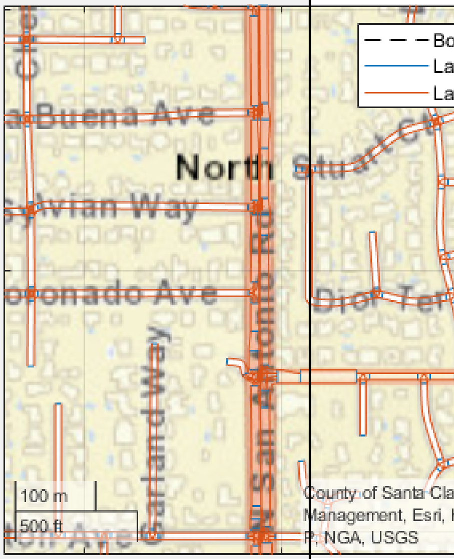
The layer objects are MATLAB structures whose properties correspond to structure fields. To access data from these fields, use dot notation. For example, this code selects the `NodeId` subfield from the `NodeAttribution` field of a layer:

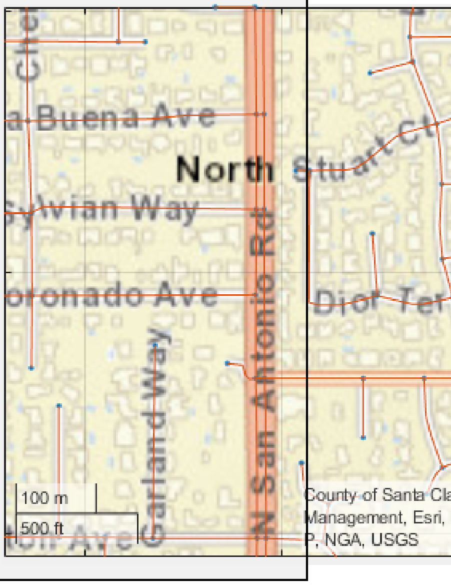
```
layerData.NodeAttribution.NodeId
```

This table summarizes the valid types of layer objects and their top-level data fields. The available layers are for the

Road Centerline Model and HD Lane Model. For an overview of HERE HDLM layers and the models that they belong to, see “HERE HD Live Map Layers” on page 4-30. For a full description of the fields, see HD Live Map Data Specification on the HERE Technologies website.

Layer Object	Description	Top-Level Data Fields (Layer Object Properties)	Plot Support
`AdasAttributes`	Precision geometry measurements, such as slope, elevation, and curvature of roads. Use this data to develop advanced driver assistance systems (ADAS).	• `HereTileId` • `LinkAttribution` • `NodeAttribution`	Not available
`ExternalReferenceAttributes`	References to external map links, nodes, and topologies for other HERE maps.	• `HereTileId` • `LinkAttribution` • `NodeAttribution`	Not available
`LaneAttributes`	Lane-level attributes, such as direction of travel and lane type.	• `HereTileId` • `LaneGroupAttribution`	Not available
`LaneGeometryPolyline`	3-D lane geometry composed of a set of 3-D points joined into polylines.	• `HereTileId` • `TileCenterHere3dCoordinate` • `LaneGroupGeometries`	Available — Use the `plot` function.  Bo Lar Lar Lar Buena Ave Nort vian Way onado Ave 100 m 500 ft County of Santa Clara Management, Esri, HE P, NGA, USGS
`LaneRoadReferences`	Road and lane group references and range information. Use this data to translate positions between the Road Centerline Model and the HD Lane Model.	• `HereTileId` • `LaneGroupLinkReferences` • `LinkLaneGroupReferences`	Not available

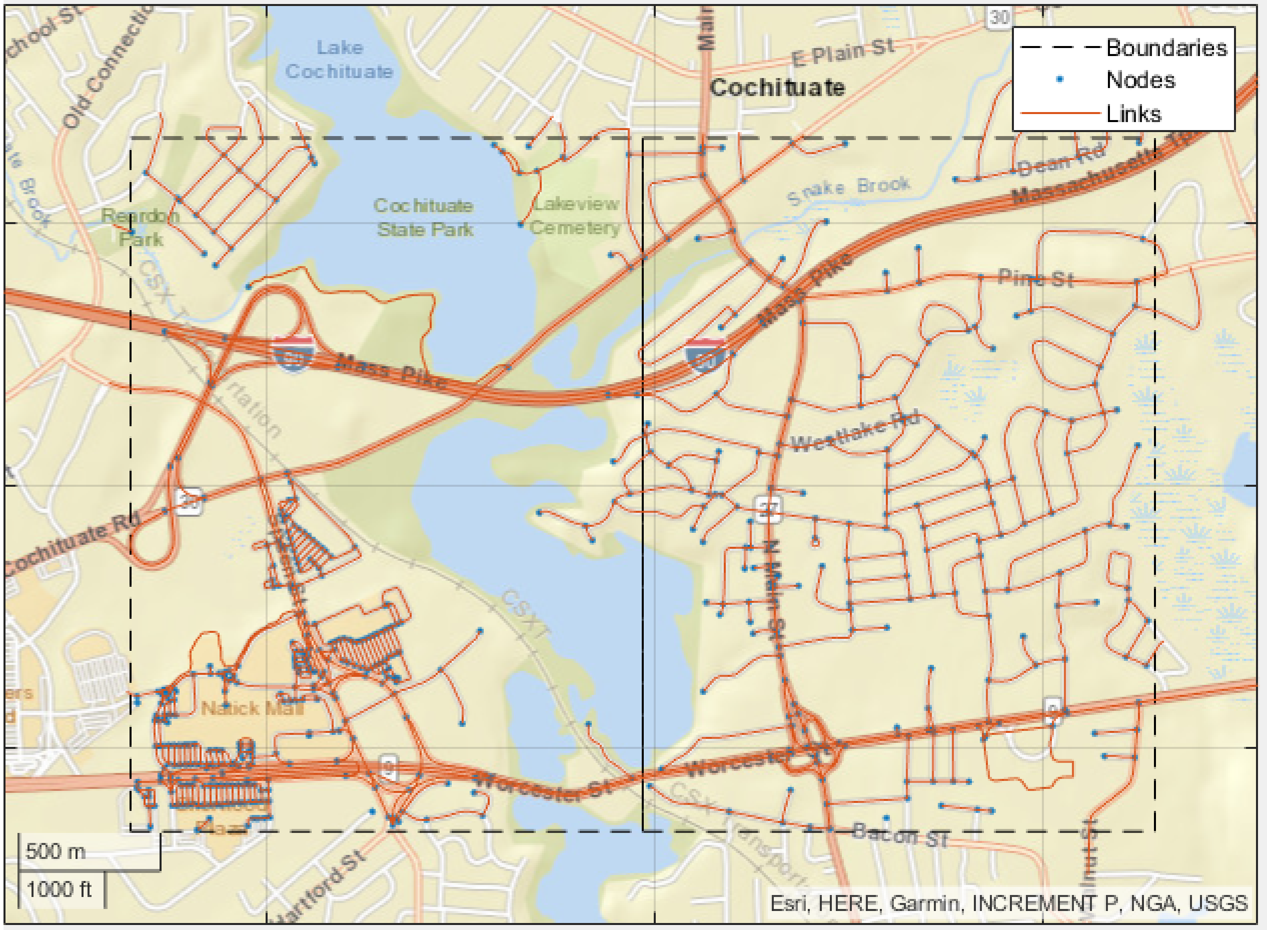
Layer Object	Description	Top-Level Data Fields (Layer Object Properties)	Plot Support
`LaneTopology`	Topologies of the HD Lane model, including lane group, lane group connector, lane, and lane connector topologies. This layer also contains the simplified 2-D boundary geometry of the lane model for determining map tile affinity and overflow.	• `HereTileId` • `TileCenterHere2dCoordinate` • `LaneGroupsStartingInTile` • `LaneGroupConnectorsInTile` • `IntersectingLaneGroupRefs`	Available — Use the `plot` function.  Bo La La Buena Ave North Stu lvian Way oronado Ave Garland Way N San Antonio Rd Dio Ter 100 m 500 ft County of Santa Cla Management, Esri, I P, NGA, USGS
`RoutingAttributes`	Road attributes related to navigation and conditions. These attributes are mapped parametrically to the 2-D polyline geometry in the topology layer.	• `HereTileId` • `LinkAttribution` • `NodeAttribution` • `StrandAttribution` • `AttributionGroupList`	Not available
`RoutingLaneAttributes`	Core navigation lane attributes and conditions, such as the number of lanes in a road. These values are mapped parametrically to 2-D polylines along the road links.	• `HereTileId` • `LinkAttribution`	Not available
`SpeedAttributes`	Speed-related road attributes, such as speed limits. These attributes are mapped to the 2-D polyline geometry of the topology layer.	• `HereTileId` • `LinkAttribution`	Not available

Layer Object	Description	Top-Level Data Fields (Layer Object Properties)	Plot Support
`TopologyGeometry`	Topology and 2-D line geometry of the road. This layer also contains definitions of the nodes and links in the map tile.	• `HereTileId` • `TileCenterHere2dCoordinate` • `NodesInTile` • `LinksStartingInTile` • `IntersectingLinkRefs`	Available — Use the `plot` function.  Che a Buena Ave North Stuart Ct ylvian Way oronado Ave Diot Ter Garland Way N San Antonio Rd 100 m 500 ft County of Santa Cla Management, Esri, P, NGA, USGS

Visualize Map Layer Data

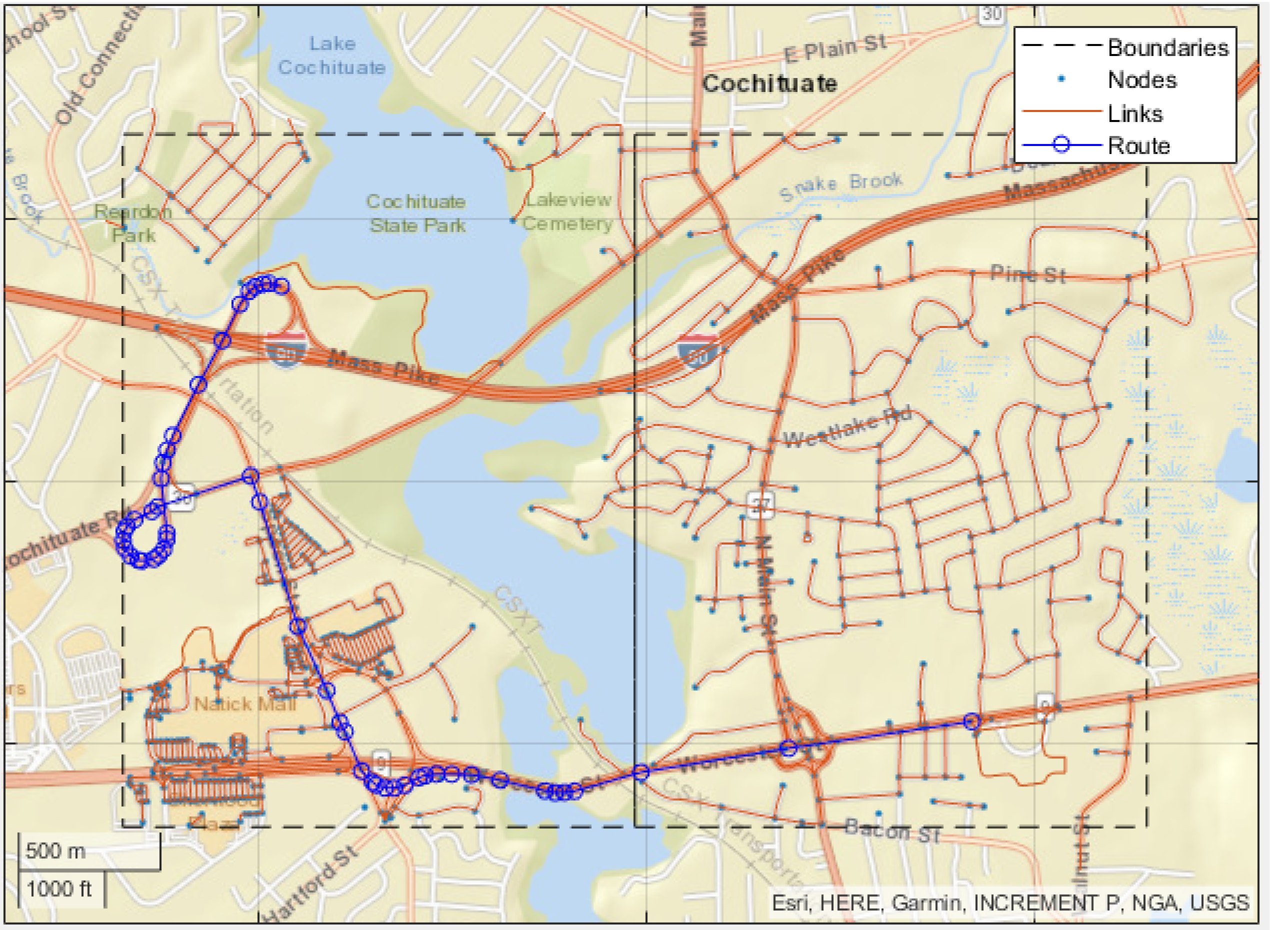
You can visualize the data of certain map layers. To visualize these layers, use the `plot` function. Plot the topology geometry of the returned map layers. The plot shows the boundaries, nodes (intersections and dead-ends), and links (streets) within the map tiles. If a link extends past the tile boundary, the layer data includes that link.

```
plot(topology)
```



Map layer plots are returned on a geographic axes. To customize map displays, you can use the properties of the geographic axes. For more details, see GeographicAxes Properties. Overlay the driving route on the plot.

```
hold on
geoplot(lat,lon,'bo-','DisplayName','Route');
hold off
```



See Also

`hereHDLMReader` | `plot` | `read`

More About

- “HERE HD Live Map Layers” on page 4-30
- “Use HERE HD Live Map Data to Verify Lane Configurations”

External Websites

- HD Live Map Data Specification

HERE HD Live Map Layers

HERE HD Live Map[6] (HERE HDLM), developed by HERE Technologies, is a cloud-based web service that enables you to access highly accurate, continuously updated map data. The data is composed of tiled map layers containing information such as the topology and geometry of roads and lanes, and road-level and lane-level attributes. The data is stored in a series of map catalogs that correspond to geographic regions.

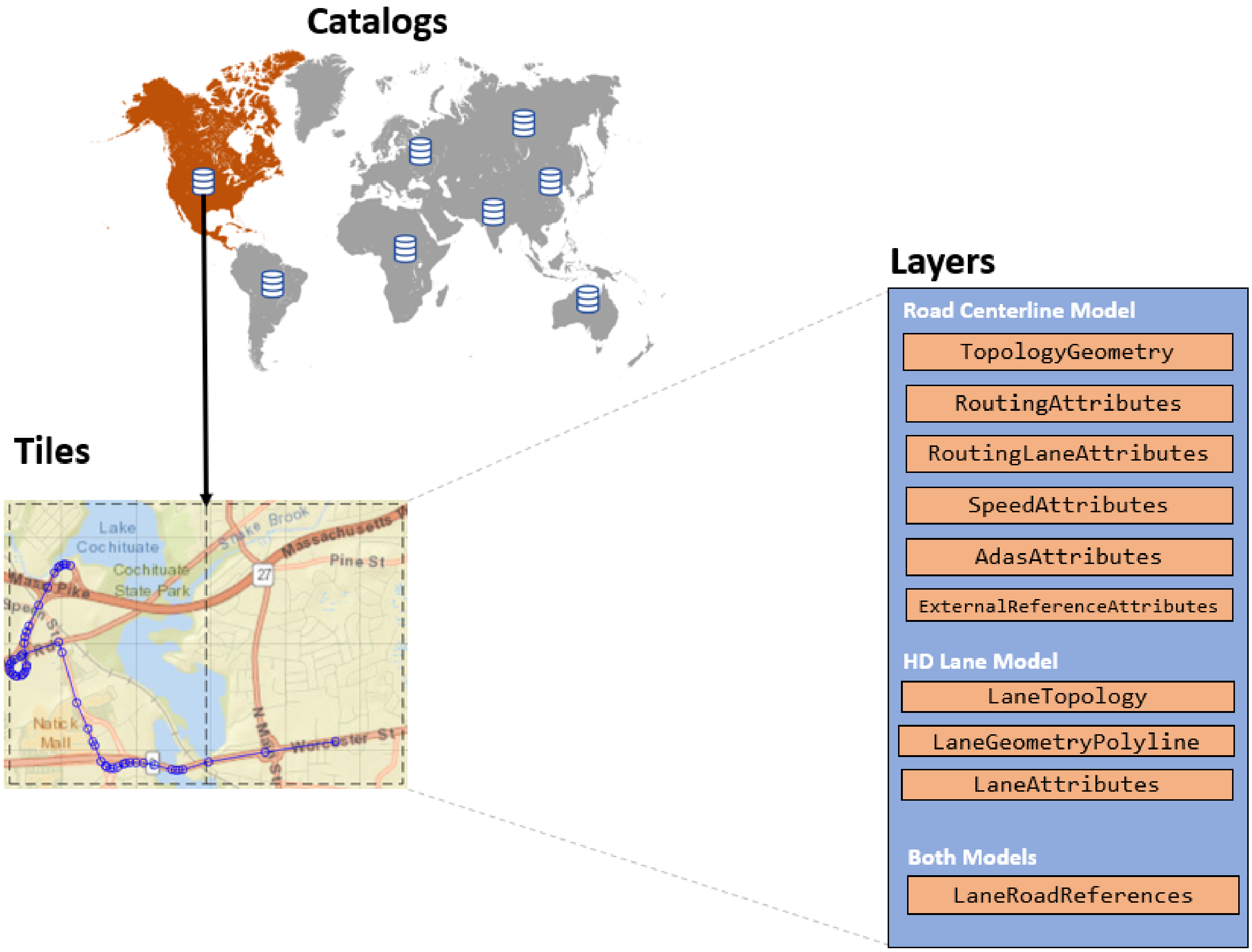
To access layer data for a selection of map tiles, use a `hereHDLMReader` object. For information on the `hereHDLMReader` workflow, see "Access HERE HD Live Map Data" on page 4-7.

The layers are grouped into these models:

- "Road Centerline Model" on page 4-31 — Provides road topology, shape geometry, and other road-level attributes
- "HD Lane Model" on page 4-33 — Contains lane topology, highly accurate geometry, and lane-level attributes
- "HD Localization Model" on page 4-34 — Includes multiple features, such as road signs, to support localization strategies

`hereHDLMReader` objects support reading layers from the Road Centerline Model and HD Lane Model only.

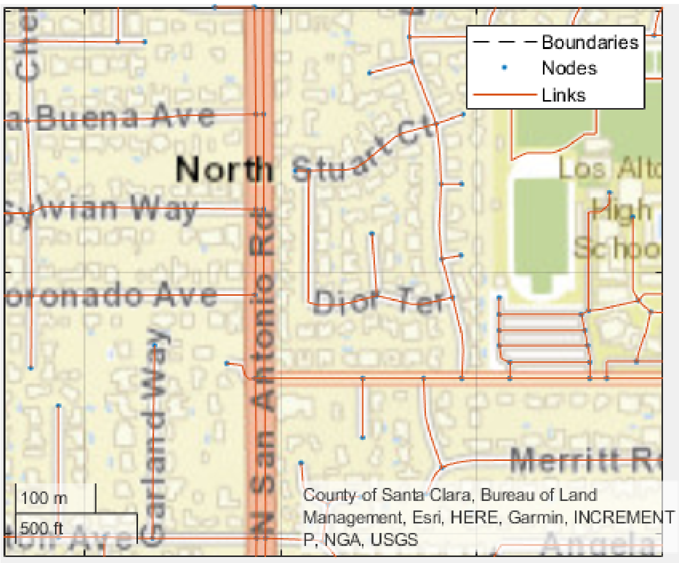
[6] You need to enter into a separate agreement with HERE in order to gain access to the HDLM services and to get the required credentials (app_id and app_code) for using the HERE Service.



Road Centerline Model

The Road Centerline Model represents the topology of the road network. It is composed of links corresponding to streets and nodes corresponding to intersections and dead-ends. For each map tile, the layers within this model contain information about these links and nodes, such as the 2-D line geometry of the road network, speed attributes, and routing attributes.

The figure shows a plot for the `TopologyGeometry` layer, which visualizes the 2-D line geometry of the nodes and links within a map tile.



This table shows the map layers of the Road Centerline Model that a `hereHDLMReader` object can read. The available layers vary by geographic region, so not all layers are available for every map tile. When you call the `read` function on a `hereHDLMReader` object and specify a map layer name, the function returns the layer data as an object. For more details about these layer objects, see the `read` function reference page.

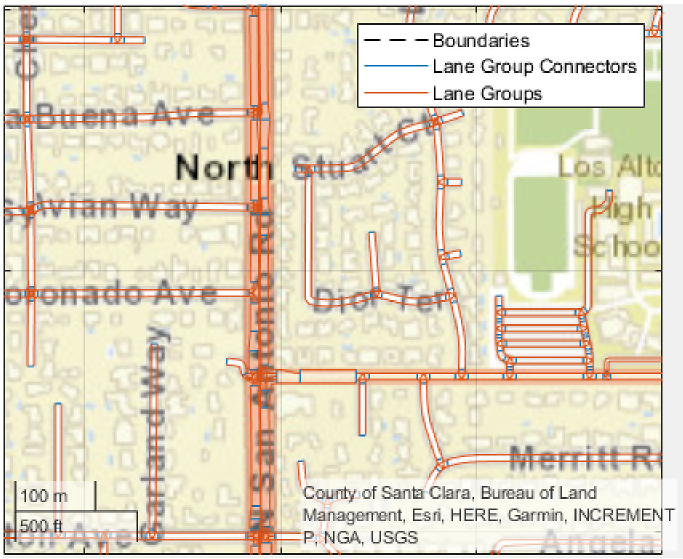
Road Centerline Model Layers	Description
`TopologyGeometry`	Topology and 2-D line geometry of the road. This layer also contains definitions of the links (streets) and nodes (intersections and dead-ends) in the map tile.
`RoutingAttributes`	Road attributes related to navigation and conditions. These attributes are mapped parametrically to the 2-D polyline geometry in the topology layer.
`RoutingLaneAttributes`	Core navigation lane attributes and conditions, such as the number of lanes in a road. These values are mapped parametrically to 2-D polylines along the road links.
`SpeedAttributes`	Speed-related road attributes, such as speed limits. These attributes are mapped to the 2-D polyline geometry of the topology layer.
`AdasAttributes`	Precision geometry measurements such as slope, elevation, and curvature of roads. Use this data to develop advanced driver assistance systems (ADAS).
`ExternalReferenceAttributes`	References to external links, nodes, and topologies for other HERE maps.

Road Centerline Model Layers	Description
`LaneRoadReferences` (also part of HD Lane Model)	Road and lane group references and range information. Use this data to translate positions between the Road Centerline Model and the HD Lane Model.

HD Lane Model

The HD Lane Model represents the topology and geometry of lane groups, which are the lanes within a link (street). In this model, the shapes of lanes are modeled with 2-D and 3-D positions and support centimeter-level accuracy. This model provides several lane attributes, including lane type, direction of travel, and lane boundary color and style.

The figure shows a plot for the `LaneTopology` layer object, which visualizes the 2-D line geometry of lane groups and their connectors within a map tile.



County of Santa Clara, Bureau of Land Management, Esri, HERE, Garmin, INCREMENT P, NGA, USGS

This table shows the map layers of the HD Lane Model that a `hereHDLMReader` object can read. The available layers vary by geographic region, so not all layers are available for every map tile. When you call the `read` function on a `hereHDLMReader` object and specify a map layer name, the function returns the layer data as an object. For more details about these layer objects, see the `read` function reference page.

HD Lane Model Layers	Description
`LaneTopology`	Topologies of the HD Lane model, including lane group, lane group connector, lane, and lane connector topologies. This layer also contains the simplified 2-D boundary geometry of the lane model for determining map tile affinity and overflow.
`LaneGeometryPolyline`	3-D lane geometry composed of a set of 3-D points joined into polylines.

HD Lane Model Layers	Description
LaneAttributes	Lane-level attributes, such as direction of travel and lane type.
LaneRoadReferences (also part of Road Centerline Model)	Road and lane group references and range information. Used to translate positions between the Road Centerline Model and the HD Lane Model.

HD Localization Model

The HD Localization Model contains data, such as traffic signs or other road objects, that helps autonomous vehicles accurately locate where they are within a road network. hereHDLMReader objects do not support reading layers from this model.

See Also

hereHDLMReader | plot | read

More About

- "Access HERE HD Live Map Data" on page 4-7
- "Use HERE HD Live Map Data to Verify Lane Configurations"

External Websites

- HD Live Map Data Specification

Rotations, Orientations, and Quaternions for Automated Driving

A quaternion is a four-part hypercomplex number used to describe three-dimensional rotations and orientations. Quaternions have applications in many fields, including aerospace, computer graphics, and virtual reality. 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 using a `quaternion` object, perform mathematical operations on it, or convert it to other rotation formats, such as Euler angles and rotation matrices.

You can use quaternions to perform 3-D point and frame rotations.

- With point rotations, you rotate points in a static frame of reference.
- With frame rotations, you rotate the frame of reference around a static point to convert the frame into the coordinate system relative to the point.

You can define these rotations by using an axis of rotation and an angle of rotation about that axis. Quaternions encapsulate the axis and angle of rotation and have an algebra for manipulating these rotations. The `quaternion` object uses the "right-hand rule" convention to define rotations. That is, positive rotations are clockwise around the axis of rotation when viewed from the origin.

Quaternion Format

A quaternion number is represented in this form:

$$a + b\mathrm{i} + c\mathrm{j} + d\mathrm{k}$$

a, *b*, *c*, and *d* are real numbers. These coefficients are known as the parts of the quaternion.

i, j, and k are the complex elements of a quaternion. These elements satisfy the equation $\mathrm{i}^2 = \mathrm{j}^2 = \mathrm{k}^2 = \mathrm{ijk} = -1$.

The quaternion parts *a*, *b*, *c*, and *d* specify the axis and angle of rotation. For a rotation of α radians about a rotation axis represented by the unit vector [*x*, *y*, *z*], the quaternion describing the rotation is given by this equation:

$$\cos\left(\frac{\alpha}{2}\right) + \sin\left(\frac{\alpha}{2}\right)(x\mathbf{i} + y\mathbf{j} + z\mathbf{k})$$

Quaternion Creation

You can create quaternions in multiple ways. For example, create a quaternion by specifying its parts.

```
q = quaternion(1,2,3,4)
```

```
q = 

  quaternion


     1 + 2i + 3j + 4k
```

You can create arrays of quaternions in the same way. For example, create a 2-by-2 quaternion array by specifying four 2-by-2 matrices.

```
qArray = quaternion([1 10; -1 1], [2 20; -2 2], [3 30; -3 3], [4 40; -4 4])
```

```
qArray = 

  2x2 quaternion array


      1 +  2i +  3j +  4k     10 + 20i + 30j + 40k
     -1 -  2i -  3j -  4k      1 +  2i +  3j +  4k
```

You can also use four-column arrays to construct quaternions, where each column represents a quaternion part. For example, create an array of quaternions that represent random rotations.

```
qRandom = randrot(4,1)
```

```
qRandom = 

  4x1 quaternion array


      0.14515 + 0.086053i +  0.96601j +  0.19583k
      0.51365 -  0.36627i +  0.77569j - 0.017661k
     -0.81811 -  0.15704i -   0.4889j +  0.25887k
      0.27111 +  0.10774i +  0.95433j - 0.064451k
```

Index and manipulate quaternions just like any other array. For example, index a quaternion from the qRandom quaternion array.

```
qRandom(3)
```

```
ans = 

  quaternion


    -0.81811 - 0.15704i -  0.4889j + 0.25887k
```

Reshape the quaternion array.

```
reshape(qRandom,2,2)
```

```
ans = 

  2x2 quaternion array


      0.14515 + 0.086053i +  0.96601j +  0.19583k     -0.81811 -  0.15704i -   0.4889j +  0.2588
      0.51365 -  0.36627i +  0.77569j - 0.017661k      0.27111 +  0.10774i +  0.95433j - 0.06445
```

Concatenate the quaternion array with the first quaternion that you created.

```
[qRandom; q]
```

```
ans = 

  5x1 quaternion array


      0.14515 + 0.086053i +  0.96601j +  0.19583k
      0.51365 -  0.36627i +  0.77569j - 0.017661k
     -0.81811 -  0.15704i -   0.4889j +  0.25887k
      0.27111 +  0.10774i +  0.95433j - 0.064451k
            1 +        2i +        3j +        4k
```

Quaternion Math

Quaternions have well-defined arithmetic operations. To apply these operations, first define two quaternions by specifying their real-number parts.

```
q1 = quaternion(1,2,3,4)
```

```
q1 = quaternion
     1 + 2i + 3j + 4k
```

```
q2 = quaternion(-5,6,-7,8)
```

```
q2 = quaternion
    -5 + 6i - 7j + 8k
```

Addition of quaternions is similar to complex numbers, where parts are added independently.

```
q1 + q2
```

```
ans = quaternion
     -4 +  8i -  4j + 12k
```

Subtraction of quaternions works similar to addition of quaternions.

```
q1 - q2
```

```
ans = quaternion
      6 -  4i + 10j -  4k
```

Because the complex elements of quaternions must satisfy the equation

$$i^2 = j^2 = k^2 = ijk = -1,$$

multiplication of quaternions is more complex than addition and subtraction. Given this requirement, multiplication of quaternions is not commutative. That is, when multiplying quaternions, reversing the order of the quaternions changes the result of their product.

```
q1 * q2
```

```
ans = quaternion
    -28 + 48i - 14j - 44k
```

```
q2 * q1
```

```
ans = quaternion
    -28 - 56i - 30j + 20k
```

However, every quaternion has a multiplicative inverse, so you can divide quaternions. Right division of `q1` by `q2` is equivalent to $q1(q2^{-1})$.

```
q1 ./ q2
```

```
ans = quaternion
      0.10345 -   0.3908i - 0.091954j + 0.022989k
```

Left division of `q1` by `q2` is equivalent to $(q2^{-1})q1$.

```
q1 .\ q2
```

```
ans = quaternion
     0.6 - 1.2i +   0j +   2k
```

The conjugate of a quaternion is formed by negating each of the complex parts, similar to conjugate of a complex number.

```
conj(q1)
```

```
ans = quaternion
     1 - 2i - 3j - 4k
```

To describe a rotation using a quaternion, the quaternion must be a *unit quaternion*. A unit quaternion has a norm of 1, where the norm is defined as

$$\text{norm}(q) = \sqrt{a^2 + b^2 + c^2 + d^2}.$$

Normalize a quaternion.

```
qNormalized = normalize(q1)
```

```
qNormalized = quaternion
     0.18257 + 0.36515i + 0.54772j +  0.7303k
```

Verify that this normalized unit quaternion has a norm of 1.

```
norm(qNormalized)
```

```
ans = 1.0000
```

The rotation matrix for the conjugate of a normalized quaternion is equal to the inverse of the rotation matrix for that normalized quaternion.

```
rotmat(conj(qNormalized),'point')

ans = 3×3

   -0.6667    0.6667    0.3333
    0.1333   -0.3333    0.9333
    0.7333    0.6667    0.1333
```

```
inv(rotmat(qNormalized,'point'))

ans = 3×3

   -0.6667    0.6667    0.3333
    0.1333   -0.3333    0.9333
    0.7333    0.6667    0.1333
```

Extract Quaternions from Transformation Matrix

If you have a 3-D transformation matrix created using functions such as `rigid3d` or `affine3d`, you can extract the rotation matrix from it and represent it in the form of a quaternion. However, before performing this conversion, you must first convert the rotation matrix from the postmultiply format to the premultiply format expected by quaternions.

Postmultiply Format

To perform rotations using the rotation matrix part of a transformation matrix, multiply an (*x*, *y*, *z*) point by this rotation matrix.

- In point rotations, this point is rotated within a frame of reference.
- In frame rotations, the frame of reference is rotated around this point.

Transformation matrices represented by `rigid3d` or `affine3d` objects use a *postmultiply format*. In this format, the point is multiplied by the rotation matrix, in that order. To satisfy the matrix multiplication, the point and its corresponding translation vector must be row vectors.

This equation shows the postmultiply format for a rotation matrix, *R,* and a translation vector, *t*.

$$[x' \; y' \; z'] = [x \; y \; z]\begin{bmatrix} R_{11} & R_{12} & R_{13} \\ R_{21} & R_{22} & R_{23} \\ R_{31} & R_{32} & R_{33} \end{bmatrix} + [t_x \; t_y \; t_z]$$

This format also applies when *R* and *t* are combined into a homogeneous transformation matrix. In this matrix, the 1 is used to satisfy the matrix multiplication and can be ignored.

$$[x' \; y' \; z' \; 1] = [x \; y \; z \; 1]\begin{bmatrix} R_{11} & R_{12} & R_{13} & 0 \\ R_{21} & R_{22} & R_{23} & 0 \\ R_{31} & R_{32} & R_{33} & 0 \\ t_x & t_y & t_z & 1 \end{bmatrix}$$

Premultiply Format

In the *premultiply format*, the rotation matrix is multiplied by the point, in that order. To satisfy the matrix multiplication, the point and its corresponding translation vector must be column vectors.

This equation shows the premultiply format, where R is the rotation matrix and t is the translation vector.

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} R_{11} & R_{12} & R_{13} \\ R_{21} & R_{22} & R_{23} \\ R_{31} & R_{32} & R_{33} \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix}$$

As with the postmultiply case, this format also applies when R and t are combined into a homogeneous transformation matrix.

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} R_{11} & R_{12} & R_{13} & t_x \\ R_{21} & R_{22} & R_{23} & t_y \\ R_{31} & R_{32} & R_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Convert from Postmultiply to Premultiply Format

To convert a rotation matrix to the premultiply format expected by quaternions, take the transpose of the rotation matrix.

Create a 3-D rigid geometric transformation object from a rotation matrix and translation vector. The angle of rotation, θ, is in degrees.

```
theta = 30;
rot = [ cosd(theta) sind(theta) 0; ...
       -sind(theta) cosd(theta) 0; ...
                  0           0 1];
trans = [2 3 4];

tform = rigid3d(rot,trans)
```

```
tform = 
  rigid3d with properties:

       Rotation: [3x3 double]
    Translation: [2 3 4]
```

The elements of the rotation matrix are ordered for rotating points using the postmultiply format. Convert the matrix to the premultiply format by taking its transpose. Notice that the second element of the first row and first column swap locations.

```
rotPost = tform.Rotation
```

```
rotPost = 3×3

    0.8660    0.5000         0
   -0.5000    0.8660         0
         0         0    1.0000
```

```
rotPre = rotPost.'
```

```
rotPre = 3×3

    0.8660   -0.5000         0
    0.5000    0.8660         0
         0         0    1.0000
```

Create a quaternion from the premultiply version of the rotation matrix. Specify that the rotation matrix is configured for point rotations.

```
q = quaternion(rotPre,'rotmat','point')
```

```
q = quaternion
     0.96593 +       0i +       0j + 0.25882k
```

To verify that the premultiplied quaternion and the postmultiplied rotation matrix produce the same results, rotate a sample point using both approaches.

```
point = [1 2 3];
rotatedPointQuaternion = rotatepoint(q,point)
```

```
rotatedPointQuaternion = 1×3

   -0.1340    2.2321    3.0000
```

```
rotatedPointRotationMatrix = point * rotPost
```

```
rotatedPointRotationMatrix = 1×3

   -0.1340    2.2321    3.0000
```

To convert back to the original rotation matrix, extract a rotation matrix from the quaternion. Then, create a `rigid3d` object by using the transpose of this rotation matrix.

```
R = rotmat(q,'point');
T = rigid3d(R',trans);
T.Rotation
```

```
ans = 3×3

    0.8660    0.5000         0
   -0.5000    0.8660         0
         0         0    1.0000
```

See Also

`affine3d` | `quaternion` | `rigid3d` | `rotateframe` | `rotatepoint`

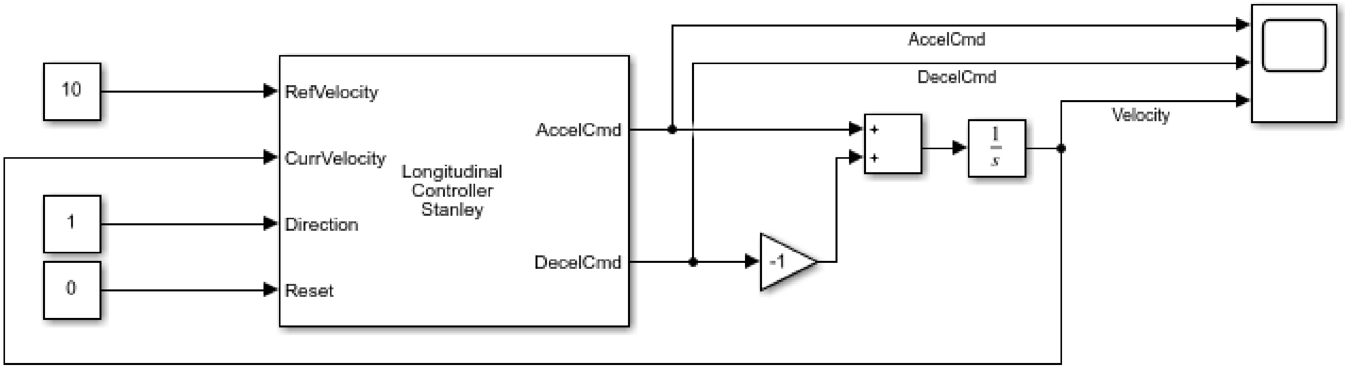
More About

- “Build a Map from Lidar Data”
- “Build a Map from Lidar Data Using SLAM”

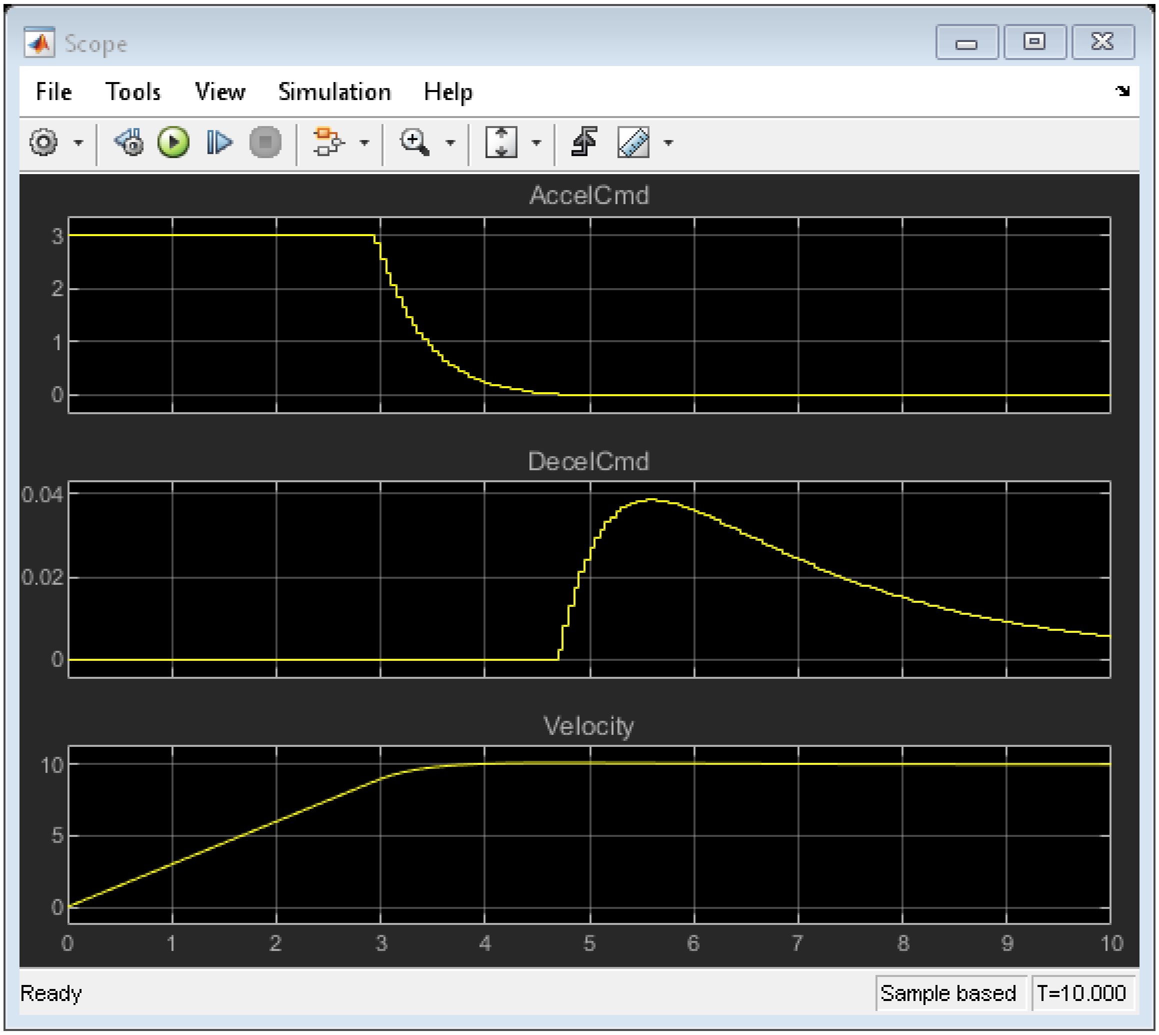
Control Vehicle Velocity

This model uses a Longitudinal Controller Stanley block to control the velocity of a vehicle in forward motion. In this model, the vehicle accelerates from 0 to 10 meters per second.

The Longitudinal Controller Stanley block is a discrete proportional-integral controller with integral anti-windup. Given the current velocity and driving direction of a vehicle, the block outputs the acceleration and deceleration commands needed to match the specified reference velocity.



Run the model. Then, open the scope to see the change in velocity and the corresponding acceleration and deceleration commands.



The Longitudinal Controller Stanley block saturates the acceleration command at a maximum value of 3 meters per second. The **Maximum longitudinal acceleration (m/s^2)** parameter of the block determines this maximum value. Try tuning this parameter and resimulating the model. Observe the effects of the change on the scope. Other parameters that you can tune include the gain coefficients of the proportional and integral components of the block, using the **Proportional gain, Kp** and **Integral gain, Ki** parameters, respectively.

See Also

Lateral Controller Stanley | Longitudinal Controller Stanley

More About

- “Automated Parking Valet in Simulink”

Velocity Profile of Straight Path

This model uses a Velocity Profiler block to generate a velocity profile for a vehicle traveling forward on a straight, 100-meter path that has no changes in direction.

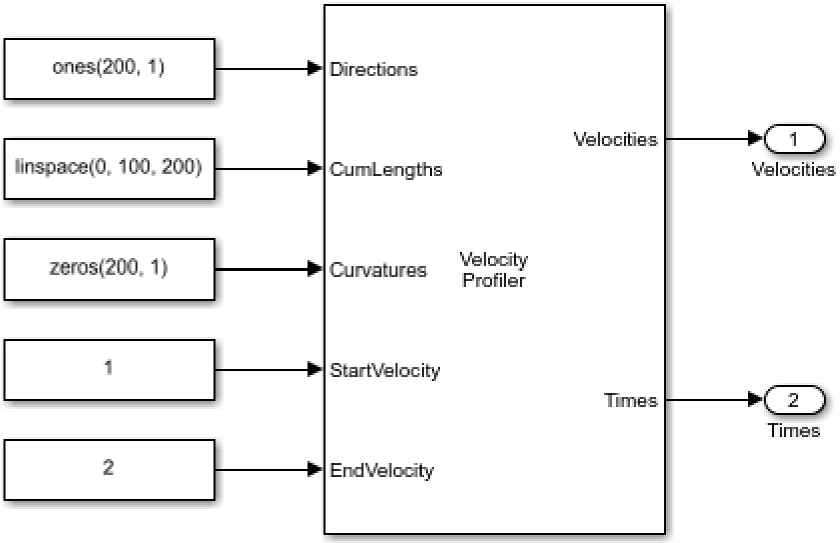
The Velocity Profiler block generates velocity profiles based on the speed, acceleration, and jerk constraints that you specify using parameters. You can use the generated velocity profile as the input reference velocities of a vehicle controller.

This model is for illustrative purposes and does not show how to use the Velocity Profiler block in a complete automated driving model. To see how to use this block in such a model, see the “Automated Parking Valet in Simulink” example.

Open and Inspect Model

The model consists of a single Velocity Profiler block with constant inputs. Open the model.

```
model = 'VelocityProfileStraightPath';
open_system(model)
```



The first three inputs specify information about the driving path.

- The **Directions** input specifies the driving direction of the vehicle along the path, where 1 means forward and –1 means reverse. Because the vehicle travels only forward, the direction is 1 along the entire path.
- The **CumLengths** input specifies the length of the path. The path is 100 meters long and is composed of a sequence of 200 cumulative path lengths.

- The **Curvatures** input specifies the curvature along the path. Because this path is straight, the curvature is 0 along the entire path.

In a complete automated driving model, you can obtain these input values from the output of a Path Smoother Spline block, which smooths a path based on a set of poses.

The **StartVelocity** and **EndVelocity** inputs specify the velocity of the vehicle at the start and end of the path, respectively. The vehicle starts the path traveling at a velocity of 1 meter per second and reaches the end of the path traveling at a velocity of 2 meters per second.

Generate Velocity Profile

Simulate the model to generate the velocity profile.

```
out = sim(model);
```

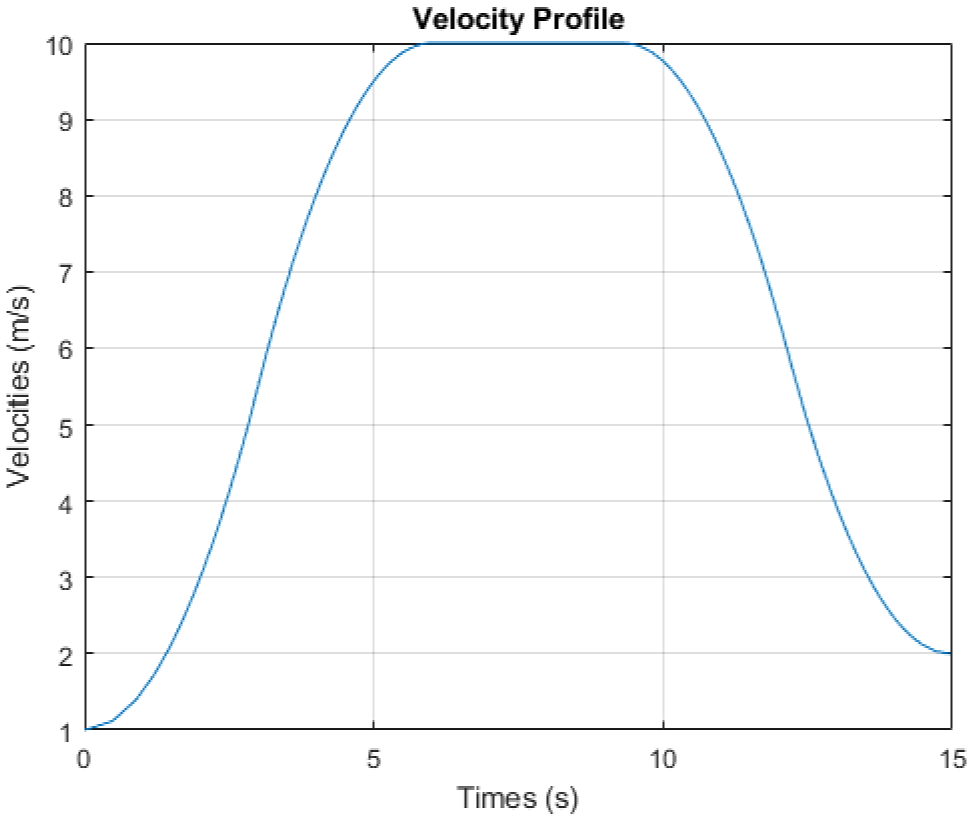
The output velocity profile is a sequence of velocities along the path that meet the speed, acceleration, and jerk constraints specified in the parameters of the Velocity Profiler block.

The block also outputs the times at which the vehicle arrives at each point along the path. You can use this output to visualize the velocities over time.

Visualize Velocity Profile

Use the simulation output to plot the velocity profile.

```
t = length(out.tout);
velocities = out.yout.signals(1).values(:,:,t);
times = out.yout.signals(2).values(:,:,t);

plot(times,velocities)
title('Velocity Profile')
xlabel('Times (s)')
ylabel('Velocities (m/s)')
grid on
```

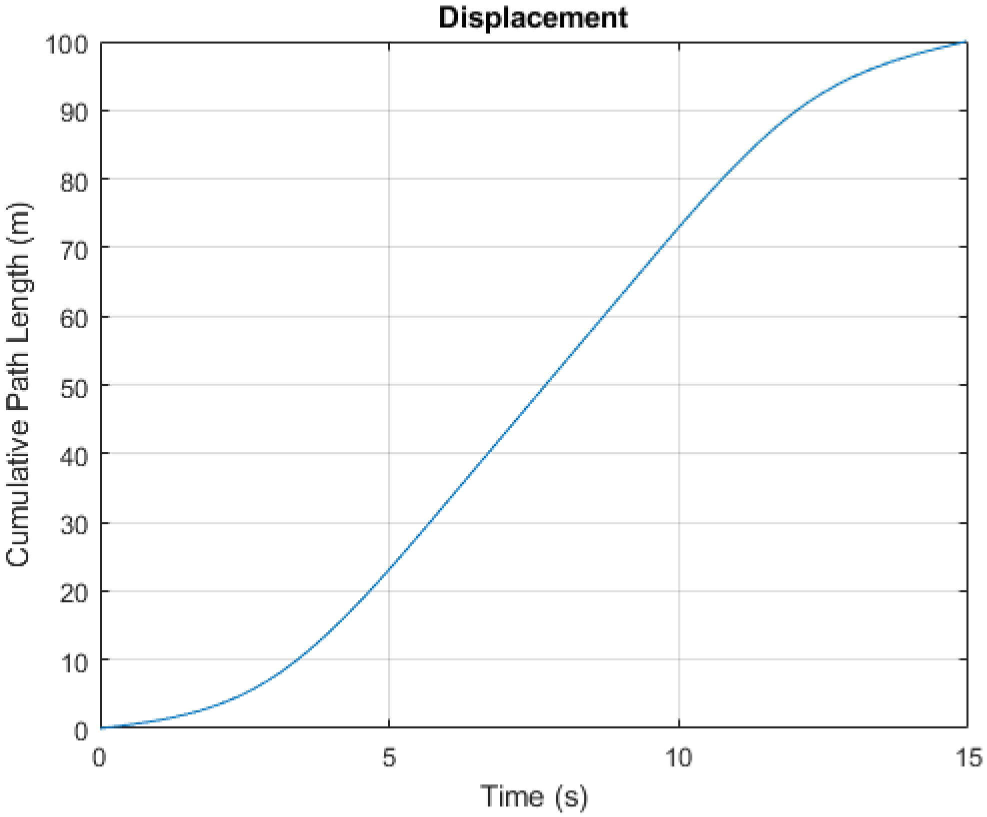


A vehicle that follows this velocity profile:

1. Starts at a velocity of 1 meter per second
2. Accelerates to a maximum speed of 10 meters per second, as specified by the **Maximum allowable speed (m/s)** parameter of the Velocity Profiler block
3. Decelerates to its ending velocity of 2 meters per second

For comparison, plot the displacement of the vehicle over time by using the cumulative path lengths.

```
figure
cumLengths = linspace(0,100,200);
plot(times,cumLengths)
title('Displacement')
xlabel('Time (s)')
ylabel('Cumulative Path Length (m)')
grid on
```



For details on how the block calculates the velocity profile, see the “Algorithms” section of the Velocity Profiler block reference page.

See Also

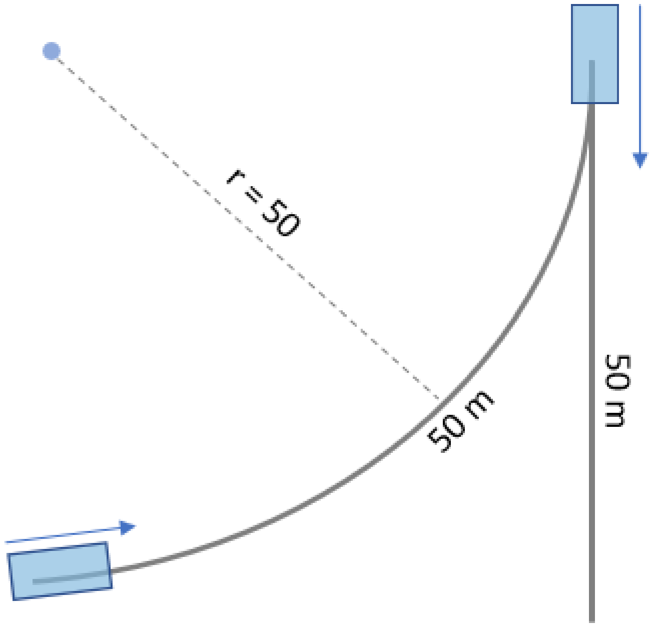
Path Smoother Spline | Velocity Profiler

More About

- “Velocity Profile of Path with Curve and Direction Change” on page 4-48
- “Automated Parking Valet in Simulink”

Velocity Profile of Path with Curve and Direction Change

This model uses a Velocity Profiler block to generate a velocity profile for a driving path that includes a curve and a change in direction. In this model, the vehicle travels forward on a curved path for 50 meters, and then travels straight in reverse for another 50 meters.



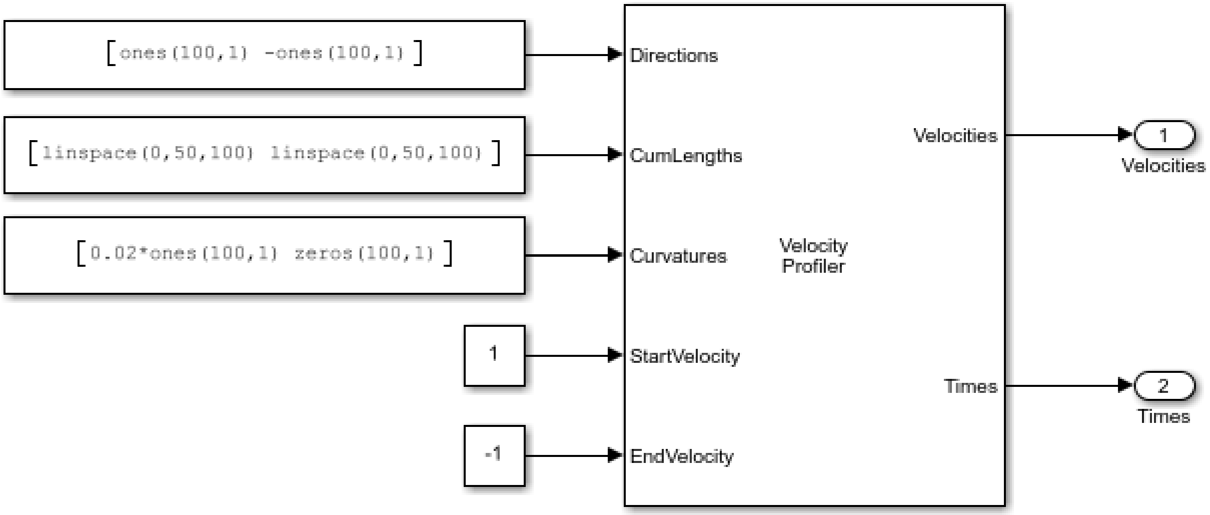
The Velocity Profiler block generates velocity profiles based on the speed, acceleration, and jerk constraints that you specify using parameters. You can use the generated velocity profile as the input reference velocities of a vehicle controller.

This model is for illustrative purposes and does not show how to use the Velocity Profiler block in a complete automated driving model. To see how to use this block in such a model, see the “Automated Parking Valet in Simulink” example.

Open and Inspect Model

The model consists of a single Velocity Profiler block with constant inputs. Open the model.

```
model = 'VelocityProfileCurvedPathDirectionChanges';
open_system(model)
```



The first three inputs specify information about the driving path.

- The **Directions** input specifies the driving direction of the vehicle along the path, where 1 means forward and –1 means reverse. In the first path segment, because the vehicle travels only forward, the direction is 1 along the entire segment. In the second path segment, because the vehicle travels only in reverse, the direction is –1 along the entire segment.
- The **CumLengths** input specifies the length of the path. The path consists of two 50-meter segments. The first segment represents a forward left turn, and the second segment represents a straight path in reverse. The path is composed of a sequence of 200 cumulative path lengths, with 100 lengths per 50-meter segment.
- The **Curvatures** input specifies the curvature along this path. The curvature of the first path segment corresponds to a turning radius of 50 meters. Because the second path segment is straight, the curvature is 0 along the entire segment.

In a complete automated driving model, you can obtain these input values from the output of a Path Smoother Spline block, which smooths a path based on a set of poses.

The **StartVelocity** and **EndVelocity** inputs specify the velocity of the vehicle at the start and end of the path, respectively. The vehicle starts the path traveling at a velocity of 1 meter per second and reaches the end of the path traveling at a velocity of –1 meters per second. The negative velocity indicates that the vehicle is traveling in reverse at the end of the path.

Generate Velocity Profile

Simulate the model to generate the velocity profile.

```
out = sim(model);
```

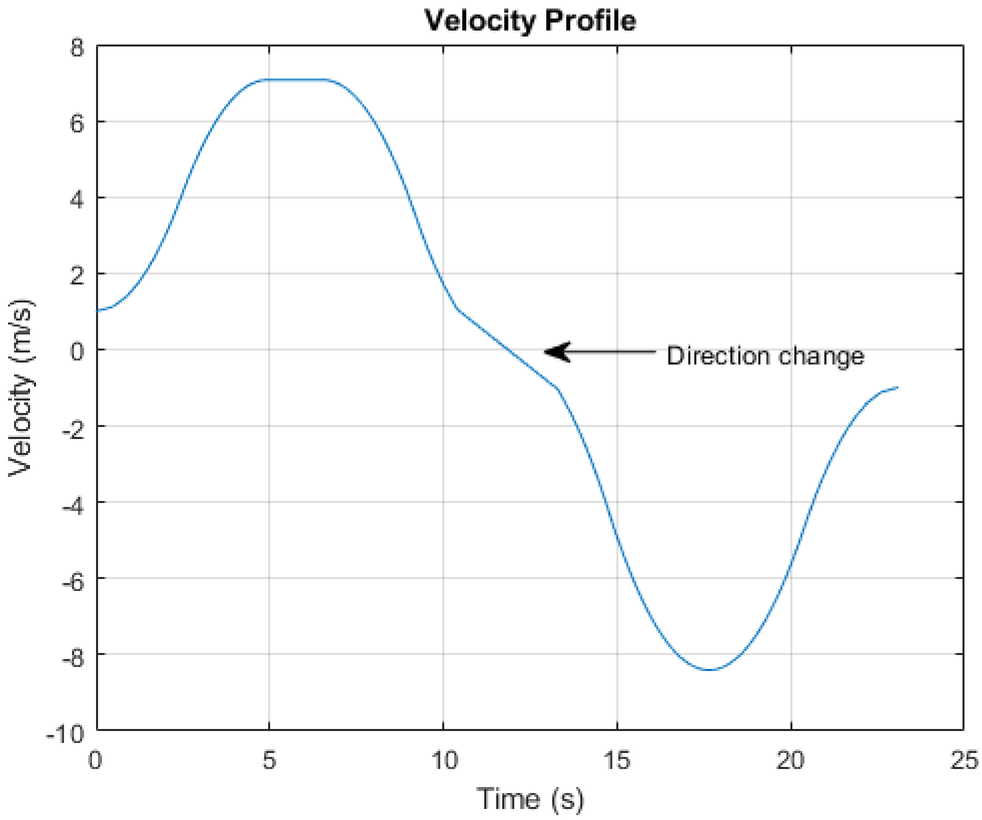
The output velocity profile is a sequence of velocities along the path that meet the speed, acceleration, and jerk constraints specified in the parameters of the Velocity Profiler block.

The block also outputs the times at which the vehicle arrives at each point along the path. You can use this output to visualize the velocities over time.

Visualize Velocity Profile

Use the simulation output to plot the velocity profile.

```
t = length(out.tout);
velocities = out.yout.signals(1).values(:,:,t);
times = out.yout.signals(2).values(:,:,t);

plot(times,velocities)
title('Velocity Profile')
xlabel('Time (s)')
ylabel('Velocity (m/s)')
annotation('textarrow',[0.63 0.53],[0.56 0.56],'String',{'Direction change'});
grid on
```



For this path, the Velocity Profiler block generates two separate velocity profiles: one for the forward left turn and one for the straight reverse motion. In the final output, the block concatenates these velocities into a single velocity profile.

A vehicle that follows this velocity profile:

1. Starts at a velocity of 1 meter per second
2. Accelerates forward
3. Decelerates until its velocity reaches 0, so that the vehicle can switch driving directions
4. Accelerates in reverse

5 Decelerates until it reaches its ending velocity

In both driving directions, the vehicle fails to reach the maximum speed specified by the **Maximum allowable speed (m/s)** parameter of the Velocity Profiler block, because the path is too short.

For details on how the block calculates the velocity profile, see the “Algorithms” section of the Velocity Profiler block reference page.

See Also

Path Smoother Spline | Velocity Profiler

More About

- “Velocity Profile of Straight Path” on page 4-44
- “Automated Parking Valet in Simulink”

Cuboid Driving Scenario Simulation

Build a Driving Scenario and Generate Synthetic Detections

This example shows you how to build a driving scenario and generate vision and radar sensor detections from it by using the **Driving Scenario Designer** app. You can use these detections to test your controllers or sensor fusion algorithms.

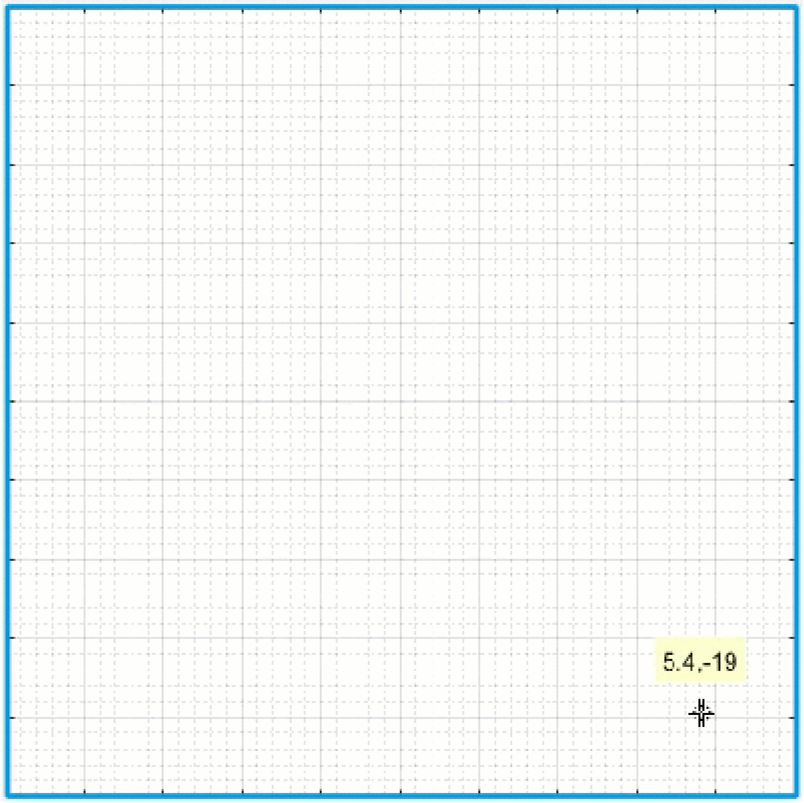
This example covers the entire workflow for creating a scenario and generating synthetic detections. Alternatively, you can generate detections from prebuilt scenarios. For more details, see “Prebuilt Driving Scenarios in Driving Scenario Designer” on page 5-14.

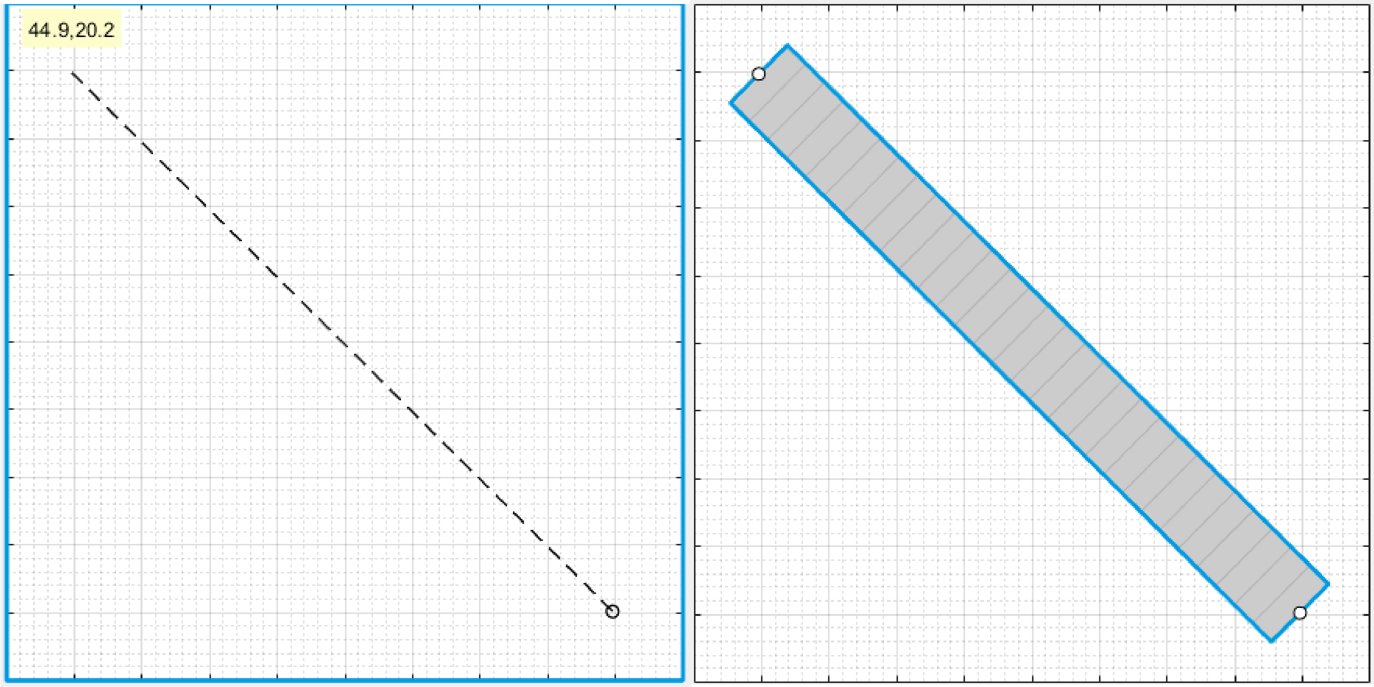
Create a New Driving Scenario

To open the app, at the MATLAB command prompt, enter `drivingScenarioDesigner`.

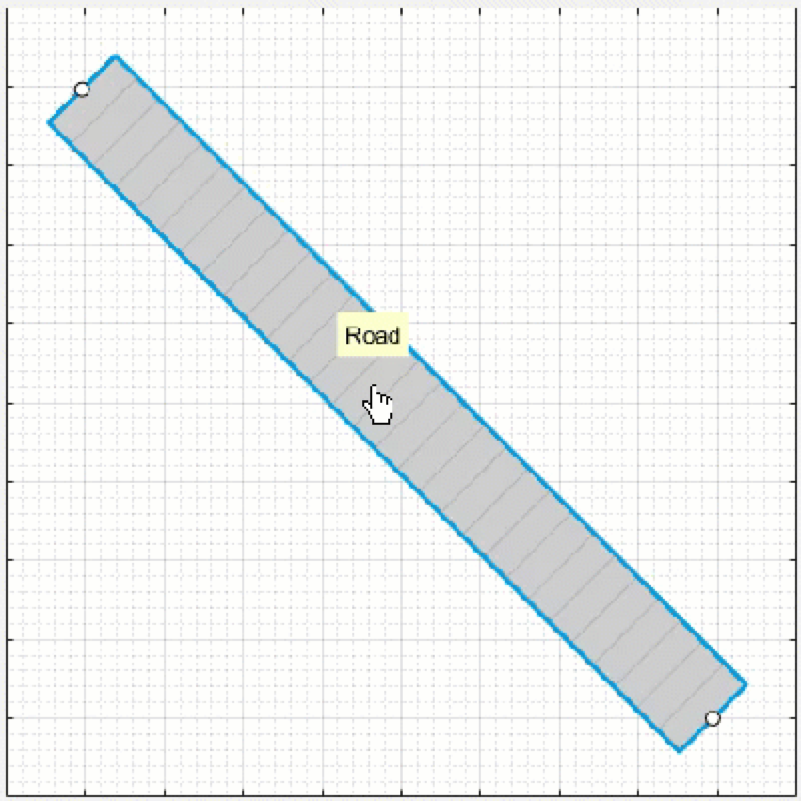
Add a Road

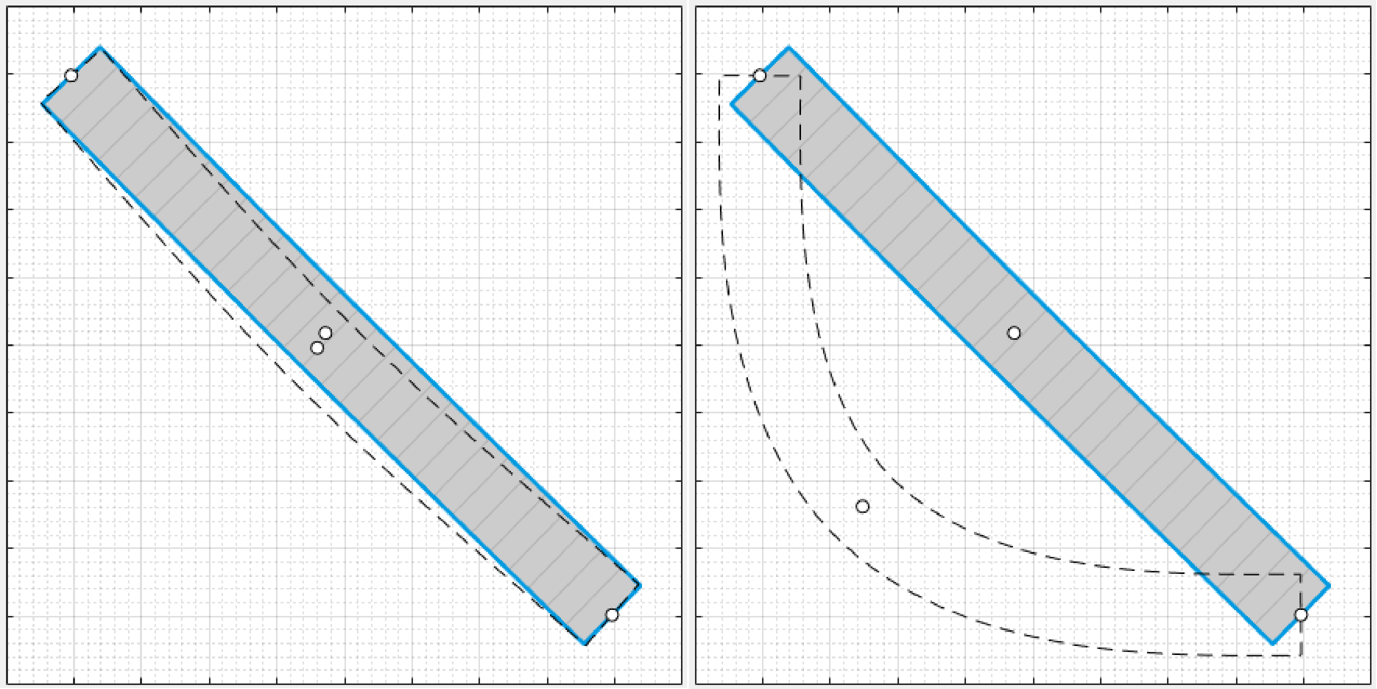
Add a curved road to the scenario canvas. On the app toolstrip, click **Add Road**. Then click one corner of the canvas, extend the road to the opposite corner, and double-click to create the road.





To make the road curve, add a road center around which to curve it. Right-click the middle of the road and select **Add Road Center**. Then drag the added road center to one of the empty corners of the canvas.

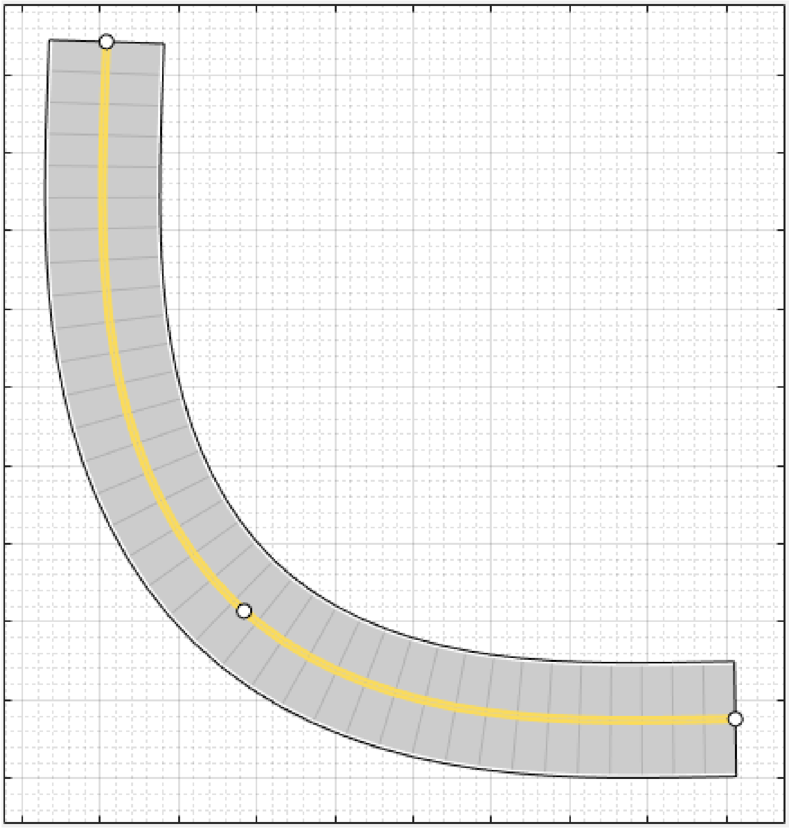




To adjust the road further, you can click and drag any of the road centers. To create more complex curves, add more road centers.

Add Lanes

By default, the road is a single lane and has no lane markings. To make the scenario more realistic, convert the road into a two-lane highway. In the left pane, on the **Roads** tab, expand the **Lanes** section. Set the **Number of lanes** to `[1 1]` and the **Lane Width** to `3.6` meters, which is a typical highway lane width.



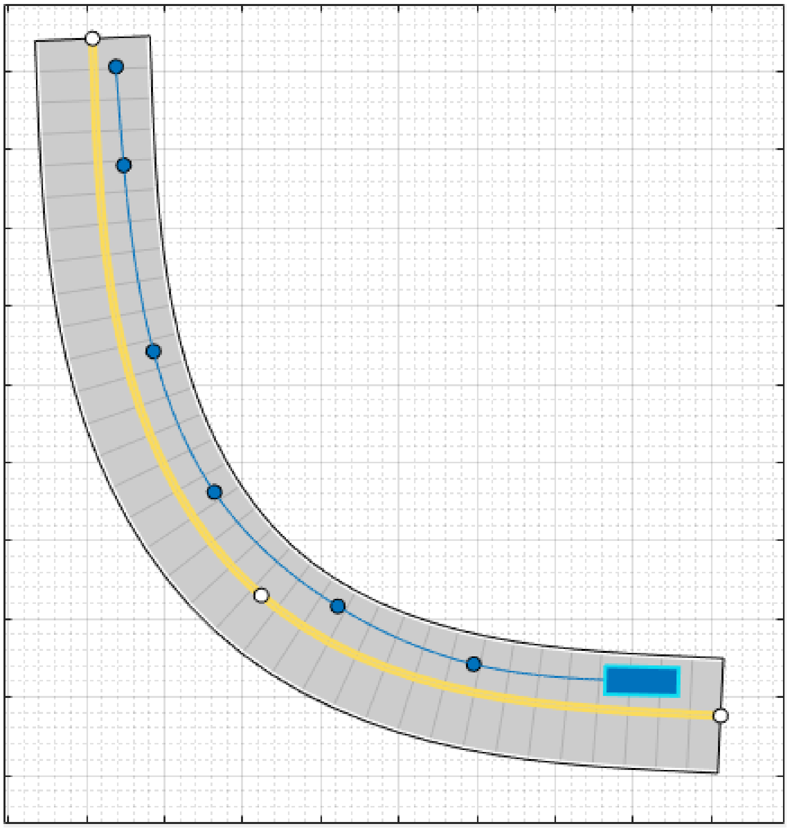
The white, solid lanes markings on either edge of the road indicate the road shoulder. The yellow, double-solid lane marking in the center indicates that the road is two-way. To inspect or modify these lanes, from the **Marking** list, select one of the lanes and modify the lane parameters.

Add Vehicles

By default, the first car that you add to a scenario is the ego vehicle, which is the main car in the driving scenario. The ego vehicle contains the sensors that detect the lane markings, pedestrians, or other cars in the scenario. Add the ego vehicle, and then add a second car for the ego vehicle to detect.

Add Ego Vehicle

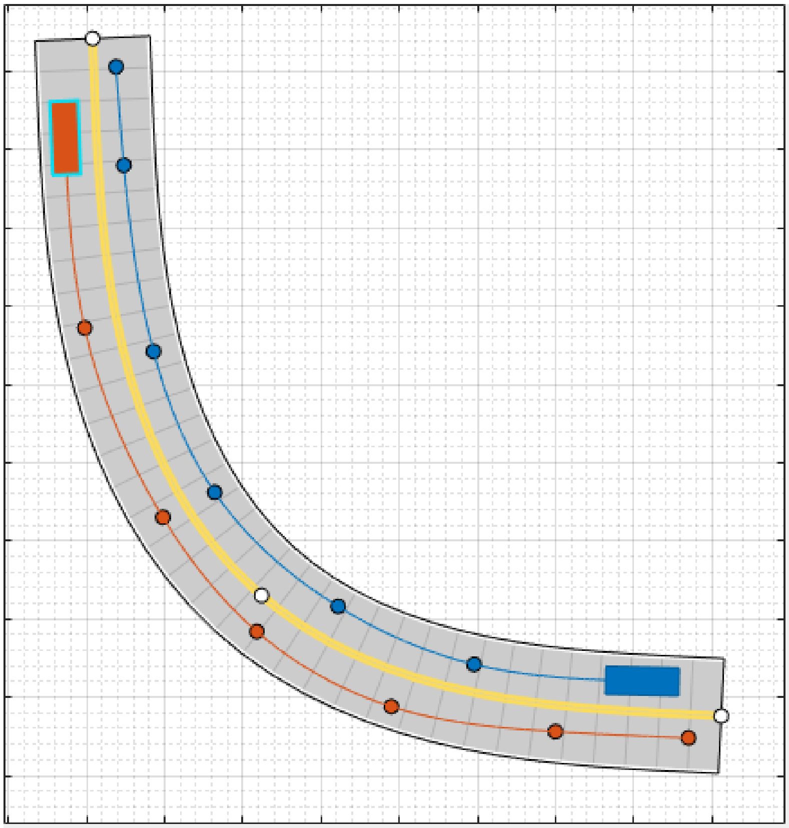
To add the ego vehicle, right-click one end of the road, and select **Add Car**. To specify the trajectory of the car, right-click the car, select **Add Waypoints**, and add waypoints along the road for the car to pass through. After you add the last waypoint along the road, press **Enter**. The car autorotates in the direction of the first waypoint. For finer precision over the trajectory, you can adjust the waypoints. You can also right-click the path to add new waypoints.



Now adjust the speed of the car. In the left pane, on the **Actors** tab, set **Constant Speed** to 15 m/s. For more control over the speed of the car, clear the **Constant Speed** check box and set the velocity between waypoints in the **Waypoints** table.

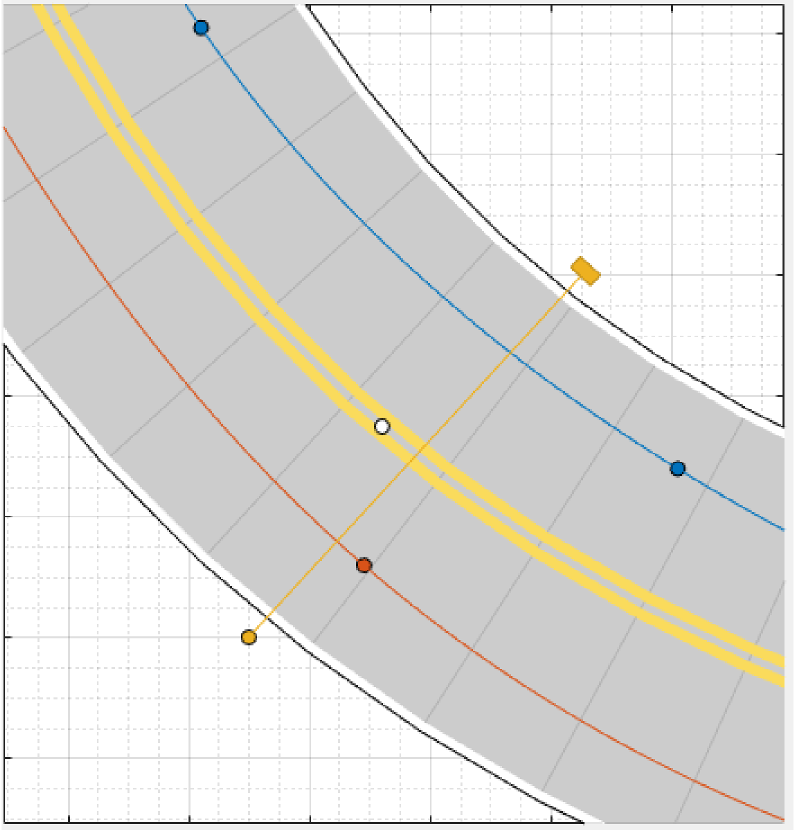
Add Second Car

Add a vehicle for the ego vehicle to detect. On the app toolstrip, click **Add Actor** and select **Car**. Add the second car with waypoints, driving in the lane opposite from the ego vehicle and on the other end of the road. Leave the speed and other settings of the car unchanged.



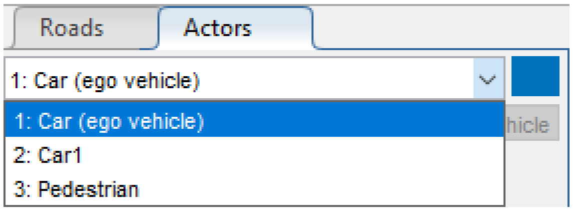
Add a Pedestrian

Add to the scenario a pedestrian crossing the road. Zoom in (**Ctrl+Plus**) on the middle of the road, right-click one side of the road, and click **Add Pedestrian**. Then, to set the path of the pedestrian, add a waypoint on the other side of the road.



By default, the color of the pedestrian nearly matches the color of the lane markings. To make the pedestrian stand out more, from the **Actors** tab, click the corresponding color patch for the pedestrian to modify its color.

To test the speed of the cars and the pedestrian, run the simulation. Adjust actor speeds or other properties as needed by selecting the actor from the left pane of the **Actors** tab.



Add Sensors

Add front-facing radar and vision (camera) sensors to the ego vehicle. Use these sensors to generate detections of the pedestrian, the lane boundaries, and the other vehicle.

Add Camera

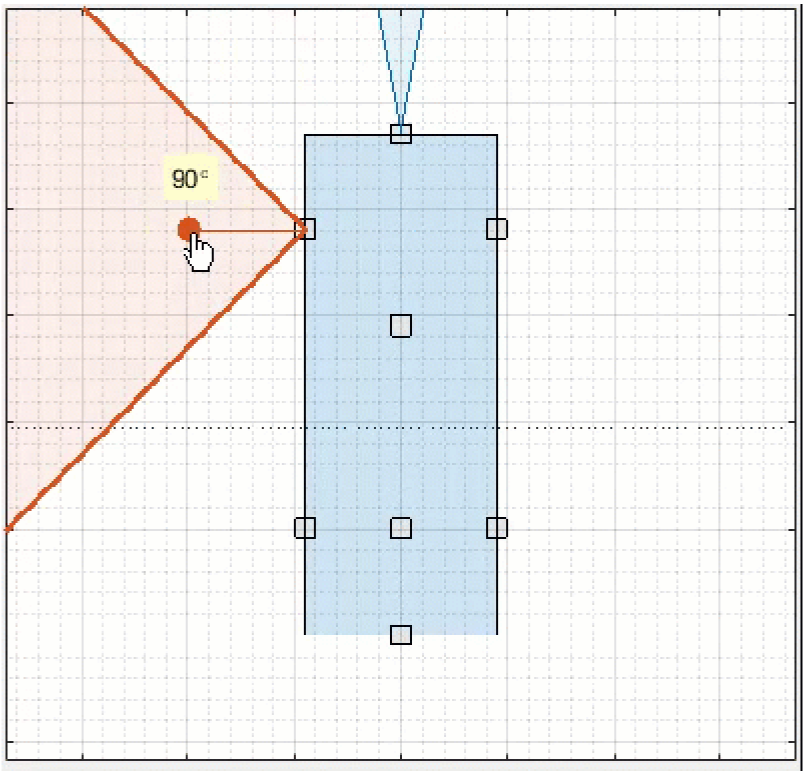
On the app toolstrip, click **Add Camera**. The sensor canvas shows standard locations at which to place sensors. Click the front-most predefined sensor location to add a camera sensor to the front bumper of the ego vehicle. To place sensors more precisely, you can disable snapping options. In the bottom-left corner of the sensor canvas, click the Configure the Sensor Canvas button ⚙.

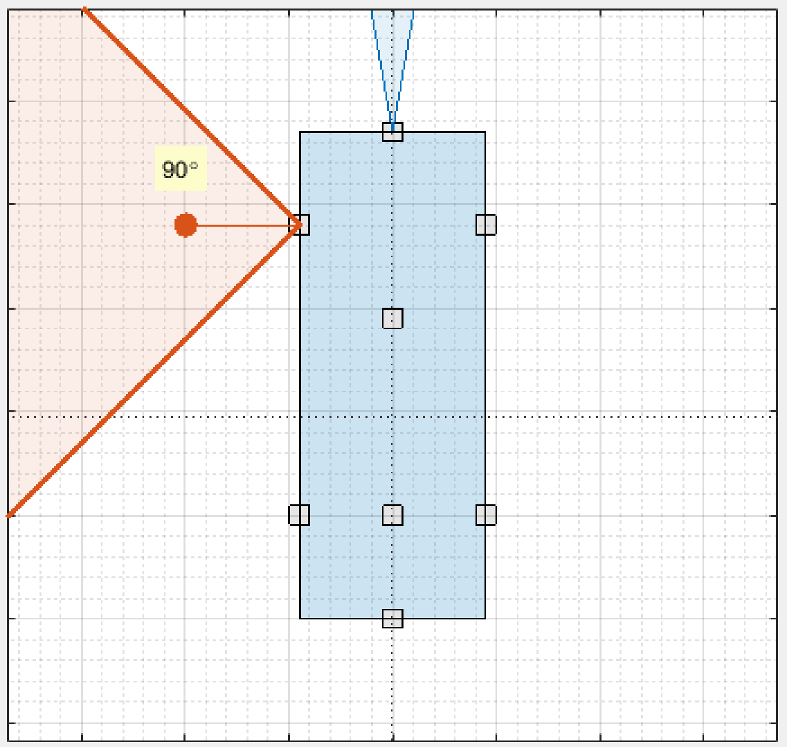
By default, the camera detects only actors and not lanes. To enable lane detections, on the **Sensors** tab in the left pane, expand the **Detection Parameters** section and set **Detection Type** to `Objects & Lanes`. Then expand the **Lane Settings** section and update the settings as needed.

Add Radar

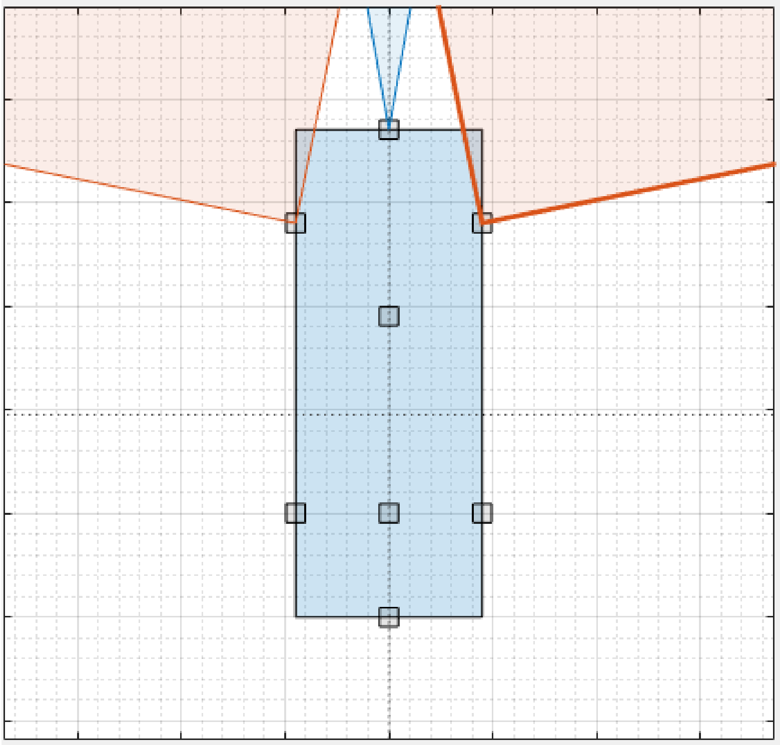
Snap a radar sensor to the front-left wheel. Right-click the predefined sensor location for the wheel and select **Add Radar**. By default, sensors added to the wheels are short range.

Tilt the radar sensor toward the front of the car. Move your cursor over the coverage area, then click and drag the angle marking.





Add an identical radar sensor to the front-right wheel. Right-click the sensor on the front-left wheel and click **Copy**. Then right-click the predefined sensor location for the front-right wheel and click **Paste**. The orientation of the copied sensor mirrors the orientation of the sensor on the opposite wheel.

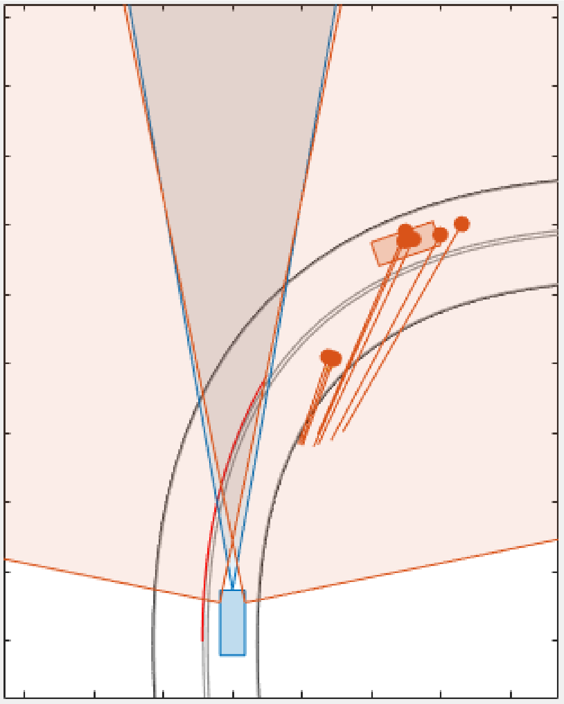


The camera and radar sensors now provide overlapping coverage of the front of the ego vehicle.

Generate Synthetic Detections

Run Scenario

To generate detections from the sensors, click **Run**. As the scenario runs, the **Ego-Centric View** displays the scenario from the perspective of the ego vehicle. The **Bird’s-Eye Plot** displays the detections.



To turn off certain types of detections, in the bottom-left corner of the bird's-eye plot, click the Configure the Bird's-Eye Plot button ⚙.

By default, the scenario ends when the first actor stops. To run the scenario for a set amount of time, on the app toolstrip, click **Settings** and change the stop condition.

Export Sensor Detections

- To export detections to the MATLAB workspace, on the app toolstrip, select **Export** > **Export Sensor Data**. Name the workspace variable and click **OK**. The app saves the sensor data as a structure containing the actor poses, object detections, and lane detections at each time step.
- To export a MATLAB function that generates the scenario and its detections, select **Export** > **Export MATLAB Function**. This function returns the sensor detections as a structure, the scenario as a `drivingScenario` object, and the sensor models as `visionDetectionGenerator` and `radarDetectionGenerator` System objects. By modifying this function, you can create variations of the original scenario. For an example of this process, see “Create Driving Scenario Variations Programmatically” on page 5-80.

Save Scenario

After you generate the detections, click **Save** to save the scenario file. In addition, you can save the sensor models as separate files. You can also save the road and actor models together as a separate scenario file.

You can reopen this scenario file from the app. Alternatively, at the MATLAB command prompt, you can use this syntax.

```
drivingScenarioDesigner(scenarioFileName)
```

You can also reopen the scenario by using the exported `drivingScenario` object. At the MATLAB command prompt, use this syntax, where `scenario` is the name of the exported object.

```
drivingScenarioDesigner(scenario)
```

To reopen sensors, use this syntax, where `sensors` is a `radarDetectionGenerator` object, `visionDetectionGenerator` object, or a cell array of such objects.

```
drivingScenarioDesigner(scenario,sensors)
```

If you are developing a driving algorithm in Simulink, you can use a Scenario Reader block to read roads and actors from the scenario file or `drivingScenario` object into your model. This block does not directly read sensor data. To add sensors created in the app to a Simulink model, you can generate a model containing your scenario and sensors by selecting **Export** > **Export Simulink Model**. In this model, a Scenario Reader block reads the scenario and Radar Detection Generator and Vision Detection Generator blocks model the sensors.

See Also

Apps
Driving Scenario Designer

Blocks
Radar Detection Generator | Scenario Reader | Vision Detection Generator

Objects
`drivingScenario` | `radarDetectionGenerator` | `visionDetectionGenerator`

More About

- “Prebuilt Driving Scenarios in Driving Scenario Designer” on page 5-14
- “Euro NCAP Driving Scenarios in Driving Scenario Designer” on page 5-36
- “Import OpenDRIVE Roads into Driving Scenario” on page 5-62
- “Create Driving Scenario Variations Programmatically” on page 5-80
- “Test Open-Loop ADAS Algorithm Using Driving Scenario” on page 5-94
- “Test Closed-Loop ADAS Algorithm Using Driving Scenario” on page 5-100

Prebuilt Driving Scenarios in Driving Scenario Designer

The **Driving Scenario Designer** app provides a library of prebuilt scenarios representing common driving maneuvers. The app also includes scenarios representing European New Car Assessment Programme (Euro NCAP®) test protocols and cuboid versions of the prebuilt scenes used in the 3D simulation environment.

Choose a Prebuilt Scenario

To get started, open the **Driving Scenario Designer** app. At the MATLAB command prompt, enter `drivingScenarioDesigner`.

In the app, the prebuilt scenarios are stored as MAT-files and organized into folders. To open a prebuilt scenario file, from the app toolstrip, select **Open > Prebuilt Scenario**. Then select a prebuilt scenario from one of the folders.

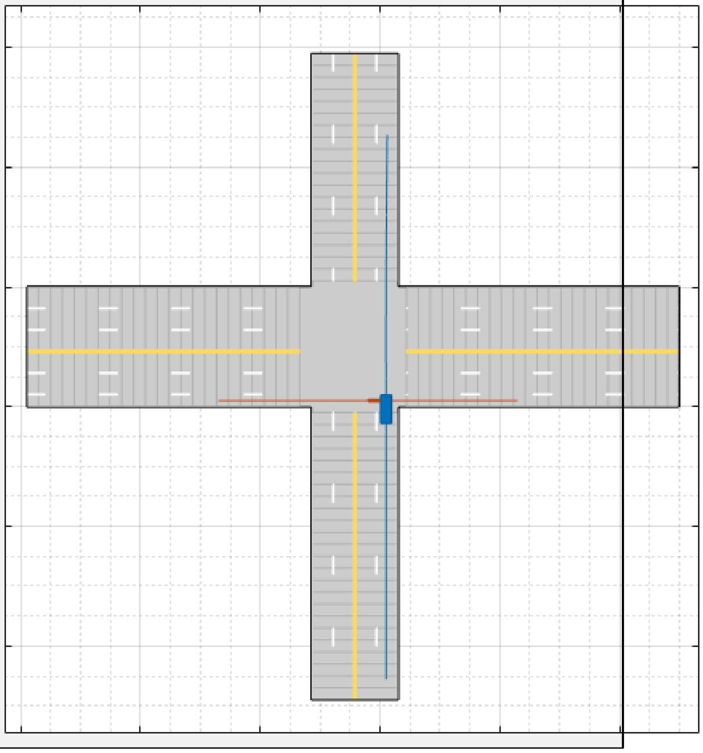
- “Euro NCAP” on page 5-14
- “Intersections” on page 5-14
- “Simulation 3D” on page 5-19
- “Turns” on page 5-19
- “U-Turns” on page 5-27

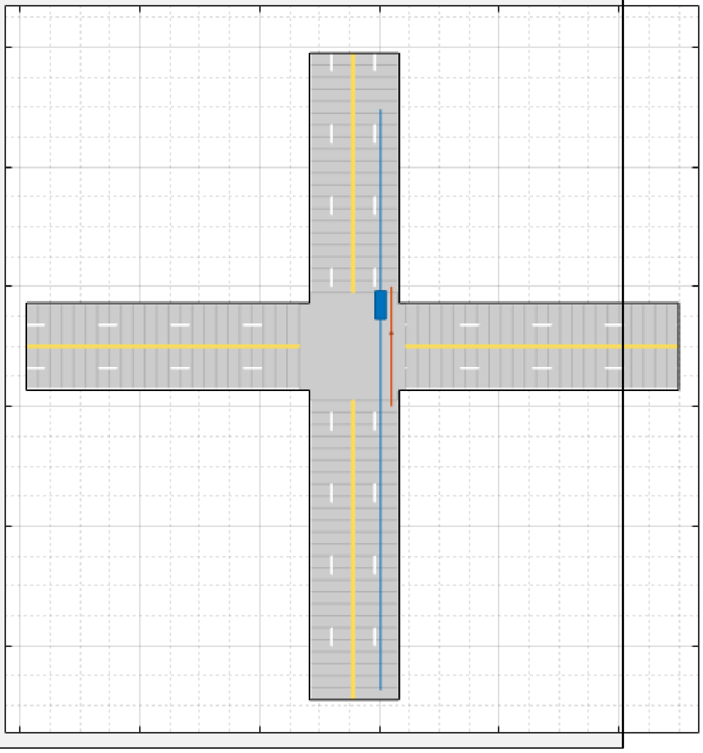
Euro NCAP

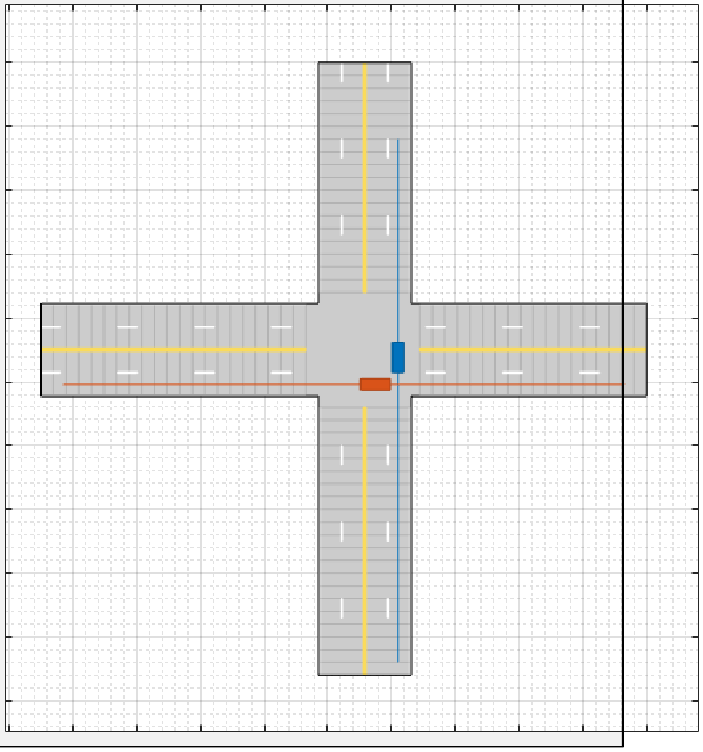
These scenarios represent Euro NCAP test protocols. The app includes scenarios for testing autonomous emergency braking, emergency lane keeping, and lane keep assist systems. For more details, see “Euro NCAP Driving Scenarios in Driving Scenario Designer” on page 5-36.

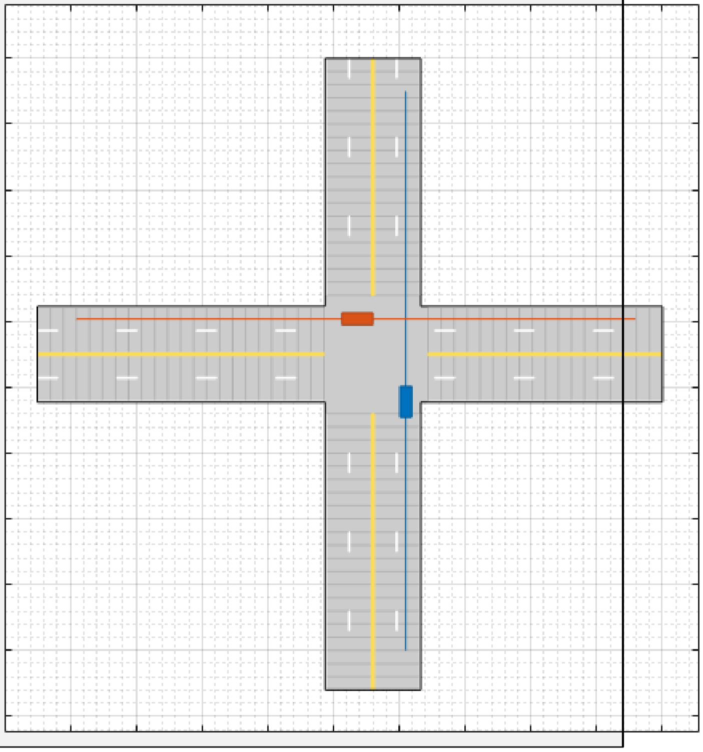
Intersections

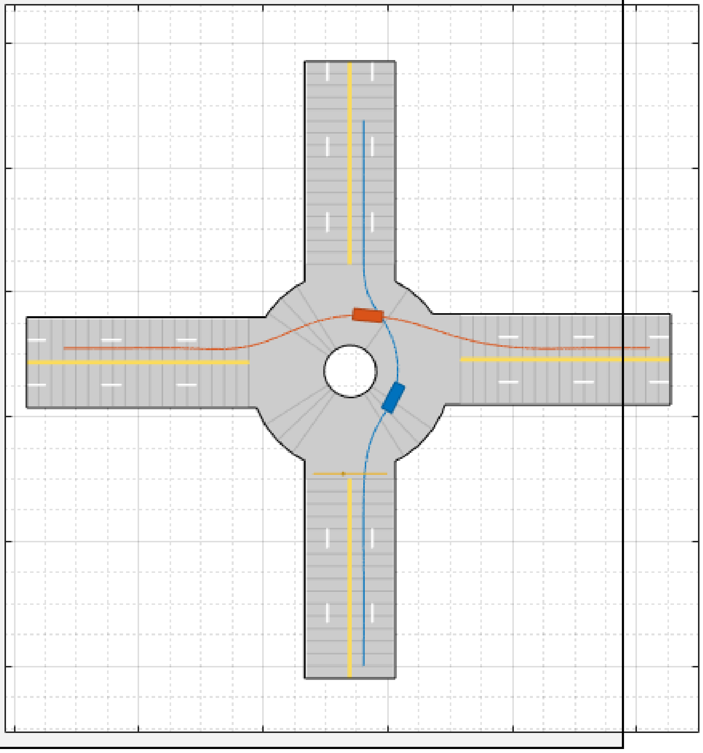
These scenarios involve common traffic patterns at four-way intersections and roundabouts.

File Name	Description
EgoVehicleGoesStraight_BicycleFromLeftGoesStraight_Collision.mat	The ego vehicle travels north and goes straight through an intersection. A bicycle coming from the left side of the intersection goes straight and collides with the ego vehicle. 

File Name	Description
`EgoVehicleGoesStraight_PedestrianToRightGoesStraight.mat`	The ego vehicle travels north and goes straight through an intersection. A pedestrian in the lane to the right of the ego vehicle also travels north and goes straight through the intersection. 

File Name	Description
`EgoVehicleGoesStraight_VehicleFromLeftGoesStraight.mat`	The ego vehicle travels north and goes straight through an intersection. A vehicle coming from the left side of the intersection also goes straight. The ego vehicle crosses in front of the other vehicle. 

File Name	Description
`EgoVehicleGoesStraight_VehicleFromRightGoesStraight.mat`	The ego vehicle travels north and goes straight through an intersection. A vehicle coming from the right side of the intersection also goes straight and crosses through the intersection first. 

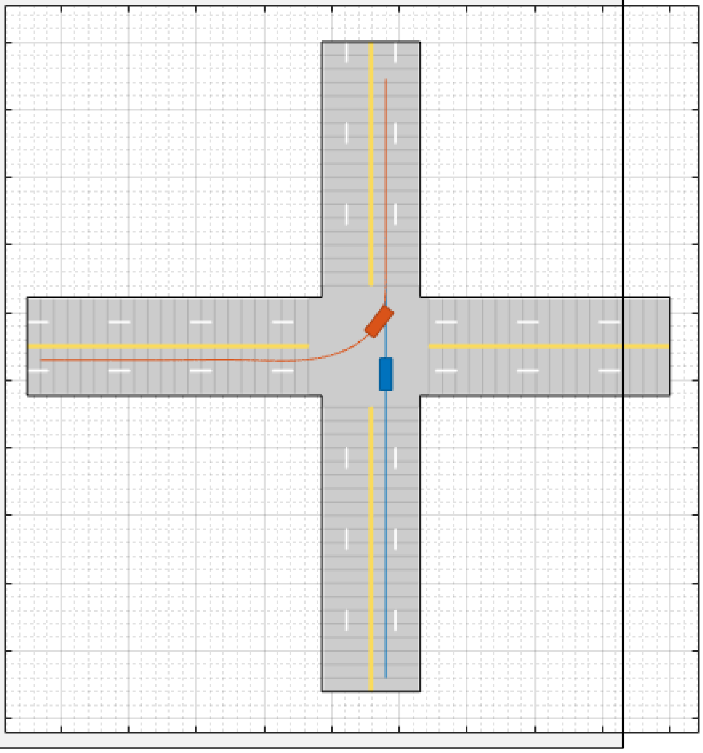
File Name	Description
Roundabout.mat	The ego vehicle travels north and crosses the path of a pedestrian while entering a roundabout. The ego vehicle then crosses the path of another vehicle as both vehicles drive through the roundabout. 

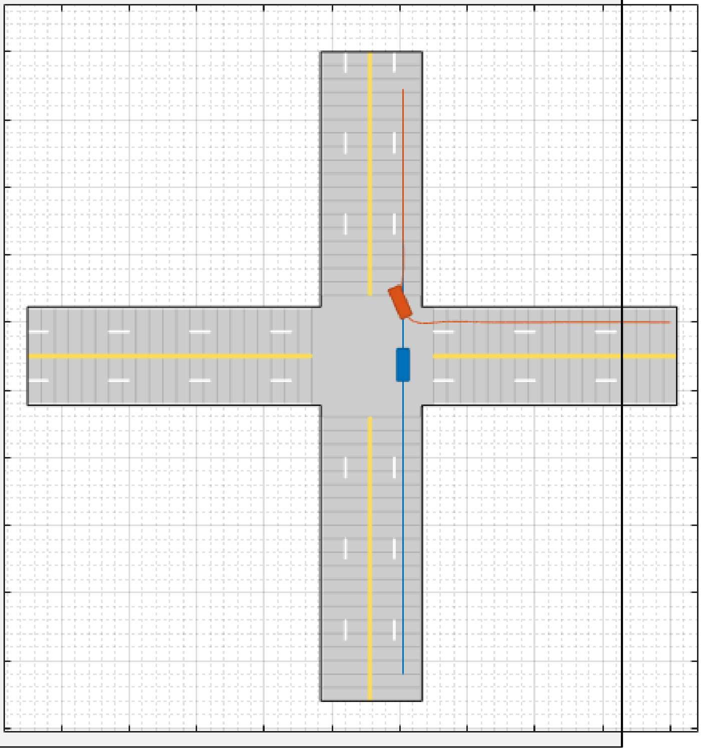
Simulation 3D

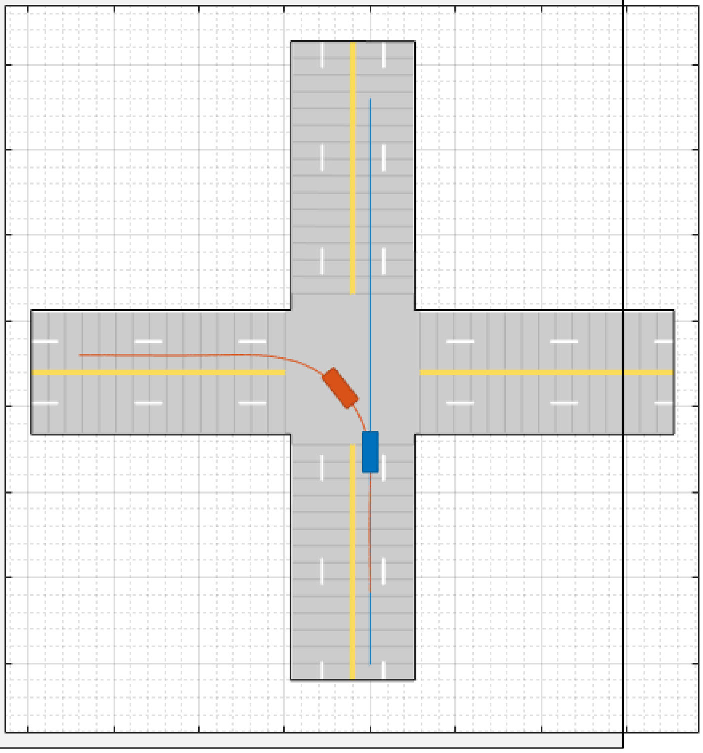
These scenarios are cuboid versions of several of the prebuilt scenes available in the 3D simulation environment. You can add vehicles and trajectories to these scenarios. Then, you can include these vehicles and trajectories in your Simulink model to simulate them in the 3D environment. This environment is rendered using the Unreal Engine from Epic Games. For more details on these scenarios, see “Cuboid Versions of 3D Simulation Scenes in Driving Scenario Designer” on page 5-55.

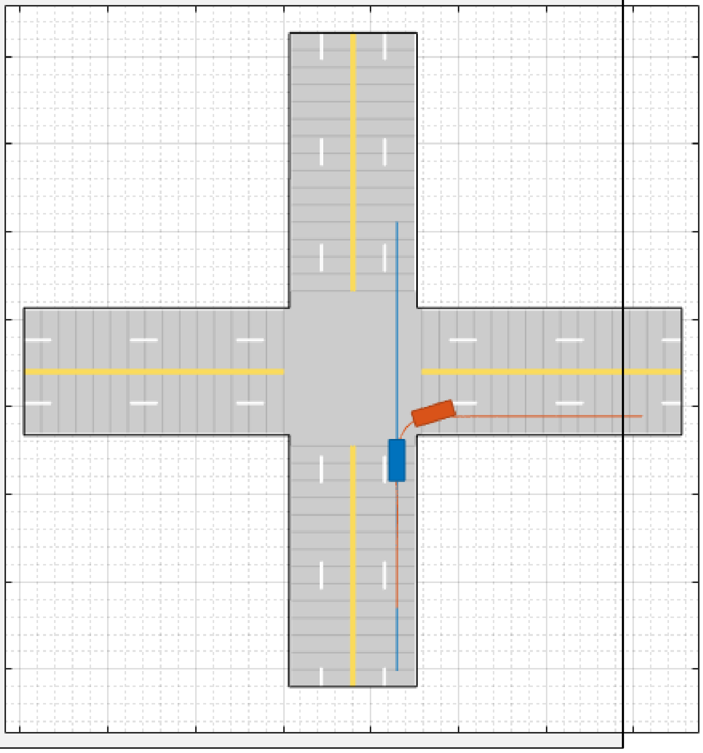
Turns

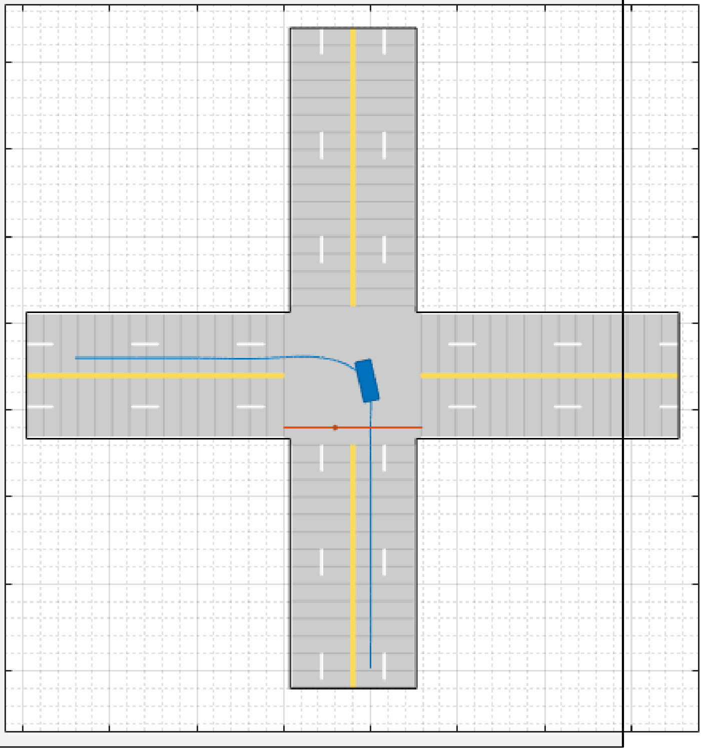
These scenarios involve turns at four-way intersections.

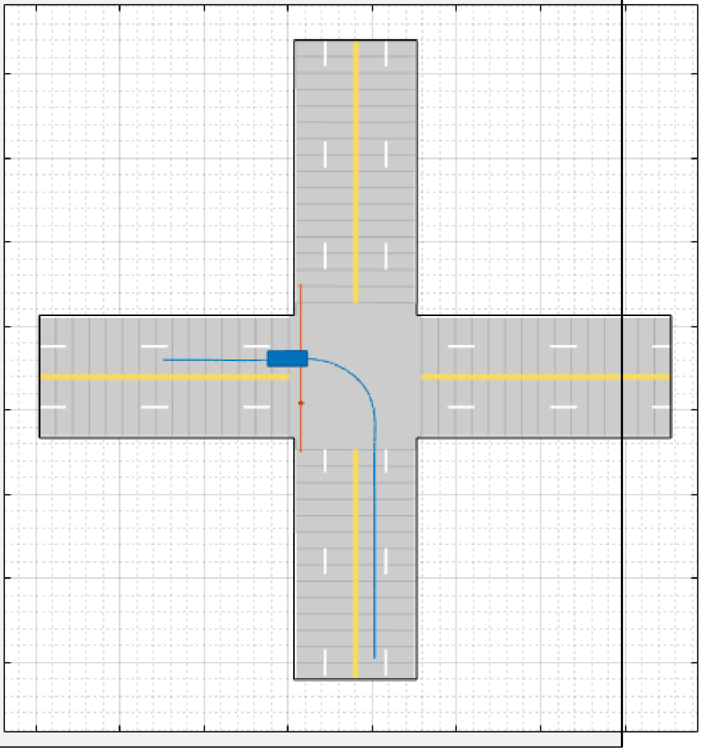
File Name	Description
EgoVehicleGoesStraight_VehicleFromLeftTurnsLeft.mat	The ego vehicle travels north and goes straight through an intersection. A vehicle coming from the left side of the intersection turns left and ends up in front of the ego vehicle. 

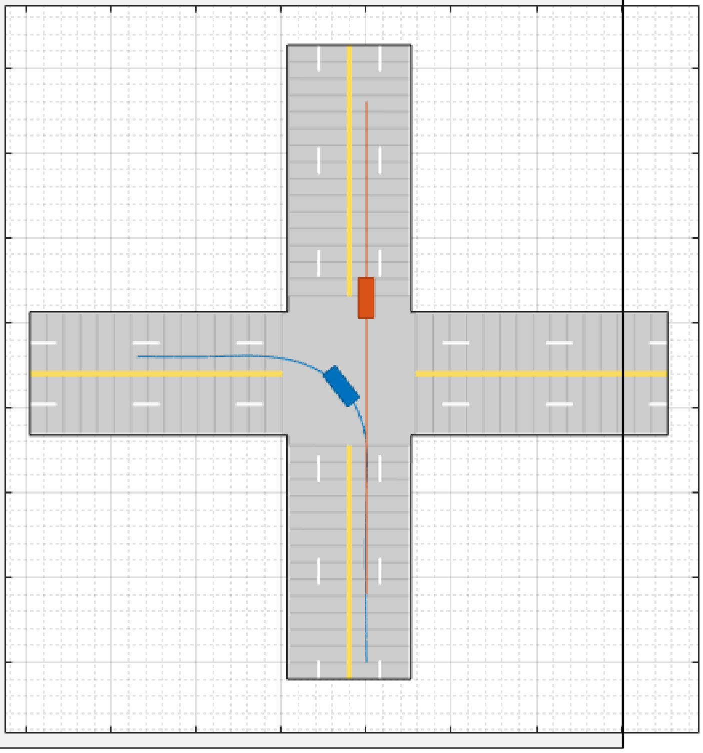
File Name	Description
EgoVehicleGoesStraight_VehicleFromRightTurnsRight.mat	The ego vehicle travels north and goes straight through an intersection. A vehicle coming from the right side of the intersection turns right and ends up in front of the ego vehicle. 

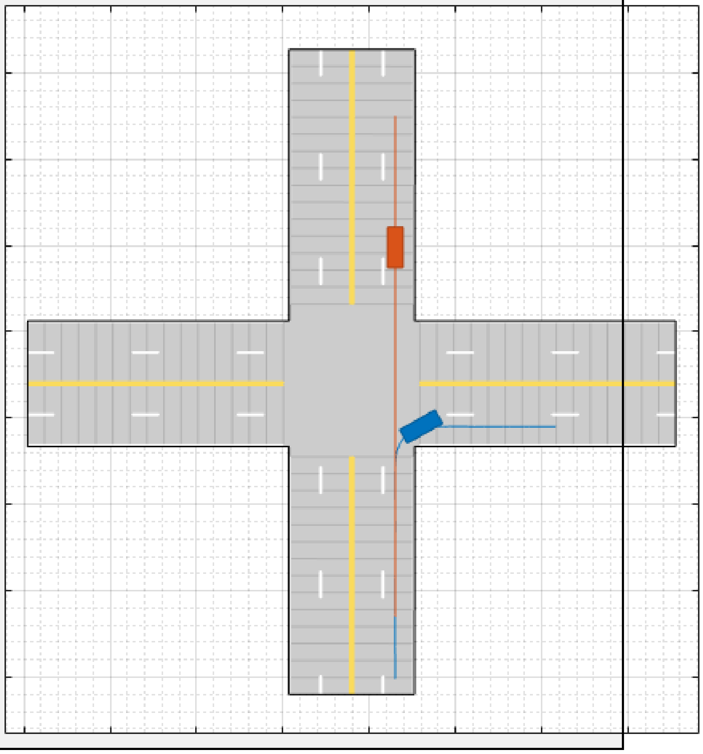
File Name	Description
`EgoVehicleGoesStraight_VehicleInFrontTurnsLeft.mat`	The ego vehicle travels north and goes straight through an intersection. A vehicle in front of the ego vehicle turns left at the intersection. 

File Name	Description
`EgoVehicleGoesStraight_VehicleInFrontTurnsRight.mat`	The ego vehicle travels north and goes straight through an intersection. A vehicle in front of the ego vehicle turns right at the intersection. 

File Name	Description
EgoVehicleTurnsLeft_PedestrianFromLeftGoesStraight.mat	The ego vehicle travels north and turns left at an intersection. A pedestrian coming from the left side of the intersection goes straight. The ego vehicle completes its turn before the pedestrian finishes crossing the intersection. 

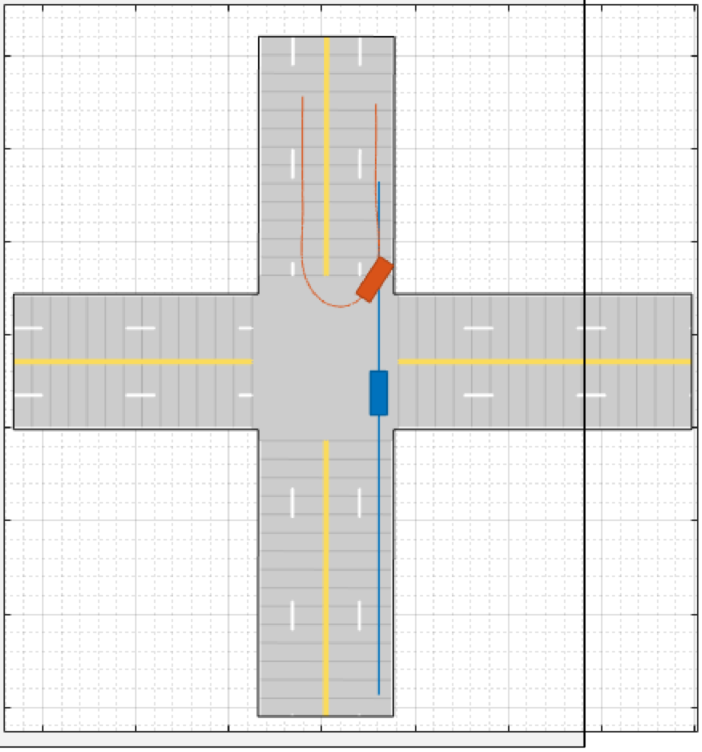
File Name	Description
EgoVehicleTurnsLeft_PedestrianInOppLaneGoesStraight.mat	The ego vehicle travels north and turns left at an intersection. A pedestrian in the opposite lane goes straight through the intersection. The ego vehicle completes its turn before the pedestrian finishes crossing the intersection. 

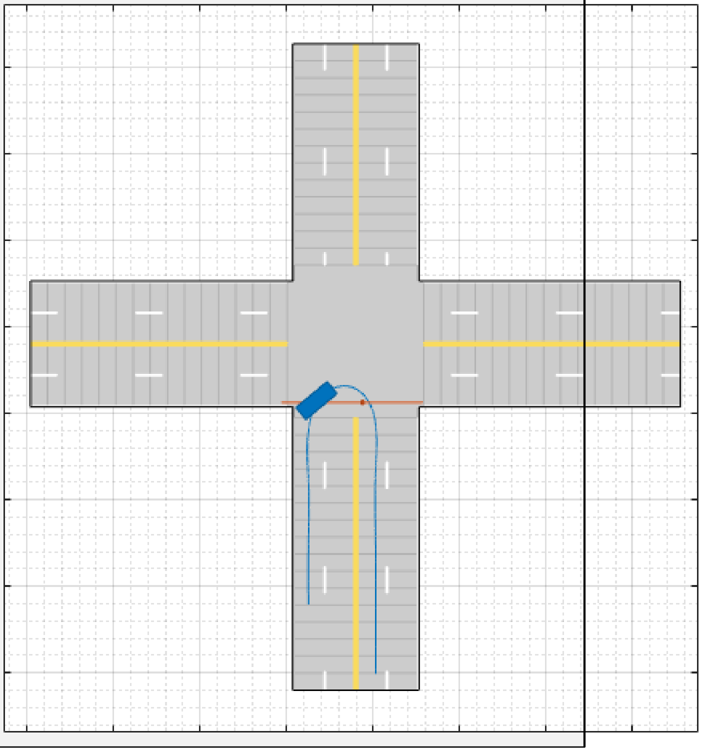
File Name	Description
`EgoVehicleTurnsLeft_VehicleInFrontGoesStraight.mat`	The ego vehicle travels north and turns left at an intersection. A vehicle in front of the ego vehicle goes straight through the intersection. 

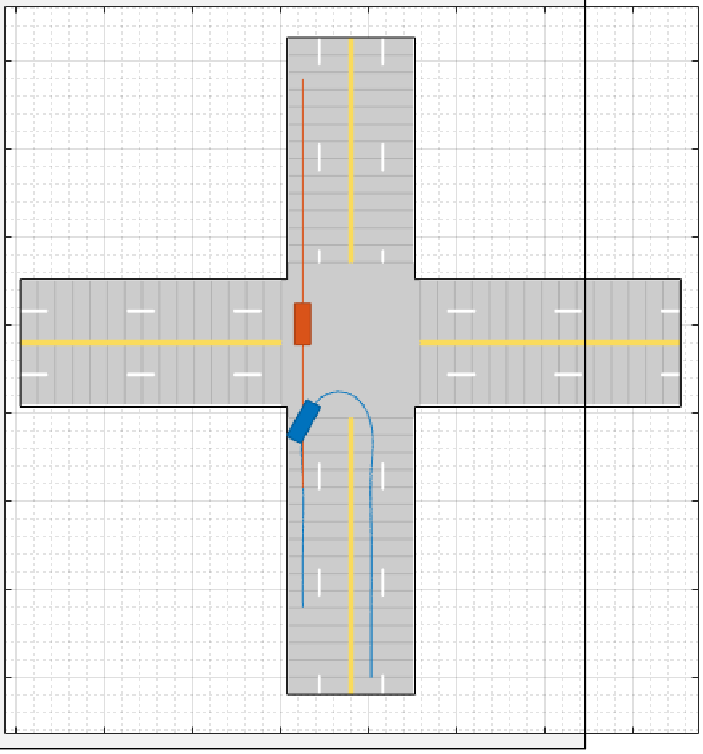
File Name	Description
EgoVehicleTurnsRight_VehicleInFrontGoesStraight.mat	The ego vehicle travels north and turns right at an intersection. A vehicle in front of the ego vehicle goes straight through the intersection. 

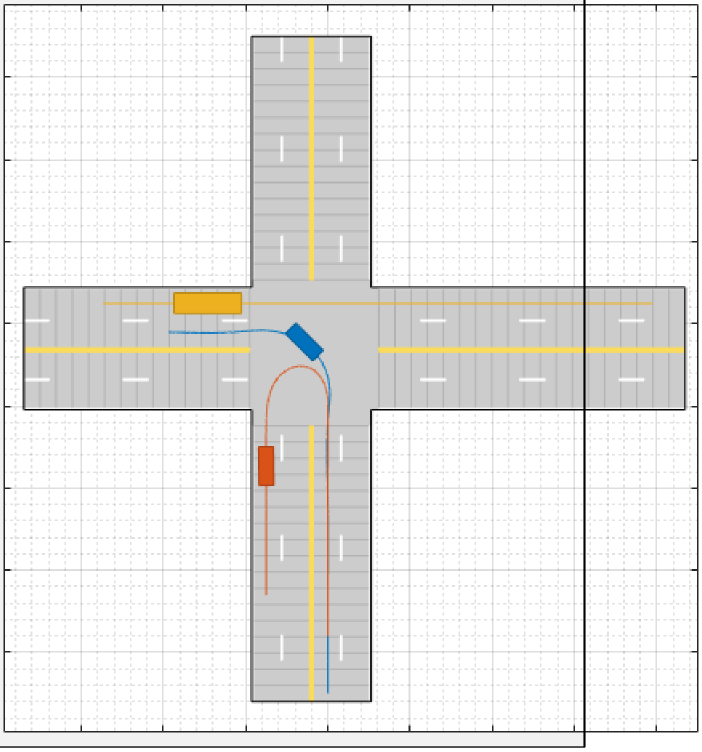
U-Turns

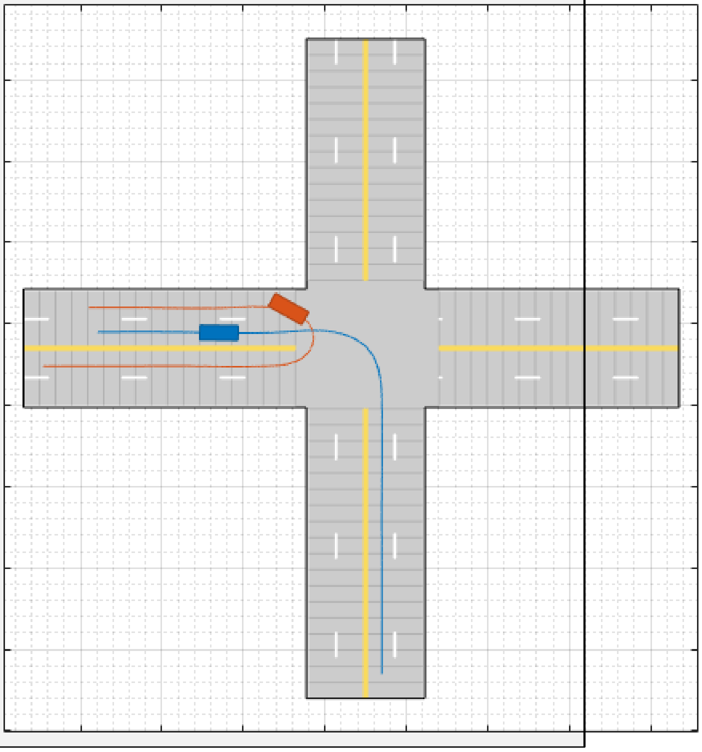
These scenarios involve U-turns at four-way intersections.

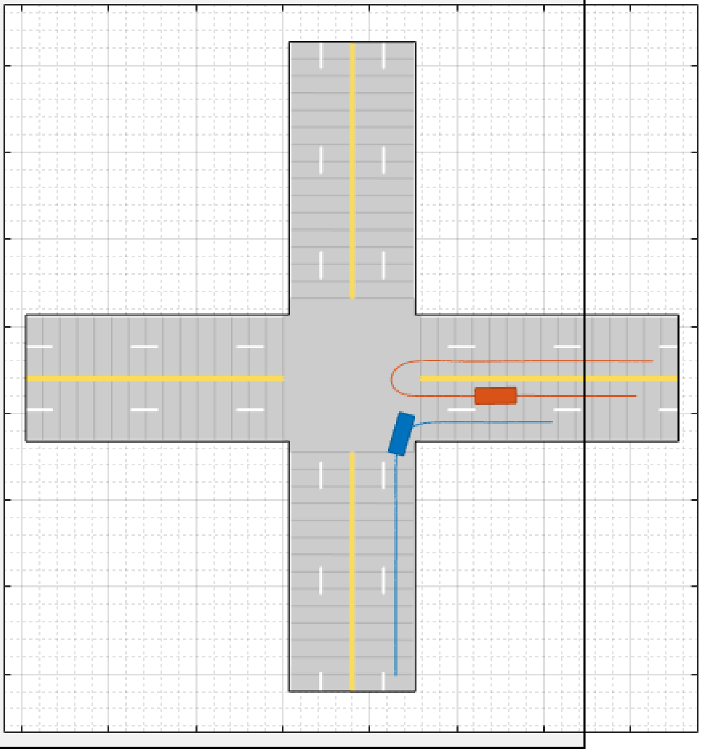
File Name	Description
EgoVehicleGoesStraight_VehicleInOppLaneMakesUTurn.mat	The ego vehicle travels north and goes straight through an intersection. A vehicle in the opposite lane makes a U-turn. The ego vehicle ends up behind the vehicle. 

File Name	Description
`EgoVehicleMakesUTurn_PedestrianFromRightGoesStraight.mat`	The ego vehicle travels north and makes a U-turn at an intersection. A pedestrian coming from the right side of the intersection goes straight and crosses the path of the U-turn. 

File Name	Description
EgoVehicleMakesUTurn_VehicleInOppLaneGoesStraight.mat	The ego vehicle travels north and makes a U-turn at an intersection. A vehicle traveling south in the opposite direction goes straight and ends up behind the ego vehicle. 

File Name	Description
`EgoVehicleTurnsLeft_Vehicle1MakesUTurn_Vehicle2GoesStraight.mat`	The ego vehicle travels north and turns left at an intersection. A vehicle in front of the ego vehicle makes a U-turn at the intersection. A second vehicle, a truck, comes from the right side of the intersection. The ego vehicle ends up in the lane next to the truck. 

File Name	Description
`EgoVehicleTurnsLeft_VehicleFromLeftMakesUTurn.mat`	The ego vehicle travels north and turns left at an intersection. A vehicle coming from the left side of the intersection makes a U-turn. The ego vehicle ends up in the lane next to the other vehicle. 

File Name	Description
EgoVehicleTurnsRight_VehicleFromRightMakesUTurn.mat	The ego vehicle travels north and turns right at an intersection. A vehicle coming from the right side of the intersection makes a U-turn. The ego vehicle ends up behind the vehicle, in an adjacent lane. 

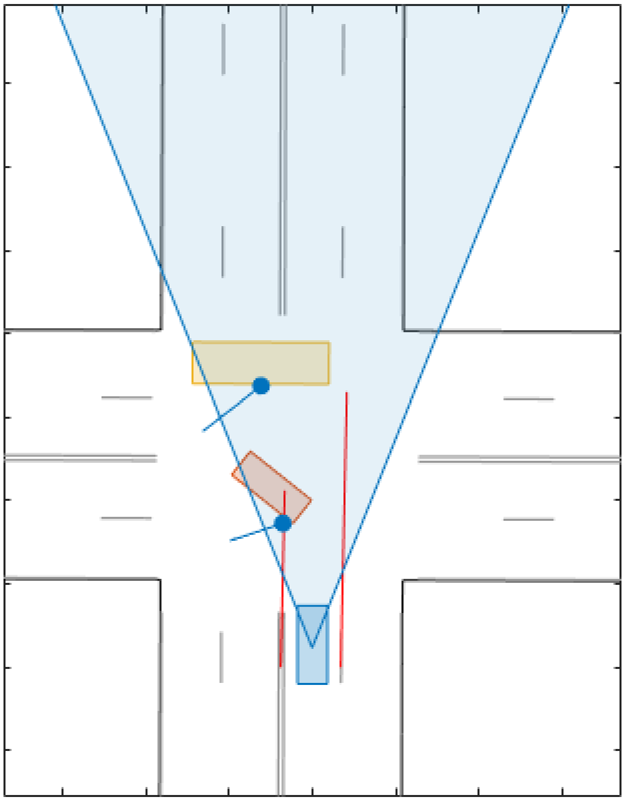
Modify Scenario

After you choose a scenario, you can modify the parameters of the roads and actors. For example, from the **Actors** tab on the left, you can change the position or velocity of the ego vehicle or other actors. From the **Roads** tab, you can change the width of the lanes or the type of lane markings.

You can also add or modify sensors. For example, from the **Sensors** tab, you can change the detection parameters or the positions of the sensors. By default, in Euro NCAP scenarios, the ego vehicle does not contain sensors. All other prebuilt scenarios have at least one front-facing camera or radar sensor, set to detect lanes and objects.

Generate Synthetic Detections

To generate detections from the sensors, on the app toolstrip, click **Run**. As the scenario runs, the **Ego-Centric View** displays the scenario from the perspective of the ego vehicle. The **Bird's-Eye Plot** displays the detections.



Export the detections.

- To export detections to the MATLAB workspace, on the app toolstrip, select **Export** > **Export Sensor Data**. Name the workspace variable and click **OK**. The app saves the sensor data as a structure containing the actor poses, object detections, and lane detections at each time step.
- To export a MATLAB function that generates the scenario and its detections, select **Export** > **Export MATLAB Function**. This function returns the sensor detections as a structure, the scenario as a `drivingScenario` object, and the sensor models as `visionDetectionGenerator` and `radarDetectionGenerator` System objects. By modifying this function, you can create variations of the original scenario. For an example of this process, see “Create Driving Scenario Variations Programmatically” on page 5-80.

Save Scenario

Because prebuilt scenarios are read-only, save a copy of the driving scenario to a new folder. To save the scenario file, on the app toolstrip, select **Save** > **Scenario File As**.

You can reopen this scenario file from the app. Alternatively, at the MATLAB command prompt, you can use this syntax.

```
drivingScenarioDesigner(scenarioFileName)
```

You can also reopen the scenario by using the exported `drivingScenario` object. At the MATLAB command prompt, use this syntax, where `scenario` is the name of the exported object.

```
drivingScenarioDesigner(scenario)
```

To reopen sensors, use this syntax, where `sensors` is a `radarDetectionGenerator` object, `visionDetectionGenerator` object, or a cell array of such objects.

```
drivingScenarioDesigner(scenario,sensors)
```

If you are developing a driving algorithm in Simulink, you can use a Scenario Reader block to read roads and actors from the scenario file or `drivingScenario` object into your model. This block does not directly read sensor data. To add sensors created in the app to a Simulink model, you can generate a model containing your scenario and sensors by selecting **Export** > **Export Simulink Model**. In this model, a Scenario Reader block reads the scenario and Radar Detection Generator and Vision Detection Generator blocks model the sensors.

See Also

Apps

Driving Scenario Designer

Blocks

Radar Detection Generator | Vision Detection Generator

Objects

`drivingScenario` | `radarDetectionGenerator` | `visionDetectionGenerator`

More About

- “Build a Driving Scenario and Generate Synthetic Detections” on page 5-2
- “Euro NCAP Driving Scenarios in Driving Scenario Designer” on page 5-36
- “Cuboid Versions of 3D Simulation Scenes in Driving Scenario Designer” on page 5-55
- “Test Open-Loop ADAS Algorithm Using Driving Scenario” on page 5-94
- “Test Closed-Loop ADAS Algorithm Using Driving Scenario” on page 5-100

Euro NCAP Driving Scenarios in Driving Scenario Designer

The **Driving Scenario Designer** app provides a library of prebuilt scenarios representing European New Car Assessment Programme (Euro NCAP) test protocols. The app includes scenarios for testing autonomous emergency braking (AEB), emergency lane keeping (ELK), and lane keep assist (LKA) systems.

Choose a Euro NCAP Scenario

To get started, open the **Driving Scenario Designer** app. At the MATLAB command prompt, enter `drivingScenarioDesigner`.

In the app, the Euro NCAP scenarios are stored as MAT-files and organized into folders. To open a Euro NCAP file, on the app toolstrip, select **Open** > **Prebuilt Scenario**. The `PrebuiltScenarios` folder opens, which includes subfolders for all prebuilt scenarios available in the app (see also "Prebuilt Driving Scenarios in Driving Scenario Designer" on page 5-14).

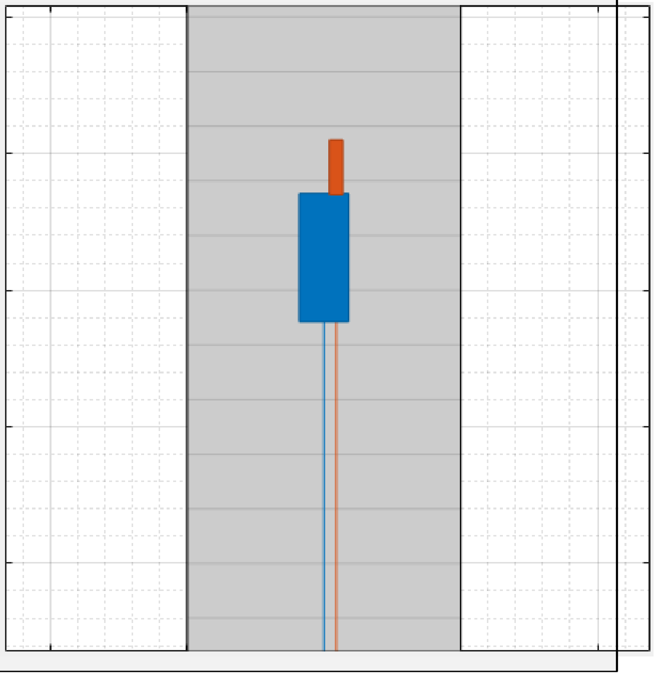
Double-click the **EuroNCAP** folder, and then choose a Euro NCAP scenario from one of these subfolders.

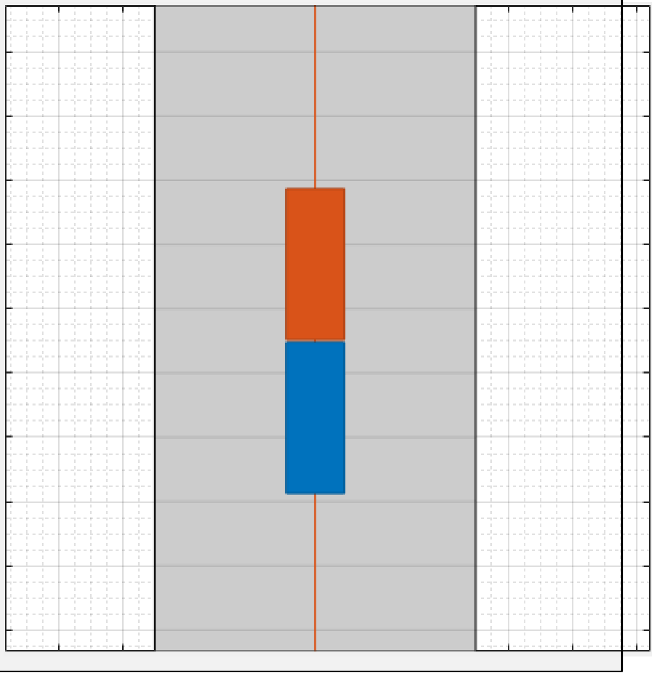
- "Autonomous Emergency Braking" on page 5-36
- "Emergency Lane Keeping" on page 5-42
- "Lane Keep Assist" on page 5-46

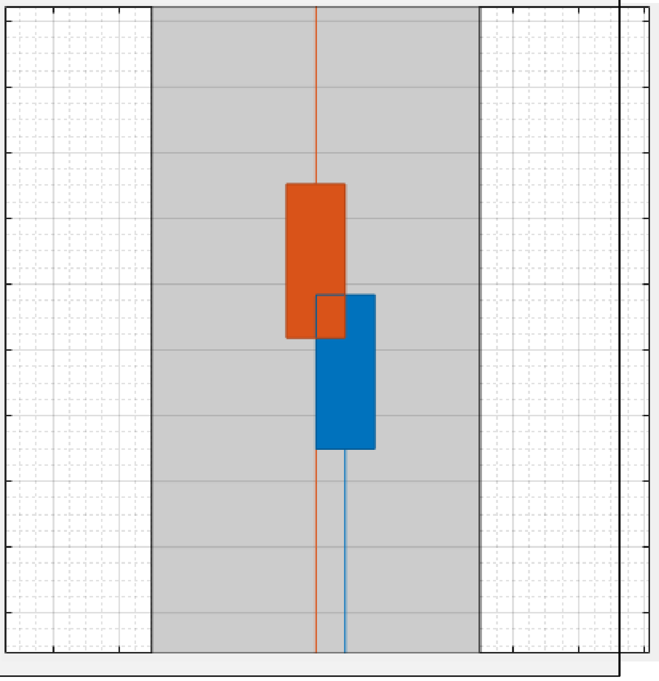
Autonomous Emergency Braking

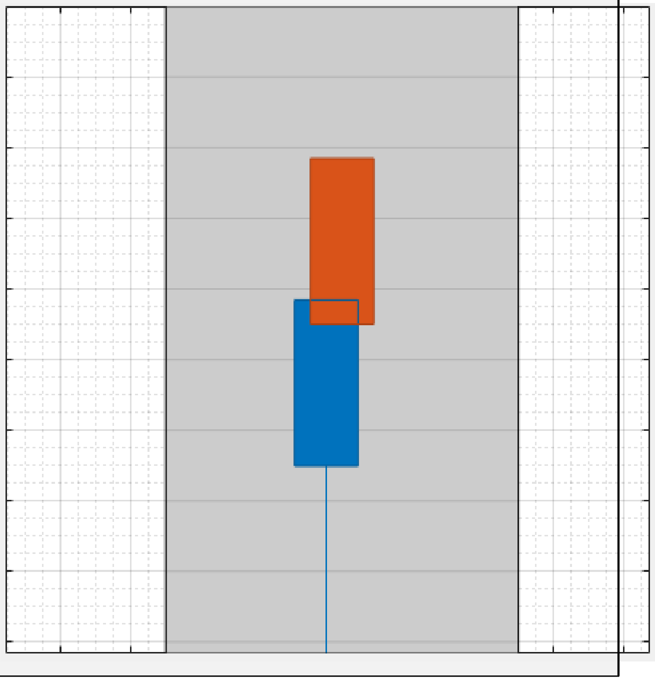
These scenarios are designed to test autonomous emergency braking (AEB) systems. AEB systems warn drivers of impending collisions and automatically apply brakes to prevent collisions or reduce the impact of collisions. Some AEB systems prepare the vehicle and restraint systems for impact.

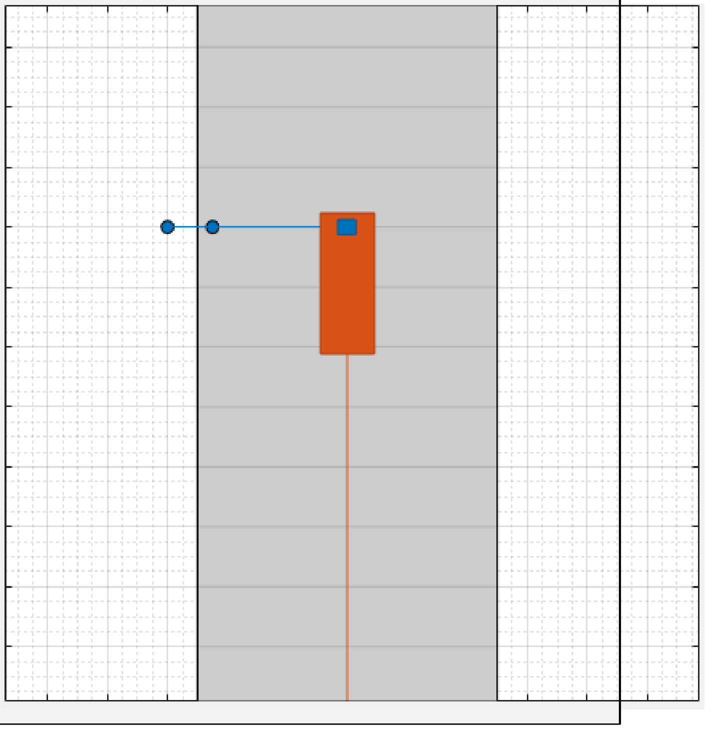
The table lists a subset of the available AEB scenarios. Other AEB scenarios in the folder vary the points of collision, the amount of overlap between vehicles, and the initial gap between vehicles.

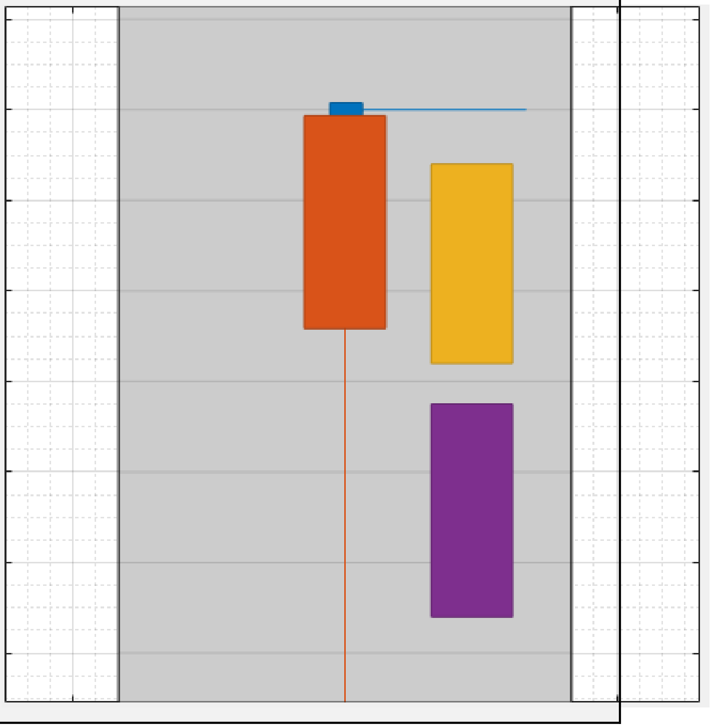
File Name	Description
`AEB_Bicyclist_Longitudinal_25width.mat`	The ego vehicle collides with the bicyclist that is in front of it. Before the collision, the bicyclist and ego vehicle are traveling in the same direction along the longitudinal axis. At collision time, the bicycle is 25% of the way across the width of the ego vehicle. 

File Name	Description
AEB_CCRb_2_initialGap_12m.mat	A car-to-car rear braking (CCRb) scenario, where the ego vehicle rear-ends a braking vehicle. The braking vehicle begins to decelerate at 2 m/s^2. The initial gap between the ego vehicle and the braking vehicle is 12 m. 

File Name	Description
`AEB_CCRm_50overlap.mat`	A car-to-car rear moving (CCRm) scenario, where the ego vehicle rear-ends a moving vehicle. At collision time, the ego vehicle overlaps with 50% of the width of the moving vehicle. 

File Name	Description
AEB_CCRs_-75overlap.mat	A car-to-car rear stationary (CCRs) scenario, where the ego vehicle rear-ends a stationary vehicle. At collision time, the ego vehicle overlaps with –75% of the width of the stationary vehicle. When the ego vehicle is to the left of the other vehicle, the percent overlap is negative. 

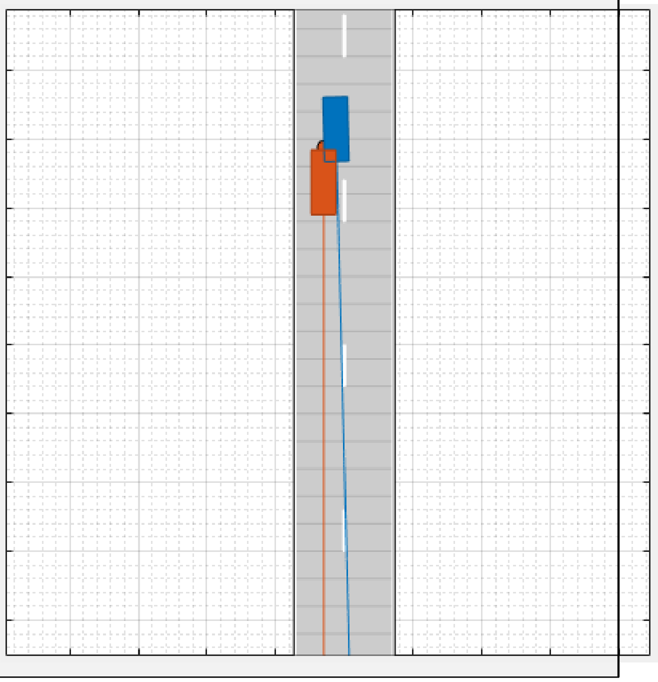
File Name	Description
AEB_Pedestrian_Farside_50width.mat	The ego vehicle collides with a pedestrian who is traveling from the left side of the road, which Euro NCAP test protocols refer to as the far side. These protocols assume that vehicles travel on the right side of the road. Therefore, the left side of the road is the side farthest from the ego vehicle. At collision time, the pedestrian is 50% of the way across the width of the ego vehicle. 

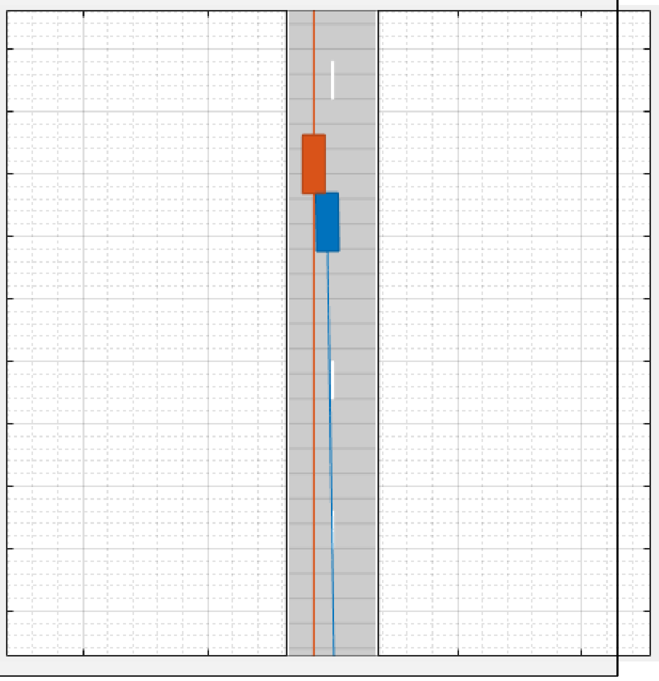
File Name	Description
AEB_PedestrianChild_Nearside_50width.mat	The ego vehicle collides with a pedestrian who is traveling from the right side of the road, which Euro NCAP test protocols refer to as the near side. These protocols assume that vehicles travel on the right side of the road. Therefore, the right side of the road is the side nearest to the ego vehicle. At collision time, the pedestrian is 50% of the way across the width of the ego vehicle. 

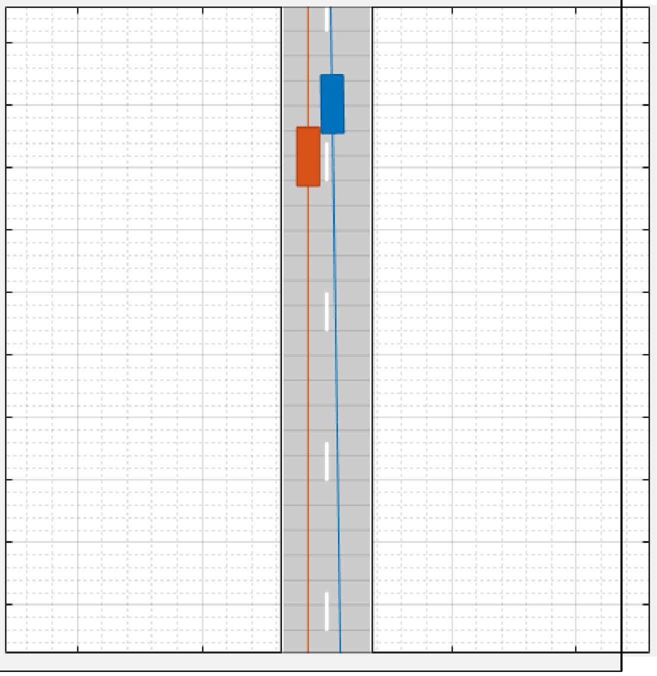
Emergency Lane Keeping

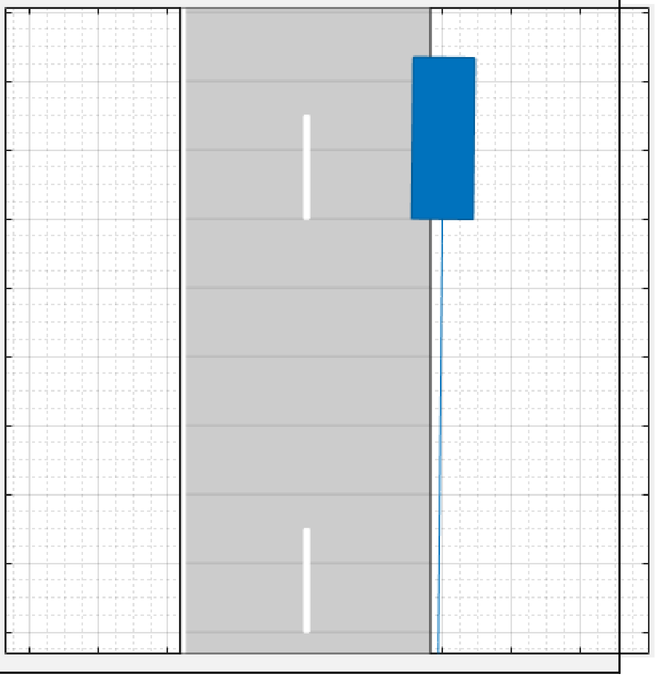
These scenarios are designed to test emergency lane keeping (ELK) systems. ELK systems prevent collisions by warning drivers of impending, unintentional lane departures.

The table lists a subset of the available ELK scenarios. Other ELK scenarios in the folder vary the lateral velocity of the ego vehicle and the lane marking types.

File Name	Description
`ELK_FasterOvertakingVeh_Intent_Vlat_0.5.mat`	The ego vehicle intentionally changes lanes and collides with a faster, overtaking vehicle that is in the other lane. The ego vehicle travels at a lateral velocity of 0.5 m/s. 

File Name	Description
`ELK_OncomingVeh_Vlat_0.3.mat`	The ego vehicle unintentionally changes lanes and collides with an oncoming vehicle that is in the other lane. The ego vehicle travels at a lateral velocity of 0.3 m/s. 

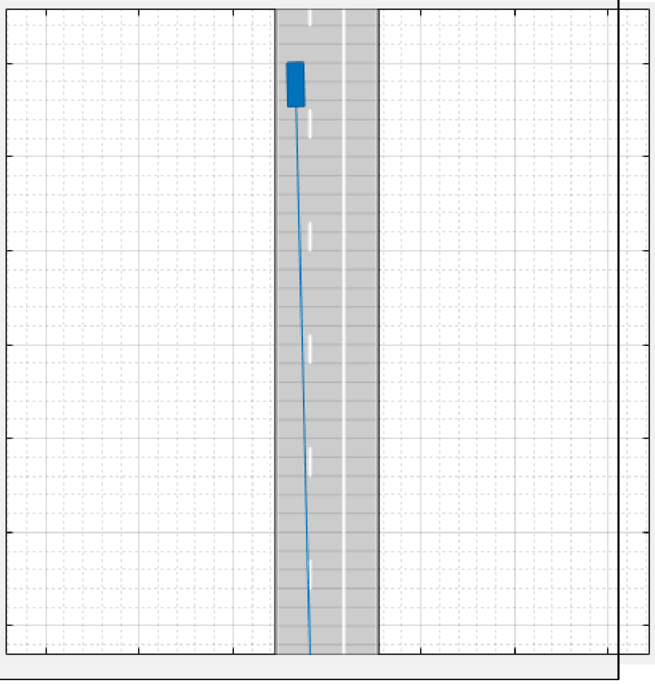
File Name	Description
`ELK_OvertakingVeh_Unintent_Vlat_0.3.mat`	The ego vehicle unintentionally changes lanes, overtakes a vehicle in the other lane, and collides with that vehicle. The ego vehicle travels at a lateral velocity of 0.3 m/s. 

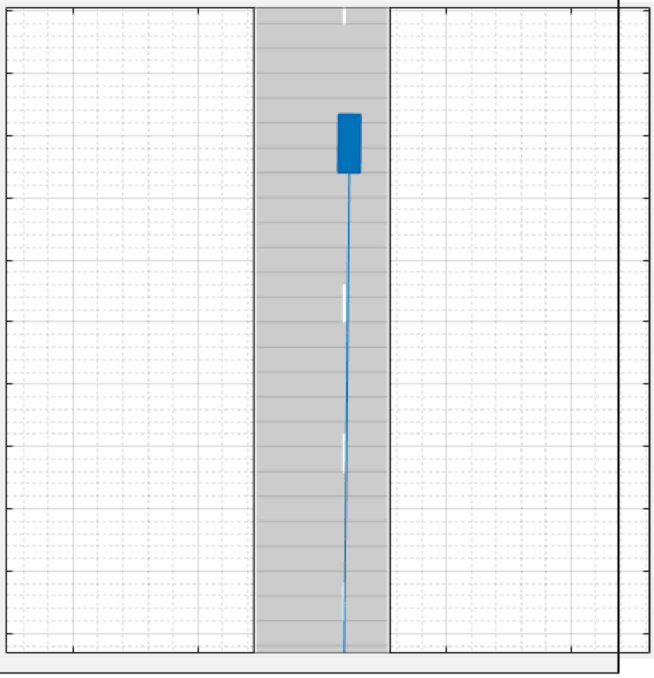
File Name	Description
`ELK_RoadEdge_NoBndry_Vlat_0.2.mat`	The ego vehicle unintentionally changes lanes and ends up on the road edge. The road edge has no lane boundary markings. The ego vehicle travels at a lateral velocity of 0.2 m/s. 

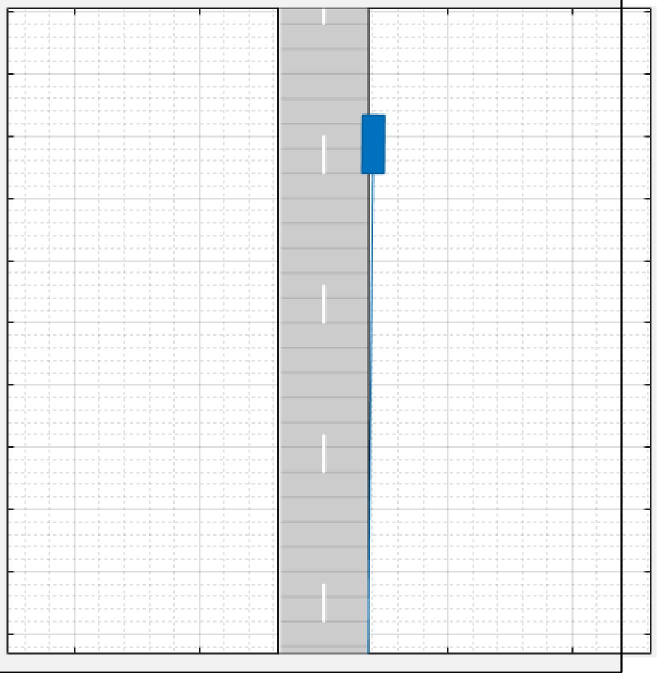
Lane Keep Assist

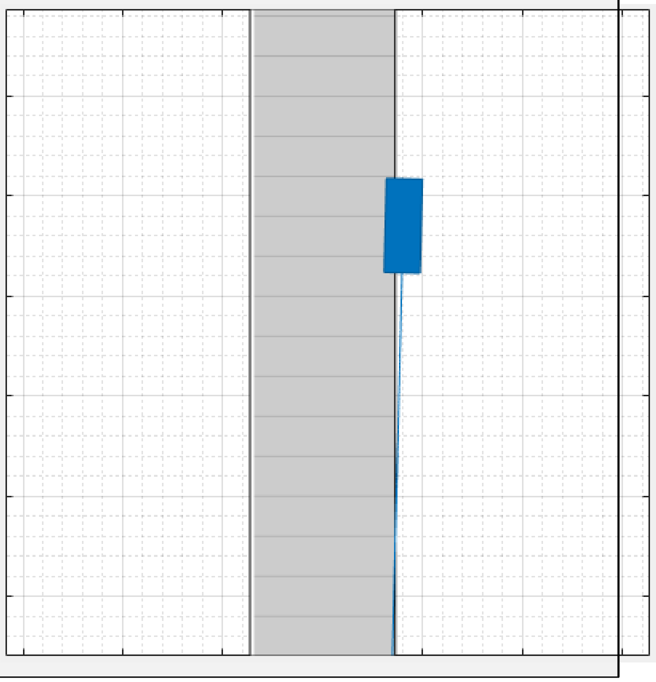
These scenarios are designed to test lane keep assist (LKA) systems. LKA systems detect unintentional lane departures and automatically adjust the steering angle of the vehicle to stay within the lane boundaries.

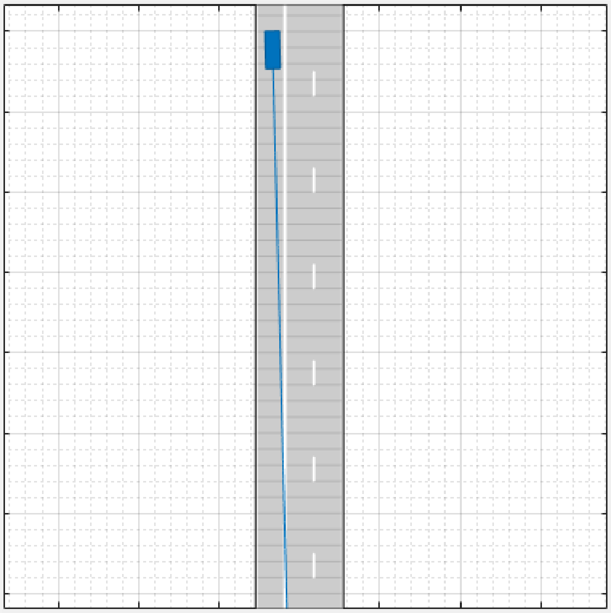
The table lists a subset of the available LKA scenarios. Other LKA scenarios in the folder vary the lateral velocity of the ego vehicle and the lane marking types.

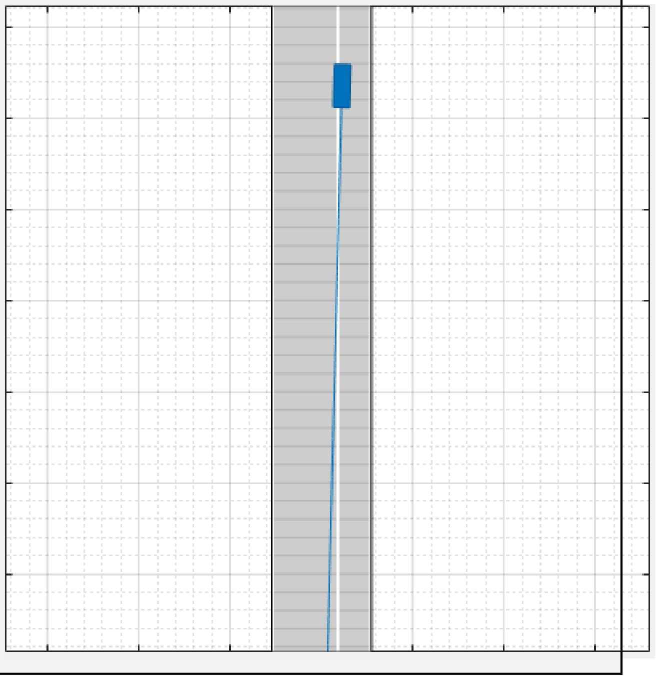
File Name	Description
LKA_DashedLine_Solid_Left_Vlat_0.5.mat	The ego vehicle unintentionally departs from a lane that is dashed on the left and solid on the right. The car departs the lane from the left (dashed) side, traveling at a lateral velocity of 0.5 m/s. 

File Name	Description
LKA_DashedLine_Unmarked_Right_Vlat_0.5 .mat	The ego vehicle unintentionally departs from a lane that is dashed on the right and unmarked on the left. The car departs the lane from the right (dashed) side, traveling at a lateral velocity of 0.5 m/s. 

File Name	Description
`LKA_RoadEdge_NoBndry_Vlat_0.5.mat`	The ego vehicle unintentionally departs from a lane and ends up on the road edge. The road edge has no lane boundary markings. The car travels at a lateral velocity of 0.5 m/s. 

File Name	Description
LKA_RoadEdge_NoMarkings_Vlat_0.5.mat	The ego vehicle unintentionally departs from a lane and ends up on the road edge. The road has no lane markings. The car travels at a lateral velocity of 0.5 m/s. 

File Name	Description
LKA_SolidLine_Dashed_Left_Vlat_0.5.mat	The ego vehicle unintentionally departs from a lane that is solid on the left and dashed on the right. The car departs the lane from the left (solid) side, traveling at a lateral velocity of 0.5 m/s. 

File Name	Description
LKA_SolidLine_Unmarked_Right_Vlat_0.5.mat	The ego vehicle unintentionally departs from a lane that is a solid on the right and unmarked on the left. The car departs the lane from the right (solid) side, traveling at a lateral velocity of 0.5 m/s. 

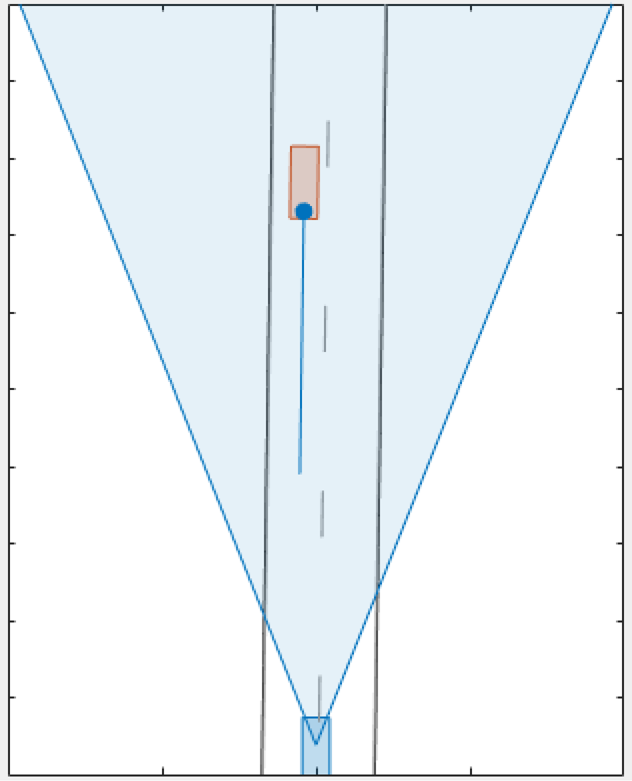
Modify Scenario

By default, in Euro NCAP scenarios, the ego vehicle does not contain sensors. If you are testing a vehicle sensor, on the app toolstrip, click **Add Camera** or **Add Radar** to add a sensor to the ego vehicle. Then, on the **Sensor** tab, adjust the parameters of the sensors to match your sensor model. If you are testing a camera sensor, to enable the camera to detect lanes, expand the **Detection Parameters** section, and set **Detection Type** to `Lanes & Objects`.

You can also adjust the parameters of the roads and actors in the scenario. For example, from the **Actors** tab on the left, you can change the position or velocity of the ego vehicle or other actors. From the **Roads** tab, you can change the width of lanes or the type of lane markings.

Generate Synthetic Detections

To generate detections from any added sensors, click **Run**. As the scenario runs, the **Ego-Centric View** displays the scenario from the perspective of the ego vehicle. The **Bird's-Eye Plot** displays the detections.



Export the detections.

- To export detections to the MATLAB workspace, on the app toolstrip, select **Export** > **Export Sensor Data**. Name the workspace variable and click **OK**. The app saves the sensor data as a structure containing the actor poses, object detections, and lane detections at each time step.
- To export a MATLAB function that generates the scenario and its detections, select **Export** > **Export MATLAB Function**. This function returns the sensor detections as a structure, the scenario as a `drivingScenario` object, and the sensor models as `visionDetectionGenerator` and `radarDetectionGenerator` System objects. By modifying this function, you can create variations of the original scenario. For an example of this process, see “Create Driving Scenario Variations Programmatically” on page 5-80.

Save Scenario

Because Euro NCAP scenarios are read-only, save a copy of the driving scenario to a new folder. To save the scenario file, on the app toolstrip, select **Save** > **Scenario File As**.

You can reopen this scenario file from the app. Alternatively, at the MATLAB command prompt, you can use this syntax.

```
drivingScenarioDesigner(scenarioFileName)
```

You can also reopen the scenario by using the exported `drivingScenario` object. At the MATLAB command prompt, use this syntax, where `scenario` is the name of the exported object.

```
drivingScenarioDesigner(scenario)
```

To reopen sensors, use this syntax, where `sensors` is a `radarDetectionGenerator` object, `visionDetectionGenerator` object, or a cell array of such objects.

```
drivingScenarioDesigner(scenario,sensors)
```

If you are developing a driving algorithm in Simulink, you can use a Scenario Reader block to read roads and actors from the scenario file or `drivingScenario` object into your model. This block does not directly read sensor data. To add sensors created in the app to a Simulink model, you can generate a model containing your scenario and sensors by selecting **Export > Export Simulink Model**. In this model, a Scenario Reader block reads the scenario and Radar Detection Generator and Vision Detection Generator blocks model the sensors.

References

[1] European New Car Assessment Programme. *Euro NCAP Assessment Protocol - SA*. Version 8.0.2. January 2018.

[2] European New Car Assessment Programme. *Euro NCAP AEB C2C Test Protocol*. Version 2.0.1. January 2018.

[3] European New Car Assessment Programme. *Euro NCAP LSS Test Protocol*. Version 2.0.1. January 2018.

See Also

Apps
Driving Scenario Designer

Blocks
Radar Detection Generator | Scenario Reader | Vision Detection Generator

Objects
`drivingScenario` | `radarDetectionGenerator` | `visionDetectionGenerator`

More About

- “Build a Driving Scenario and Generate Synthetic Detections” on page 5-2
- “Prebuilt Driving Scenarios in Driving Scenario Designer” on page 5-14
- “Create Driving Scenario Variations Programmatically” on page 5-80
- “Autonomous Emergency Braking with Sensor Fusion”
- “Test Open-Loop ADAS Algorithm Using Driving Scenario” on page 5-94
- “Test Closed-Loop ADAS Algorithm Using Driving Scenario” on page 5-100

External Websites

- Euro NCAP Safety Assist Protocols

Cuboid Versions of 3D Simulation Scenes in Driving Scenario Designer

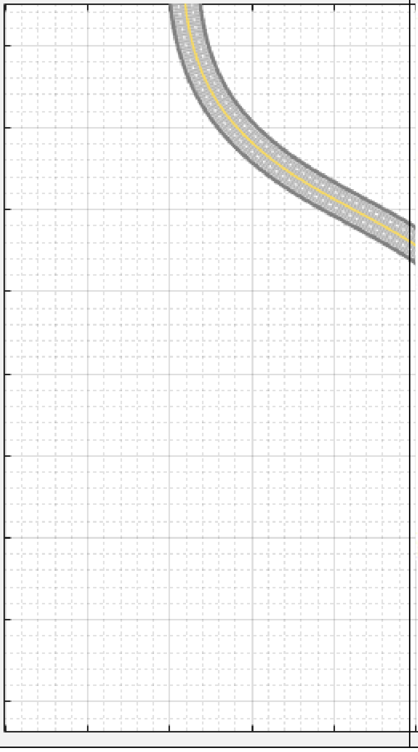
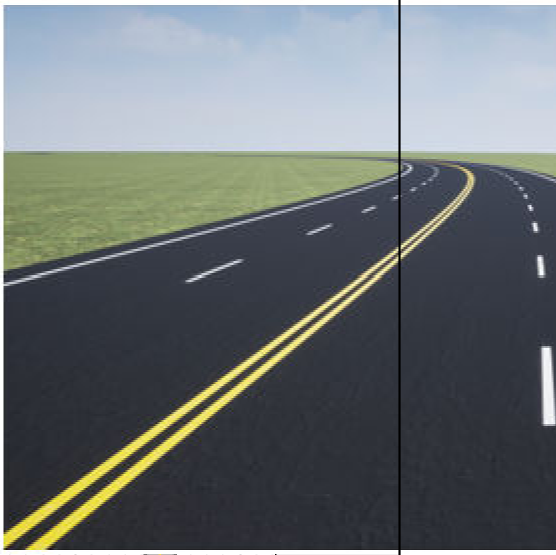
The **Driving Scenario Designer** app provides prebuilt scenarios that recreate scenes from the 3D simulation environment within the cuboid simulation environment. In these cuboid versions of the scenes, you can add vehicles represented using simple box shapes and specify their trajectories. Then, you 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 these scenes using the Unreal Engine from Epic Games. For more details about the environment, see “3D Simulation for Automated Driving” on page 6-2.

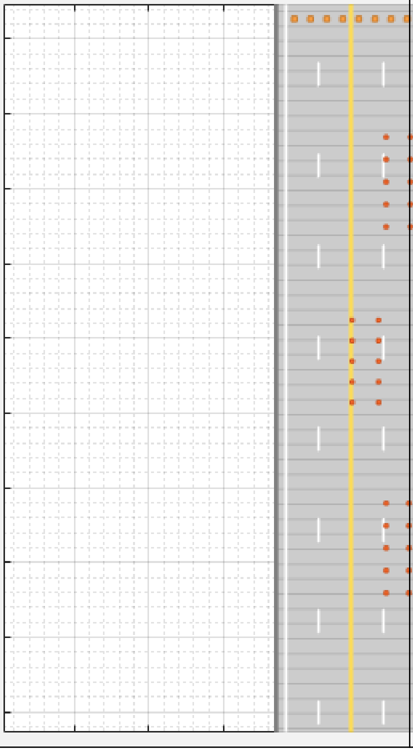
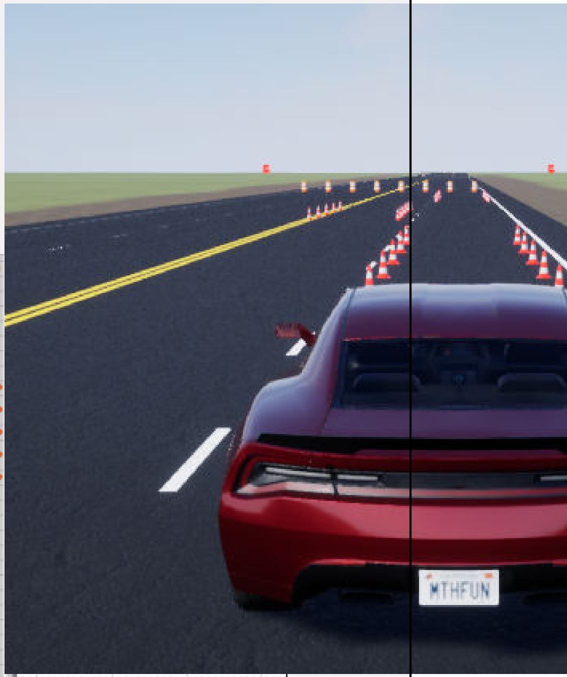
Choose 3D Simulation Scenario

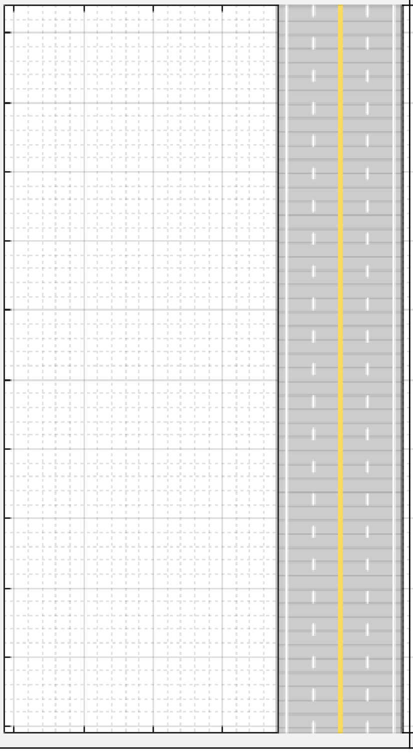
Open the **Driving Scenario Designer** app. At the MATLAB command prompt, enter `drivingScenarioDesigner`.

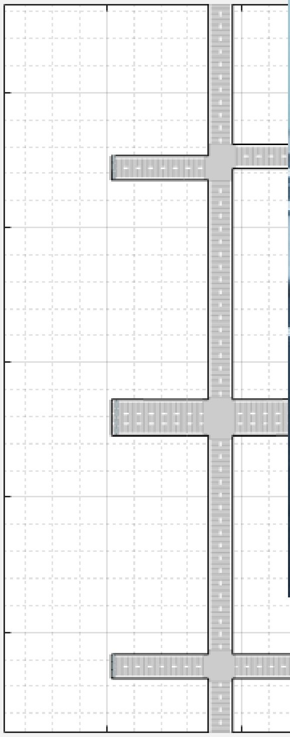
The app stores the 3D simulation scenarios as MAT-files called scenario files. To open a scenario file, first select **Open** > **Prebuilt Scenario** on the app toolstrip. The **PrebuiltScenarios** folder that opens includes subfolders for all prebuilt scenarios available in the app.

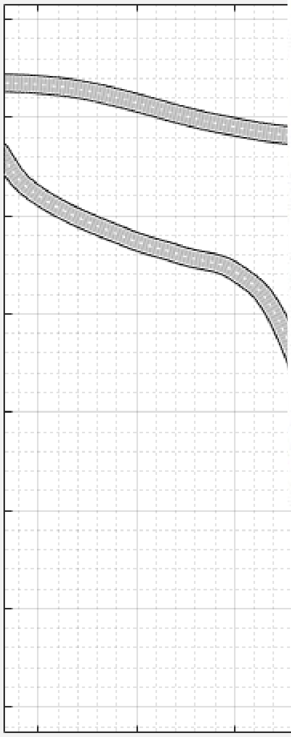
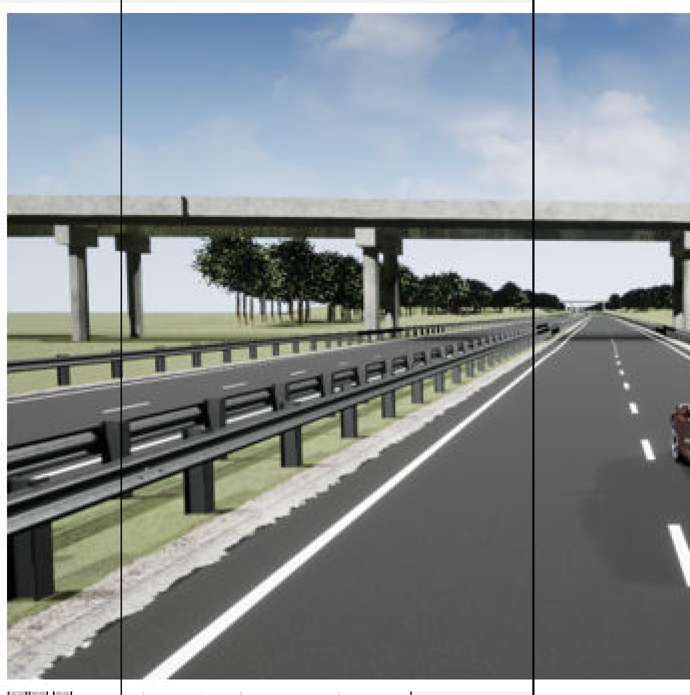
Double-click the **Simulation3D** folder, and then choose one of the scenarios described in this table.

File Name of Cuboid Scenario	Description	Corresponding 3D Scene
`CurvedRoad.mat`	Curved, looped road 	**Curved Road** 

File Name of Cuboid Scenario	Description	Corresponding 3D Scene
`DoubleLaneChange.mat`	Straight road with traffic cones and traffic barrels that are set up for executing a double lane change The cuboid version does not include the traffic signs or traffic light that are in the corresponding 3D scene. 	**Double Lane Change** 

File Name of Cuboid Scenario	Description	Corresponding 3D Scene
`StraightRoad.mat`	Straight road segment 	**Straight Road**

File Name of Cuboid Scenario	Description	Corresponding 3D Scene
`USCityBlock.mat`	City block with intersections and barriers The cuboid version doe include the traffic light in the corresponding 3I It also does not include crosswalk or pedestriar markings at intersectio objects inside the city b such as buildings, trees hydrants. 	**US City Block**

File Name of Cuboid Scenario	Description	Corresponding 3D Scene
`USHighway.mat`	Highway with traffic cones and barriers The cuboid version doe include the traffic signs guard rails that are in t corresponding 3D scen 	**US Highway** 

Note The **Driving Scenario Designer** app does not include cuboid versions of these scenes:

- **Large Parking Lot**
- **Open Surface**
- **Parking Lot**
- **Virtual Mcity**

To generate vehicle trajectories for these unsupported scenes, use the process described in the "Select Waypoints for 3D Simulation" example.

Modify Scenario

With the scenario loaded, you can now add vehicles to the scenario, set their trajectories, and designate one of the vehicles as the ego vehicle. For an example that shows how to do complete these actions, see "Build a Driving Scenario and Generate Synthetic Detections" on page 5-2.

If you plan to simulate these vehicles in the corresponding 3D scene, avoid modifying the road network or existing road objects, such as barriers and traffic cones. Otherwise, you can break parity between the two versions and simulation results might differ. To prevent such accidental changes to the road network, road interactions are disabled by default. If you want to modify the road network, in the bottom-left corner of the **Scenario Canvas** pane, click the Configure the Scenario Canvas button ⚙. Then, select **Enable road interactions**.

You can add sensors to the ego vehicle, but you cannot recreate these sensors in the 3D environment. The environment has its own sensors in the form of Simulink blocks. For more details, see “Choose a Sensor for 3D Simulation” on page 6-16.

Save Scenario

Because these scenarios are read-only, to save your scenario file, you must save a copy of it to a new folder. On the app toolstrip, select **Save > Scenario File As**.

You can reopen the saved scenario file from the app. Alternatively, at the MATLAB command prompt, enter this command, where *scenarioFileName* represents the scenario file to open.

```
drivingScenarioDesigner(scenarioFileName)
```

Recreate Scenario in Simulink for 3D Environment

After you save the scenario file containing the vehicles and other actors that you added, you can recreate these vehicles in trajectories in Simulink. At a high level, follow this workflow:

1. Configure 3D scene — In a new model, add a Simulation 3D Scene Configuration block and specify the 3D scene that corresponds to your scenario file.
2. Read actor poses from scenario file — Add a Scenario Reader block and read the actor poses at each time step from your scenario file. These poses comprise the trajectories of the actors.
3. Transform actor poses — Output the actors, including the ego vehicle, from the Scenario Reader block. Use Vehicle To World and Cuboid To 3D Simulation blocks to convert the actor poses to the coordinate system used in the 3D environment.
4. Specify actor poses to vehicles — Add Simulation 3D Vehicle with Ground Following blocks that correspond to the vehicles in your model. Specify the converted actor poses as inputs to the vehicle blocks.
5. Add sensors and simulate — Add sensors, simulate in the 3D environment, and view sensor data using the **Bird's-Eye Scope**.

For an example that follows this workflow, see “Visualize 3D Simulation Sensor Coverages and Detections” on page 6-35.

See Also

Apps

Driving Scenario Designer

Blocks

Simulation 3D Scene Configuration | Cuboid To 3D Simulation | Scenario Reader | Vehicle To World

More About

- “Prebuilt Driving Scenarios in Driving Scenario Designer” on page 5-14
- “Visualize 3D Simulation Sensor Coverages and Detections” on page 6-35

Import OpenDRIVE Roads into Driving Scenario

OpenDRIVE [1] is an open file format that enables you to specify large and complex road networks. Using the **Driving Scenario Designer** app, you can import roads and lanes from an OpenDRIVE file into a driving scenario. You can then add actors and sensors to the scenario and generate synthetic lane and object detections for testing your driving algorithms developed in MATLAB. Alternatively, to test driving algorithms developed in Simulink, you can use a Scenario Reader block to read the road network and actors into a model.

To import OpenDRIVE roads and lanes into a `drivingScenario` object instead of into the app, use the `roadNetwork` function.

Import OpenDRIVE File

Open the **Driving Scenario Designer** app. At the MATLAB command prompt, enter:

```
drivingScenarioDesigner
```

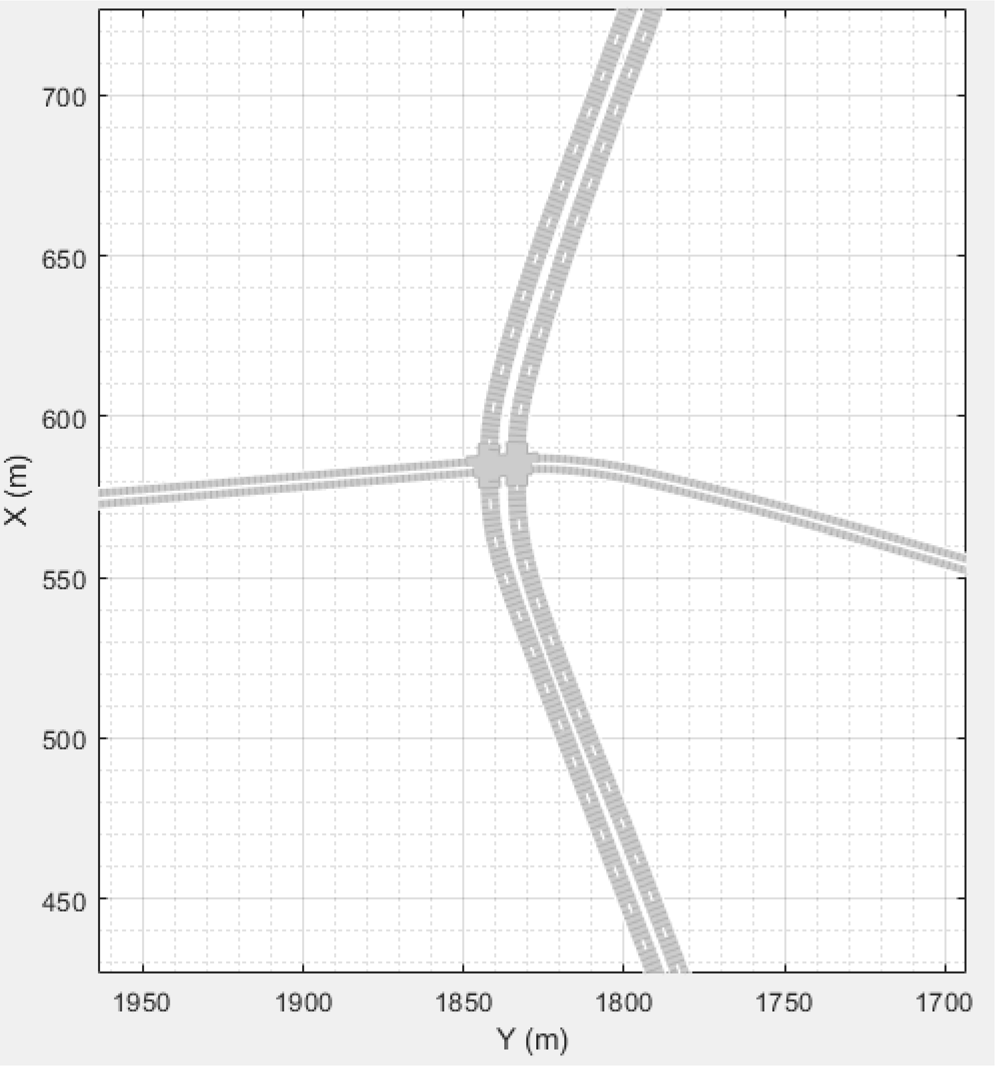
To import an OpenDRIVE file, on the app toolstrip, select **Import > OpenDRIVE Road Network**. The file you select must be a valid OpenDRIVE file of type `.xodr` or `.xml`. In addition, the file must conform with OpenDRIVE format specification version 1.4H.

From your MATLAB root folder, navigate to and open this file:

```
matlabroot/examples/driving/data/intersection.xodr
```

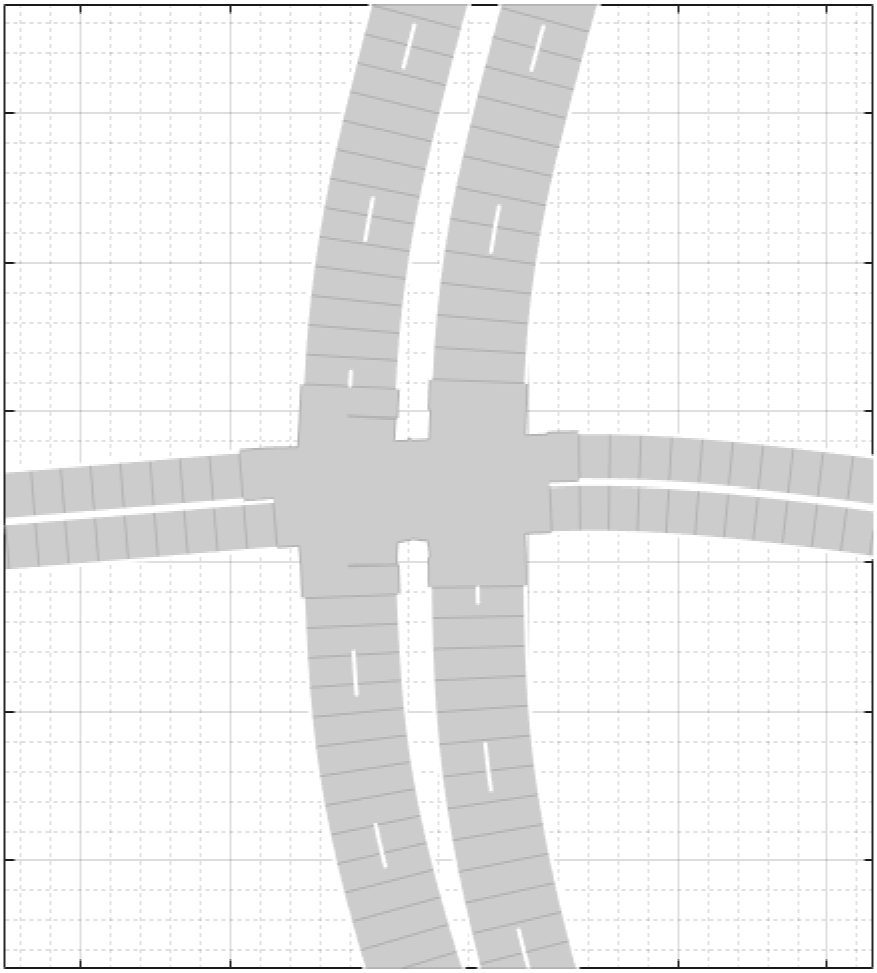
Because you cannot import an OpenDRIVE road network into an existing scenario file, the app prompts you to save your current driving scenario.

The **Scenario Canvas** of the app displays the imported road network. The roads in this network are thousands of meters long. Zoom in (press **Ctrl+Plus**) on the road to inspect it more closely.



Inspect Roads

The imported road network shows a pair of two-lane roads intersecting with a single two-lane road.



Verify that the road network imported as expected, keeping in mind the following limitations and behaviors within the app.

OpenDRIVE Import Limitations

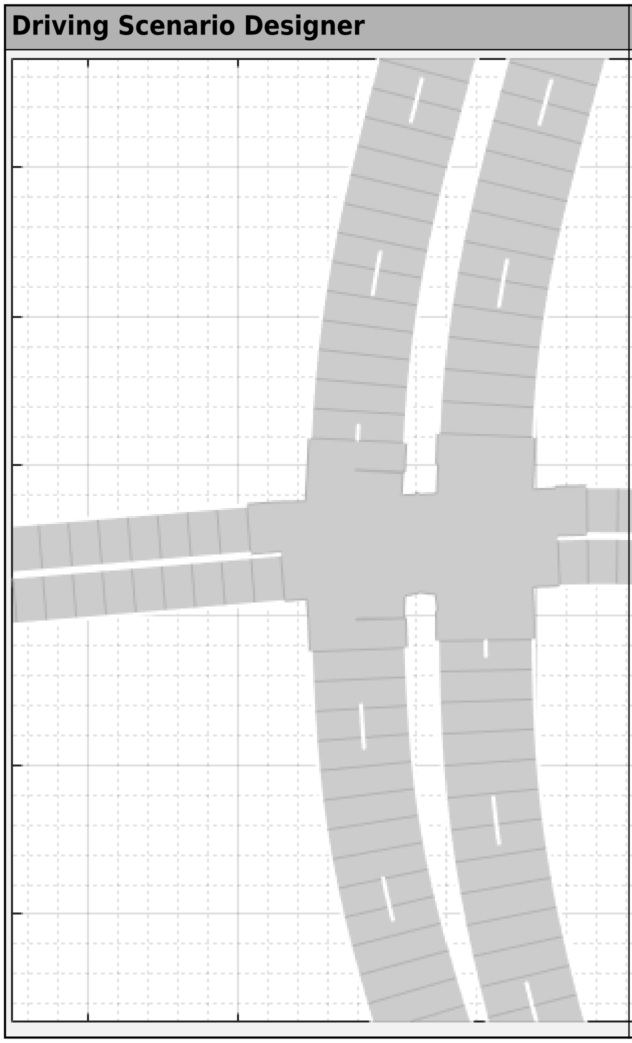
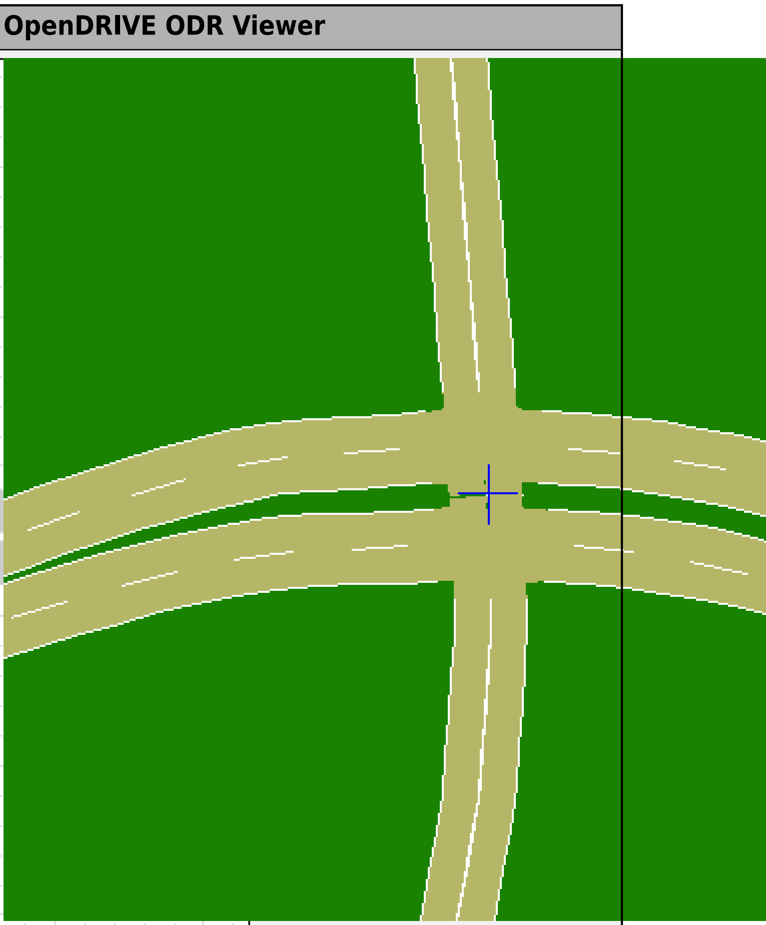
The **Driving Scenario Designer** app does not support all components of the OpenDRIVE specification.

- You can import only lanes, lane type information, and roads. The import of road objects and traffic signals is not supported.
- OpenDRIVE files containing large road networks can take up to several minutes to load. In addition, these road networks can cause slow interactions on the app canvas. Examples of large road networks include ones that model the roads of a city or ones with roads that are thousands of meters long.
- Lanes with variable widths are not supported. The width is set to the highest width found within that lane. For example, if a lane has a width that varies from 2 meters to 4 meters, the app sets the lane width to 4 meters throughout.
- Roads with lane type information specified as `driving`, `border`, `restricted`, `shoulder`, and `parking` are supported. Lanes with any other lane type information are imported as border lanes.
- Roads with multiple lane marking styles specified as `'Unmarked'`, `'Solid'`, `'DoubleSolid'`, `'Dashed'`, `'DoubleDashed'`, `'SolidDashed'`, and `'DashedSolid'` are supported.

- Lane marking styles `Bott Dots`, `Curbs`, and `Grass` are not supported. Lanes with these marking styles are imported as unmarked.

Road Orientation

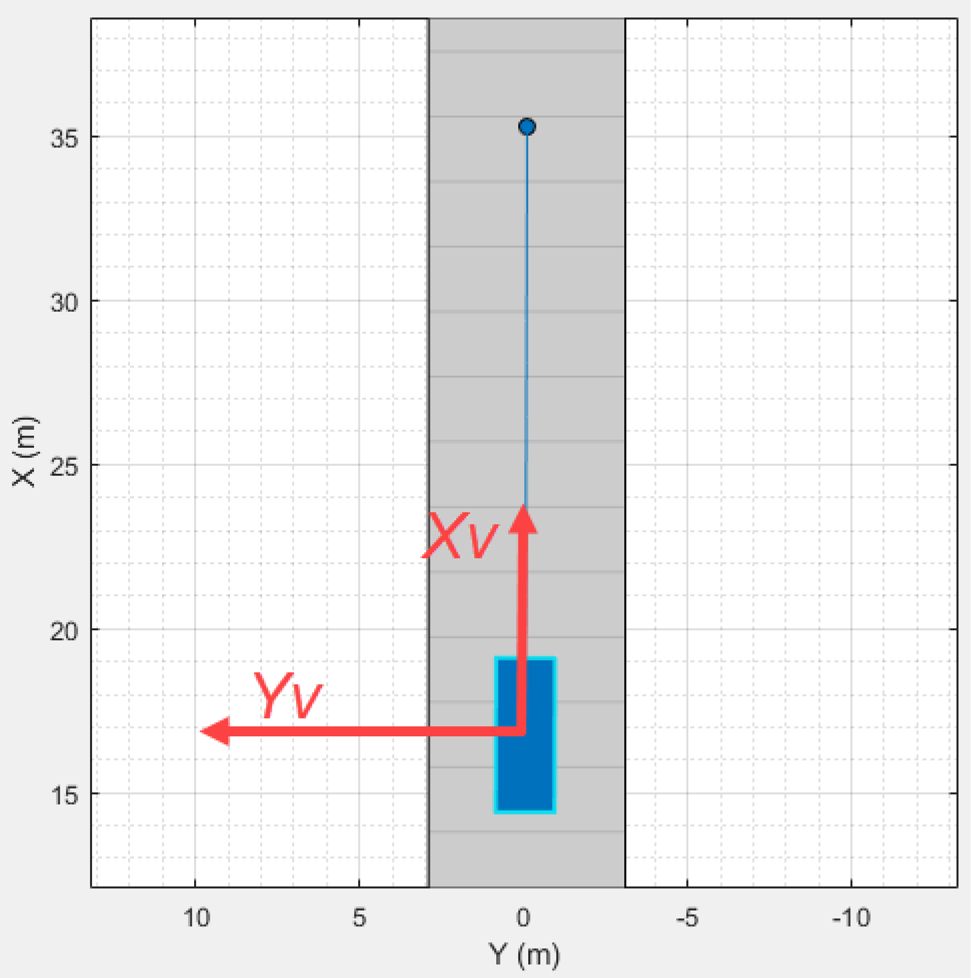
In the **Driving Scenario Designer** app, the orientation of roads can differ from the orientation of roads in other tools that display OpenDRIVE roads. The table shows this difference in orientation between the app and the OpenDRIVE ODR Viewer.

Driving Scenario Designer	OpenDRIVE ODR Viewer
	

In the OpenDRIVE ODR viewer, the *X*-axis runs along the bottom of the viewer, and the *Y*-axis runs along the left side of the viewer.

In the **Driving Scenario Designer** app, the *Y*-axis runs along the bottom of the canvas, and the *X*-axis runs along the left side of the canvas. This world coordinate system in the app aligns with the vehicle coordinate system (X_V,Y_V) used by vehicles in the driving scenario, where:

- The X_V-axis (longitudinal axis) points forward from a vehicle in the scenario.
- The Y_V-axis (lateral axis) points to the left of the vehicle, as viewed when facing forward.



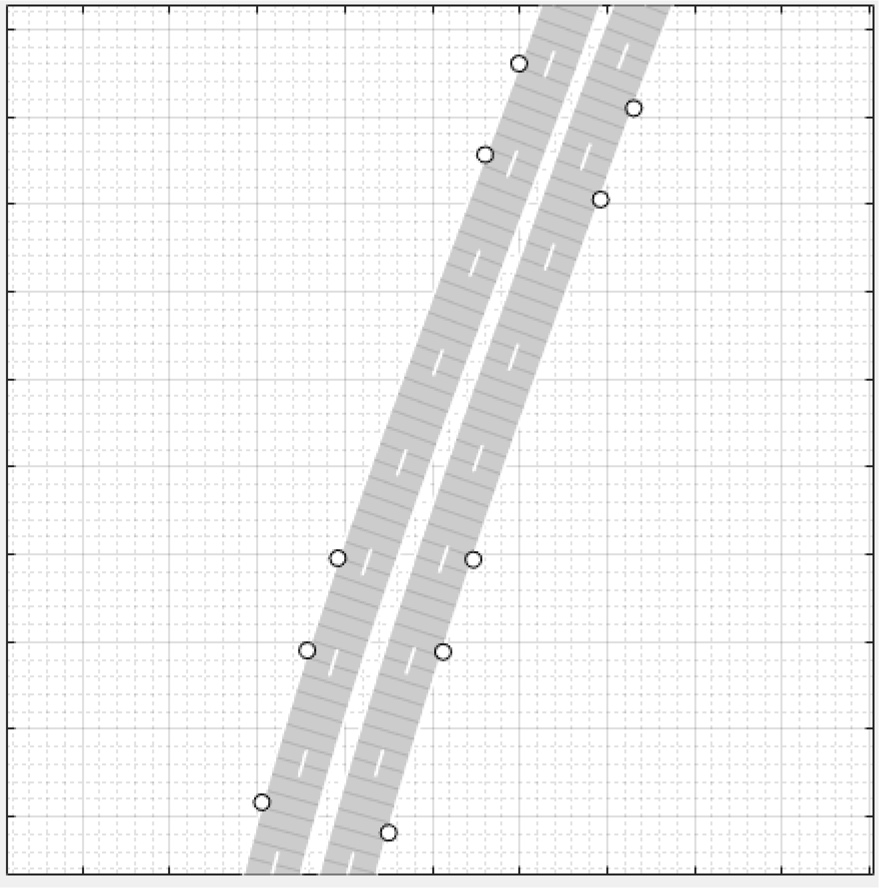
For more details about the coordinate systems, see “Coordinate Systems in Automated Driving Toolbox” on page 1-2.

Road Centers on Edges

In the **Driving Scenario Designer** app, the location and orientation of roads are defined by road centers. When you create a road in the app, the road centers are always in the middle of the road. When you import OpenDRIVE road networks into the app, however, some roads have their road centers on the road edges. This behavior occurs when the OpenDRIVE roads are explicitly specified as being right lanes or left lanes.

Consider the divided highway in the imported OpenDRIVE file. First, enable road interactions so that you can see the road centers. In the bottom-left corner of the **Scenario Canvas**, click the Configure the Scenario Canvas button ⚙, and then select **Enable road interactions**.

- The lanes on the right side of the highway have their road centers on the right edge.
- The lanes on the left side of the highway have their road centers on the left edge.

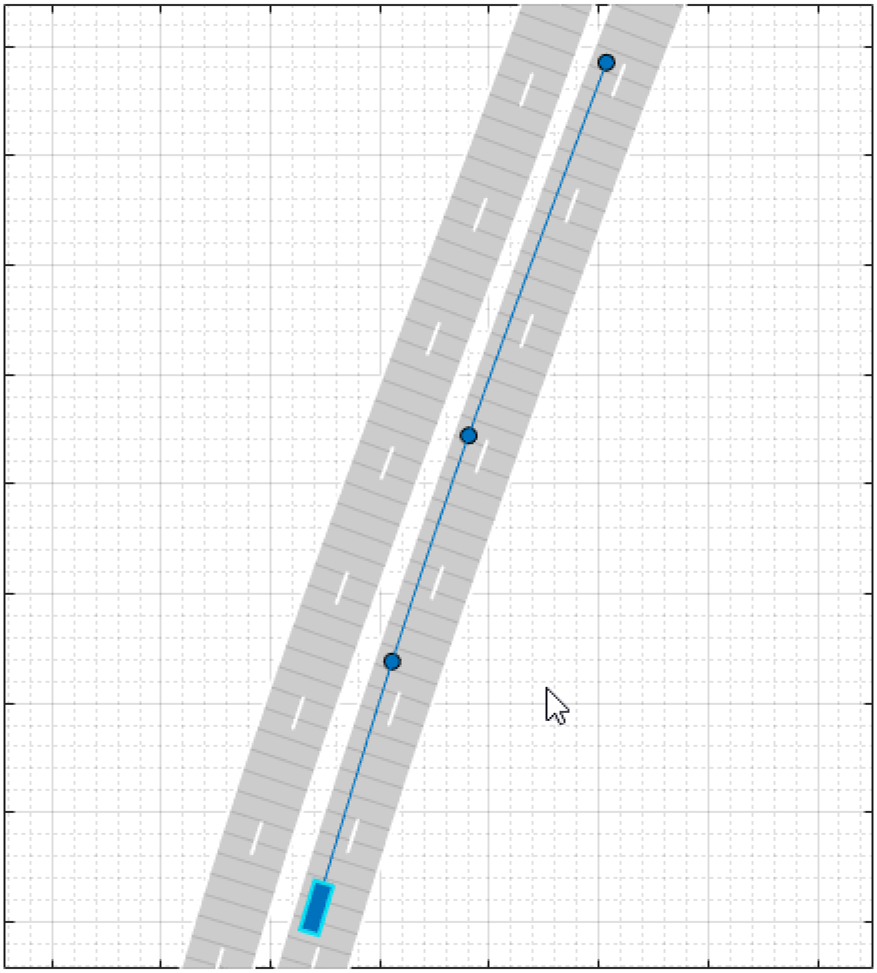


Add Actors and Sensors to Scenario

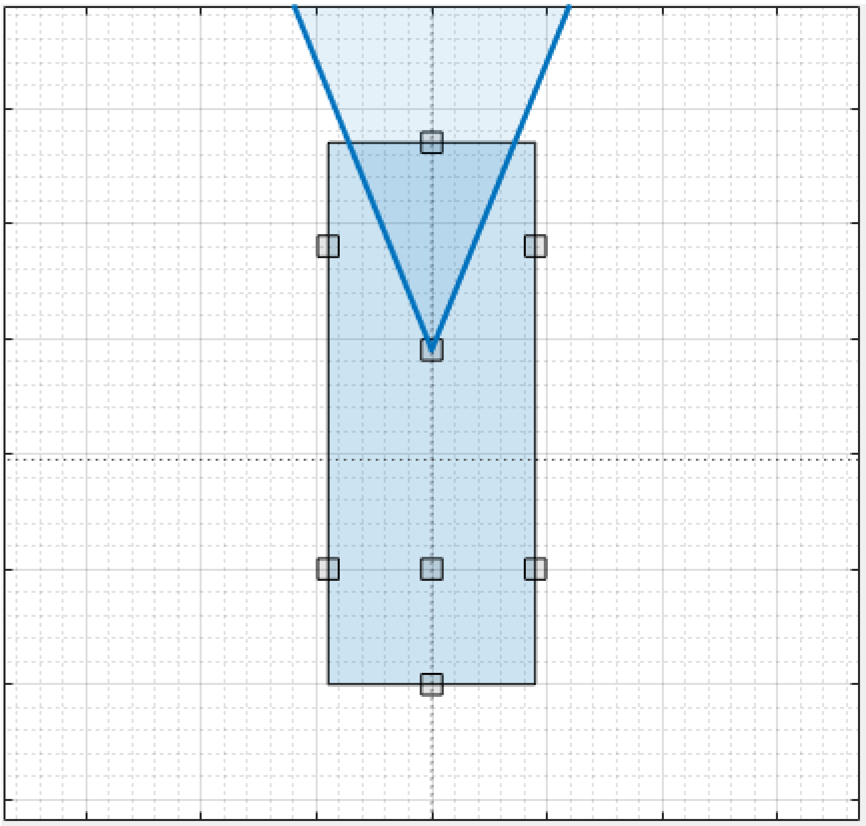
You can add actors and sensors to a scenario containing OpenDRIVE roads. However, you cannot add other roads to the scenario. If a scenario contains an OpenDRIVE road network, the **Add Road** button in the app toolstrip is disabled. In addition, you cannot import additional OpenDRIVE road networks into a scenario.

Before adding an actor and sensors, if you have road interactions enabled, consider disabling them to prevent you from accidentally dragging road centers and changing the road network. If road interactions are enabled, in the bottom-left corner of the **Scenario Canvas**, click the Configure the Scenario Canvas button ⚙, and then clear **Disable road interactions**.

Add an ego vehicle to the scenario by right-clicking one of the roads in the canvas and selecting **Add Car**. To specify the trajectory of the car, right-click the car in the canvas, select **Add Waypoints**, and add waypoints along the road for the car to pass through. After you add the last waypoint along the road, press **Enter**. The car autorotates in the direction of the first waypoint.



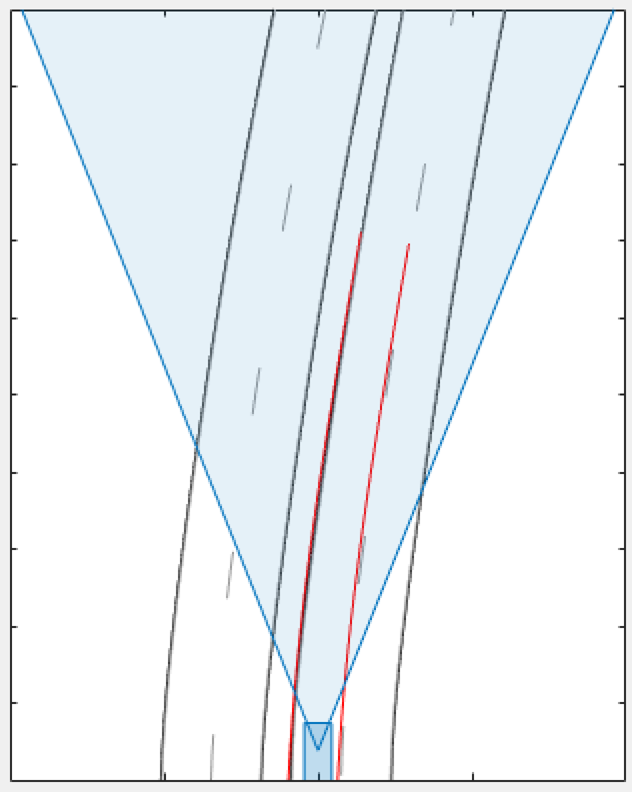
Add a camera sensor to the ego vehicle. On the app toolstrip, click **Add Camera**. Then, on the sensor canvas, add the camera to the predefined location representing the front window of the car.



Configure the camera to detect lanes. In the left pane, on the **Sensors** tab, expand the **Detection Parameters** section. Then, set the **Detection Type** parameter to `Lanes`.

Generate Synthetic Detections

To generate lane detections from the camera, on the app toolstrip, click **Run**. As the scenario runs, the **Ego-Centric View** displays the scenario from the perspective of the ego vehicle. The **Bird's-Eye Plot** displays the left-lane and right-lane boundaries of the ego vehicle.



To export the detections to the MATLAB workspace, on the app toolstrip, click **Export** > **Export Sensor Data**. Name the workspace variable and click **OK**.

The **Export** > **Export MATLAB Function** option is disabled. If a scenario contains OpenDRIVE roads, then you cannot export a MATLAB function that generates the scenario and its detections.

Save Scenario

After you generate the detections, click **Save** to save the scenario file. In addition, you can save the sensor models as separate files. You can also save the road and actor models together as a separate scenario file.

You can reopen this scenario file from the app. Alternatively, at the MATLAB command prompt, you can use this syntax.

```
drivingScenarioDesigner(scenarioFileName)
```

When you reopen this file, the **Add Road** button remains disabled.

If you are developing a driving algorithm in Simulink, you can use a Scenario Reader block to read the roads and actors from the scenario file into your model. Scenario files containing large OpenDRIVE road networks can take up to several minutes to read into models.

If you are developing a driving algorithm in Simulink, you can use a Scenario Reader block to read roads and actors from the scenario file or `drivingScenario` object into your model. This block does not directly read sensor data. To add sensors created in the app to a Simulink model, you can generate a model containing your scenario and sensors by selecting **Export** > **Export Simulink**

Model. In this model, a Scenario Reader block reads the scenario and Radar Detection Generator and Vision Detection Generator blocks model the sensors.

References

[1] Dupuis, Marius, et al. *OpenDRIVE Format Specification*. Revision 1.4, Issue H, Document No. VI2014.106. Bad Aibling, Germany: VIRES Simulationstechnologie GmbH, November 4, 2015.

See Also

Apps
Driving Scenario Designer

Blocks
Scenario Reader

Objects
`drivingScenario`

Functions
`roadNetwork`

More About

- “Build a Driving Scenario and Generate Synthetic Detections” on page 5-2
- “Prebuilt Driving Scenarios in Driving Scenario Designer” on page 5-14
- “Coordinate Systems in Automated Driving Toolbox” on page 1-2
- “Scenario Generation from Recorded Vehicle Data”

External Websites

- opendrive.org

Import HERE HD Live Map Roads into Driving Scenario

HERE HD Live Map[7] (HERE HDLM), developed by HERE Technologies, is a cloud-based web service that enables you to access highly accurate, continuously updated map data. Using the **Driving Scenario Designer** app, you can import map data from the HERE HDLM service and use it to generate roads for your driving scenarios.

This example focuses on importing map data in the app. Alternatively, to import HERE HDLM roads into a `drivingScenario` object, use the `roadNetwork` function.

Set Up HERE HDLM Credentials

To access the HERE HDLM web service, you need to enter valid HERE credentials obtained from HERE Technologies. Set up these credentials by using the `hereHDLMCredentials` function. At the MATLAB command prompt, enter:

```
hereHDLMCredentials setup
```

In the HERE HD Live Map Credentials dialog box, enter a valid **App ID** and **App Code**. To save your credentials for future MATLAB sessions on your machine, in the dialog box, select **Save my credentials between MATLAB sessions** and click **OK**. The credentials are now saved for the rest of your MATLAB session on your machine.

If you need to change your credentials, you can delete them and set up new ones by using the `hereHDLMCredentials` function.

Specify Geographic Coordinates

To select the roads you want to import, you need to specify a region of interest from which to obtain the road data. To define this region of interest, specify latitude and longitude coordinates that are near that road data. You can specify coordinates for a single point or a set of points, such as ones that make up a driving route.

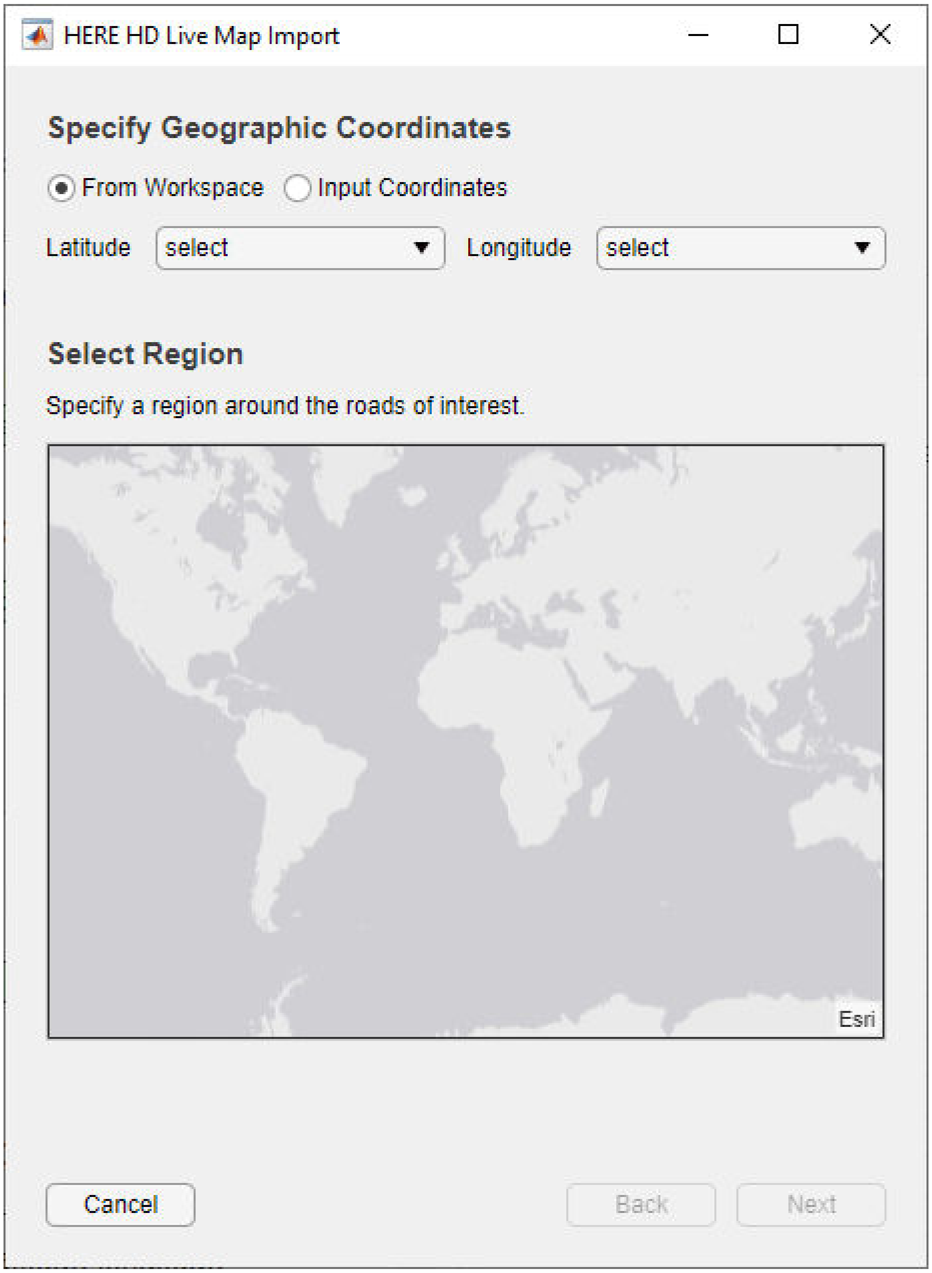
Specify the coordinates from a driving route.

1. Load a sequence of latitude and longitude coordinates that make up a driving route. At the MATLAB command prompt, enter these commands:

   ```
   data = load('geoSequence.mat');
   lat = data.latitude;
   lon = data.longitude;
   ```
2. Open the app.

   ```
   drivingScenarioDesigner
   ```
3. On the app toolstrip, select **Import** > **HERE HD Live Map**. If you previously entered or saved HERE credentials, then the dialog box opens directly to the page where you can specify geographic coordinates.

[7] 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 (app_id and app_code) for using the HERE Service.



HERE HD Live Map Import

Specify Geographic Coordinates

● From Workspace ○ Input Coordinates

Latitude: select ▼ Longitude: select ▼

Select Region

Specify a region around the roads of interest.

4 Leave **From Workspace** selected, and then select the variables for the route coordinates.

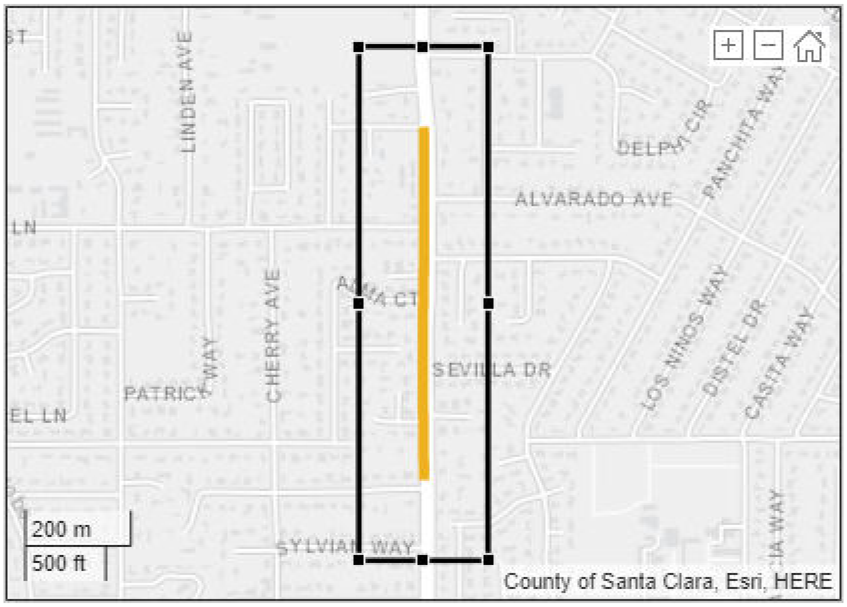
- Set the **Latitude** parameter to `lat`.
- Set the **Longitude** parameter to `lon`.

This table describes the complete list of options for specifying latitude and longitude coordinates.

Specify Geographic Coordinates Parameter Value	Description	Latitude Parameter Value	Longitude Parameter Value
From Workspace	Specify a set of latitude and longitude coordinates, such as from a driving route obtained through a GPS. These coordinates must be stored as variables in the MATLAB workspace.	Workspace variables containing vectors of decimal values in the range [–90, 90]. Units are in degrees. **Latitude** and **Longitude** must be the same size. After you select a **Latitude** variable, the **Longitude** list includes only variables of the same size as your **Latitude** selection.	Workspace variables containing vectors of decimal values in the range [–180, 180]. Units are in degrees. **Latitude** and **Longitude** must be the same size. After you select a **Longitude** variable, if you select a **Latitude** variable of a different size, the dialog box clears your **Longitude** selection.
Input Coordinates	Specify latitude and longitude coordinates for a single geographic point.	Decimal scalar in the range [–90, 90]. Units are in degrees.	Decimal scalar in the range [–180, 180]. Units are in degrees.

Select Region Containing Roads

After you specify the latitude and longitude coordinates, the **Select Region** section of the dialog box displays these coordinates in orange on a map. The coordinates are connected in a line. A rectangular region of interest displays around the coordinates. In the next page of the dialog box, you select the roads to import based on which roads are at least partially within this region.

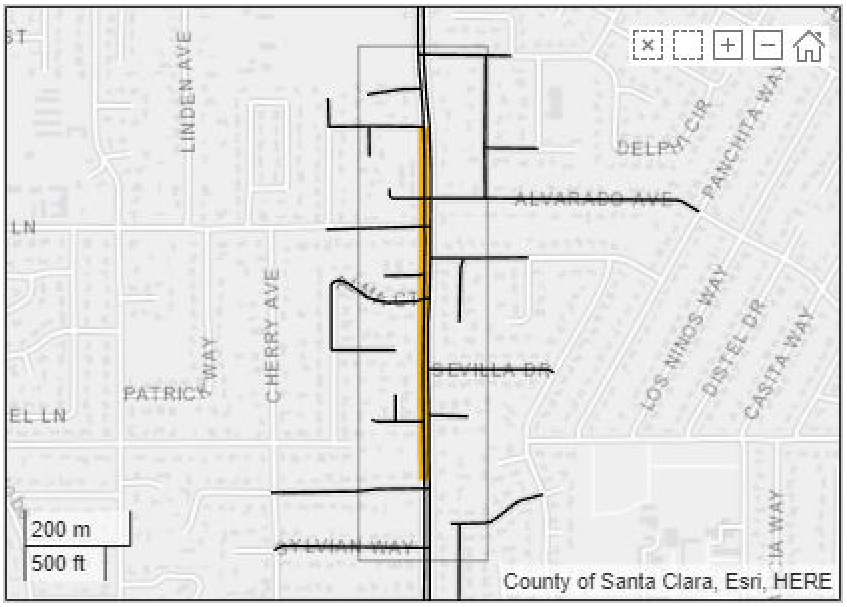


You can change the size of this region or move it around to select different roads. To zoom in and out of the region, use the buttons in the top-right corner of the map display.

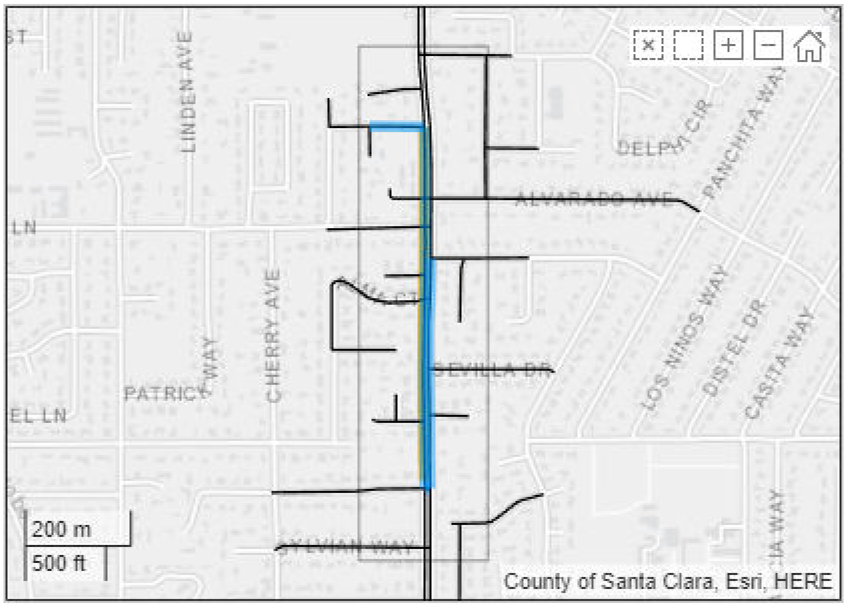
With the coordinates still enclosed within the region, click **Next**.

Select Roads to Import

After you select a region, the **Select Roads** section of the dialog box displays selectable roads in black.



Using the selected region from the previous section, select the roads that are nearest to the driving route by clicking **Select Nearest Roads**. The selected roads are overlaid onto the driving route and appear in blue.



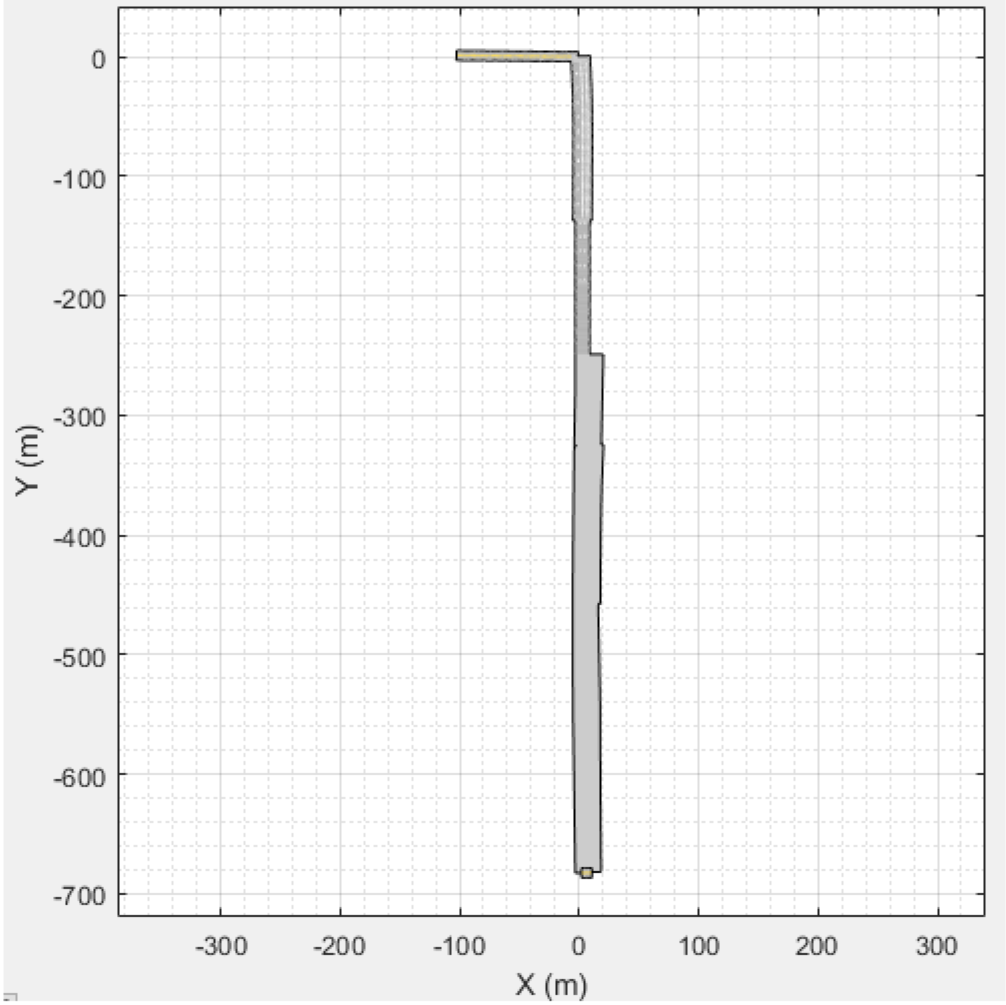
This table describes additional actions you can take for selecting roads from a region.

Goal	Action
Select individual roads from the region.	Click the individual roads to select them.
Select all roads from the region.	Click **Select All**.
Select all but a few roads from the region.	Click **Select All**, and then click the individual roads to deselect them.
Select roads from the region that are nearest to the specified coordinates.	Click **Select Nearest Roads**. Use this option when you have a sequence of nonsparse coordinates. If your coordinates are sparse or the underlying HERE HDLM data for those coordinates are sparse, then the app might not select the nearest roads.
Select a subset of roads from a region, such as all roads in the upper half of the region.	In the top-left corner of the map display, click the Select Roads button ⬚. Then, draw a rectangle around the roads to select. • To deselect a subset of roads from this selection, click the Deselect Roads button ☒. Then, draw a rectangle around the roads to deselect. • To deselect all roads and start over, click **Deselect All**.

Note The number of roads you select has a direct effect on app performance. Select the fewest roads that you need to create your driving scenario.

Import Roads

With the roads nearest to the route still selected, click **Import**. The app imports the HERE HDLM roads and generates a road network.



To maintain the same alignment with the geographic map display, the *X*-axis of the **Scenario Canvas** is on the bottom and the *Y*-axis is on the left. In driving scenarios that are not imported from maps, the *X*-axis is on the left and the *Y*-axis is on the bottom. This alignment is consistent with the Automated Driving Toolbox world coordinate system.

The origin of the scenario is the first point specified in the driving route. Even if you select roads from the end of a driving route, the origin is still anchored to this first point. If you specified a single geographic point by using the **Input Coordinates** option, then the origin is that point.

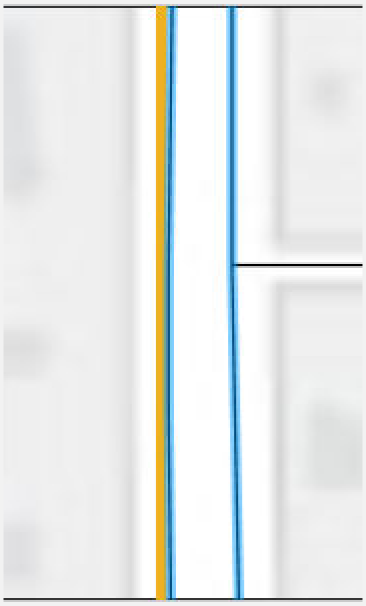
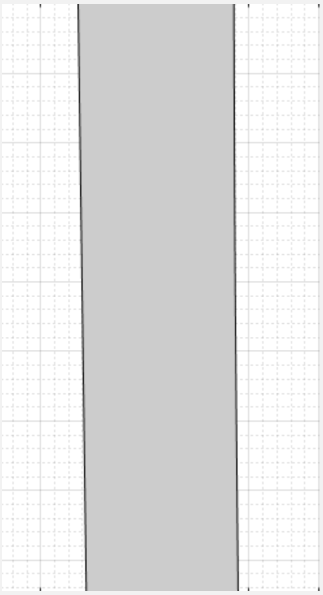
By default, road interactions are disabled. Disabled road interactions prevent you from accidentally modifying the network and reduces visual clutter by hiding the road centers. If you want to modify the roads, in the bottom-left corner of the **Scenario Canvas**, click the Configure the Scenario Canvas button ⚙. Then, select **Enable road interactions**.

Note In some cases, the app is unable to import all selected roads. This issue can occur if the curvature of the road is too sharp for the app to render it properly. In these cases, the app pauses the import, and the dialog box highlights the nonimportable roads in red. To continue importing all other selected roads, click **Continue**.

Compare Imported Roads Against Map Data

The generated road network in the app has several differences from the actual HERE HDLM road network. For example, the actual HERE HDLM road network contains roads with varying widths and varying numbers of lanes. The **Driving Scenario Designer** app does not support these features. Instead, the app sets each road to have the maximum width and the maximum number of lanes found along its entire length. These changes increase the widths of the roads and causes roads to overlap and appear as one road. Sensors that detect lanes are unable to detect the covered lanes.

This table shows the difference between a portion of the HERE HDLM road network and the imported driving scenario.

HERE HDLM Road Network	Imported Driving Scenario
	

Road networks generated from imported HERE HDLM data can have several differences from the actual HERE HDLM road network. For more details on the unsupported HERE HDLM road and lane features, see the “Limitations” section of the **Driving Scenario Designer** app reference page.

Save Scenario

Save the scenario file. After you save the scenario, you cannot import additional HERE HDLM roads into it. Instead, you need to create a new scenario and import a new road network.

You can now add actors and sensors to the scenario, generate synthetic lane and object detections for testing your driving algorithms, or import the scenario into Simulink.

See Also

Apps
Driving Scenario Designer

Blocks
Scenario Reader

Objects
`drivingScenario`

Functions
`roadNetwork`

More About

- “Access HERE HD Live Map Data” on page 4-7
- “Coordinate Systems in Automated Driving Toolbox” on page 1-2
- “Build a Driving Scenario and Generate Synthetic Detections” on page 5-2

External Websites

- HERE Technologies

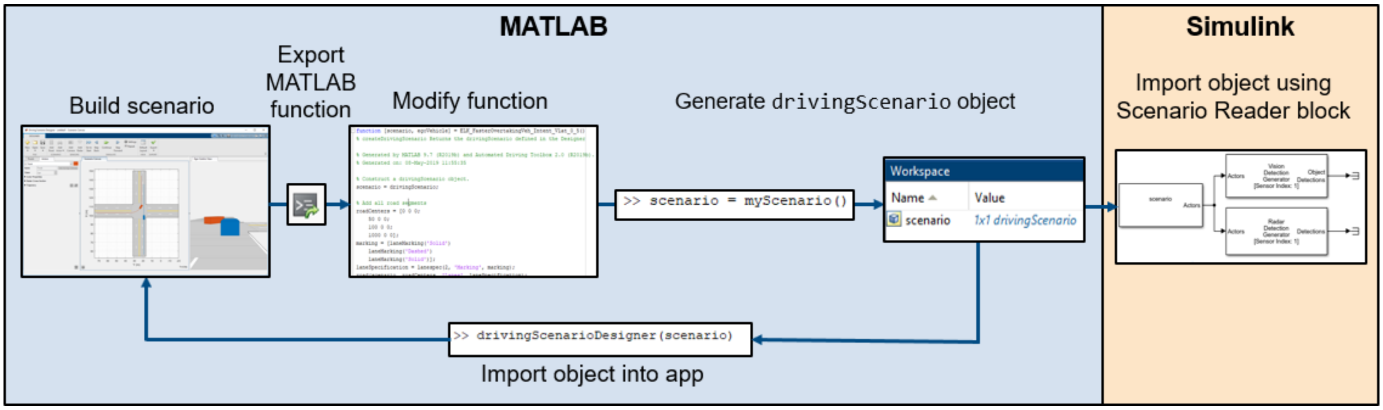
Create Driving Scenario Variations Programmatically

This example shows how to programmatically create variations of a driving scenario that was built using the Driving Scenario Designer app. Programmatically creating variations of a scenario enables you to rapidly test your driving algorithms under multiple conditions.

To create programmatic variations of a driving scenario, follow these steps:

1. Interactively build a driving scenario by using the Driving Scenario Designer app.
2. Export a MATLAB® function that generates the MATLAB code that is equivalent to this scenario.
3. In the MATLAB Editor, modify the exported function to create variations of the original scenario.
4. Call the function to generate a `drivingScenario` object that represents the scenario.
5. Import the scenario object into the app to simulate the modified scenario or generate additional scenarios. Alternatively, to simulate the modified scenario in Simulink®, import the object into a Simulink model by using a Scenario Reader block.

The diagram shows a visual representation of this workflow.



```
>> scenario = myScenario()
```

```
>> drivingScenarioDesigner(scenario)
```

Before beginning this example, add the example file folder to the MATLAB search path.

```
addpath(genpath(fullfile(matlabroot,'examples','driving')))
```

Build Scenario in App

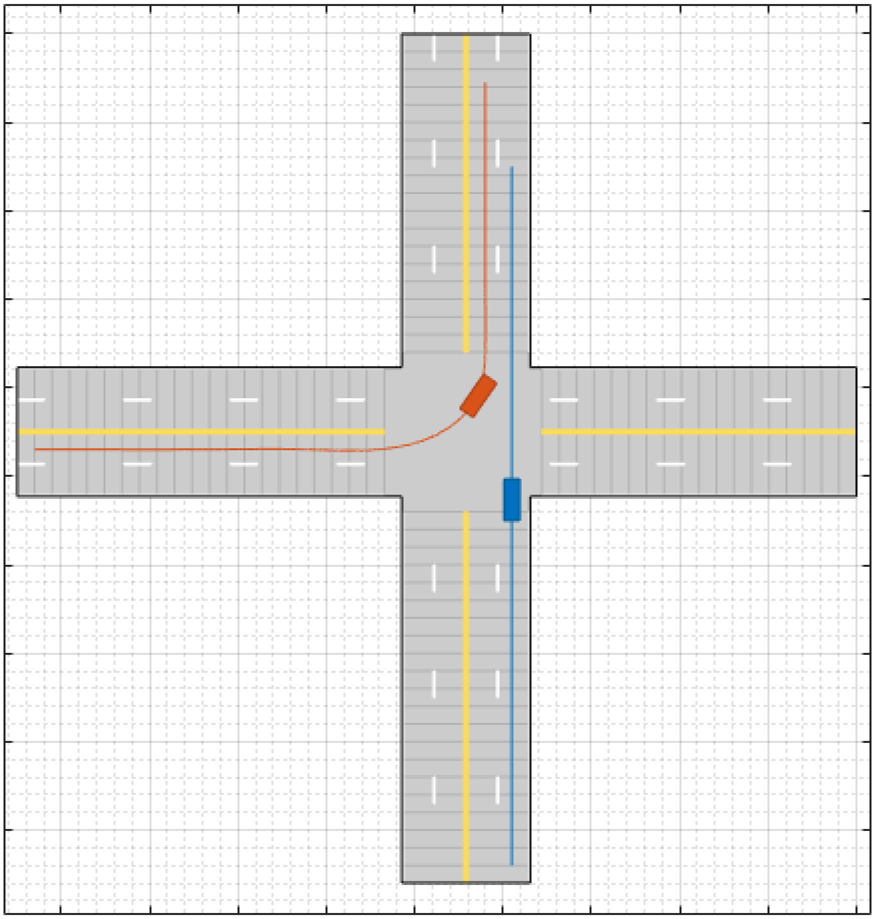
Use the Driving Scenario Designer to interactively build a driving scenario on which to test your algorithms. For more details on building scenarios, see “Build a Driving Scenario and Generate Synthetic Detections” on page 5-2.

This example uses a driving scenario that is based on one of the prebuilt scenarios that you can load from the Driving Scenario Designer app.

Open the scenario file in the app.

```
drivingScenarioDesigner('LeftTurnScenarioNoSensors.mat')
```

Click **Run** to simulate the scenario. In this scenario, the ego vehicle travels north and goes straight through an intersection. Meanwhile, a vehicle coming from the left side of the intersection turns left and ends up in front of the ego vehicle, in the adjacent lane.

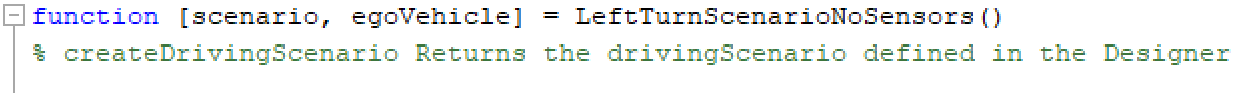


For simplicity, this scenario does not include sensors mounted on the ego vehicle.

Export MATLAB Function of Scenario

After you view and simulate the scenario, you can export the scenario to the MATLAB command line. From the Driving Scenario Designer app toolstrip, select **Export > Export MATLAB Function**. The exported function contains the MATLAB code used to produce the scenario created in the app. Open the exported function.

```
open LeftTurnScenarioNoSensors.m
```



```
function [scenario, egoVehicle] = LeftTurnScenarioNoSensors()
% createDrivingScenario Returns the drivingScenario defined in the Designer
```

Calling this function returns these aspects of the driving scenario.

- `scenario` — Roads and actors of the scenarios, returned as a `drivingScenario` object.
- `egoVehicle` — Ego vehicle defined in the scenario, returned as a `Vehicle` object. For details, see the `vehicle` function.

If your scenario contains sensors, then the returned function includes additional code for generating the sensors. If you simulated the scenario containing those sensors, then the function can also generate the detections produced by those sensors.

Modify Function to Create Scenario Variations

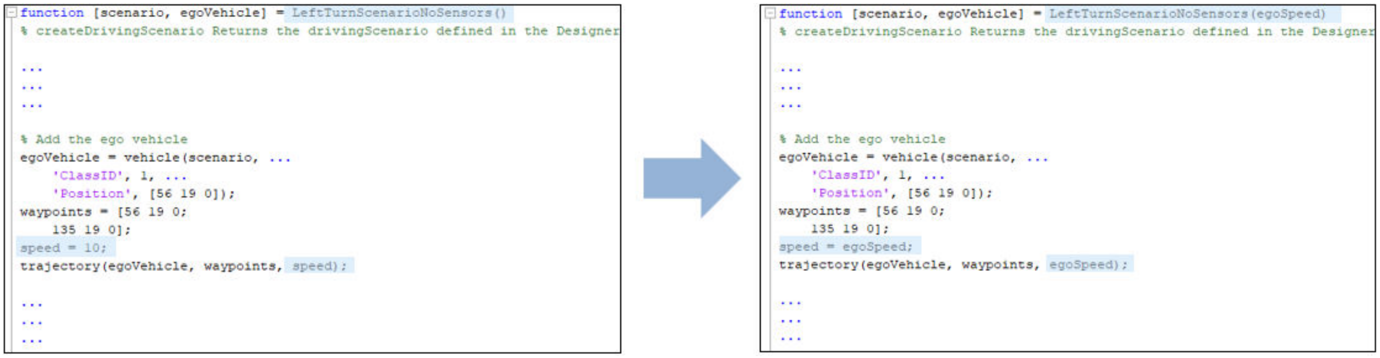
By modifying the code in the exported MATLAB function, you can generate multiple variations of a single scenario. One common variation is to test the ego vehicle at different speeds. In the exported MATLAB function, the speed of the ego vehicle is set to a constant value of 10 meters per second (`speed = 10`). To generate varying ego vehicle speeds, you can convert the speed variable into an input argument to the function. Open the script containing a modified version of the exported function.

```
open LeftTurnScenarioNoSensorsModified.m
```

In this modified function:

- `egoSpeed` is included as an input argument.
- `speed`, the constant variable, is deleted.
- To compute the ego vehicle trajectory, `egoSpeed` is used instead of `speed`.

This figure shows these script modifications.



```
function [scenario, egoVehicle] = LeftTurnScenarioNoSensors()
% createDrivingScenario Returns the drivingScenario defined in the Designer

...
...
...

% Add the ego vehicle
egoVehicle = vehicle(scenario, ...
    'ClassID', 1, ...
    'Position', [56 19 0]);
waypoints = [56 19 0;
    135 19 0];
speed = 10;
trajectory(egoVehicle, waypoints, speed);

...
...
...
```

```
function [scenario, egoVehicle] = LeftTurnScenarioNoSensors(egoSpeed)
% createDrivingScenario Returns the drivingScenario defined in the Designer

...
...
...

% Add the ego vehicle
egoVehicle = vehicle(scenario, ...
    'ClassID', 1, ...
    'Position', [56 19 0]);
waypoints = [56 19 0;
    135 19 0];
speed = egoSpeed;
trajectory(egoVehicle, waypoints, egoSpeed);

...
...
...
```

To produce additional variations, consider:

- Modifying the road and lane parameters to view the effect on lane detections
- Modifying the trajectory or starting positions of the vehicles
- Modifying the dimensions of the vehicles

Call Function to Generate Programmatic Scenarios

Using the modified function, generate a variation of the scenario in which the ego vehicle travels at a constant speed of 20 meters per second.

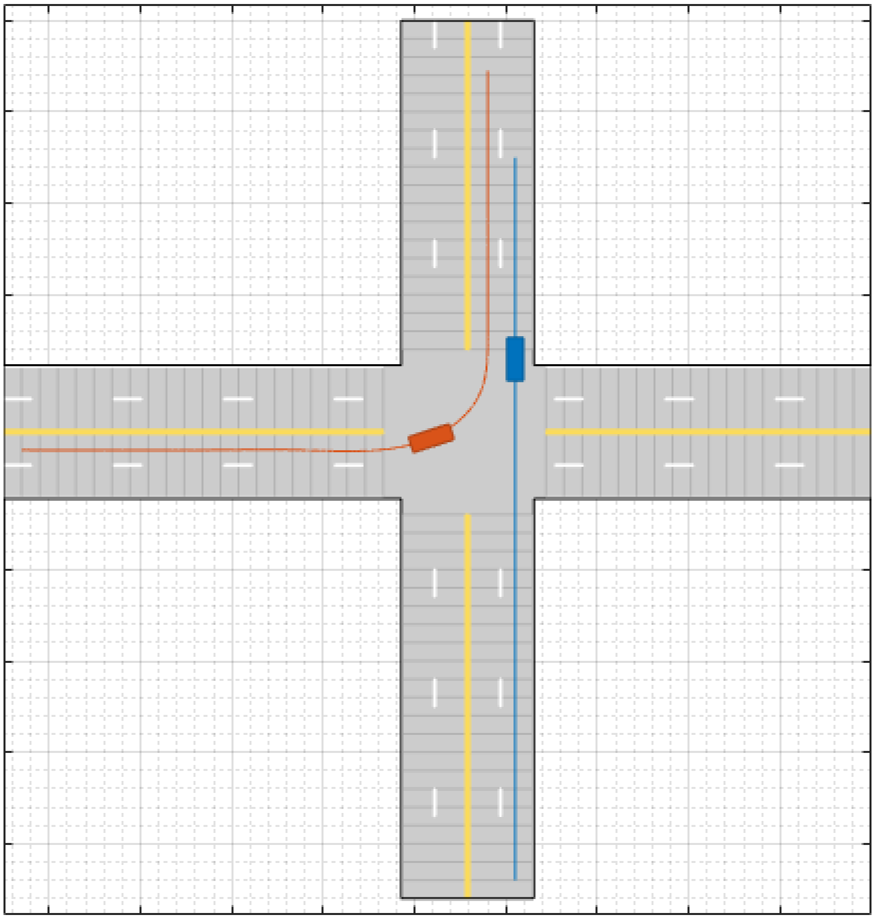
```
scenario = LeftTurnScenarioNoSensorsModified(20) % m/s
```

```
scenario = 
  drivingScenario with properties:

        SampleTime: 0.0400
          StopTime: Inf
    SimulationTime: 0
         IsRunning: 1
            Actors: [1x2 driving.scenario.Vehicle]
```

Import Modified Scenario into App

To import the modified scenario with the modified vehicle into the app, use the `drivingScenarioDesigner` function. Specify the `drivingScenario` object as an input argument.

```
drivingScenarioDesigner(scenario)
```



Previously, the other vehicle passed through the intersection first. Now, with the speed of the ego vehicle increased from 10 to 20 meters per second, the ego vehicle passes through the intersection first.

When working with `drivingScenario` objects in the app, keep these points in mind.

- To try out different ego vehicle speeds, call the exported function again, and then import the new `drivingScenario` object using the `drivingScenarioDesigner` function. The app does not include a menu option for importing these objects.
- If your scenario includes sensors, you can reopen both the scenario and sensors by using this syntax: `drivingScenarioDesigner(scenario,sensors)`.
- If you make significant changes to the dimensions of an actor, be sure that the `ClassID` property of the actor corresponds to a **Class ID** value specified in the app. For example, in the app, cars have a **Class ID** of `1` and trucks have a **Class ID** of `2`. If you programmatically change a car to have the dimensions of a truck, update the `ClassID` property of that vehicle from `1` (car) to `2` (truck).

Import Modified Scenario into Simulink

To import the modified scenario into a Simulink model, use a Scenario Reader block. This block reads the roads and actors from either a scenario file saved from the app or a `drivingScenario` variable saved to the MATLAB workspace or the model workspace. Add a Scenario Reader block to your model and set these parameters.

1. Set **Source of driving scenario** to `From workspace`.
2. Set **MATLAB or model workspace variable name** to the name of the `drivingScenario` variable in your workspace.

When working with `drivingScenario` objects in Simulink, keep these points in mind.

- When **Source of ego vehicle** is set to `Scenario`, the model uses the ego vehicle defined in your `drivingScenario` object. The block determines which actor is the ego vehicle based on the specified `ActorID` property of the actor. This actor must be a `Vehicle` object (see `vehicle`). To change the designated ego vehicle, update the **Ego vehicle ActorID** parameter.
- When connecting the output actor poses to Radar Detection Generator or Vision Detection Generator blocks, update these sensor blocks to obtain the actor profiles directly from the `drivingScenario` object. By default, these blocks use the same set of actor profiles for all actors, where the profiles are defined on the **Actor Profiles** tab of the blocks. To obtain the profiles from the object, on the **Actor Profiles** tab of each sensor block, set the **Select method to specify actor profiles** parameter to `MATLAB expression`. Then, set the **MATLAB expression for actor profiles** parameter to call the `actorProfiles` function on the object. For example: `actorProfiles(scenario)`.

When you are done with this example, remove the example file folder from the MATLAB search path.

```
rmpath(genpath(fullfile(matlabroot,'examples','driving')))
```

See Also

Apps
Driving Scenario Designer

Blocks
Radar Detection Generator | Scenario Reader | Vision Detection Generator

Functions
`actorProfiles` | `vehicle`

Objects
`drivingScenario`

More About

- “Build a Driving Scenario and Generate Synthetic Detections” on page 5-2
- “Prebuilt Driving Scenarios in Driving Scenario Designer” on page 5-14
- “Euro NCAP Driving Scenarios in Driving Scenario Designer” on page 5-36
- “Driving Scenario Tutorial”

Generate Sensor Detection Blocks Using Driving Scenario Designer

This example shows how to update the radar and camera sensors of a Simulink® model by using the Driving Scenario Designer app. The Driving Scenario Designer app enables you to generate multiple sensor configurations quickly and interactively. You can then use these generated sensor configurations in your existing Simulink models to test your driving algorithms more thoroughly.

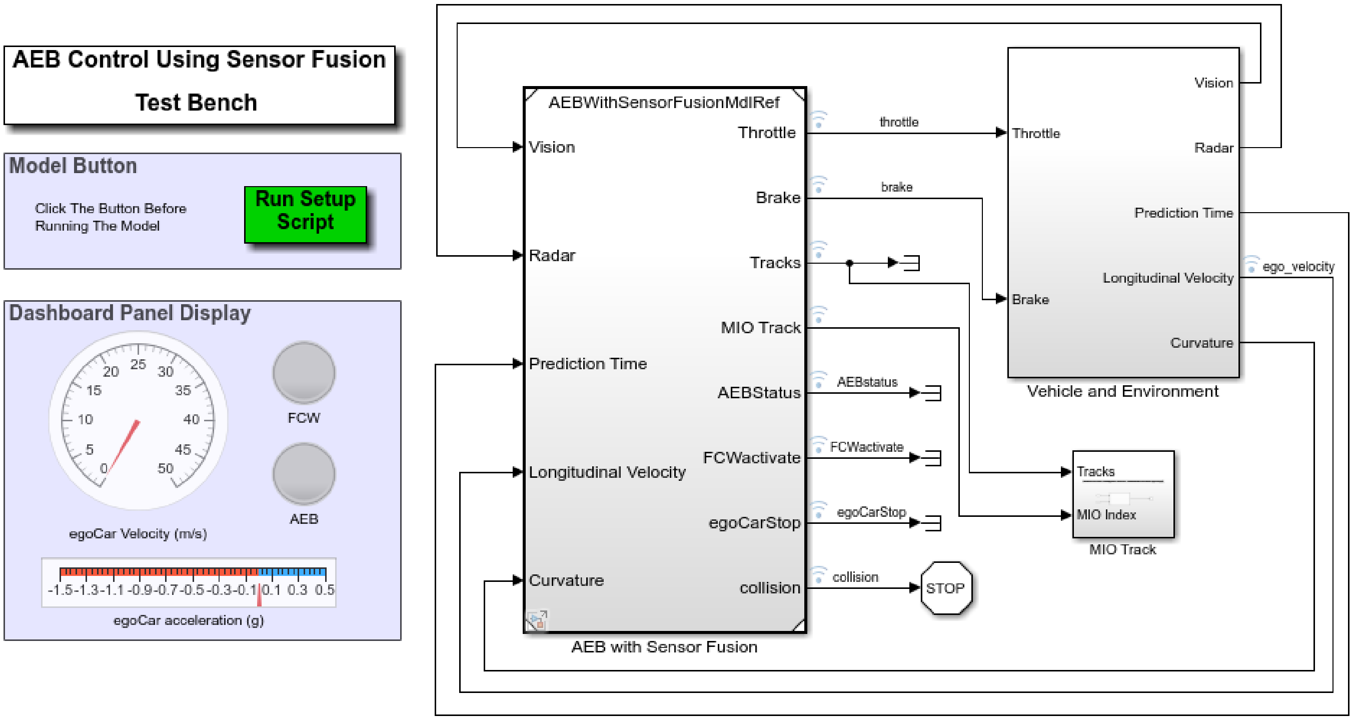
Before beginning this example, add the example file folder to the MATLAB® search path.

```
addpath(genpath(fullfile(matlabroot,'examples','driving')))
```

Inspect and Simulate Model

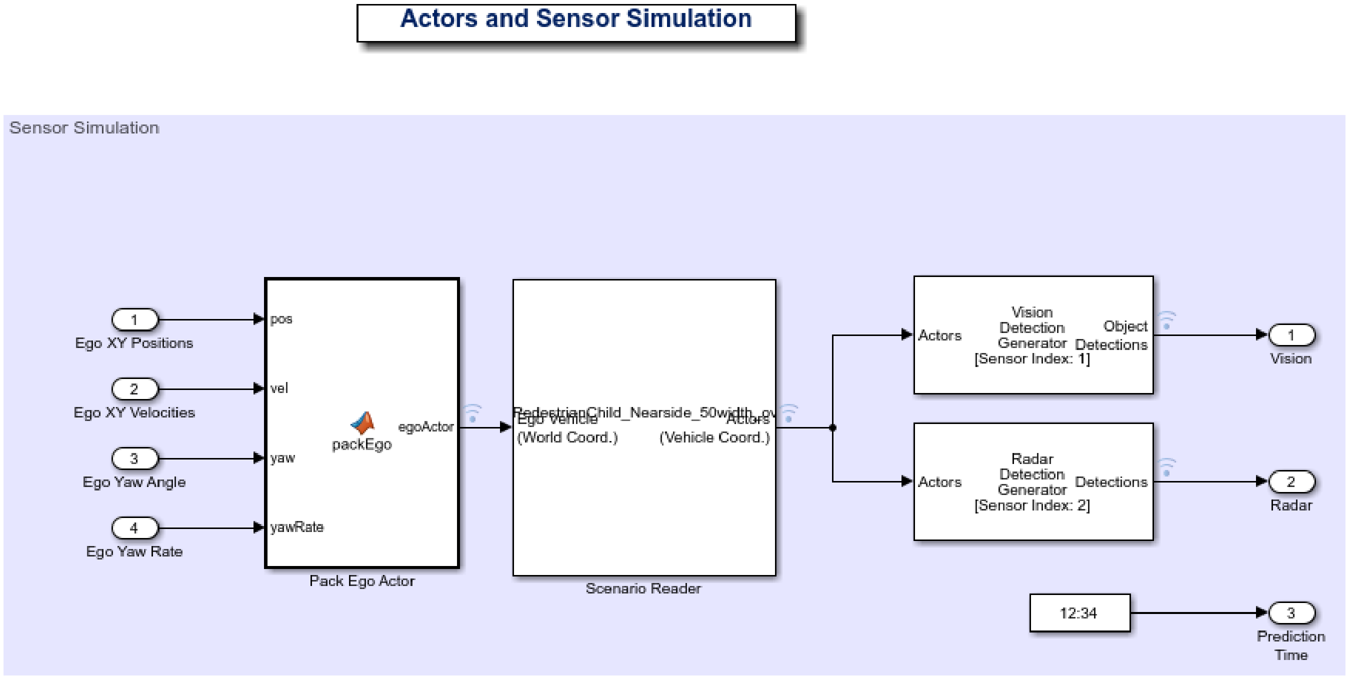
The model used in this example implements an autonomous emergency braking (AEB) sensor fusion algorithm. For more details about this model, see the “Autonomous Emergency Braking with Sensor Fusion” example. Open this model.

```
open_system('AEBTestBenchExample')
```



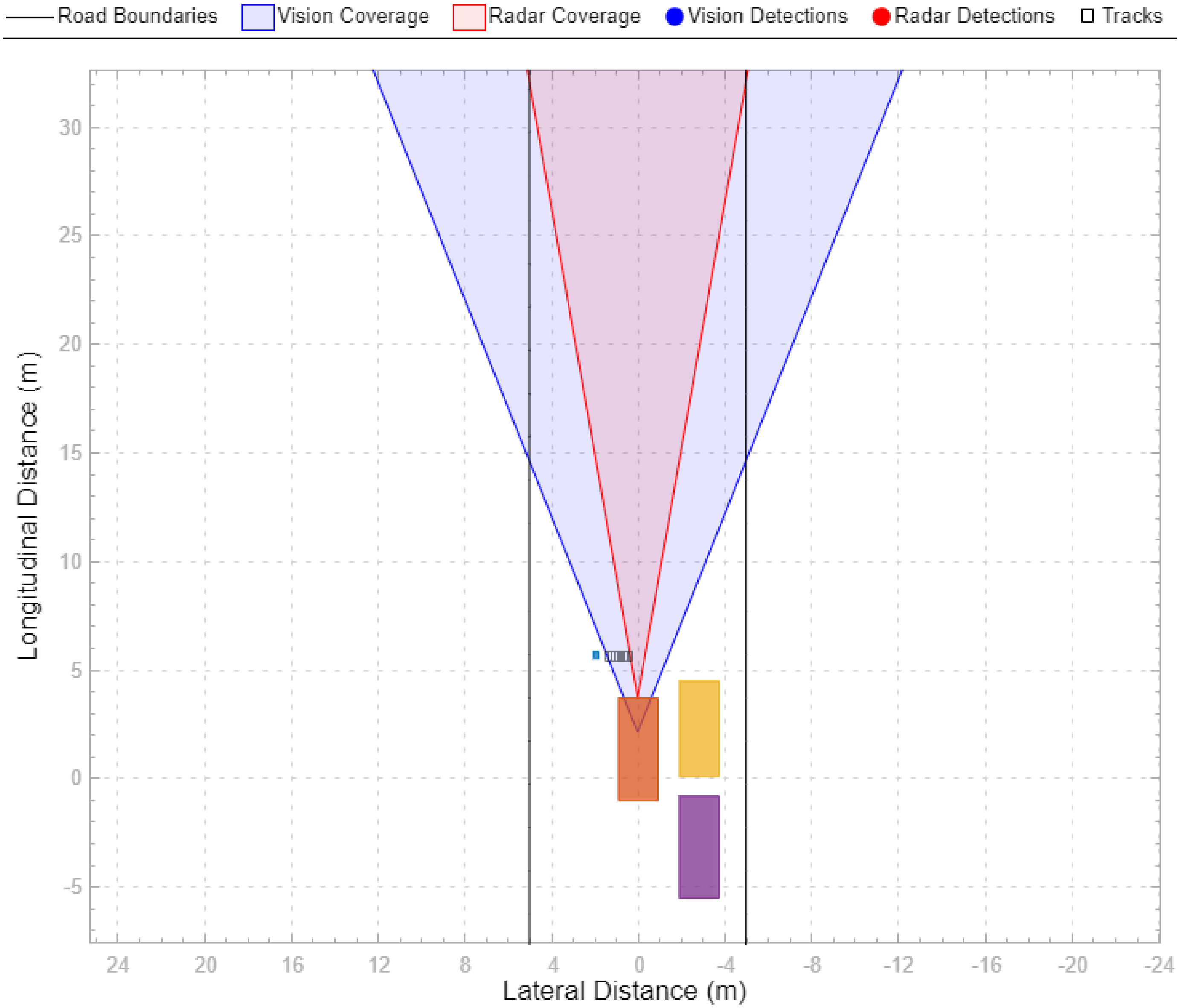
The driving scenario and sensor detection generators used to test the algorithm are located in the **Vehicle Environment > Actors and Sensor Simulation** subsystem. Open this subsystem.

```
open_system('AEBTestBenchExample/Vehicle and Environment/Actors and Sensor Simulation')
```



A Scenario Reader block reads the actors and roads from the specified Driving Scenario Designer file. The block outputs the non-ego actors. These actors are then passed to Radar Detection Generator and Vision Detection Generator sensor blocks. During simulation, these blocks generate detections of the non-ego actors.

Simulate and visualize the scenario on the Bird's-Eye Scope. On the model toolstrip, under **Review Results**, click **Bird's-Eye Scope**. In the scope, click **Find Signals**, and then click **Run** to run the simulation. In this scenario, the AEB model causes the ego vehicle to brake in time to avoid a collision with a pedestrian child who is crossing the street.



During this example, you replace the existing sensors in this model with new sensors created in the Driving Scenario Designer app.

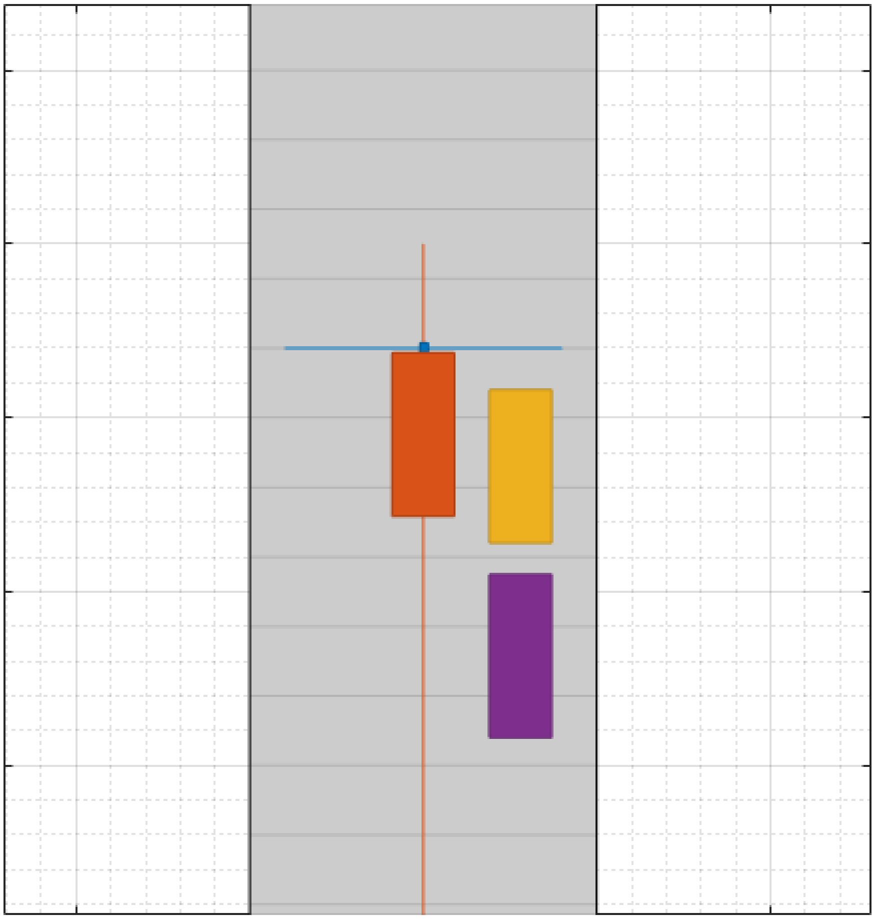
Load Scenario in App

The model uses a driving scenario that is based on one of the prebuilt Euro NCAP test protocol scenarios. You can load these scenarios from the Driving Scenario Designer app. For more details on these scenarios, see “Euro NCAP Driving Scenarios in Driving Scenario Designer” on page 5-36.

Load the scenario file into the app.

```
drivingScenarioDesigner('AEB_PedestrianChild_Nearside_50width_overrun.mat')
```

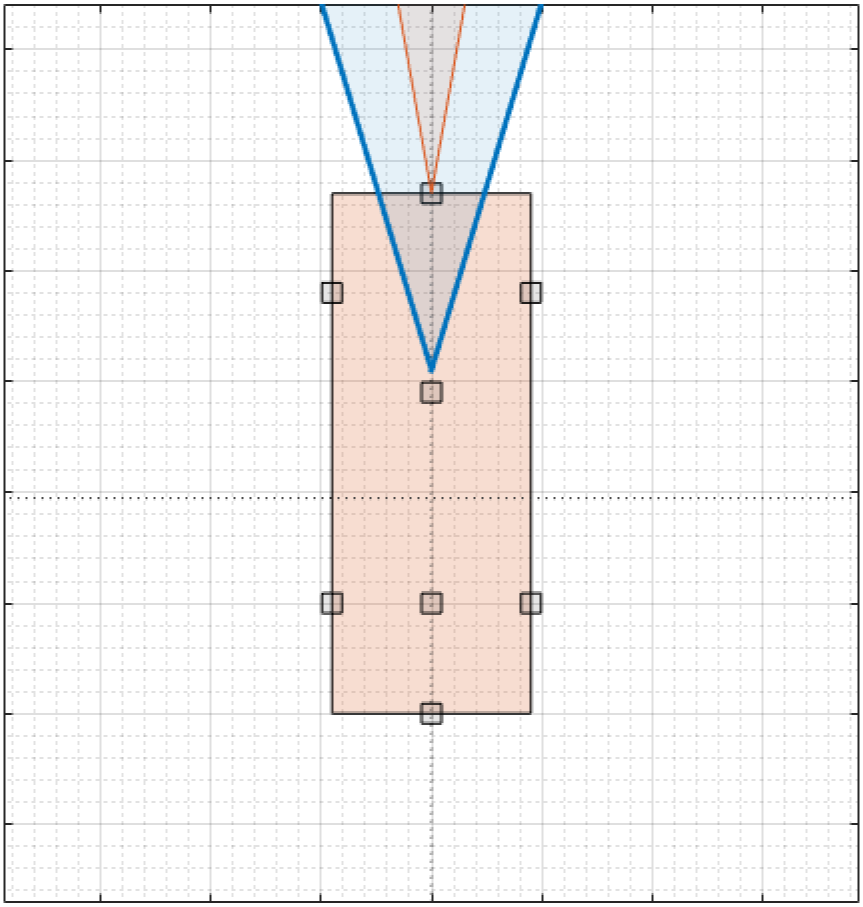
To simulate the scenario in the app, click **Run**. In the app simulation, unlike in the model simulation, the ego vehicle collides with the pedestrian. The app uses a predefined ego vehicle trajectory, whereas the model uses the AEB algorithm to control the trajectory and cause the ego vehicle to brake.



Load Sensors

The loaded scenario file contains only the roads and actors in the scenario. A separate file contains the sensors. To load these sensors into the scenario, on the app toolstrip, select **Open > Sensors**. Open the `AEBSensor.mat` file located in the example folder. Alternatively, from your MATLAB root folder, navigate to and open this file: *`matlabroot`*`/examples/driving/AEBSensors.mat`.

A radar sensor is mounted to the front bumper of the ego vehicle. A camera sensor is mounted to the front window of the ego vehicle.



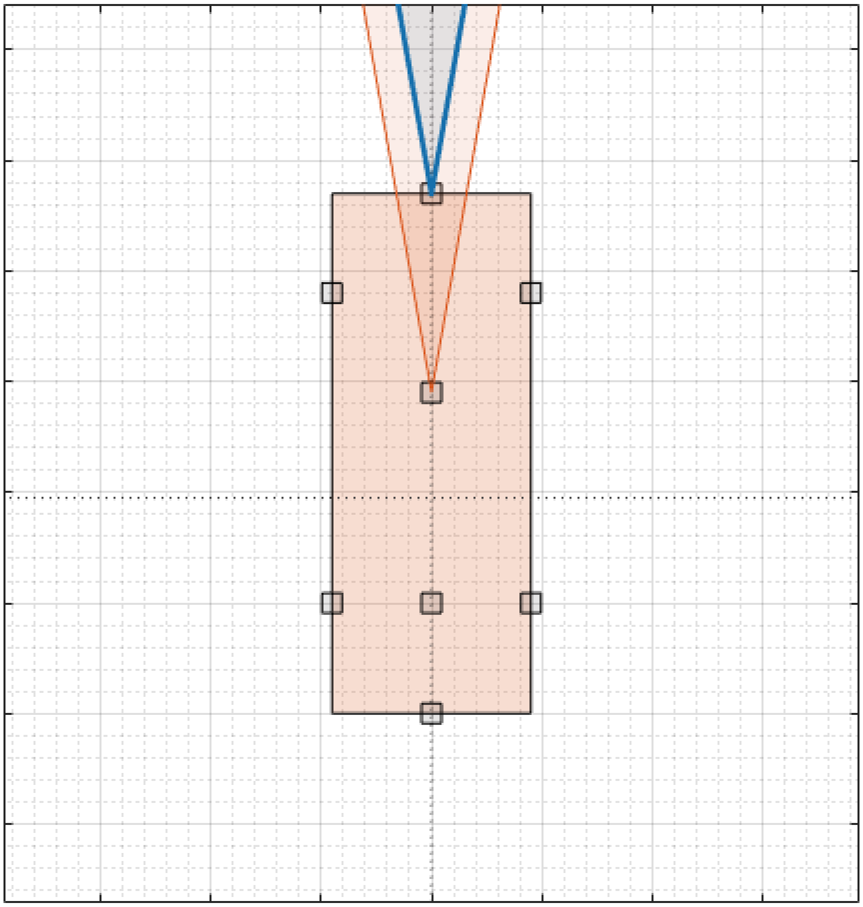
Update Sensors

Update the radar and camera sensors by changing their locations on the ego vehicles.

1. On the **Sensor Canvas**, click and drag the radar sensor to the predefined `Front Window` location.
2. Click and drag the camera sensor to the predefined `Front Bumper` location. At this predefined location, the app updates the camera from a short-range sensor to a long-range sensor.
3. Optionally, in the left pane, on the **Sensors** tab, try modifying the parameters of the camera and radar sensors. For example, you can change the detection probability or the accuracy and noise settings.
4. Save a copy of this new scenario and sensor configuration to a writeable location.

For more details on working with sensors in the app, see “Build a Driving Scenario and Generate Synthetic Detections” on page 5-2.

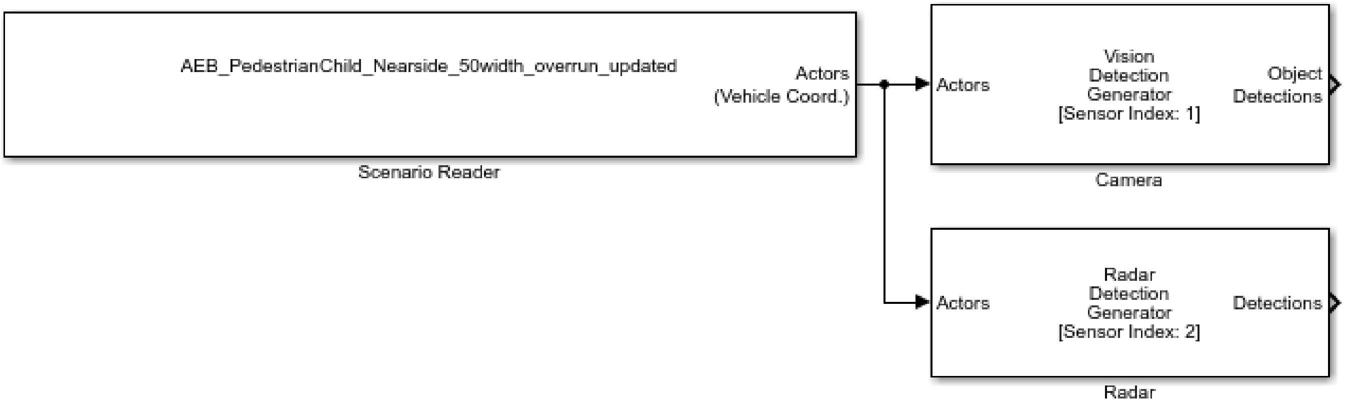
This image shows a sample updated sensor configuration.



Export Scenario and Sensors to Simulink

To generate Simulink blocks for the scenario and its sensors, on the app toolstrip, select **Export > Export Simulink Model**. This model shows sample blocks that were exported from the app.

```
open_system('AEBGeneratedScenarioAndSensors')
```



If you made no changes to the roads and actors in the scenario, then the Scenario Reader block reads the same road and actor data that was used in the AEB model. The Radar Detection Generator and Vision Detection Generator blocks model the radar and camera that you created in the app.

Copy Exported Scenario and Sensors into Existing Model

Replace the scenario and sensors in the AEB model with the newly generated scenario and sensors. Even if you did not modify the roads and actors and read data from the same scenario file, replacing the existing Scenario Reader block is still a best practice. Using this generated block keeps the bus names for scenario and sensors consistent as data passes between them.

To get started, in the AEB model, reopen the **Vehicle Environment > Actors and Sensor Simulation** subsystem.

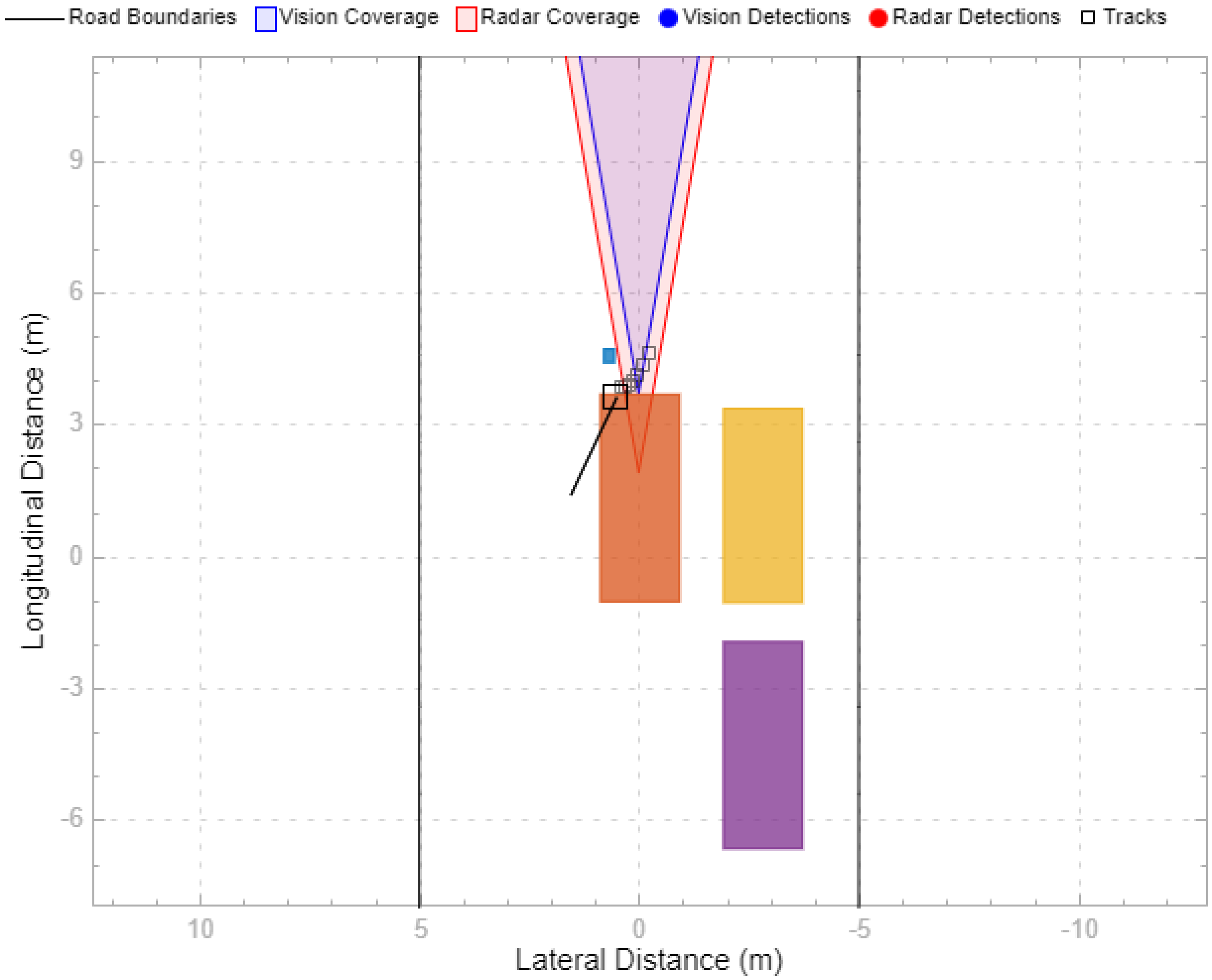
```
open_system('AEBTestBenchExample/Vehicle and Environment/Actors and Sensor Simulation')
```

Next, to cope the scenario and sensor blocks with the generated ones, follow these steps:

1. Delete the existing Scenario Reader, Radar Detection Generator, and Vision Detection Generator blocks. Do not delete the signal lines that are input to the Scenario Reader block or output from the sensor blocks. Alternatively, disconnect these blocks without deleting them, and comment them out of the model. Using this option, you can compare the existing blocks to the new one and revert back if needed. Select each block. Then, on the **Block** tab, select **Comment Out**.
2. Copy the blocks from the generated model into the AEB model.
3. Open the copied-in Scenario Reader block and set the **Source of ego vehicle** parameter to `Input port`. Click **OK**. The AEB model defines the ego vehicle in the Pack Ego Actor block, which you connect to the **Ego Vehicle** port of the Scenario Reader block.
4. Connect the existing signal lines to the copied-in blocks. To clean up the layout of the model, on the **Format** tab of the model, select **Auto Arrange**.
5. Verify that the updated subsystem block diagram resembles the pre-existing block diagram. Then, save the model, or save a copy of the model to a writeable location.

Simulate Updated Model

To visualize the updated scenario simulation, reopen the Bird's-Eye Scope, click **Find Signals**, and then click **Run**. With this updated sensor configuration, the ego vehicle does not brake in time.



To try different sensor configurations, reload the scenario and sensors in the app, export new scenarios and sensors, and copy them into the AEB model.

When you are done simulating the model, remove the example file folder from the MATLAB search path.

```
rmpath(genpath(fullfile(matlabroot,'examples','driving')))
```

See Also

Apps
Bird's-Eye Scope | Driving Scenario Designer

Blocks
Radar Detection Generator | Scenario Reader | Vision Detection Generator

More About

- “Build a Driving Scenario and Generate Synthetic Detections” on page 5-2
- “Euro NCAP Driving Scenarios in Driving Scenario Designer” on page 5-36

- “Test Open-Loop ADAS Algorithm Using Driving Scenario” on page 5-94
- “Test Closed-Loop ADAS Algorithm Using Driving Scenario” on page 5-100
- “Autonomous Emergency Braking with Sensor Fusion”

Test Open-Loop ADAS Algorithm Using Driving Scenario

This example shows how to test an open-loop ADAS (advanced driver assistance system) algorithm in Simulink®. In an open-loop ADAS algorithm, the ego vehicle behavior is predefined and does not change as the scenario advances during simulation.

To test the scenario, you use a driving scenario that was saved from the Driving Scenario Designer app. In this example, you read in a scenario using a Scenario Reader block, and then visually verify the performance of sensor algorithms on the Bird's-Eye Scope.

Before beginning this example, add the example file folder to the MATLAB® search path.

```
addpath(genpath(fullfile(matlabroot,'examples','driving')))
```

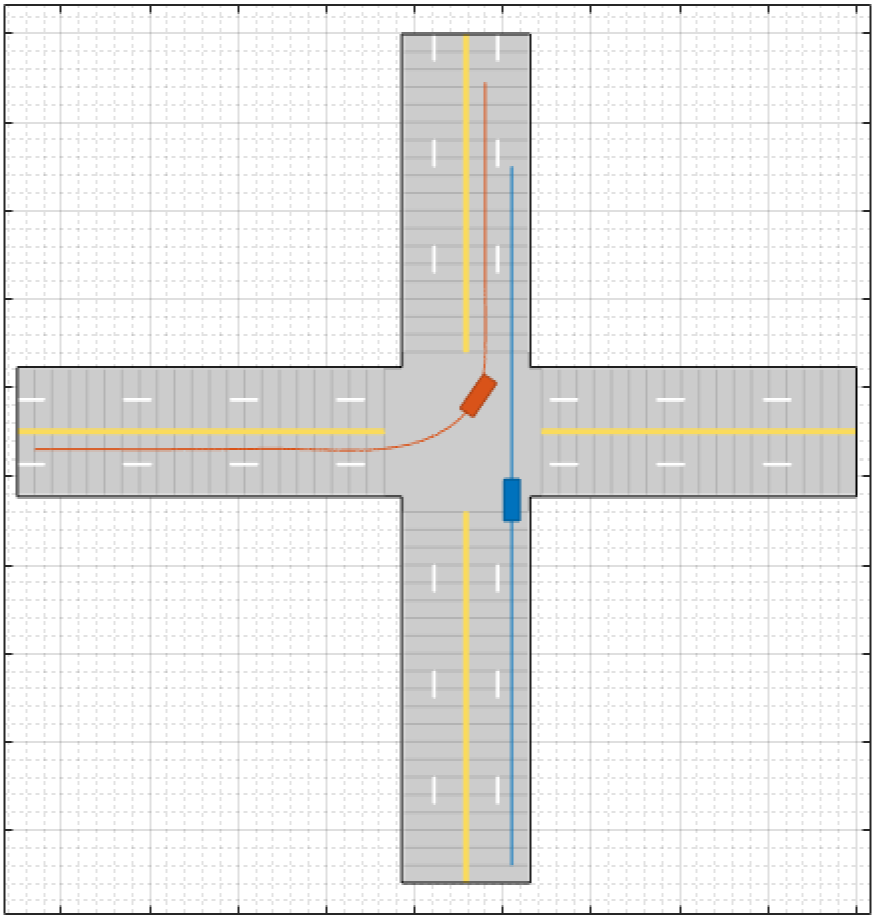
Inspect Driving Scenario

This example uses a driving scenario that is based on one of the prebuilt scenarios that you can access through the Driving Scenario Designer app. For more details on these scenarios, see “Prebuilt Driving Scenarios in Driving Scenario Designer” on page 5-14.

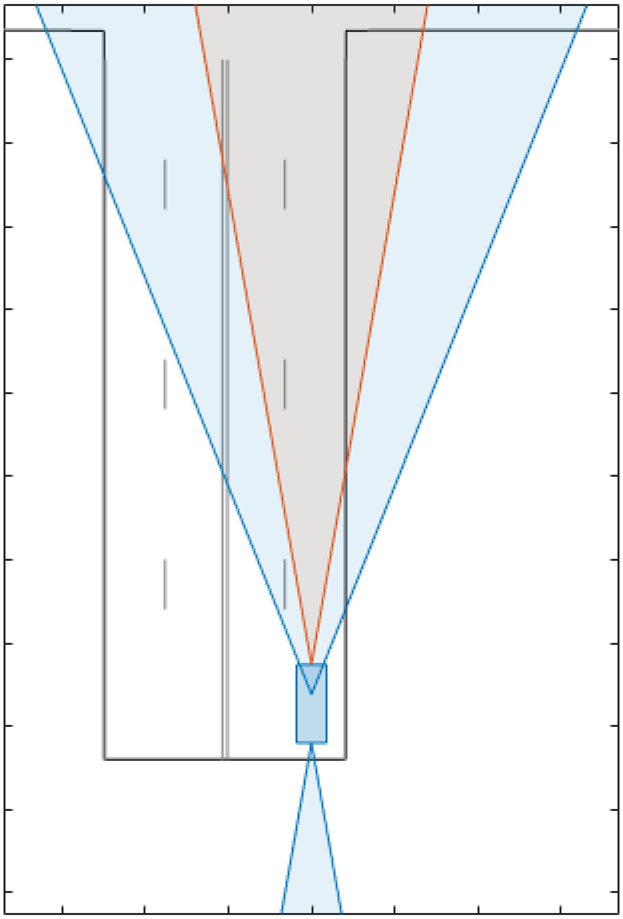
Open the scenario file in the app.

```
drivingScenarioDesigner('LeftTurnScenario.mat')
```

Click **Run** to simulate the scenario. In this scenario, the ego vehicle travels north and goes straight through an intersection. Meanwhile, a vehicle coming from the left side of the intersection turns left and ends up in front of the ego vehicle.



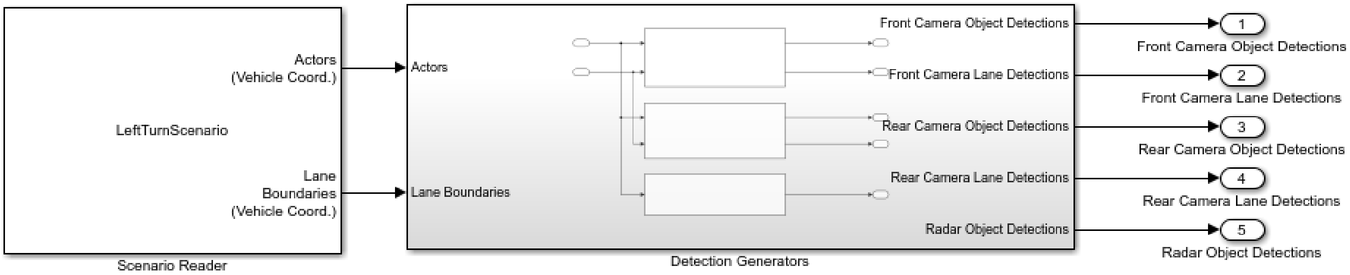
The ego vehicle also includes a front-facing radar for generating object detections and front-facing and rear-facing cameras for generating object and lane boundary detections.



Inspect Model

The model used in this example was generated from the app by selecting **Export > Export Simulink Model**. In the model, a Scenario Reader block reads the actors and roads from the scenario file and outputs the non-ego actors and lane boundaries. Open the model.

```
open_system('OpenLoopWithScenarios.slx')
```



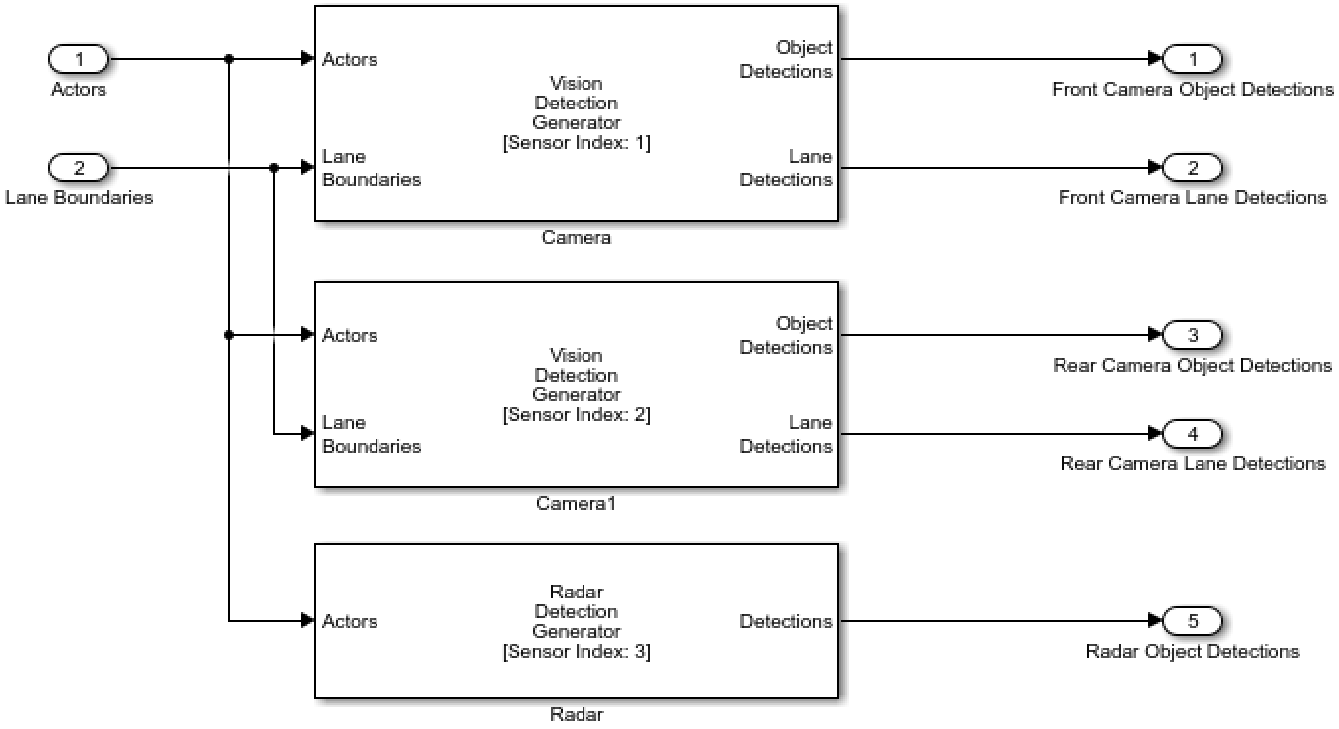
Actors
(Vehicle Coord.)
LeftTurnScenario
Lane
Boundaries
(Vehicle Coord.)
Scenario Reader
Actors
Lane Boundaries
Front Camera Object Detections
Front Camera Lane Detections
Rear Camera Object Detections
Rear Camera Lane Detections
Radar Object Detections
Detection Generators
1
Front Camera Object Detections
2
Front Camera Lane Detections
3
Rear Camera Object Detections
4
Rear Camera Lane Detections
5
Radar Object Detections

In the Scenario Reader block, the **Driving Scenario Designer file name** parameter specifies the name of the scenario file. You can specify a scenario file that is on the MATLAB search path, such as the scenario file used in this example, or the full path to a scenario file. Alternatively, you can specify a `drivingScenario` object by setting **Source of driving scenario** to `From workspace` and then setting **MATLAB or model workspace variable name** to the name of a valid `drivingScenario` object workspace variable.

The Scenario Reader block outputs the poses of the non-ego actors in the scenario and the left-lane and right-lane boundaries of the ego vehicle. To output all lane boundaries of the road on which the ego vehicle is traveling, select the corresponding option for the **Lane boundaries to output** parameter.

The actors and lane boundaries are passed to a subsystem containing the sensor blocks. Open the subsystem.

```
open_system('OpenLoopWithScenarios/Detection Generators')
```


1
Actors
2
Lane Boundaries
Actors
Lane Boundaries
Vision Detection Generator [Sensor Index: 1]
Object Detections
Lane Detections
Camera
1
Front Camera Object Detections
2
Front Camera Lane Detections
Actors
Lane Boundaries
Vision Detection Generator [Sensor Index: 2]
Object Detections
Lane Detections
Camera1
3
Rear Camera Object Detections
4
Rear Camera Lane Detections
Actors
Radar Detection Generator [Sensor Index: 3]
Detections
Radar
5
Radar Object Detections

The Radar Detection Generator block accepts the actors as input. The Vision Detection Generator block accepts the actors and lane boundaries as input. These sensor blocks produce synthetic detections from the scenario. The outputs are in vehicle coordinates, where:

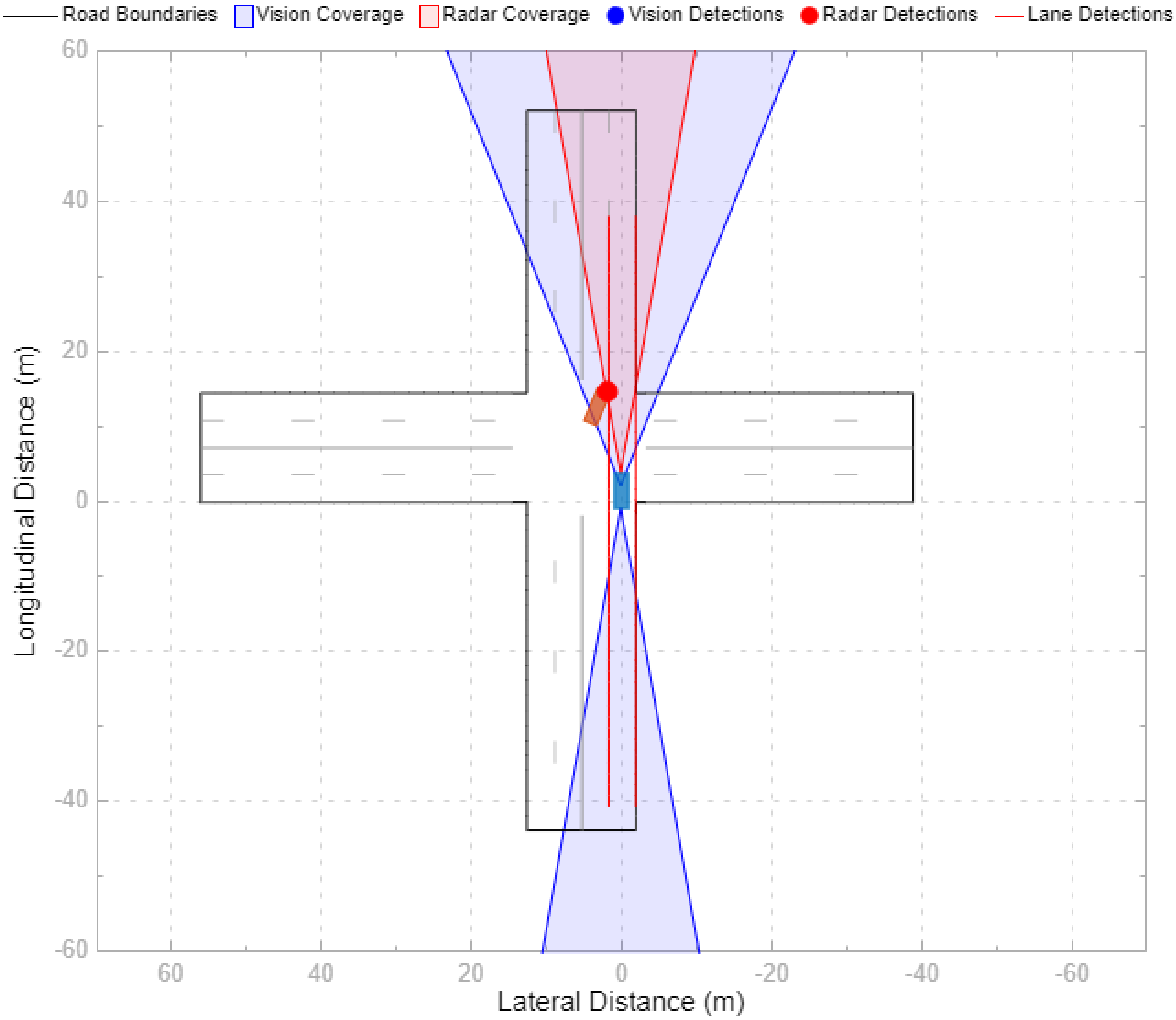
- The *X*-axis points forward from the ego vehicle.
- The *Y*-axis points to the left of the ego vehicle.
- The origin is located at the center of the rear axle of the ego vehicle.

If a scenario has multiple ego vehicles, in the Scenario Reader block, set the **Coordinate system of outputs** parameter to `World coordinates` instead of `Vehicle coordinates`. In the world coordinate system, the actors and lane boundaries are in the world coordinates of the driving scenario. When this parameter is set to `World coordinates`, however, visualization of the scenario using the Bird's-Eye Scope is not supported.

Because this model is open loop, the ego vehicle behavior does not change as the simulation advances. Therefore, the **Source of ego vehicle** parameter is set to `Scenario`, and the block reads the predefined ego vehicle pose and trajectory from the scenario file. For vehicle controllers and other closed-loop models, set the **Source of ego vehicle** parameter to `Input port`. With this option, you specify an ego vehicle that is defined in the model as an input to the Scenario Reader block. For an example, see "Test Closed-Loop ADAS Algorithm Using Driving Scenario" on page 5-100.

Visually Verify Algorithm

To visualize the scenario and the object and lane boundary detections, use the Bird's-Eye Scope. From the Simulink toolstrip, under **Review Results**, click **Bird's-Eye Scope**. Then, in the scope, click **Find Signals** and run the simulation. The sensors generate detections for the non-ego actor and lane boundaries.



Update Simulation Settings

This model uses the default simulation stop time of 10 seconds. However, because the scenario is only about 5 seconds long, the simulation continues to run in the Bird's-Eye Scope even after the scenario has ended. To synchronize the simulation and scenario stop times, in the Simulink model toolbar, set the simulation stop time to `5.2` seconds, which is the exact stop time of the app scenario. After you run the simulation, the app displays this value in the bottom-right corner of the scenario canvas.

If the simulation runs too fast in the Bird's-Eye Scope, you can slow down the simulation by using simulation pacing. From the Simulink toolstrip, select **Run > Simulation Pacing**. Select the **Enable pacing to slow down simulation** check box and decrease the simulation time to slightly less than 1 second per wall-clock second, such as 0.8 seconds. Then, rerun the simulation in the Bird's-Eye Scope.

When you are done with this example, remove the example file folder from the MATLAB search path.

```
rmpath(genpath(fullfile(matlabroot,'examples','driving')))
```

See Also

Apps
Bird's-Eye Scope | **Driving Scenario Designer**

Blocks
Radar Detection Generator | Scenario Reader | Vision Detection Generator

More About

- “Sensor Fusion Using Synthetic Radar and Vision Data in Simulink”
- “Test Closed-Loop ADAS Algorithm Using Driving Scenario” on page 5-100
- “Create Driving Scenario Variations Programmatically” on page 5-80
- “Generate Sensor Detection Blocks Using Driving Scenario Designer” on page 5-85

Test Closed-Loop ADAS Algorithm Using Driving Scenario

This model shows how to test a closed-loop ADAS (advanced driver assistance system) algorithm in Simulink®. In a closed-loop ADAS algorithm, the ego vehicle is controlled by changes in its scenario environment as the simulation advances.

To test the scenario, you use a driving scenario that was saved from the Driving Scenario Designer app. In this model, you read in a scenario using a Scenario Reader block, and then visually verify the performance of the algorithm, an autonomous emergency braking (AEB) system, on the Bird's-Eye Scope.

Before beginning this example, add the example file folder to the MATLAB® search path.

```
addpath(genpath(fullfile(matlabroot,'examples','driving')))
```

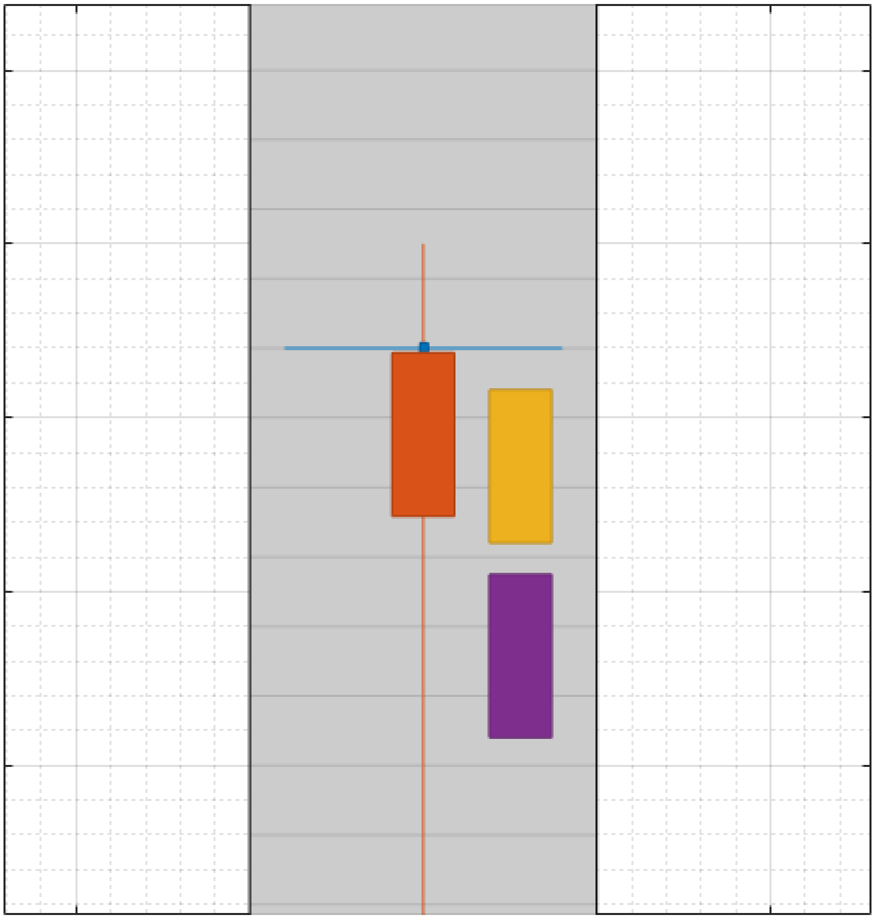
Inspect Driving Scenario

This example uses a driving scenario that is based on one of the prebuilt Euro NCAP test protocol scenarios that you can access through the Driving Scenario Designer app. For more details on these scenarios, see “Euro NCAP Driving Scenarios in Driving Scenario Designer” on page 5-36.

Open the scenario file in the app.

```
drivingScenarioDesigner('AEB_PedestrianChild_Nearside_50width_overrun.mat')
```

Click **Run** to simulate the scenario. In this scenario, the ego vehicle collides with a pedestrian child who is crossing the street.

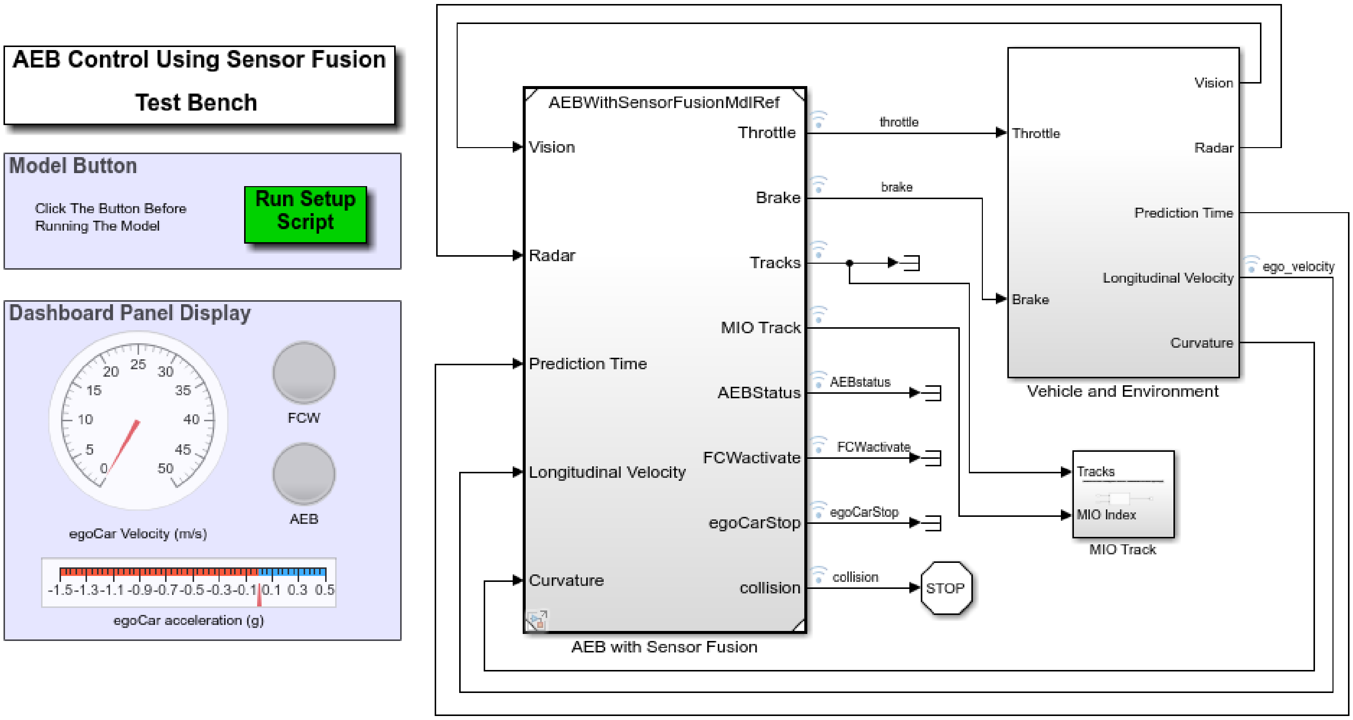


In the model used in this example, you use an AEB sensor fusion algorithm to detect the pedestrian child and test whether the ego vehicle brakes in time to avoid a collision.

Inspect Model

The model implements the AEB algorithm described in the “Autonomous Emergency Braking with Sensor Fusion” example. Open the model.

```
open_system('AEBTestBenchExample')
```

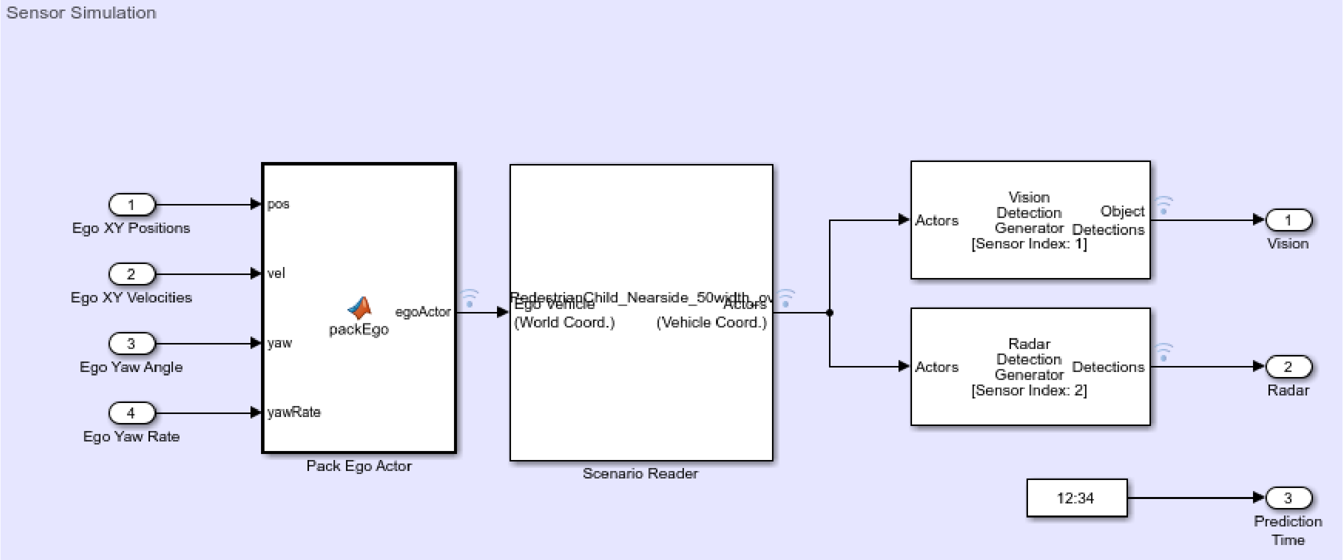


A Scenario Reader block reads the non-ego actors and roads from the specified scenario file and outputs the non-ego actors. The ego vehicle is passed into the block through an input port.

The Scenario Reader block is located in the **Vehicle Environment > Actors and Sensor Simulation** subsystem. Open this subsystem.

```
open_system('AEBTestBenchExample/Vehicle and Environment/Actors and Sensor Simulation')
```

Actors and Sensor Simulation



In the Scenario Reader block, the **Driving Scenario Designer file name** parameter specifies the name of the scenario file. You can specify a scenario file that is on the MATLAB search path, such as the scenario file used in this example, or the full path to a scenario file. Alternatively, you can specify a `drivingScenario` object by setting **Source of driving scenario** to `From workspace` and then setting **MATLAB or model workspace variable name** to the name of a valid `drivingScenario` object workspace variable. In closed-loop simulations, specifying the `drivingScenario` object is useful because it enables you finer control over specifying the initial position of the ego vehicle in your model.

The Scenario Reader block outputs the poses of the non-ego actors in the scenario. These poses are passed to vision and radar sensors, whose detections are used to determine the behavior of the AEB controller.

The actor poses are output in vehicle coordinates, where:

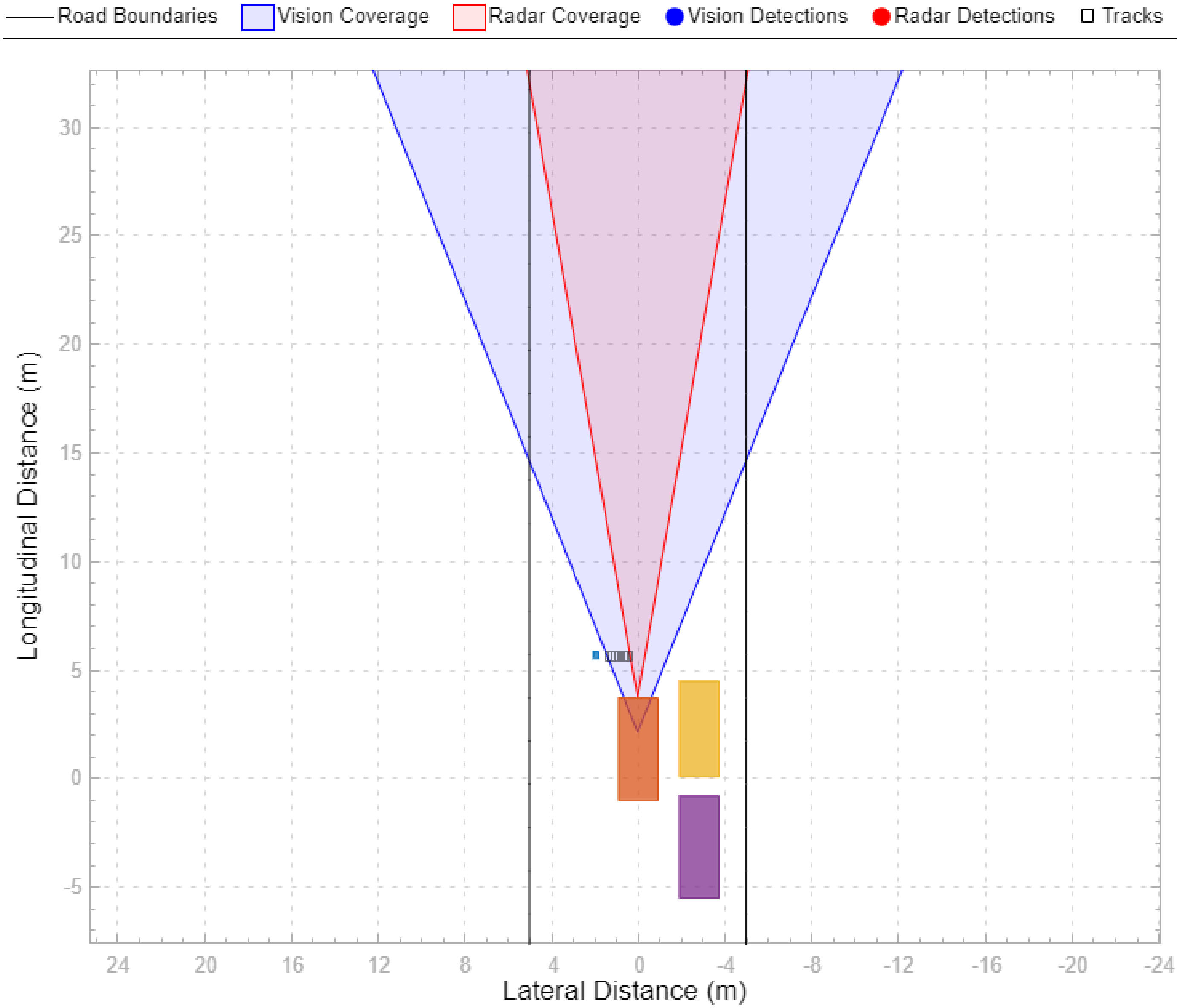
- The *X*-axis points forward from the ego vehicle.
- The *Y*-axis points to the left of the ego vehicle.
- The origin is located at the center of the rear axle of the ego vehicle.

If a scenario has multiple ego vehicles, in the Scenario Reader block, set the **Coordinate system of outputs** parameter to `World coordinates` instead of `Vehicle coordinates`. In the world coordinate system, the actors and lane boundaries are in the world coordinates of the driving scenario. When this parameter is set to `World coordinates`, however, visualization of the scenario using the Bird's-Eye Scope is not supported.

Although this scenario includes a predefined ego vehicle, the Scenario Reader block is configured to ignore this ego vehicle definition. Instead, the ego vehicle is defined in the model and specified as an input to the Scenario Reader block (the **Source of ego vehicle** parameter is set to `Input port`). As the simulation advances, the AEB algorithm determines the pose and trajectory of the ego vehicle. If you are developing an open-loop algorithm, where the ego vehicle is predefined in the driving scenario, set the **Source of ego vehicle** parameter to `Scenario`. For an example, see “Test Open-Loop ADAS Algorithm Using Driving Scenario” on page 5-94.

Visually Verify Algorithm

To visualize the scenario, use the Bird's-Eye Scope. From the Simulink toolstrip, under **Review Results**, click **Bird's-Eye Scope**. Then, in the scope, click **Find Signals** and run the simulation. With the AEB algorithm, the ego vehicle brakes in time to avoid a collision.



When you are done verifying the algorithm, remove the example file folder from the MATLAB search path.

```
rmpath(genpath(fullfile(matlabroot,'examples','driving')))
```

See Also

Apps
Bird's-Eye Scope | **Driving Scenario Designer**

Blocks
Radar Detection Generator | Scenario Reader | Vision Detection Generator

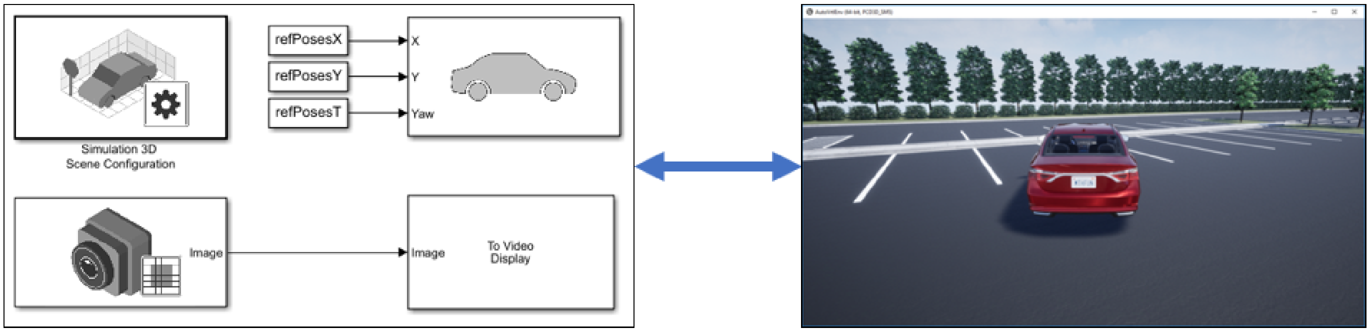
More About

- “Autonomous Emergency Braking with Sensor Fusion”
- “Lateral Control Tutorial”
- “Test Open-Loop ADAS Algorithm Using Driving Scenario” on page 5-94
- “Create Driving Scenario Variations Programmatically” on page 5-80
- “Generate Sensor Detection Blocks Using Driving Scenario Designer” on page 5-85

3D Simulation – User's Guide

3D Simulation for Automated Driving

Automated Driving Toolbox provides a co-simulation framework that models driving algorithms in Simulink and visualizes their performance in a 3D environment. This 3D simulation environment uses the Unreal Engine from Epic Games.



Simulink blocks related to the 3D simulation environment can be found in the **Automated Driving Toolbox** > **Simulation 3D** block library. These blocks provide the ability to:

- 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.

This simulation tool is commonly used to supplement real data when developing, testing, and verifying the performance of automated driving algorithms. In conjunction with a vehicle model, you can use these blocks to perform realistic closed-loop simulations that encompass the entire automated driving stack, from perception to control.

For more details on the simulation environment, see “How 3D Simulation for Automated Driving Works” on page 6-8.

3D Simulation Blocks

To access the **Automated Driving Toolbox** > **Simulation 3D** library, at the MATLAB command prompt, enter `drivingsim3d`.

Scenes

To configure a model to co-simulate with the 3D simulation environment, add a Simulation 3D Scene Configuration block to the model. Using this block, you can choose from a set of prebuilt 3D scenes where you can test and visualize your driving algorithms. The following image is from the Virtual Mcity scene.

The toolbox includes these scenes.

Scene	Description
Straight Road	Straight road segment
Curved Road	Curved, looped road
Parking Lot	Empty parking lot
Double Lane Change	Straight road with barrels and traffic signs that are set up for executing a double lane change maneuver
Open Surface	Flat, black pavement surface with no road objects
US City Block	City block with intersections, barriers, and traffic lights
US Highway	Highway with cones, barriers, traffic lights, and traffic signs
Large Parking Lot	Parking lot with parked cars, cones, curbs, and traffic signs
Virtual Mcity	City environment that represents the University of Michigan proving grounds (see Mcity Test Facility); includes cones, barriers, an animal, traffic lights, and traffic signs

If you have the Automated Driving Toolbox Interface for Unreal Engine 4 Projects support package, then you can modify these scenes or create new ones. For more details, see “Customize 3D Scenes for Automated Driving” on page 6-43.

Vehicles

To define a virtual vehicle in a scene, add a Simulation 3D Vehicle with Ground Following block to your model. Using this block, you can control the movement of the vehicle by supplying the X, Y, and yaw values that define its position and orientation at each time step. The vehicle automatically moves along the ground.

You can also specify the color and type of vehicle. The toolbox includes these vehicle types:

- **Box Truck**
- **Hatchback**
- **Muscle Car**
- **Sedan**
- **Small Pickup Truck**
- **Sport Utility Vehicle**

Sensors

You can define virtual sensors and attach them at various positions on the vehicles. The toolbox includes these sensor modeling and configuration blocks.

Block	Description
Simulation 3D Camera	Camera model with lens. Includes parameters for image size, focal length, distortion, and skew.
Simulation 3D Fisheye Camera	Fisheye camera that can be described using the Scaramuzza camera model. Includes parameters for distortion center, image size, and mapping coefficients.
Simulation 3D Lidar	Scanning lidar sensor model. Includes parameters for detection range, resolution, and fields of view.
Simulation 3D Probabilistic Radar	Probabilistic radar model that returns a list of detections. Includes parameters for radar accuracy, radar bias, detection probability, and detection reporting. It does not simulate radar at an electromagnetic wave propagation level.
Simulation 3D Probabilistic Radar Configuration	Configures radar signatures for all actors detected by the Simulation 3D Probabilistic Radar blocks in a model.

For more details on choosing a sensor, see “Choose a Sensor for 3D Simulation” on page 6-16.

Algorithm Testing and Visualization

Automated Driving Toolbox 3D simulation blocks provide the tools for testing and visualizing path planning, vehicle control, and perception algorithms.

Path Planning and Vehicle Control

You can use the 3D simulation environment to visualize the motion of a vehicle in a prebuilt scene. This environment provides you with a way to analyze the performance of path planning and vehicle control algorithms. After designing these algorithms in Simulink, you can use the `drivingsim3d` library to visualize vehicle motion in one of the prebuilt scenes.

For an example of path planning and vehicle control algorithm visualization, see “Visualize Automated Parking Valet Using 3D Simulation”.

Perception

Automated Driving Toolbox provides several blocks for detailed camera, radar, and lidar sensor modeling. By mounting these sensors on vehicles within the virtual environment, you can generate synthetic sensor data or sensor detections to test the performance of your sensor models against perception algorithms.

- For an example of building a lidar perception algorithm using synthetic sensor data from the 3D simulation environment, see “Simulate Lidar Sensor Perception Algorithm”.
- For an example of generating radar detections, see “Simulate Radar Sensors in 3D Environment”.

You can also output and visualize ground truth data to validate depth estimation algorithms and train semantic segmentation networks. For an example, see “Visualize Depth and Semantic Segmentation Data in 3D Environment” on page 6-30.

Closed-Loop Systems

After you design and test a perception system within the 3D simulation environment, you can then use it to drive a control system that actually steers a vehicle. In this case, rather than manually set up a trajectory, the vehicle uses the perception system to drive itself. By combining perception and control into a closed-loop system in the 3D simulation environment, you can develop and test more complex algorithms, such as lane keeping assist and adaptive cruise control.

For an example of a closed-loop system in the 3D environment, see “Highway Lane Following”.

See Also

More About

- “3D Simulation Environment Requirements and Limitations” on page 6-6
- “Simulate a Simple Driving Scenario and Sensor in 3D Environment” on page 6-21
- “Coordinate Systems for 3D Simulation in Automated Driving Toolbox” on page 6-10
- “Customize 3D Scenes for Automated Driving” on page 6-43

3D Simulation Environment Requirements and Limitations

Automated Driving Toolbox provides an interface to a 3D simulation environment that is visualized using the Unreal Engine from Epic Games. Version 4.23 of this visualization engine comes installed with Automated Driving Toolbox. When simulating in the 3D environment, keep these requirements and limitations in mind.

Software Requirements

- Windows® 64-bit platform
- Visual Studio® 2017 or newer (for customizing scenes)
- Microsoft® DirectX® — If this software is not already installed on your machine and you try to simulate in the 3D environment, Automated Driving Toolbox prompts you to install it. Once you install the software, you must restart the simulation.

In you are customizing scenes, verify that your Unreal Engine project is compatible with the Unreal Engine version supported by your MATLAB release.

MATLAB Release	Unreal Engine Version
R2019b	4.19
R2020a	4.23

Note Mac and Linux® platforms are not supported.

Minimum Hardware Requirements

- Graphics card (GPU) — Virtual reality-ready with 8 GB of on-board RAM
- Processor (CPU) — 2.60 GHz
- Memory (RAM) — 12 GB

Limitations

The 3D simulation environment blocks do not support:

- Code generation
- Model reference
- Multiple instances of the Simulation 3D Scene Configuration block
- Multiple instances of the 3D simulation environment
- Parallel simulations
- Rapid accelerator mode

In addition, when using these blocks in a closed-loop simulation, all 3D simulation environment blocks must be in the same subsystem.

See Also

Simulation 3D Scene Configuration

More About

- “3D Simulation for Automated Driving” on page 6-2
- “How 3D Simulation for Automated Driving Works” on page 6-8

External Websites

- Unreal Engine

How 3D Simulation for Automated Driving Works

Automated Driving Toolbox provides a co-simulation framework that you can use to model driving algorithms in Simulink and visualize their performance in a 3D environment. This 3D simulation environment uses the Unreal Engine by Epic Games.

Understanding how this simulation environment works can help you troubleshoot issues and customize your models.

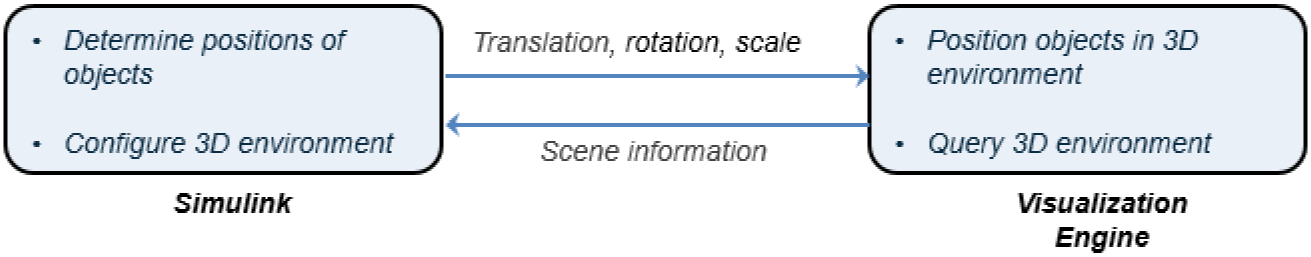
Communication with 3D Simulation Environment

When you use Automated Driving Toolbox to run your algorithms, Simulink co-simulates the algorithms in the visualization engine.

In the Simulink environment, Automated Driving Toolbox:

- Configures the 3D visualization environment, specifically the ray tracing, scene capture from cameras, and initial object positions
- Determines the next position of the objects by using the 3D simulation environment feedback

The diagram summarizes the communication between Simulink and the visualization engine.


• Determine positions of objects
• Configure 3D environment
Simulink
Translation, rotation, scale
Scene information
• Position objects in 3D environment
• Query 3D environment
Visualization Engine

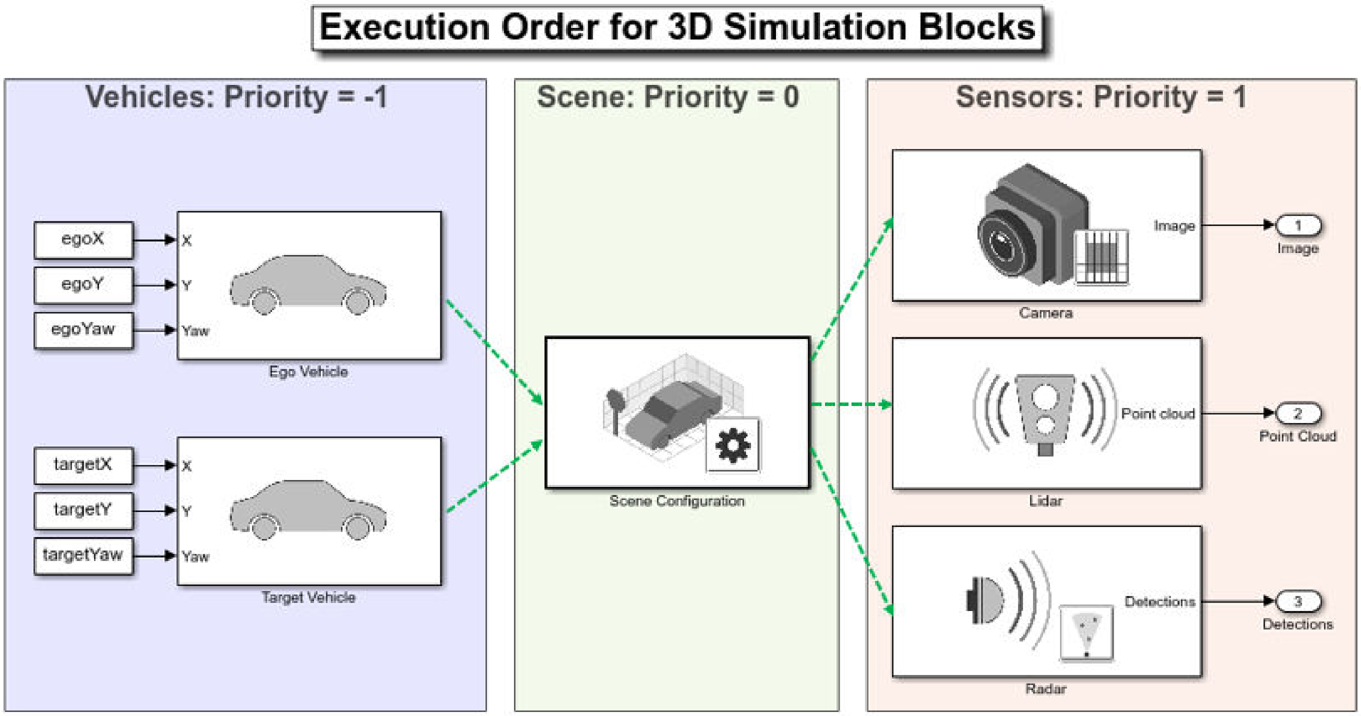
Block Execution Order

During simulation, the 3D simulation blocks follow a specific execution order:

1. The Simulation 3D Vehicle with Ground Following blocks initialize the vehicles and send their **X**, **Y**, and **Yaw** signal data to the Simulation 3D Scene Configuration block.
2. The Simulation 3D Scene Configuration block receives the vehicle data and sends it to the sensor blocks.
3. The sensor blocks receive the vehicle data and use it to accurately locate and visualize the vehicles.

The **Priority** property of the blocks controls this execution order. To access this property for any block, right-click the block, select **Properties**, and click the **General** tab. By default, Simulation 3D Vehicle with Ground Following blocks have a priority of `-1`, Simulation 3D Scene Configuration blocks have a priority of `0`, and sensor blocks have a priority of `1`.

The diagram shows this execution order.



If your sensors are not detecting vehicles in the scene, it is possible that the 3D simulation blocks are executing out of order. Try updating the execution order and simulating again. For more details on execution order, see “Control and Display Execution Order” (Simulink).

Also be sure that all 3D simulation blocks are located in the same subsystem. Even if the blocks have the correct **Priority** settings, if they are located in different subsystems, they still might execute out of order.

See Also

More About

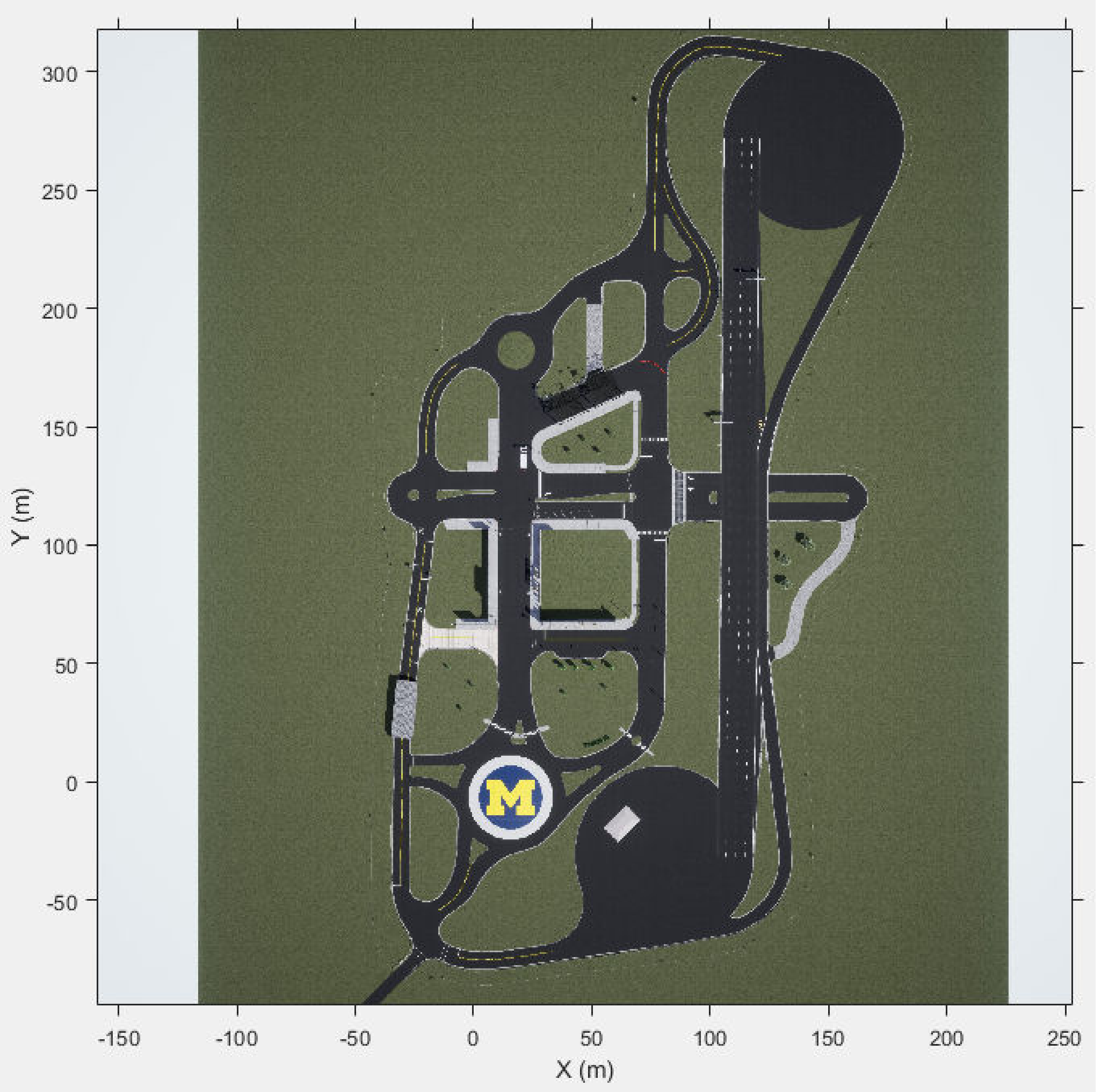
- “3D Simulation for Automated Driving” on page 6-2
- “3D Simulation Environment Requirements and Limitations” on page 6-6
- “Choose a Sensor for 3D Simulation” on page 6-16
- “Coordinate Systems for 3D Simulation in Automated Driving Toolbox” on page 6-10

Coordinate Systems for 3D Simulation in Automated Driving Toolbox

Automated Driving Toolbox enables you to simulate your driving algorithms in a 3D environment that uses the Unreal Engine from Epic Games. In general, the coordinate systems used in this environment follow the conventions described in “Coordinate Systems in Automated Driving Toolbox” on page 1-2. However, when simulating in this environment, it is important to be aware of the specific differences and implementation details of the 3D simulation coordinate systems.

World Coordinate System

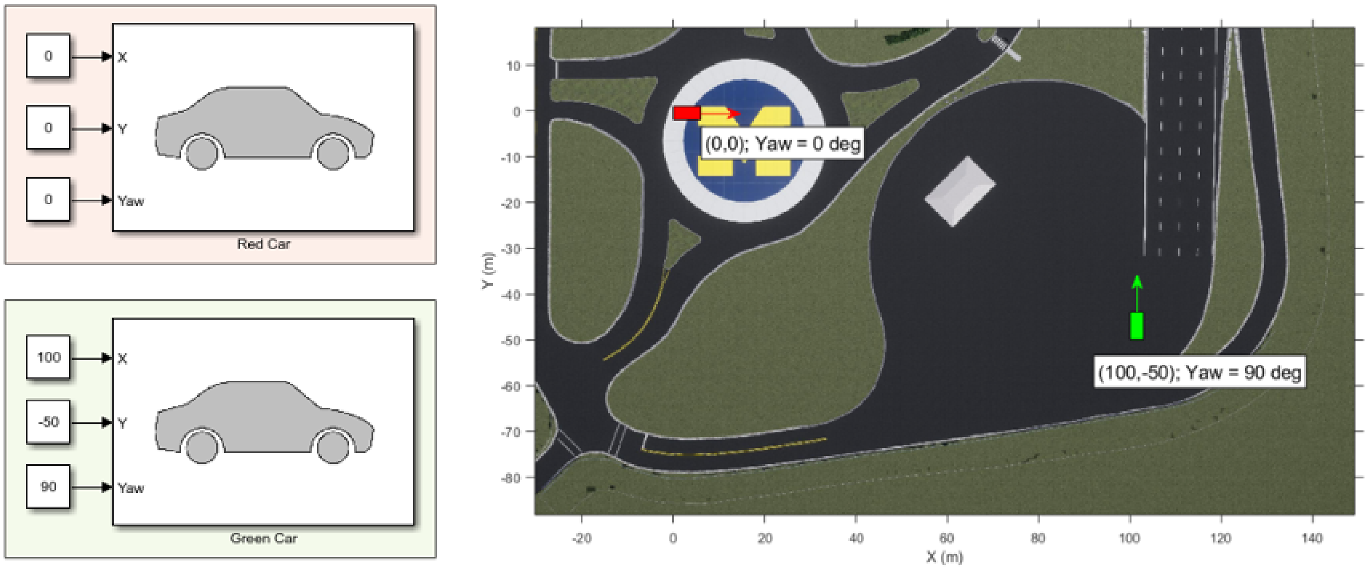
As with other Automated Driving Toolbox functionality, the 3D simulation environment uses the right-handed Cartesian world coordinate system defined in ISO 8855. The following 2D top-view image of the **Virtual Mcity** scene shows the *X*- and *Y*-coordinates of the scene.



In this coordinate system, when looking in the positive direction of the *X*-axis, the positive *Y*-axis points left. The positive *Z*-axis points from the ground up. The yaw, pitch, and roll angles are clockwise-positive, when looking in the positive directions of the *Z*-, *Y*-, and *X*-axes, respectively. If you view a scene from a 2D top-down perspective, then the yaw angle is counterclockwise-positive, because you are viewing the scene in the negative direction of the *Z*-axis.

Placing Vehicles in a Scene

Vehicles are placed in the world coordinate system of the scenes. The figure shows how specifying the **X**, **Y**, and **Yaw** ports in the Simulation 3D Vehicle with Ground Following blocks determines their placement in a scene.



The elevation and banking angle of the ground determine the *Z*-axis, roll angle, and pitch angle of the vehicles.

Difference from Unreal Editor World Coordinates

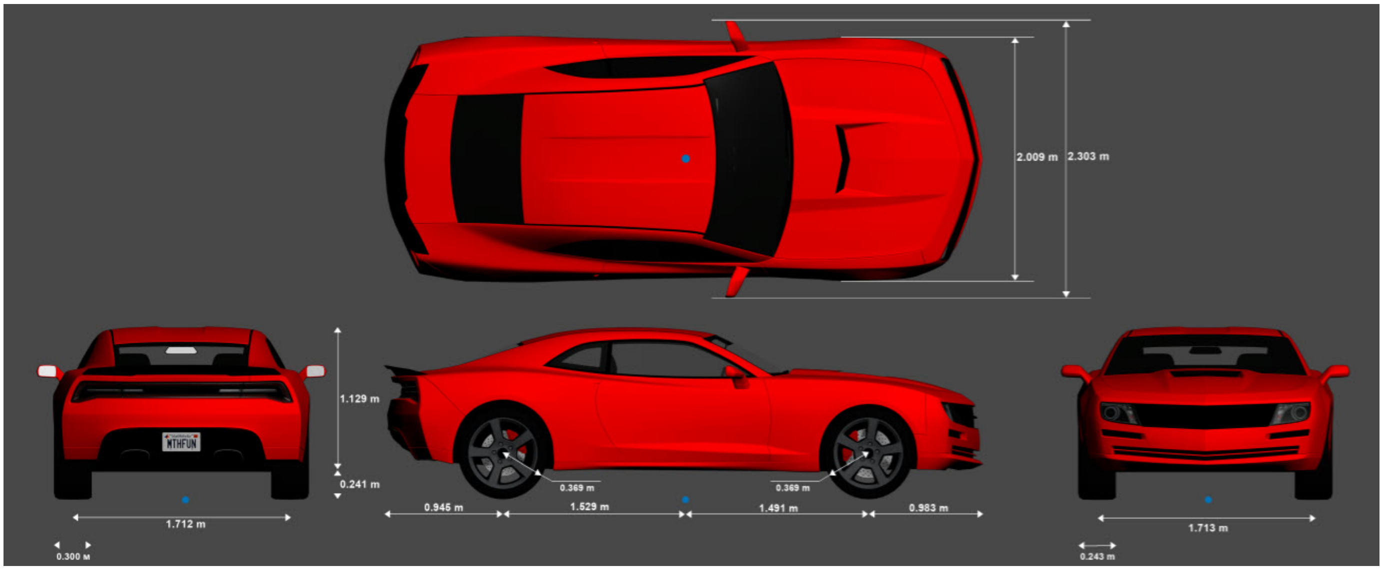
The Unreal® Editor uses a left-handed world Cartesian coordinate system in which the positive *Y*-axis points right. If you are converting from the Unreal Editor coordinate system to the coordinate system of the 3D environment, you must flip the sign of the *Y*-axis and pitch angle. The *X*-axis, *Z*-axis, roll angle, and yaw angle are the same in both coordinate systems.

Vehicle Coordinate System

The vehicle coordinate system is based on the world coordinate system. In this coordinate system:

- The *X*-axis points forward from the vehicle.
- The *Y*-axis points to the left of the vehicle.
- The *Z*-axis points up from the ground.
- Roll, pitch, and yaw are clockwise-positive when looking in the forward direction of the *X*-, *Y*-, and *Z*-axes, respectively. As with the world coordinate system, when looking at a vehicle from the top down, then the yaw angle is counterclockwise-positive.

The vehicle origin is on the ground, at the geometric center of the vehicle. In this figure, the blue dot represents the vehicle origin.



Mounting Sensors on a Vehicle

When you add a sensor block, such as a Simulation 3D Camera block, to your model, you can mount the sensor to a predefined vehicle location, such as the front bumper of the root center. These mounting locations are in the vehicle coordinate system. When you specify an offset from these locations, you offset from the origin of the mounting location, not from the vehicle origin.

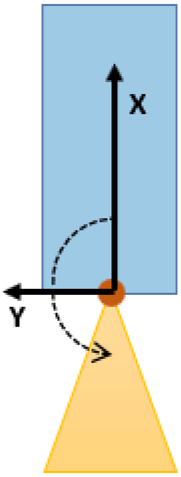
These equations define the vehicle coordinates for a sensor with location (X, Y, Z) and orientation ($Roll$, $Pitch$, Yaw):

- $(X, Y, Z) = (X_{\text{mount}} + X_{\text{offset}}, Y_{\text{mount}} + Y_{\text{offset}}, Z_{\text{mount}} + Z_{\text{offset}})$
- $(Roll, Pitch, Yaw) = (Roll_{\text{mount}} + Roll_{\text{offset}}, Pitch_{\text{mount}} + Pitch_{\text{offset}}, Yaw_{\text{mount}} + Yaw_{\text{offset}})$

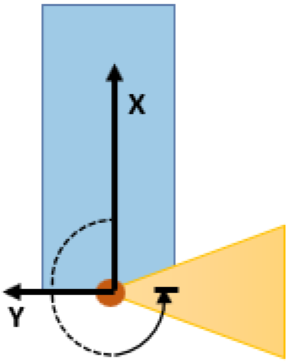
The "mount" variables refer to the predefined mounting locations relative to the vehicle origin. You define these mounting locations in the **Mounting location** parameter of the sensor block.

The "offset" variables refer to the amount of offset from these mounting locations. You define these offsets in the **Relative translation [X, Y, Z] (m)** and **Relative rotation [Roll, Pitch, Yaw] (deg)** parameters of the sensor block.

For example, consider a sensor mounted to the `Rear bumper` location. Relative to the vehicle origin, the sensor has an orientation of (0, 0, 180). In other words, when looking at the vehicle from the top down, the yaw angle of the sensor is rotated counterclockwise 180 degrees.

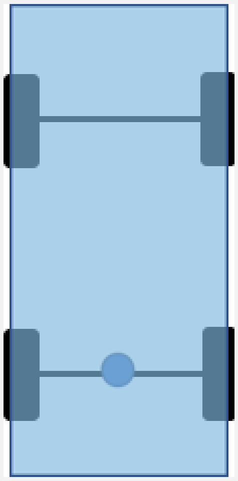
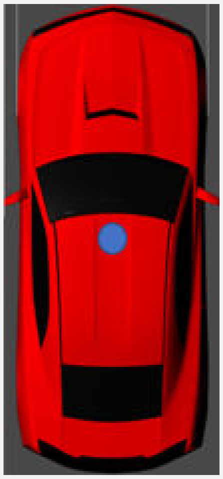


To point the sensor 90 degrees further to the right, you need to set the **Relative rotation [Roll, Pitch, Yaw] (deg)** parameter to `[0,0,90]`. In other words, the sensor is rotated 270 degrees counterclockwise relative to the vehicle origin, but it is rotated only 90 degrees counterclockwise relative to the origin of the predefined rear bumper location.



Difference from Cuboid Vehicle Origin

In the cuboid simulation environment, as described in “Cuboid Driving Scenario Simulation”, the origin is on the ground, below the center of the rear axle of the vehicle.

Cuboid Vehicle Origin	3D Simulation Vehicle Origin
	

If you are converting sensor positions between coordinate systems, then you need to account for this difference in origin by using a Cuboid To 3D Simulation block. For an example model that uses this block, see “Highway Lane Following”.

Difference from Unreal Editor Vehicle Coordinates

The Unreal Editor uses a left-handed Cartesian vehicle coordinate system in which the positive *Y*-axis points right. If you are converting from the Unreal Editor coordinate system to the coordinate system of the 3D environment, you must flip the sign of the *Y*-axis and pitch angle. The *X*-axis, *Z*-axis, roll angle, and yaw angle are the same in both coordinate systems.

See Also

Cuboid To 3D Simulation | Simulation 3D Vehicle with Ground Following

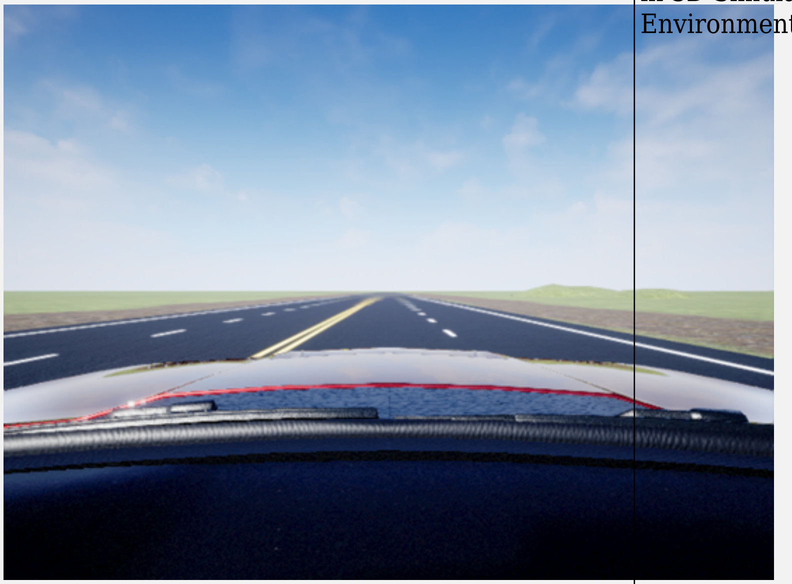
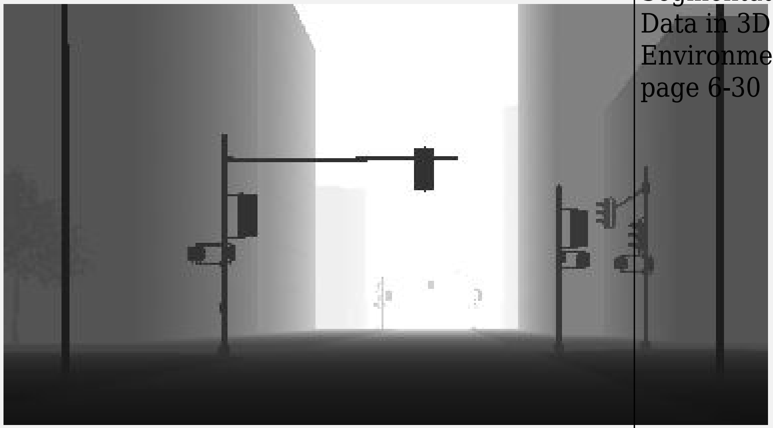
More About

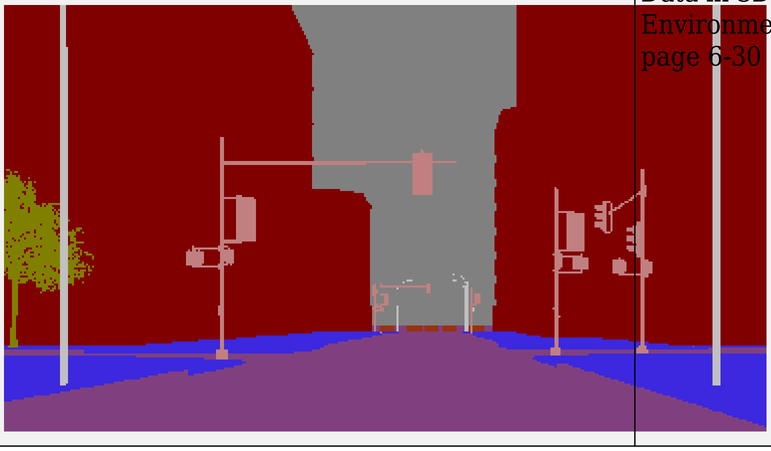
- “How 3D Simulation for Automated Driving Works” on page 6-8
- “Coordinate Systems in Automated Driving Toolbox” on page 1-2
- “Coordinate Systems in Vehicle Dynamics Blockset” (Vehicle Dynamics Blockset)

Choose a Sensor for 3D Simulation

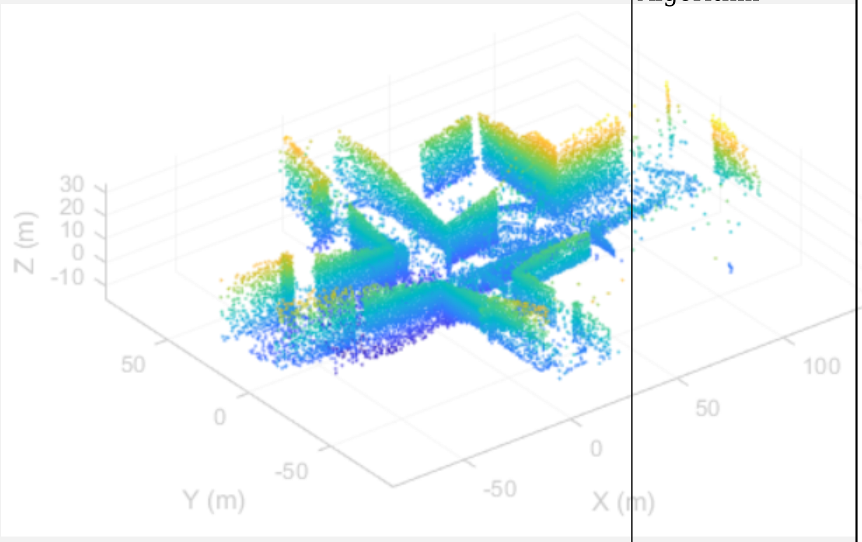
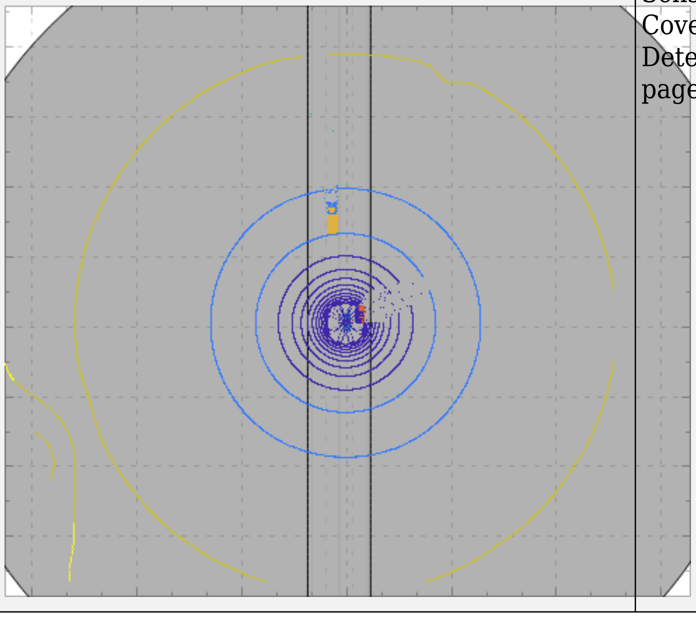
You can use the 3D simulation environment in Automated Driving Toolbox to obtain high-fidelity sensor data. This environment is rendered using the Unreal Engine from Epic Games.

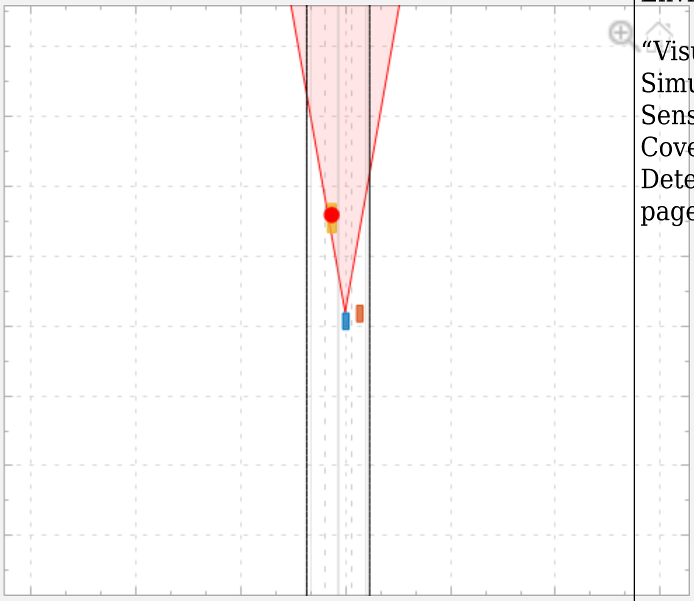
The table summarizes the sensor blocks that you can simulate in this environment.

Sensor Block	Description	Visualization	Example
Simulation 3D Camera	• Camera with lens that is based on the ideal pinhole camera. See “What Is Camera Calibration?” (Computer Vision Toolbox) • Includes parameters for image size, focal length, distortion, and skew • Includes options to output ground truth for depth estimation and	Display camera images by using a Video Viewer or To Video Display block. Sample visualization: 	“Design of Lane Marker Detector in 3D Simulation Environment”
		Display depth maps by using a Video Viewer or To Video Display block. Sample visualization: 	“Visualize Depth and Semantic Segmentation Data in 3D Environment” on page 6-30

Sensor Block	Description	Visualization	Example
	semantic segmentation	Display semantic segmentation maps by using a Video Viewer or To Video Display block. Sample visualization: 	“Visualize Depth and Semantic Segmentation Data in 3D Environment” on page 6-30

Sensor Block	Description	Visualization	Example
Simulation 3D Fisheye Camera	• Fisheye camera that can be described using the Scaramuzza camera model. See "Fisheye Calibration Basics" (Computer Vision Toolbox) • Includes parameters for distortion center, image size, and mapping coefficients	Display camera images by using a Video Viewer or To Video Display block. Sample visualization: 	"Simulate a Simple Driving Scenario and Sensor in 3D Environment" on page 6-21

Sensor Block	Description	Visualization	Example
Simulation 3D Lidar	- Scanning lidar sensor model - Includes parameters for detection range, resolution, and fields of view	Display point cloud data by using `pcplayer` within a MATLAB Function block. Sample visualization:  	"Simulate Lidar Sensor Perception Algorithm"
		Display lidar coverage areas and detections by using the **Bird's-Eye Scope**. Sample visualization: 	"Visualize 3D Simulation Sensor Coverages and Detections" on page 6-35

Sensor Block	Description	Visualization	Example
Simulation 3D Probabilistic Radar	- Probabilistic radar model that returns a list of detections - Includes parameters for radar accuracy, radar bias, detection probability, and detection reporting	Display radar coverage areas and detections by using the **Bird's-Eye Scope**. Sample visualization: 	“Simulate Radar Sensors in 3D Environment” “Visualize 3D Simulation Sensor Coverages and Detections” on page 6-35

See Also

Blocks
Simulation 3D Probabilistic Radar Configuration | Simulation 3D Scene Configuration | Simulation 3D Vehicle with Ground Following

More About

- “3D Simulation for Automated Driving” on page 6-2

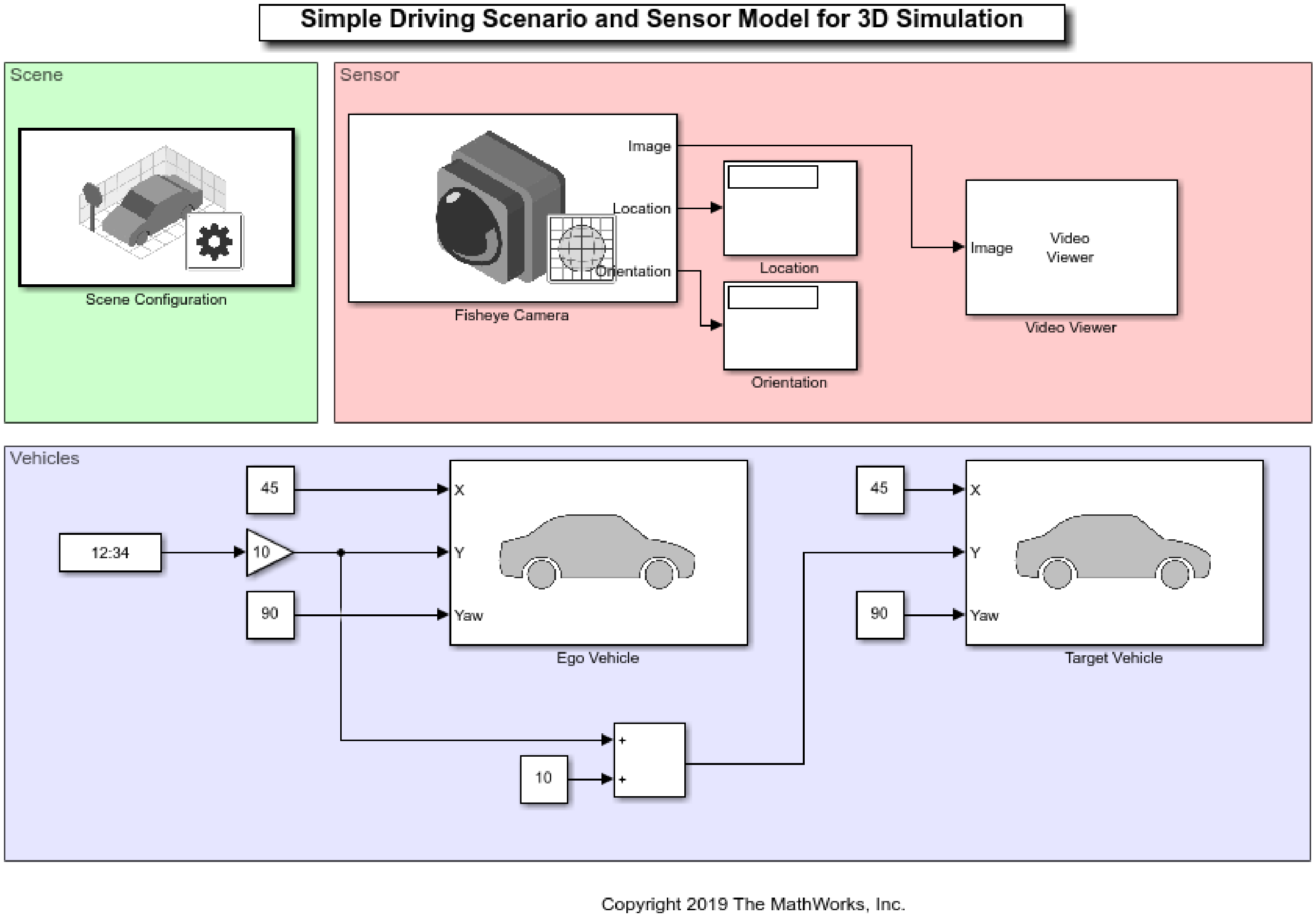
Simulate a Simple Driving Scenario and Sensor in 3D Environment

Automated Driving Toolbox™ provides blocks for visualizing sensors in a 3D simulation environment that uses the Unreal Engine® from Epic Games®. This model simulates a simple driving scenario in a prebuilt 3D scene and captures data from the scene using a fisheye camera sensor. Use this model to learn the basics of configuring and simulating scenes, vehicles, and sensors. For more background on the 3D simulation environment, see “3D Simulation for Automated Driving” on page 6-2.

Model Overview

The model consists of these main components:

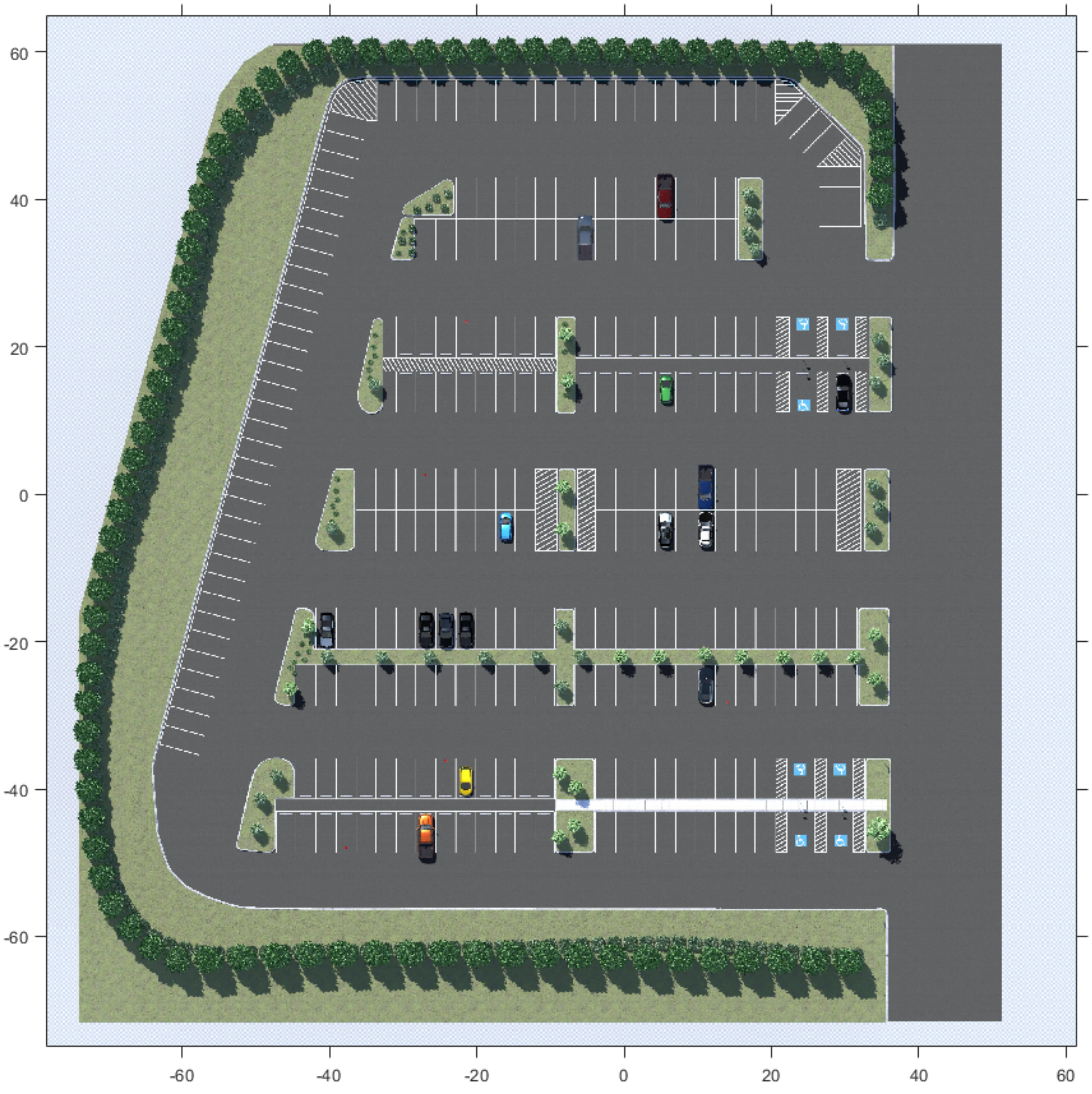
- Scene — A Simulation 3D Scene Configuration block configures the scene in which you simulate.
- Vehicles — Two Simulation 3D Vehicle with Ground Following blocks configure the vehicles within the scene and specify their trajectories.
- Sensor — A Simulation 3D Fisheye Camera configures the mounting position and parameters of the fisheye camera used to capture simulation data. A Video Viewer block visualizes the simulation output of this sensor.



Inspect Scene

In the Simulation 3D Scene Configuration block, the **Scene name** parameter determines the scene where the simulation takes place. This model uses the Large Parking Lot scene, but you can choose among several prebuilt scenes. To explore a scene, you can open the 2D image corresponding to the 3D scene.

```
data = load('sim3d_SpatialReferences.mat');
spatialRef = data.spatialReference.LargeParkingLot;
figure; imshow('sim3d_LargeParkingLot.jpg',spatialRef)
set(gca,'YDir','normal')
```



To learn how to explore other scenes, see the corresponding scene reference pages.

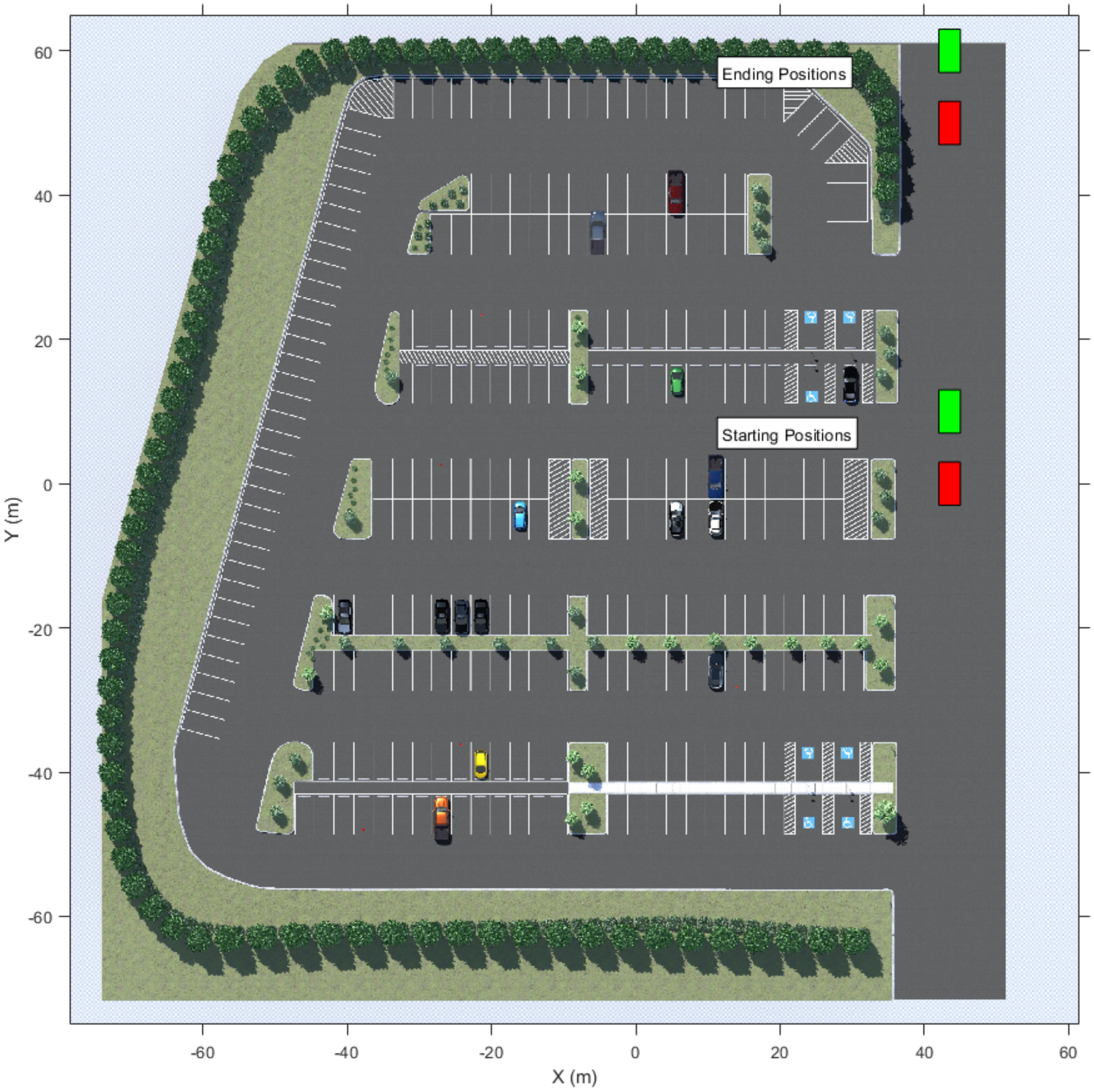
The **Scene view** parameter of this block determines the view from which the Unreal Engine window displays the scene. In this block, **Scene view** is set to `EgoVehicle`, which is the name of the ego vehicle (the vehicle with the sensor) in this scenario. During simulation, the Unreal Engine window displays the scene from behind the ego vehicle. You can also change the scene view to the other vehicle. To display the scene from the root of the scene (the scene origin), select `root`.

Inspect Vehicles

The Simulation 3D Vehicle with Ground Following blocks model the vehicles in the scenario.

- The Ego Vehicle block vehicle contains the fisheye camera sensor. This vehicle is modeled as a red hatchback.
- The Target Vehicle block is the vehicle from which the sensor captures data. This vehicle is modeled as a green SUV.

During simulation, both vehicles travel straight in the parking lot for 50 meters. The target vehicle is 10 meters directly in front of the ego vehicle.



The **X**, **Y**, and **Yaw** input ports control the trajectories of these vehicles. **X** and **Y** are in the world coordinates of the scene, which are in meters. **Yaw** is the orientation angle of the vehicle and is in degrees.

The ego vehicle travels from a position of (45,0) to (45,50), oriented 90 degrees counterclockwise from the origin. To model this position, the input port values are as follows:

- **X** is a constant value of `45`.
- **Y** is a multiple of the simulation time. A Digital Clock block outputs the simulation time every 0.1 second for 5 seconds, which is the stop time of the simulation. These simulation times are then multiplied by 10 to produce **Y** values of `[0 1 2 3 ... 50]`, or 1 meter for up to a total of 50 meters.

- **Yaw** is a constant value of `90`.

The target vehicle has the same **X** and **Yaw** values as the ego vehicle. The **Y** value of the target vehicle is always 10 meters more than the **Y** value of the ego vehicle.

In both vehicles, the **Initial position [X, Y, Z] (m)** and **Initial rotation [Roll, Pitch, Yaw] (deg)** parameters reflect the initial `[X, Y, Z]` and `[Yaw, Pitch, Roll]` values of the vehicles at the beginning of simulation.

To create more realistic trajectories, you can obtain waypoints from a scene interactively and specify these waypoints as inputs to the Simulation 3D Vehicle with Ground Following blocks. See “Select Waypoints for 3D Simulation”.

Inspect Sensor

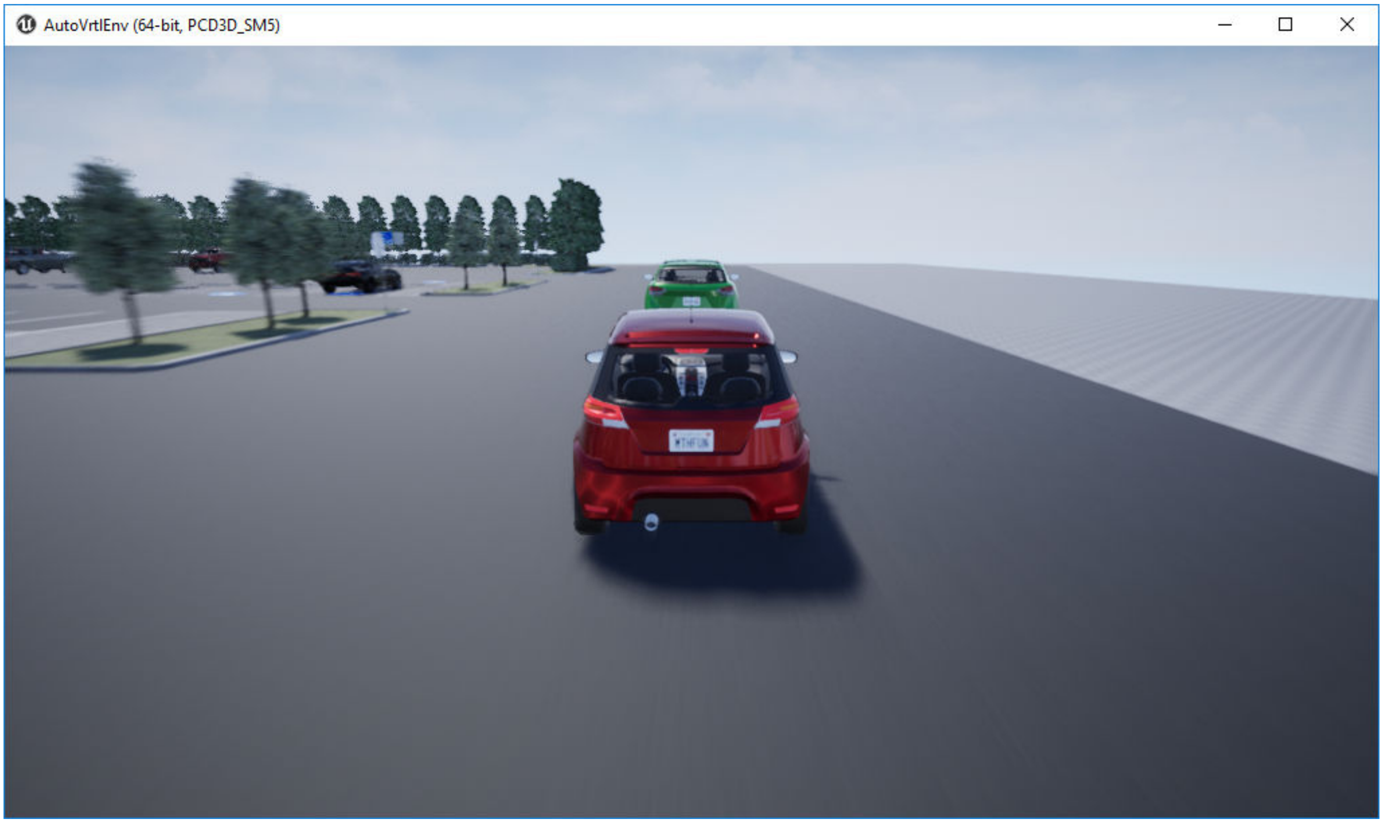
The Simulation 3D Fisheye Camera block models the sensor used in the scenario. Open this block and inspect its parameters.

- The **Mounting** tab contains parameters that determine the mounting location of the sensor. The fisheye camera sensor is mounted to the center of the roof of the ego vehicle.
- The **Parameters** tab contains the intrinsic camera parameters of a fisheye camera. These parameters are set to their default values.
- The **Ground Truth** tab contains a parameter for outputting the location and orientation of the sensor in meters and radians. In this model, the block outputs these values so you can see how they change during simulation.

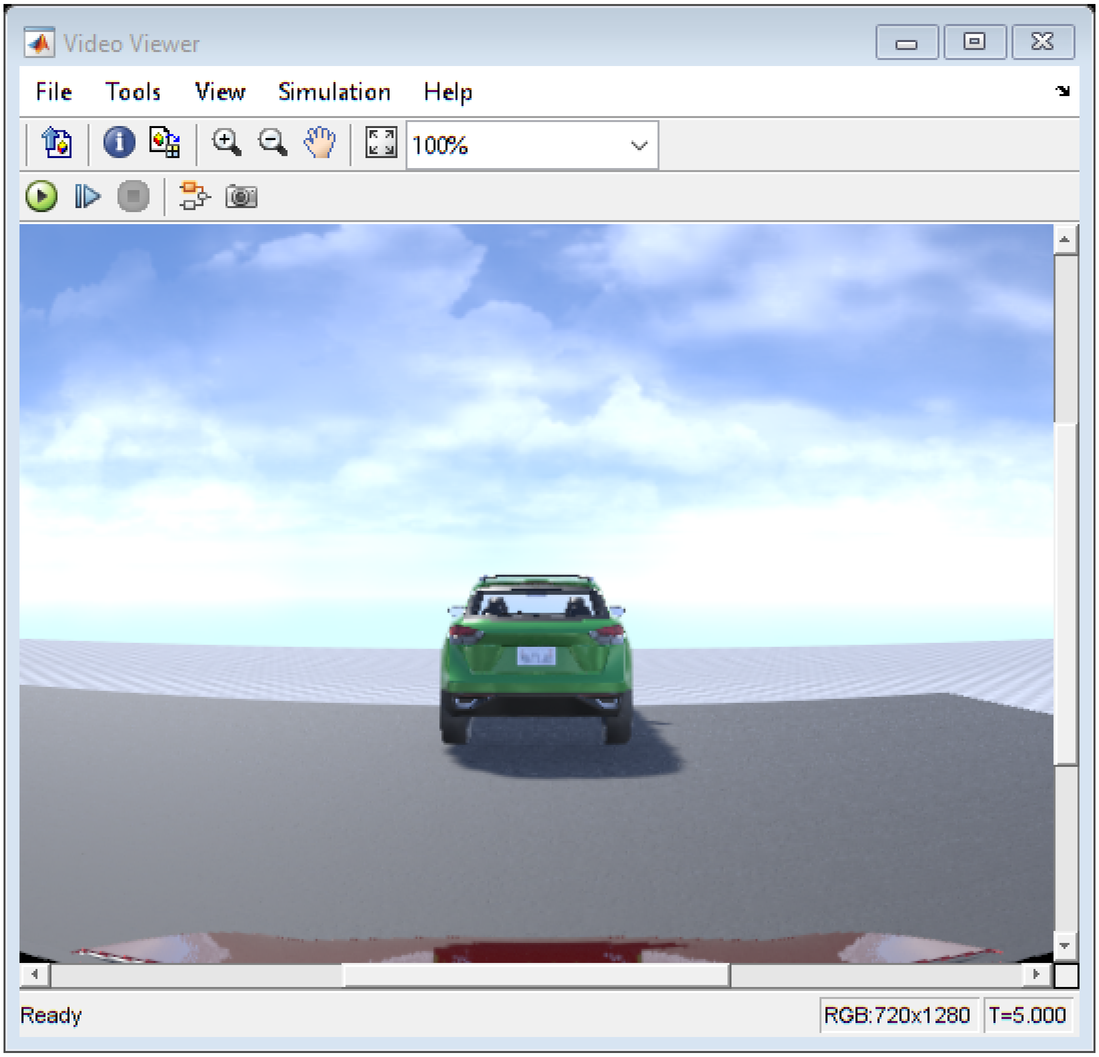
The block outputs images captured from the simulation. During simulation, the Video Viewer block displays these images.

Simulate Model

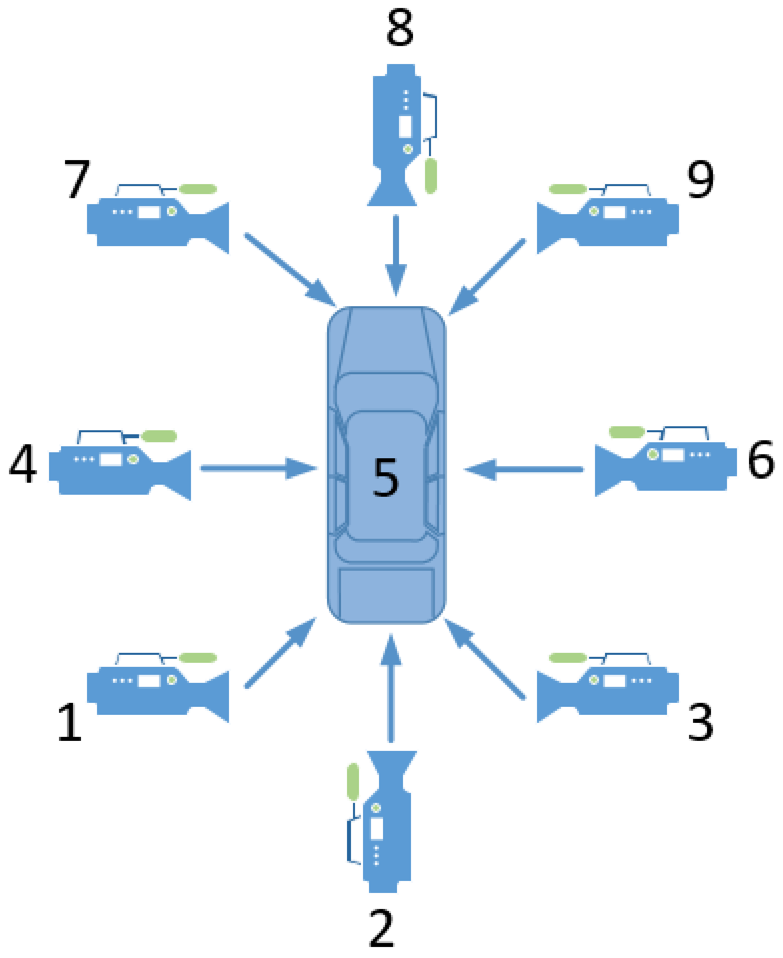
Simulate the model. When the simulation begins, it can take a few seconds for the visualization engine to initialize, especially when you are running it for the first time. The `AutoVrtlEnv` window shows a view of the scene in the 3D environment.



The Video Viewer block shows the output of the fisheye camera.



To change the view of the scene during simulation, use the numbers 1–9 on the numeric keypad.



For a bird's-eye view of the scene, press 0.

After simulating the model, try modifying the intrinsic camera parameters and observe the effects on simulation. You can also change the type of sensor block. For example, try substituting the 3D Simulation Fisheye Camera with a 3D Simulation Camera block. For more details on the available sensor blocks, see "Choose a Sensor for 3D Simulation" on page 6-16.

See Also

Simulation 3D Camera | Simulation 3D Fisheye Camera | Simulation 3D Lidar | Simulation 3D Probabilistic Radar | Simulation 3D Vehicle with Ground Following | Simulation 3D Scene Configuration

More About

- "3D Simulation for Automated Driving" on page 6-2
- "3D Simulation Environment Requirements and Limitations" on page 6-6
- "How 3D Simulation for Automated Driving Works" on page 6-8
- "Coordinate Systems in Automated Driving Toolbox" on page 1-2

- “Select Waypoints for 3D Simulation”
- “Design of Lane Marker Detector in 3D Simulation Environment”

Visualize Depth and Semantic Segmentation Data in 3D Environment

This example shows how to visualize depth and semantic segmentation data captured from a camera sensor in the Automated Driving Toolbox™ 3D simulation environment. This 3D environment uses the Unreal Engine® from Epic Games®.

You can use depth visualizations to validate depth estimation algorithms for your sensors. You can use semantic segmentation visualizations to analyze the classification scheme used for generating synthetic semantic segmentation data from the 3D environment.

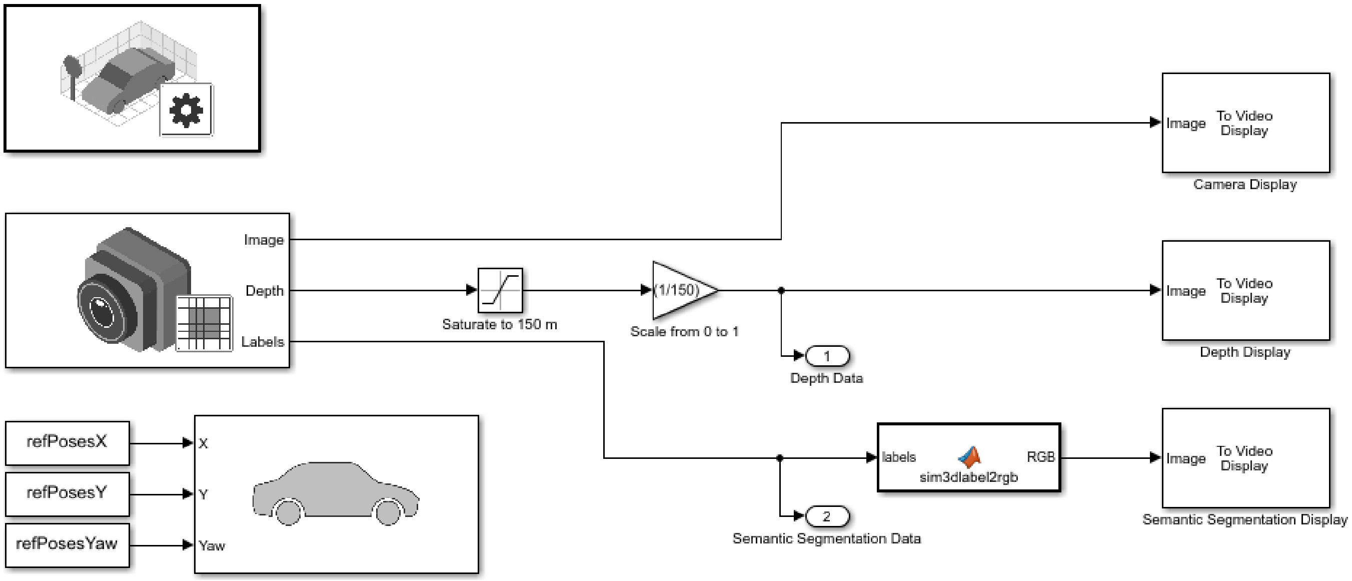
Model Setup

The model used in this example simulates a vehicle driving in a city scene.

- A Simulation 3D Scene Configuration block sets up simulation with the US City Block scene.
- A Simulation 3D Vehicle with Ground Following block specifies the driving route of the vehicle. The waypoint poses that make up this route were obtained using the technique described in the “Select Waypoints for 3D Simulation” example.
- A Simulation 3D Camera block mounted to the rearview mirror of the vehicle captures data from the driving route. This block outputs the camera, depth, and semantic segmentation displays by using To Video Display blocks.

Load the MAT-file containing the waypoint poses. Add timestamps to the poses and then open the model.

```
load smoothedPoses.mat;

refPosesX   = [linspace(0,20,1000)', smoothedPoses(:,1)];
refPosesY   = [linspace(0,20,1000)', smoothedPoses(:,2)];
refPosesYaw = [linspace(0,20,1000)', smoothedPoses(:,3)];

open_system('DepthSemanticSegmentation.slx')
```



Depth Visualization

A depth map is a grayscale representation of camera sensor output. These maps visualize camera images in grayscale, with brighter pixels indicating objects that are farther away from the sensor. You can use depth maps to validate depth estimation algorithms for your sensors.

The **Depth** port of the Simulation 3D Camera block outputs a depth map of values in the range of 0 to 1000 meters. In this model, for better visibility, a Saturation block saturates the depth output to a maximum of 150 meters. Then, a Gain block scales the depth map to the range [0, 1] so that the To Video Display block can visualize the depth map in grayscale.

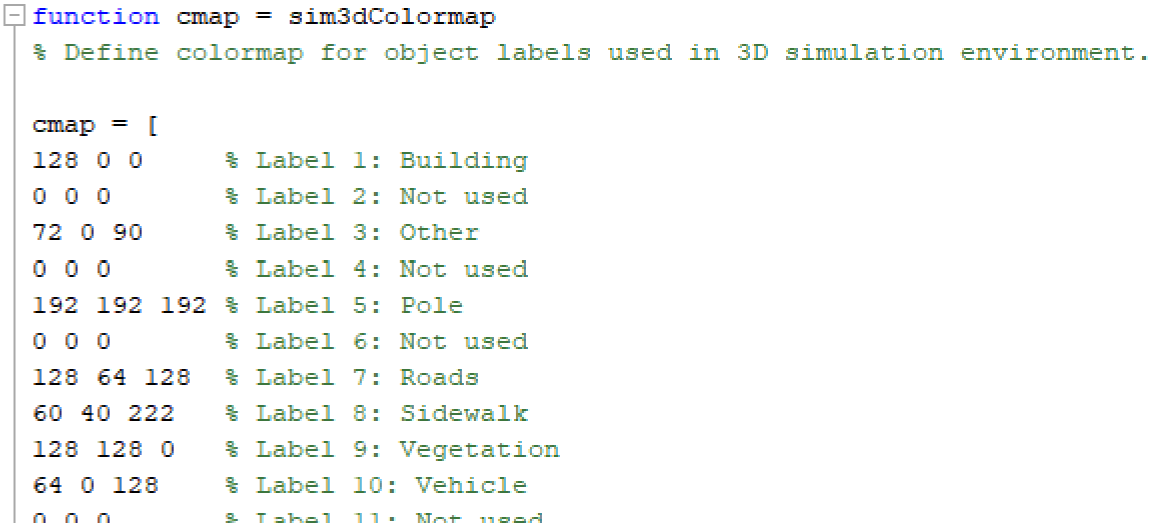
Semantic Segmentation Visualization

Semantic segmentation describes the process of associating each pixel of an image with a class label, such as *road*, *building*, or *traffic sign*. In the 3D simulation environment, you generate synthetic semantic segmentation data according to a label classification scheme. You can them use these labels to train a neural network for automated driving applications, such as road segmentation. By visualizing the semantic segmentation data, you can verify your classification scheme.

The **Labels** port of the Simulation 3D Camera block outputs a set of labels for each pixel in the output camera image. Each label corresponds to an object class. For example, in the default classification scheme used by the block, `1` corresponds to buildings. A label of `0` refers to objects of an unknown class and appears as black. For a complete list of label IDs and their corresponding object descriptions, see the **Labels** port description on the Simulation 3D Camera block reference page.

The MATLAB Function block uses the `label2rgb` function to convert the labels to a matrix of RGB triplets for visualization. The colormap is based on the colors used in the CamVid dataset, as shown in the example “Semantic Segmentation Using Deep Learning” (Computer Vision Toolbox). The colors are mapped to the predefined label IDs used in the default 3D simulation scenes. The helper function `sim3dColormap` defines the colormap. Inspect these colormap values.

```
open sim3dColormap.m
```



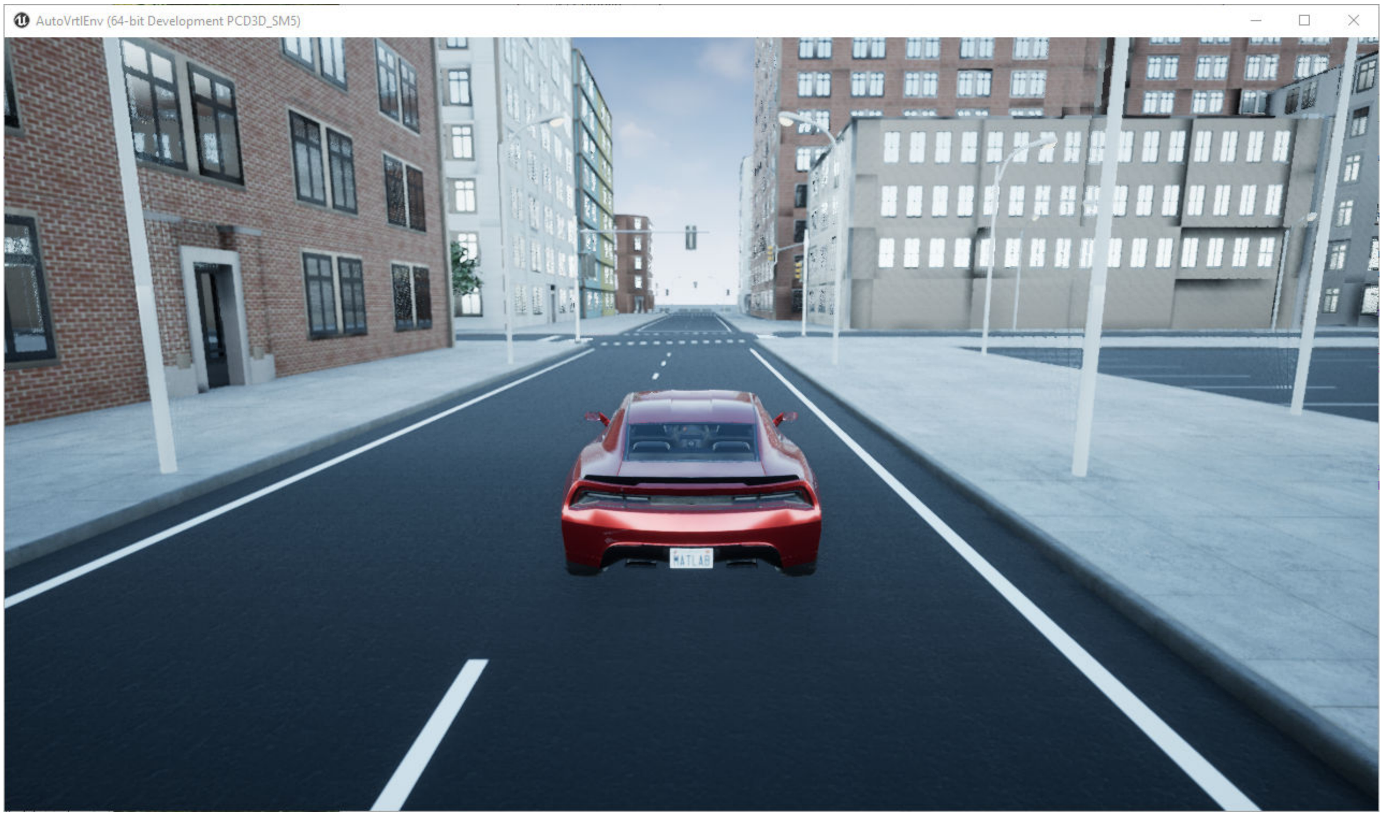
```
function cmap = sim3dColormap
% Define colormap for object labels used in 3D simulation environment.

cmap = [
128 0 0      % Label 1: Building
0 0 0        % Label 2: Not used
72 0 90      % Label 3: Other
0 0 0        % Label 4: Not used
192 192 192 % Label 5: Pole
0 0 0        % Label 6: Not used
128 64 128   % Label 7: Roads
60 40 222    % Label 8: Sidewalk
128 128 0    % Label 9: Vegetation
64 0 128     % Label 10: Vehicle
0 0 0        % Label 11: Not used
```

Model Simulation

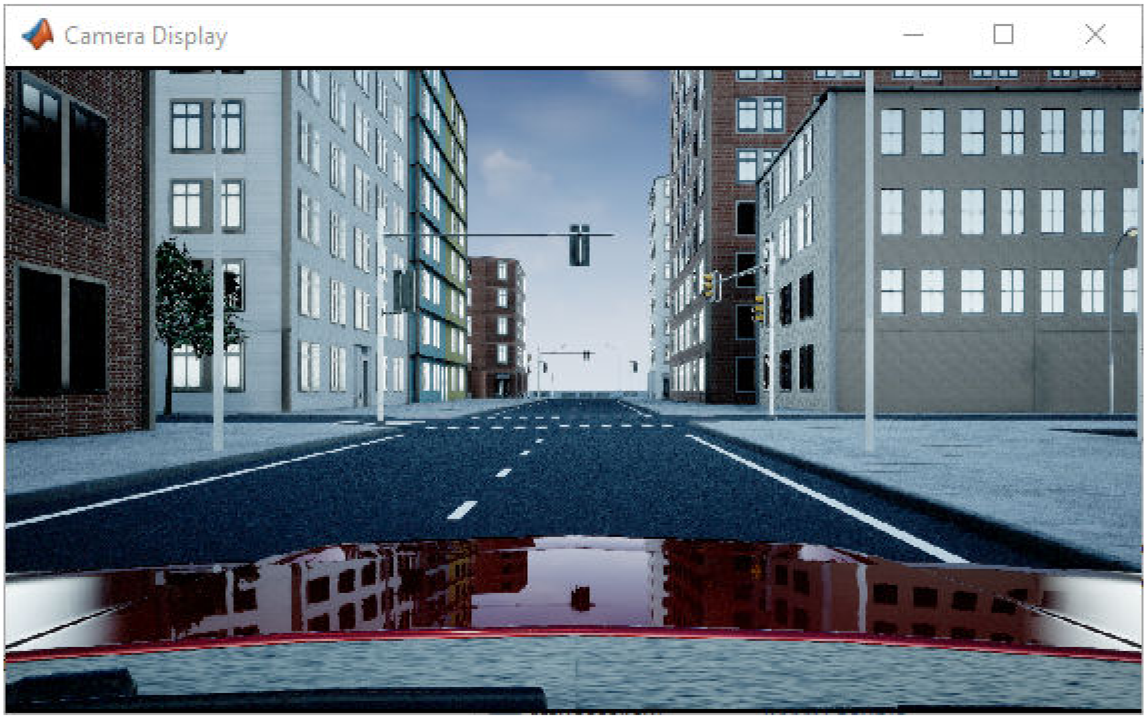
Run the model.

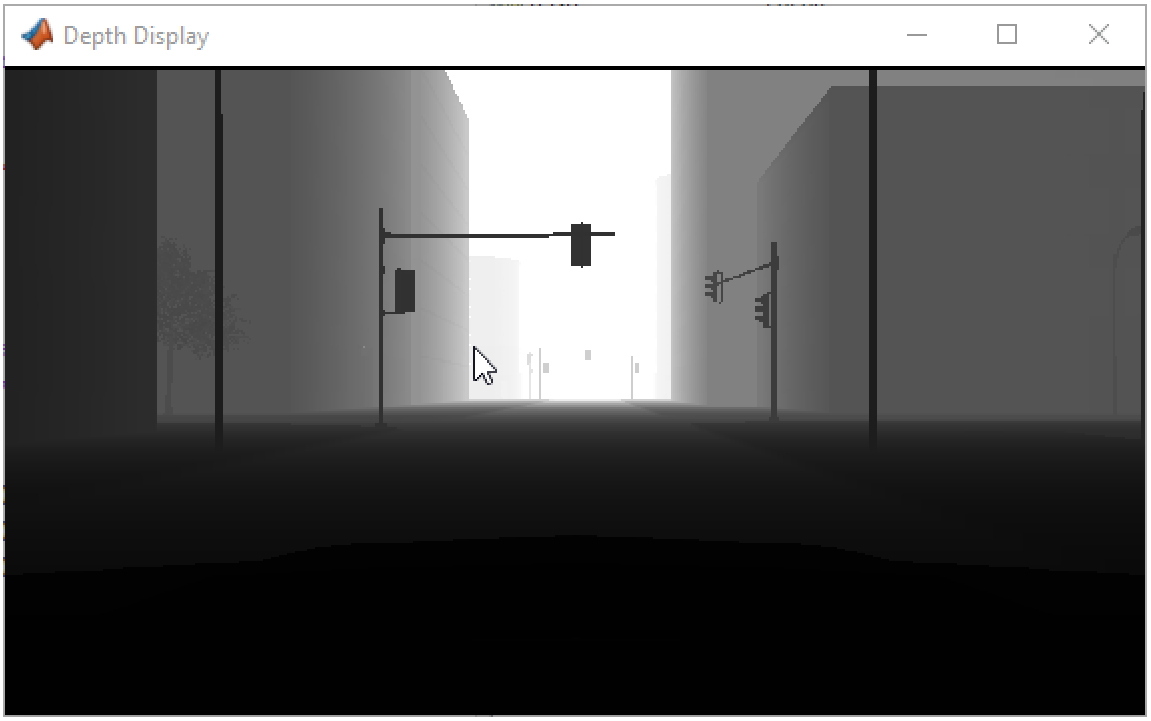
```
sim('DepthSemanticSegmentation.slx');
```

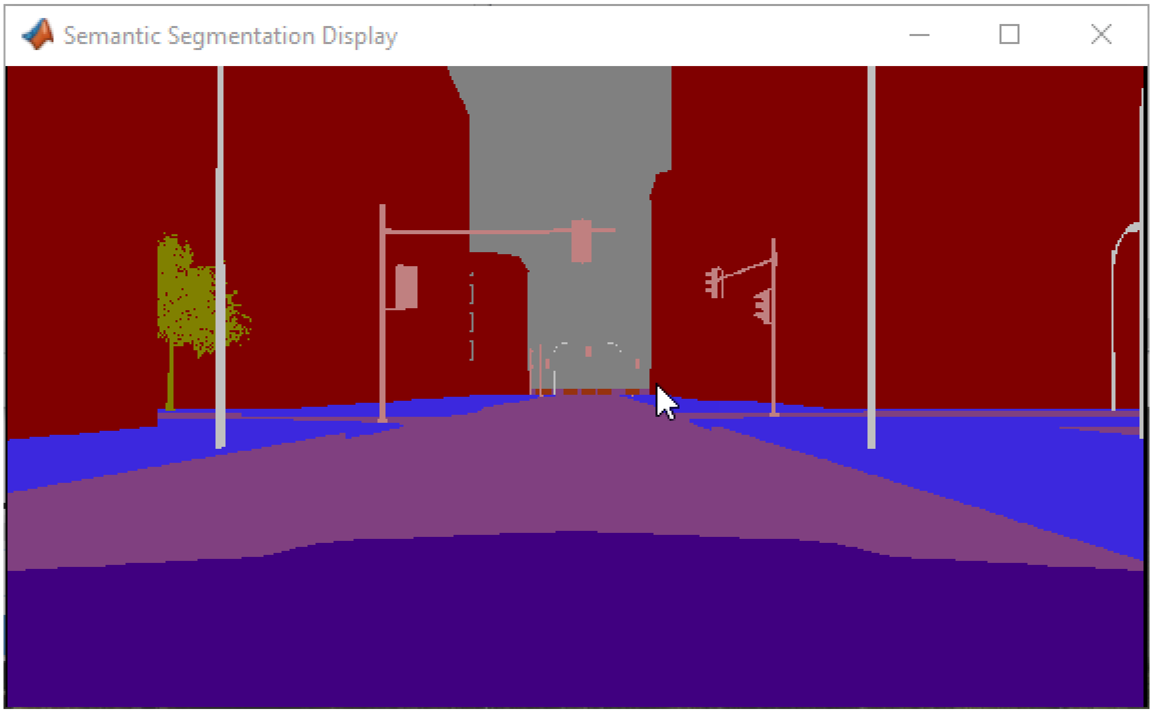
When the simulation begins, it can take a few seconds for the visualization engine to initialize, especially when you are running it for the first time. The `AutoVrtlEnv` window displays the scene from behind the ego vehicle. In this scene, the vehicles drives several blocks around the city. Because this example is mainly for illustrative purposes, the vehicle does not always follow the direction of traffic or the pattern of the changing traffic lights.



The Camera Display, Depth Display, and Semantic Segmentation Display blocks display the outputs from the camera sensor.


Camera Display


Depth Display



To change the visualization range of the output depth data, try updating the values in the Saturation and Gain blocks.

To change the semantic segmentation colors, try modifying the color values defined in the `sim3dColormap` function. Alternatively, in the `sim3dlabel2rgb` MATLAB Function block, try replacing the input colormap with your own colormap or a predefined colormap. See `colormap`.

See Also

Simulation 3D Camera | Simulation 3D Scene Configuration | Simulation 3D Vehicle with Ground Following

More About

- “Simulate a Simple Driving Scenario and Sensor in 3D Environment” on page 6-21
- “Select Waypoints for 3D Simulation”
- “Semantic Segmentation Using Deep Learning” (Computer Vision Toolbox)

Visualize 3D Simulation Sensor Coverages and Detections

This example shows how to visualize sensor coverages and detections obtained from high-fidelity radar and lidar sensors in a 3D simulation environment. In this example, you learn how to:

1. Configure Simulink® models to simulate within the 3D environment. This environment is rendered using the Unreal Engine® from Epic Games®.
2. Read ground truth data and vehicle trajectories from a scenario authored using the **Driving Scenario Designer** app, and then recreate this scenario in the Simulink model.
3. Add radar and lidar sensors to these models by using Simulation 3D Probabilistic Radar and Simulation 3D Lidar blocks.
4. Visualize the driving scenario and generated sensor data in the **Bird's-Eye Scope**.

You can use these visualizations and sensor data to test and improve your automated driving algorithms. You can also extend this example to fuse detections and visualize object tracking results, as shown in the “Sensor Fusion Using Synthetic Radar and Vision Data in Simulink” example.

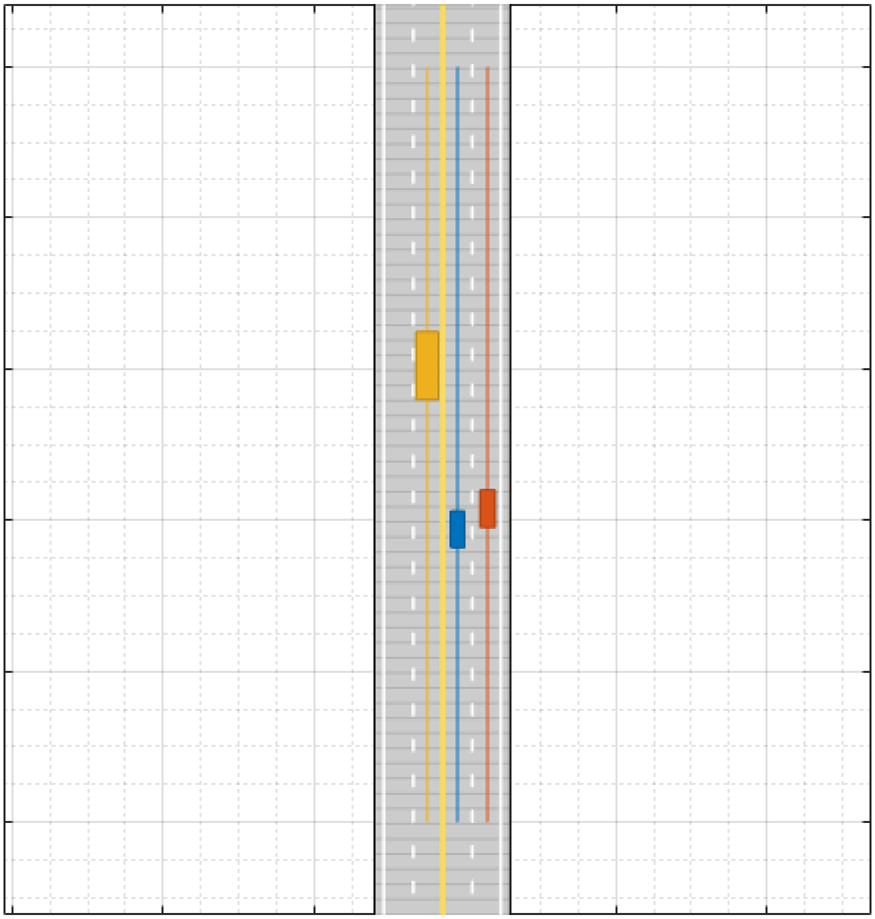
Inspect Cuboid Driving Scenario

In this example, the ground truth (roads, lanes, and actors) and vehicle trajectories come from a scenario that was authored in the **Driving Scenario Designer** app. In this app, vehicles and other actors are represented as simple box shapes called *cuboids*. For more details about authoring cuboid scenarios, see the “Build a Driving Scenario and Generate Synthetic Detections” on page 5-2 example.

Open the cuboid driving scenario file in the app.

```
drivingScenarioDesigner('StraightRoadScenario.mat')
```

In the app, run the scenario simulation. In this scenario, the ego vehicle (a blue car) travels north along a straight road at a constant speed. In the adjacent lane, an orange car travels north at a slightly higher constant speed. In the opposite lane, a yellow truck drives south at a constant speed.



When authoring driving scenarios that you later recreate in the 3D simulation environment, you must use a road network identical to one from the default 3D scenes. Otherwise, in the recreated scenario, the positions of vehicles and sensors are inaccurate. This driving scenario uses a recreation of the Straight Road scene. To select a different cuboid version of a 3D scene, on the app toolstrip, select **Open > Prebuilt Scenario > Simulation3D** and choose from the available scenes. Not all 3D scenes have corresponding versions in the app.

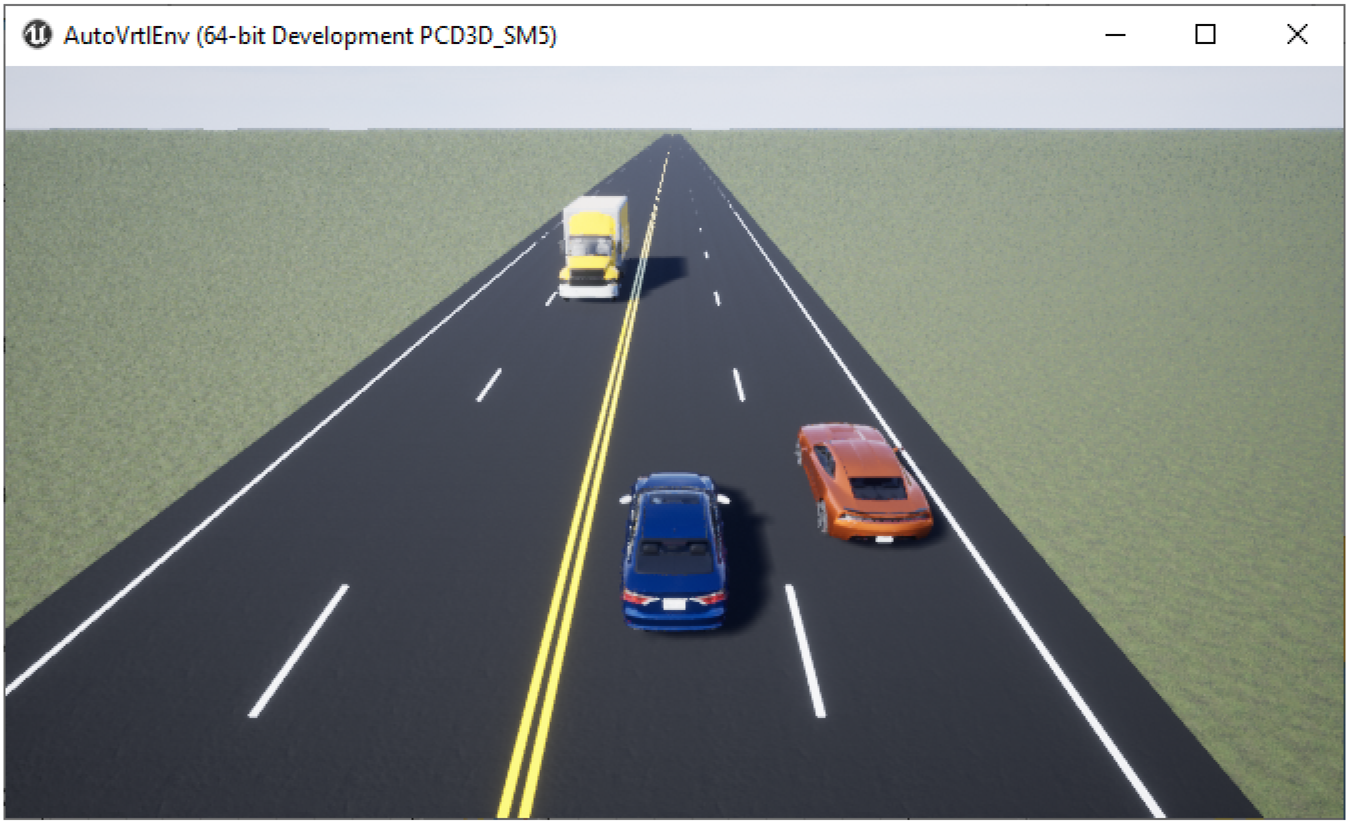
- For a list of supported scenes and additional details about each scene, see “Cuboid Versions of 3D Simulation Scenes in Driving Scenario Designer” on page 5-55.
- To generate vehicle trajectories for scenes that are not available in the app, use the process described in the “Select Waypoints for 3D Simulation” example instead.

The dimensions of vehicles in the cuboid scenarios must also match the dimensions of one of the predefined 3D simulation vehicle types. On the app toolstrip, under **3D Display**, the **Use 3D Simulation Actor Dimensions** selection sets each cuboid vehicle to have the dimensions of a 3D vehicle type. In this scenario, the vehicles have these 3D display types and corresponding vehicle dimensions.

- `Ego Vehicle` — Sedan vehicle dimensions
- `Vehicle in Adjacent Lane` — Muscle Car vehicle dimensions
- `Vehicle in Opposite Lane` — Box Truck vehicle dimensions

To change a vehicle to a different display type, on the **Actors** tab in the left pane of the app, update the **3D Display Type** parameter for that vehicle. To change the color of a vehicle, select the color patch next to the selected vehicle and choose a new color.

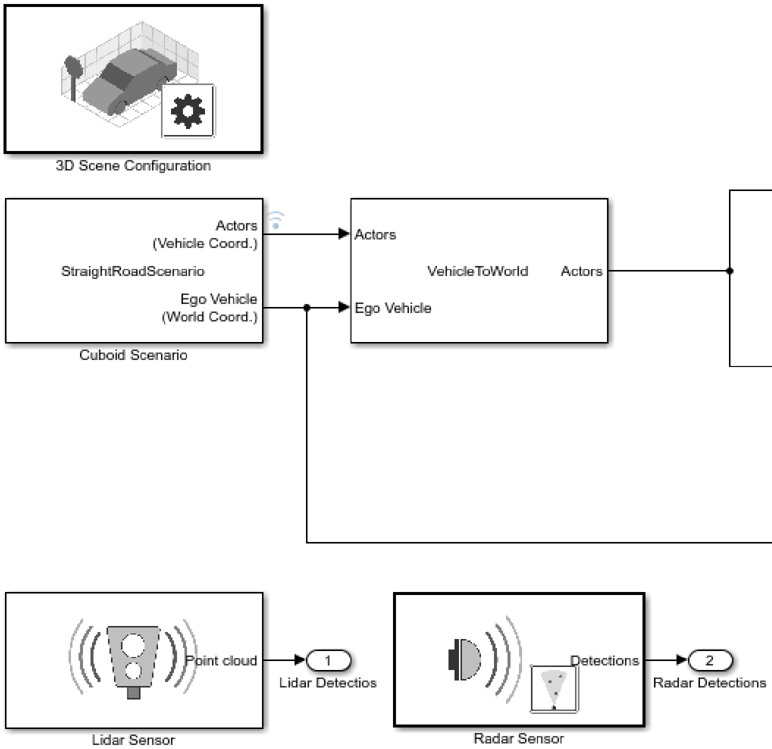
To preview how the vehicles display in the 3D environment, use the 3D display window available from the app. On the app toolstrip, select **3D Display > View Simulation in 3D Display** and rerun the simulation.



Open 3D Simulation Model

The model used in this example recreates the cuboid driving scenario. The model also defines high-fidelity sensors that generate synthetic detections from the environment. Open the model.

```
open_system('Visualize3DSimulationSensorCoveragesDetections')
```



Inspect Scene Configuration

The Simulation 3D Scene Configuration block configures the model to simulate in the 3D environment.

- The **Scene name** parameter is set to the default `Straight road` scene. This scene corresponds to the cuboid version defined in the app scenario file.
- The **Scene view** parameter is set to `Ego Vehicle`. During simulation, the 3D simulation window displays the scene from behind the ego vehicle.

The Scenario Reader block reads the ground truth data (road boundaries, lane markings, and actor poses) from the app scenario file. The **Bird's-Eye Scope** visualizes this ground truth data, not the ground truth data of the 3D simulation environment. To use the same scene for the cuboid and 3D simulation environments, the ground truth data for both environments must match. If you are creating a new scenario, you can generate a Scenario Reader block that reads data from your scenario file. First, open the scenario file in the **Driving Scenario Designer** app. Then, on the app toolstrip, select **Export > Export Simulink Model**. If you update the scenario, you do not need to generate a new Scenario Reader block.

The Simulation 3D Scene Configuration block and Scenario Reader block both have their **Sample time** parameter set to `0.1`. In addition, all other 3D simulation vehicle and sensor blocks inherit their sample time from the Simulation 3D Scene Configuration block. By setting a single sample time across the entire model, the **Bird's-Eye Scope** displays data from all blocks at a constant rate. If the ground truth and sensor data have different sample times, then the scope visualizes them at different time intervals. This process causes the ground truth and sensor data visualizations to flicker.

Inspect Vehicle Configuration

The Simulation 3D Vehicle with Ground Following blocks specify the appearances and trajectories of the vehicles in the 3D simulation environment. Each vehicle is a direct counterpart to one of the vehicles defined in the **Driving Scenario Designer** app scenario file.

In the 3D environment, vehicle positions are in world coordinates. However, the Scenario Reader block outputs the poses of non-ego actors in ego vehicle coordinates. A Vehicle To World block converts these non-ego actor poses into world coordinates. Because the ego vehicle is output in world coordinates, this conversion is not necessary for the ego vehicle. For more details about the vehicle and world coordinate systems, see “Coordinate Systems in Automated Driving Toolbox” on page 1-2.

Locations of vehicle origins differ between cuboid and 3D scenarios.

- In cuboid scenarios, the vehicle origin is on the ground, at the center of the rear axle.
- In 3D scenarios, the vehicle origin is on ground, at the geometric center of the vehicle.

The Cuboid To 3D Simulation blocks convert the cuboid origin positions to the 3D simulation origin positions. In the **ActorID used for conversion** parameters of these blocks, the specified `ActorID` of each vehicle determines which vehicle origin to convert. The Scenario Reader block outputs `ActorID` values in its **Actors** output port. In the **Driving Scenario Designer** app, you can find the corresponding `ActorID` values on the **Actors** tab, in the actor selection list. The `ActorID` for each vehicle is the value that precedes the colon.

Each Cuboid To 3D Simulation block outputs **X**, **Y**, and **Yaw** values that feed directly into their corresponding vehicle blocks. In the 3D simulation environment, the ground terrain of the 3D scene determines the *Z*-position (elevation), roll angle, and pitch angle of the vehicles.

In each Simulation 3D Vehicle with Ground Following block, the **Type** parameter corresponds to the **3D Display Type** selected for that vehicle in the app. In addition, the **Color** parameter corresponds to the vehicle color specified in the app. To maintain similar vehicle visualizations between the **Bird's-Eye Scope** and the 3D simulation window, the specified type and color must match. To change the color of a vehicle in the app, on the **Actors** tab, click the color patch to the right of the actor name in the actor selection list. Choose the color that most closely matches the colors available in the **Color** parameter of the Simulation 3D Vehicle with Ground Following block.

Inspect Sensor Configuration

The model includes two sensor blocks with default parameter settings. These blocks generate detections from the 3D simulation environment.

- The Simulation 3D Probabilistic Radar sensor block generates object detections based on a statistical model. This sensor is mounted to the front bumper of the ego vehicle.
- The Simulation 3D Lidar sensor block generates detections in the form of a point cloud. This sensor is mounted to the center of the roof of the ego vehicle.

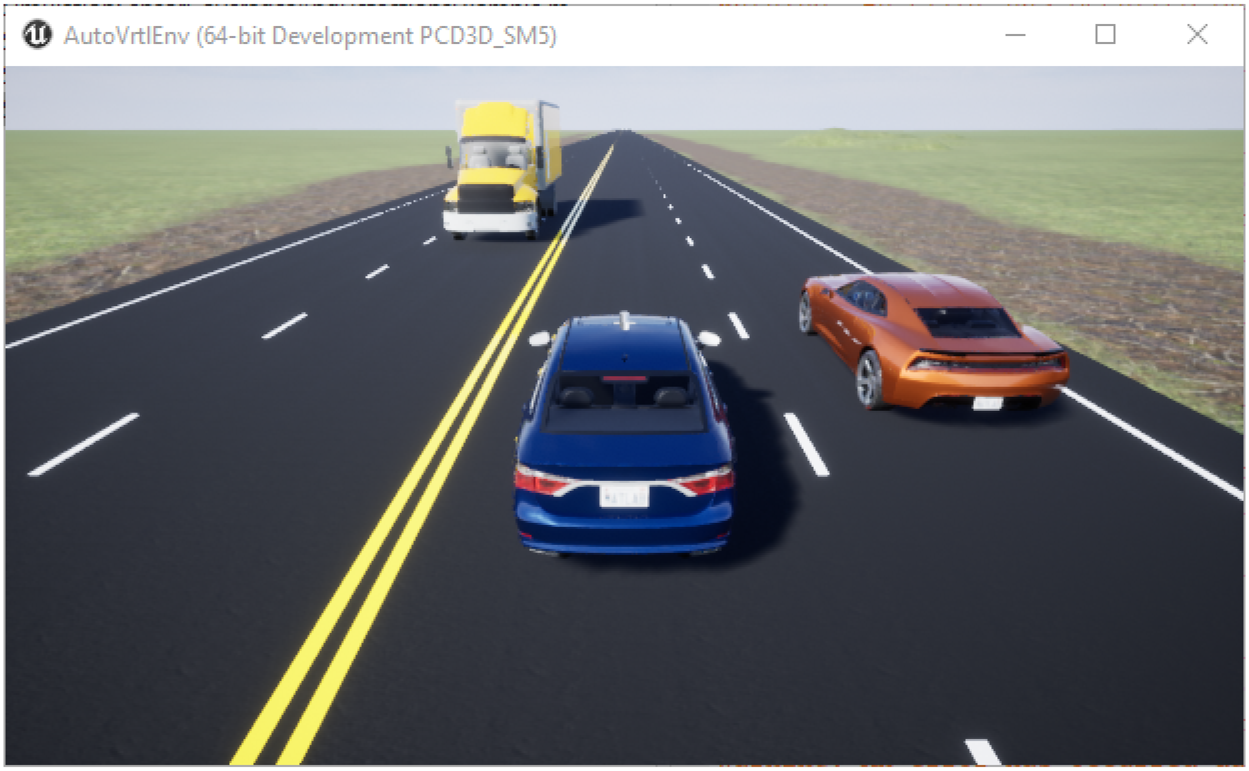
Although you can specify sensors in the **Driving Scenario Designer** app and export them to Simulink, the exported blocks are not compatible with the 3D simulation environment. You must specify 3D simulation sensors in the model directly.

Simulate and Visualize Scenario

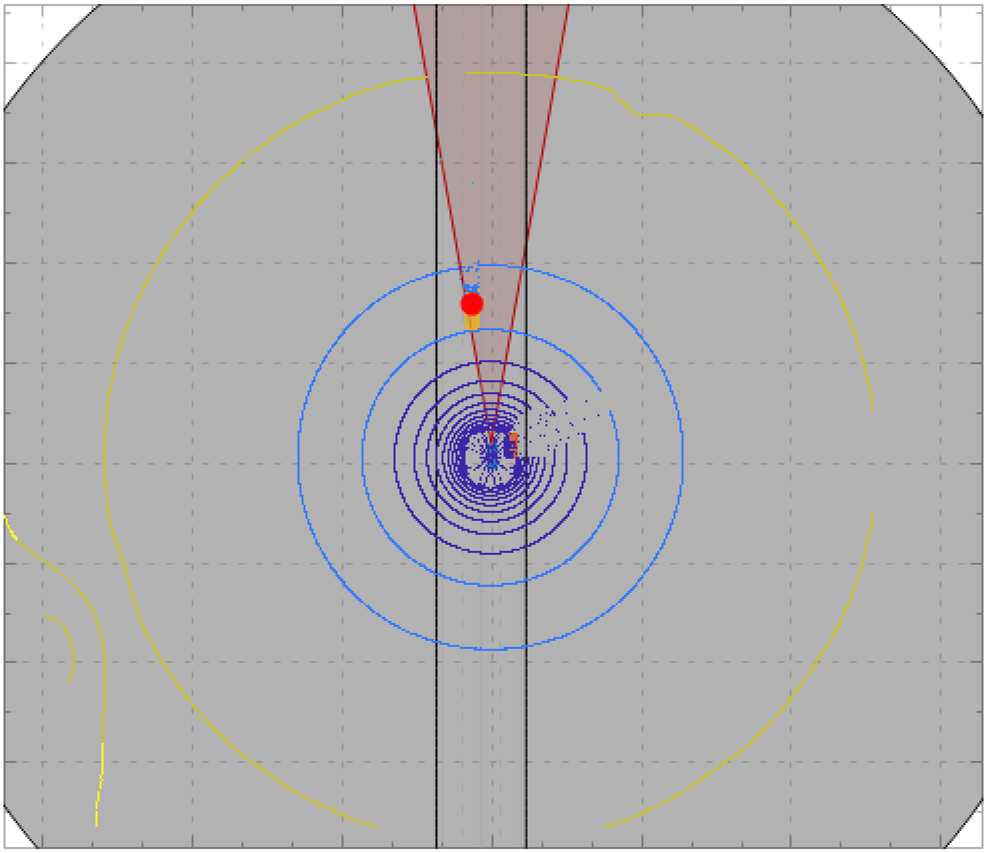
During simulation, you can visualize the scenario in both the 3D simulation window and the **Bird's-Eye Scope**.

First, open the scope. On the Simulink toolstrip, under **Review Results**, click **Bird's-Eye Scope**. Then, to find signals that the scope can display, click **Find Signals**.

To run the simulation, click **Run** in either the model or scope. When the simulation begins, it can take a few seconds for the 3D simulation window to initialize, especially when you run it for the first time in a Simulink session. When this window opens, it displays the scenario with high-fidelity graphics but does not display detections or sensor coverages.



The **Bird's-Eye Scope** displays detections and sensor coverages by using a cuboid representation. The radar coverage area and detections are in red. The lidar coverage area is in gray, and its point cloud detections display as a parula colormap.



The model runs the simulation at a pace of 0.5 seconds per wall-clock second. To adjust the pacing, from the Simulink toolstrip, select **Run > Simulation Pacing**, and then move the slider to increase or decrease the speed of the simulation.

Modify the Driving Scenario

When modifying your driving scenario, you might need to update the scenario in the **Driving Scenario Designer** app, the Simulink model, or in both places, depending on what you change.

- **Modify the road network** — In the app, select a new prebuilt scene from the **Simulation3D** folder. Do not modify these road networks or the roads will not match the roads in the selected 3D scene. In the model, in the Simulation 3D Scene Configuration block, select the corresponding scene in the **Scene name** parameter.
- **Modify vehicle trajectories** — In the app, modify the vehicle trajectories and resave the scenario. In the model, you do not need to update anything to account for this change. The Scenario Reader block automatically picks up these changes.
- **Modify vehicle appearances** — In the app, update the color and **3D Display Type** parameter of the vehicles. Also make sure that the **3D Display > Use 3D Simulation Actor Dimensions** option is selected. In the model, update the **Color** and **Type** parameters of the corresponding Simulation 3D Vehicle with Ground Following blocks.
- **Add a new vehicle** — In the app, create a new vehicle and specify a trajectory, color, and 3D display type. In the model, add a new Simulation 3D Vehicle with Ground Following block and corresponding Cuboid To 3D Simulation block. Set up these blocks similar to how the existing non-ego vehicles are set up. In the Cuboid To 3D Simulation block, set the `ActorID` of the new vehicle.

- **Set a new ego vehicle** — In the app, on the **Actors** tab, select the vehicle that you want to set as the ego vehicle and click **Set As Ego Vehicle**. In the model, in the Cuboid To 3D Simulation blocks, update the **ActorID used for conversion** parameters to account for which vehicle is the new ego vehicle. In the sensor blocks, set the **Parent name** parameters such that the sensors are mounted to the new ego vehicle.
- **Modify or add sensors** — In the app, you do not need to make any changes. In the model, modify or add sensor blocks. When adding sensor blocks, set the **Parent name** of all sensors to the ego vehicle.

To visualize any updated scenario in the **Bird's-Eye Scope**, you must click **Find Signals** again. If you modify a scenario or are interested in only visualizing sensor data, consider turning off the 3D window during simulation. In the Simulation 3D Scene Configuration block, clear the **Display 3D simulation window** parameter.

See Also

Apps
Bird's-Eye Scope | **Driving Scenario Designer**

Blocks
Cuboid To 3D Simulation | Scenario Reader | Simulation 3D Scene Configuration | Simulation 3D Vehicle with Ground Following | Vehicle To World

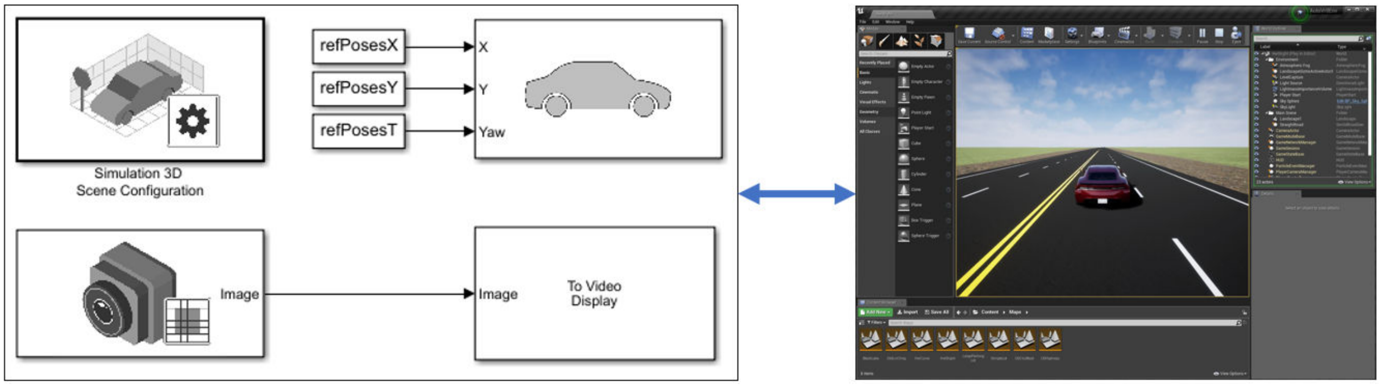
More About

- “Choose a Sensor for 3D Simulation” on page 6-16
- “Cuboid Versions of 3D Simulation Scenes in Driving Scenario Designer” on page 5-55
- “Highway Lane Following”

Customize 3D Scenes for Automated Driving

Automated Driving Toolbox comes installed with prebuilt 3D scenes in which to simulate and visualize the performance of driving algorithms modeled in Simulink. These 3D scenes are visualized using the Unreal Engine from Epic Games. By using the Unreal Editor and the Automated Driving Toolbox Interface for Unreal Engine 4 Projects, you can customize these scenes. You can also use the Unreal Editor and the support package to simulate within scenes from your own custom project.

With custom scenes, you can co-simulate in both Simulink and the Unreal Editor so that you can modify your scenes between simulation runs. You can also package your scenes into an executable file so that you do not have to open the editor to simulate with these scenes.



To customize 3D scenes for automated driving, follow these steps:

1. “Install Support Package for Customizing Scenes” on page 6-44
2. “Customize Scenes Using Simulink and Unreal Editor” on page 6-47
3. “Package Custom Scenes into Executable” on page 6-52

See Also

Simulation 3D Scene Configuration

More About

- “3D Simulation for Automated Driving” on page 6-2

Install Support Package for Customizing Scenes

To customize scenes in the Unreal Editor and use them in Simulink, you must install the Automated Driving Toolbox Interface for Unreal Engine 4 Projects.

Verify Software and Hardware Requirements

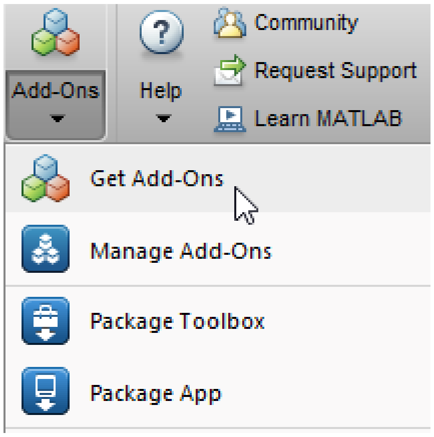
Before installing the support package, make sure that your environment meets the minimum software and hardware requirements described in “3D Simulation Environment Requirements and Limitations” on page 6-6. In particular, verify that you have Visual Studio 2017 or newer installed. This software is required for using the Unreal Editor to customize scenes.

In addition, verify that your project is compatible with Unreal Engine, Version 4.23. If your project was created with an older version of the Unreal Editor, upgrade your project to version 4.23.

Install Support Package

To install the Automated Driving Toolbox Interface for Unreal Engine 4 Projects support package, follow these steps:

1 On the MATLAB **Home** tab, in the **Environment** section, select **Add-Ons** > **Get Add-Ons**.



2 In the Add-On Explorer window, search for the Automated Driving Toolbox Interface for Unreal Engine 4 Projects support package. Click **Install**.

Note You must have write permission for the installation folder.

Set Up Scene Customization Using Support Package

The Automated Driving Toolbox Interface for Unreal Engine 4 Projects support package includes these components:

- An Unreal Engine project file (`AutoVrtlEnv.uproject`) and its associated files. This project file includes editable versions of the prebuilt 3D scenes that you can select from the **Scene description** parameter of the Simulation 3D Scene Configuration block.
- A plugin file, `MathWorkSimulation.uplugin`. This plugin establishes the connection between Simulink and the Unreal Editor and is required for co-simulation.

To set up scene customization, you must copy this project and plugin onto your local machine.

Copy Project to Local Folder

Copy the `AutoVrtlEnv` project folder into a folder on your local machine.

1. Specify the path to the support package folder that contains the project. If you previously downloaded the support package, specify only the latest download path, as shown here. Also specify a local folder destination in which to copy the project. This code specifies a local folder of `C:\Local`.

   ```
   supportPackageFolder = fullfile( ...
       matlabshared.supportpkg.getSupportPackageRoot, ...
       "toolbox","shared","sim3dprojects","driving");
   localFolder = "C:\Local";
   ```
2. Copy the `AutoVrtlEnv` project from the support package folder to the local destination folder.

   ```
   projectFolderName = "AutoVrtlEnv";
   projectSupportPackageFolder = fullfile(supportPackageFolder,projectFolderName);
   projectLocalFolder = fullfile(localFolder,projectFolderName);
   if ~exist(projectLocalFolder,"dir")
       copyfile(projectSupportPackageFolder,projectLocalFolder);
   end
   ```

   The `AutoVrtlEnv.uproject` file and all of its supporting files are now located in a folder named `AutoVrtlEnv` within the specified local folder. For example: `C:\Local\AutoVrtlEnv`.

Copy Plugin to Unreal Editor

Copy the `MathWorksSimulation` plugin into the `Plugins` folder of your Unreal Engine installation.

1. Specify the local folder containing your Unreal Engine installation. This code shows the default installation location for the editor on a Windows machine.

   ```
   ueInstallFolder = "C:\Program Files\Epic Games\UE_4.23";
   ```
2. Copy the plugin from the support package into the `Plugins` folder.

   ```
   supportPackageFolder = fullfile( ...
       matlabshared.supportpkg.getSupportPackageRoot, ...
       "toolbox","shared","sim3dprojects","driving");

   mwPluginName = "MathWorksSimulation.uplugin";
   mwPluginFolder = fullfile(supportPackageFolder,"PluginResources","UE423");
   uePluginFolder = fullfile(ueInstallFolder,"Engine","Plugins");
   uePluginDestination = fullfile(uePluginFolder,"Marketplace","MathWorks");

   cd(uePluginFolder)
   foundPlugins = dir("**/" + mwPluginName);

   if ~isempty(foundPlugins)
       numPlugins = size(foundPlugins,1);
       msg2 = cell(1,numPlugins);
       pluginCell = struct2cell(foundPlugins);

       msg1 = "Plugin(s) already exist here:" + newline + newline;
       for n = 1:numPlugins
           msg2{n} = "    " + pluginCell{2,n} + newline;
   ```

```
        end
        msg3 = newline + "Please remove plugin folder(s) and try again.";
        msg  = msg1 + msg2 + msg3;
        warning(msg);
    else
        copyfile(mwPluginFolder, uePluginDestination);
        disp("Successfully copied MathWorksSimulation plugin to UE4 engine plugins!")
    end
```

After you install and set up the support package, you can begin customizing scenes. See “Customize Scenes Using Simulink and Unreal Editor” on page 6-47.

See Also

More About

- “3D Simulation for Automated Driving” on page 6-2
- “3D Simulation Environment Requirements and Limitations” on page 6-6

Customize Scenes Using Simulink and Unreal Editor

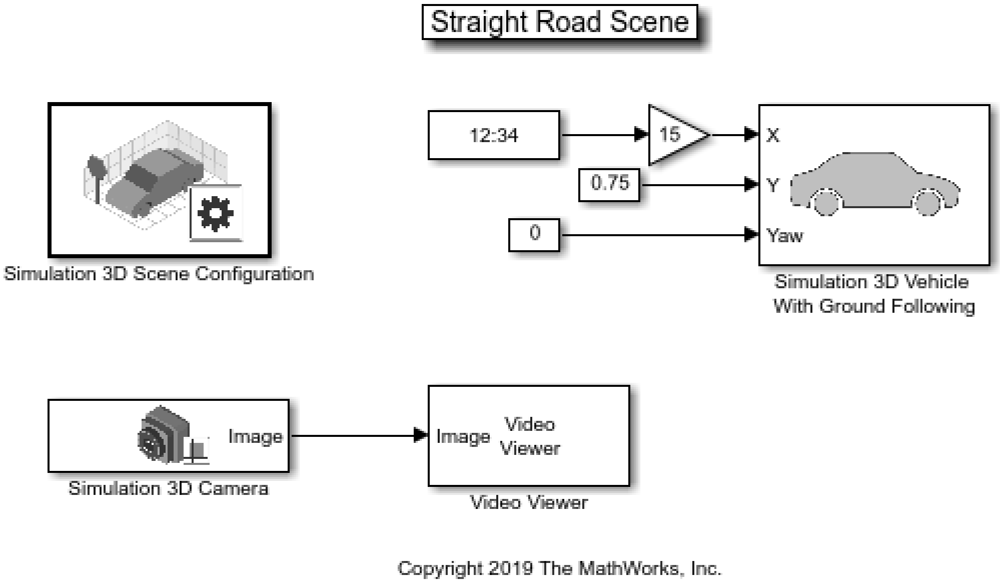
After you install the Automated Driving Toolbox Interface for Unreal Engine 4 Projects support package as described in “Install Support Package for Customizing Scenes” on page 6-44, you can simulate in custom scenes simultaneously from both the Unreal Editor and Simulink. By using this co-simulation framework, you can add vehicles and sensors to a Simulink model and then run this simulation in your custom scene.

Open Unreal Editor from Simulink

If you open your Unreal project file directly in the Unreal Editor, Simulink is unable to establish a connection with the editor. To establish this connection, you must open your project from a Simulink model.

1 Open a Simulink model configured to simulate in the 3D environment. At a minimum, the model must contain a Simulation 3D Scene Configuration block. For example, open a simple model that simulates a vehicle driving on a straight highway. This model is used in the “Design of Lane Marker Detector in 3D Simulation Environment” example.

```
open_system('straightRoadSim3D');
```


Straight Road Scene
Simulation 3D Scene Configuration
12:34
15
0.75
0
X
Y
Yaw
Simulation 3D Vehicle With Ground Following
Image
Simulation 3D Camera
Image
Video Viewer
Video Viewer
Copyright 2019 The MathWorks, Inc.

2 In the Simulation 3D Scene Configuration block of this model, set the **Scene source** parameter to `Unreal Editor`.

3 In the **Project** parameter, browse for the project file that contains the scenes that you want to customize.

For example, this sample path specifies the `AutoVrtlEnv` project that comes installed with the Automated Driving Toolbox Interface for Unreal Engine 4 Projects support package.

```
C:\Local\AutoVrtlEnv\AutoVrtlEnv.uproject
```

This sample path specifies a custom project.

```
Z:\UnrealProjects\myProject\myProject.uproject
```

4. Click **Open Unreal Editor**. The Unreal Editor opens and loads a scene from your project.

The first time that you open the Unreal Editor from Simulink, you might be asked to rebuild `UE4Editor` DLL files or the `AutoVrtlEnv` module. Click **Yes** to rebuild these files or modules. The editor also prompts you that new plugins are available. Click **Manage Plugins** and verify that the **MathWorks Interface** plugin is installed. This plugin is the `MathWorksSimulation.uplugin` file that you copied into your Unreal Editor installation in “Install Support Package for Customizing Scenes” on page 6-44.

When the editor opens, you can ignore any warning messages about files with the name `'_BuiltData'` that failed to load.

If you receive a warning that the lighting needs to be rebuilt, from the toolbar above the editor window, select **Build** > **Build Lighting Only**. The editor issues this warning the first time you open a scene or when you add new elements to a scene.

Reparent Actor Blueprint

Note If you are using a scene from the `AutoVtrlEnv` project that comes installed with the Automated Driving Toolbox Interface for Unreal Engine 4 Projects support package, skip this section. However, if you create a new scene based off of one of the scenes in this project, then you must complete this section.

The first time that you open a custom scene from Simulink, you need to associate, or reparent, this project with the **Sim3dLevelScriptActor** level blueprint used in Automated Driving Toolbox. The level blueprint controls how objects interact with the 3D environment once they are placed in it. Simulink returns an error at the start of simulation if the project is not reparented. You must reparent each scene in a custom project separately.

To reparent the level blueprint, follow these steps:

1. In the Unreal Editor toolbar, select **Blueprints** > **Open Level Blueprint**.
2. In the Level Blueprint window, select **File** > **Reparent Blueprint**.
3. Click the **Sim3dLevelScriptActor** blueprint. If you do not see the **Sim3dLevelScriptActor** blueprint listed, use these steps to check that you have the `MathWorksSimulation` plugin installed and enabled:

   a. In the Unreal Editor toolbar, select **Settings** > **Plugins**.

   b. In the Plugins window, verify that the **MathWorks Interface** plugin is listed in the installed window. If the plugin is not already enabled, select the **Enabled** check box.

      If you do not see the **MathWorks Interface** plugin in this window, repeat the steps under “Copy Plugin to Unreal Editor” on page 6-45 and reopen the editor from Simulink.

   c. Close the editor and reopen it from Simulink.
4. Close the Level Blueprint window.

Create or Modify Scenes in Unreal Editor

After you open the editor from Simulink, you can modify the scenes in your project or create new scenes.

Open Scene

In the Unreal Editor, scenes within a project are referred to as levels. Levels come in several types, and scenes have a level type of map.

To open a prebuilt scene from the `AutoVrtlEnv.uproject` file, in the **Content Browser** pane below the editor window, navigate to the **Content** > **Maps** folder. Then, select the map that corresponds to the scene you want to modify.

Unreal Editor Map	Automated Driving Toolbox Scene
`HwCurve`	**Curved Road**
`DblLnChng`	**Double Lane Change**
`BlackLake`	**Open Surface**
`LargeParkingLot`	**Large Parking Lot**
`SimpleLot`	**Parking Lot**
`HwStrght`	**Straight Road**
`USCityBlock`	**US City Block**
`USHighway`	**US Highway**

Note The `AutoVrtlEnv.uproject` file does not include the **Virtual Mcity** scene.

To open a scene within your own project, in the **Content Browser** pane, navigate to the folder that contains your scenes.

Create New Scene

To create a new scene in your project, from the top-left menu of the editor, select **File** > **New Level**.

Alternatively, you can create a new scene from an existing one. This technique is useful if you want to use one of the prebuilt scenes in the `AutoVtrlEnv` project as a starting point for creating your own scene. To save a version of the currently opened scene to your project, from the top-left menu of the editor, select **File** > **Save Current As**. The new scene is saved to the same location as the existing scene.

Add Assets to Scene

In the Unreal Editor, elements within a scene are referred to as assets. To add assets to a scene, you can browse or search for them in the **Content Browser** pane at the bottom and drag them into the editor window.

When adding assets to a scene that is in the `AutoVrtlEnv` project, you can choose from a library of driving-related assets. These assets are built as static meshes and begin with the prefix `SM_`. Search for these objects in the **Content Browser** pane.

For example, add a stop sign to a scene in the `AutoVrtlEnv` project.

1. In the **Content Browser** pane at the bottom of the editor, navigate to the **Content** folder.
2. In the search bar, search for `SM_StopSign`. Drag the stop sign from the **Content Browser** into the editing window. You can then change the position of the stop sign in the editing window or on the **Details** pane on the right, in the **Transform** section.

The Unreal Editor uses a left-hand *Z*-up coordinate system, where the *Y*-axis points to the right. Automated Driving Toolbox uses a right-hand *Z*-up coordinate system, where the *Y*-axis points to the left. When positioning objects in a scene, keep this coordinate system difference in mind. In the two coordinate systems, the positive and negative signs for the *Y*-axis and pitch angle values are reversed.

For more information on modifying scenes and adding assets, see Unreal Engine 4 Documentation.

To migrate assets from the `AutoVrtlEnv` project into your own project file, see Migrating Assets in the Unreal Engine documentation.

To obtain semantic segmentation data from a scene, then you must apply stencil IDs to the objects added to a scene. For more information, see “Apply Semantic Segmentation Labels to Custom Scenes” on page 6-54.

Run Simulation

Verify that the Simulink model and Unreal Editor are configured to co-simulate by running a test simulation.

1. In the Simulink model, click **Run**.

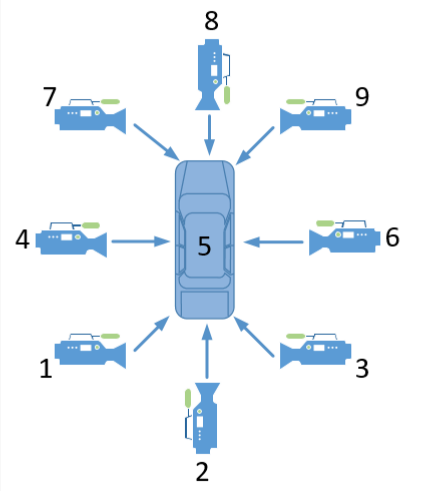
   Because the source of the scenes is the project opened in the Unreal Editor, the simulation does not start. Instead, you must start the simulation from the editor.
2. Verify that the Diagnostic Viewer window in Simulink displays this message:

   ```
   In the Simulation 3D Scene Configuration block, you set the scene source
   to 'Unreal Editor'. In Unreal Editor, select 'Play' to view the scene.
   ```

   This message confirms that Simulink has instantiated vehicles and other objects in the Unreal Engine 3D environment.
3. In the Unreal Editor, click **Play**. The simulation runs in the scene currently open in the Unreal Editor.

   - If your Simulink model contains vehicles, these vehicles drive through the scene that is open in the editor.
   - If your Simulink model includes sensors, these sensors capture data from the scene that is open in the editor.

To control the view of the scene during simulation, in the Simulation 3D Scene Configuration block, select the vehicle name from the **Scene view** parameter. To change the scene view as the simulation runs, use the numeric keypad in the editor. The table shows the position of the camera displaying the scene, relative to the vehicle selected in the **Scene view** parameter.

Key	Camera View	
1	Back left	
2	Back	
3	Back right	
4	Left	
5	Internal	
6	Right	
7	Front left	
8	Front	
9	Front right	
0	Overhead	

To restart a simulation, click **Run** in the Simulink model, wait until the Diagnostic Viewer displays the confirmation message, and then click **Play** in the editor. If you click **Play** before starting the simulation in your model, the connection between Simulink and the Unreal Editor is not established, and the editor displays an empty scene.

If you are co-simulating a custom project, to enable the numeric keypad, copy the `DefaultInput.ini` file from the support package installation folder to your custom project folder. For example, copy `DefaultInput.ini` from:

```
C:\ProgramData\MATLAB\SupportPackages\<MATLABRelease>\toolbox\shared\sim3dprojects\driving\AutoV
```

to:

```
C:\<yourproject>.project\Config
```

After tuning your custom scene based on simulation results, you can then package the scene into an executable. For more details, see “Package Custom Scenes into Executable” on page 6-52.

See Also

Simulation 3D Scene Configuration

More About

- “Apply Semantic Segmentation Labels to Custom Scenes” on page 6-54

Package Custom Scenes into Executable

When you finish modifying a custom scene, you can package the project file containing this scene into an executable. You can then configure your model to simulate from this executable by using the Simulation 3D Scene Configuration block. Executable files can improve simulation performance and do not require opening the Unreal Editor to simulate your scene. Instead, the scene runs by using the Unreal Engine that comes installed with Automated Driving Toolbox.

Package Scene into Executable Using Unreal Editor

1. Open the project containing the scene in the Unreal Editor. You must open the project from a Simulink model that is configured to co-simulate with the Unreal Editor. For more details on this configuration, see “Customize Scenes Using Simulink and Unreal Editor” on page 6-47.
2. In the Unreal Editor toolbar, select **Settings** > **Project Settings** to open the Project Settings window.
3. In the left pane, in the **Project** section, click **Packaging**.
4. In the **Packaging** section, set or verify the options in the table. If you do not see all these options, at the bottom of the **Packaging** section, click the **Show Advanced** expander

.

Packaging Option	Enable or Disable
Use Pak File	Enable
Cook everything in the project content directory (ignore list of maps below)	Disable
Cook only maps (this only affects cookall)	Enable
Create compressed cooked packages	Enable
Exclude editor content while cooking	Enable

5. Specify the scene from the project that you want to package into an executable.
   - a In the **List of maps to include in a packaged build** option, click the **Adds Element** button ✚.
   - b Specify the path to the scene that you want to include in the executable. By default, the Unreal Editor saves maps to the `/Game/Maps` folder. For example, if the `/Game/Maps` folder has a scene named `myScene` that you want to include in the executable, enter `/Game/Maps/myScene`.
   - c Add or remove additional scenes as needed.
6. Rebuild the lighting in your scenes. If you do not rebuild the lighting, the shadows from the light source in your executable file are incorrect and a warning about rebuilding the lighting displays during simulation. In the Unreal Editor toolbar, select **Build** > **Build Lighting Only**.
7. (Optional) If you plan to semantic segmentation data from the scene by using a Simulation 3D Camera block, enable rendering of the stencil IDs. In the left pane, in the **Engine** section, click **Rendering**. Then, in the main window, in the **Postprocessing** section, set **Custom Depth-Stencil Pass** to `Enabled with Stencil`. For more details on applying stencil IDs for semantic segmentation, see “Apply Semantic Segmentation Labels to Custom Scenes” on page 6-54.

8. Close the **Project Settings** window.
9. In the top-left menu of the editor, select **File** > **Package Project** > **Windows** > **Windows (64-bit)**. Select a local folder in which to save the executable, such as to the root of the project file (for example, `C:/Local/myProject`).

   Note Packaging a project into an executable can take several minutes. The more scenes that you include in the executable, the longer the packaging takes.

   Once packaging is complete, the folder where you saved the package contains a `WindowsNoEditor` folder that includes the executable file. This file has the same name as the project file.

   Note If you repackage a project into the same folder, the new executable folder overwrites the old one.

   Suppose you package a scene that is from the `myProject.uproject` file and save the executable to the `C:/Local/myProject` folder. The editor creates a file named `myProject.exe` with this path:

   ```
   C:/Local/myProject/WindowsNoEditor/myProject.exe
   ```

Simulate Scene from Executable in Simulink

1. In the Simulation 3D Scene Configuration block of your Simulink model, set the **Scene source** parameter to `Unreal Executable`.
2. Set the **File name** parameter to the name of your Unreal Editor executable file. You can either browse for the file or specify the full path to the file by using backslashes. For example:

   ```
   C:\Local\myProject\WindowsNoEditor\myProject.exe
   ```
3. Set the **Scene** parameter to the name of a scene from within the executable file. For example:

   ```
    /Game/Maps/myScene
   ```
4. Run the simulation. The model simulates in the custom scene that you created.

If you are simulating a scene from a project that is not based on the `AutoVtrlEnv` project, then the scene simulates in full screen mode. To use the same window size as the default scenes, copy the `DefaultGameUserSettings.ini` file from the support package installation folder to your custom project folder. For example, copy `DefaultGameUserSettings.ini` from:

```
C:\ProgramData\MATLAB\SupportPackages\<MATLABrelease>\toolbox\shared\sim3dprojects\automotive\Aut
```

to:

```
C:\<yourproject>.project\Config
```

Then, package scenes from the project into an executable again and retry the simulation.

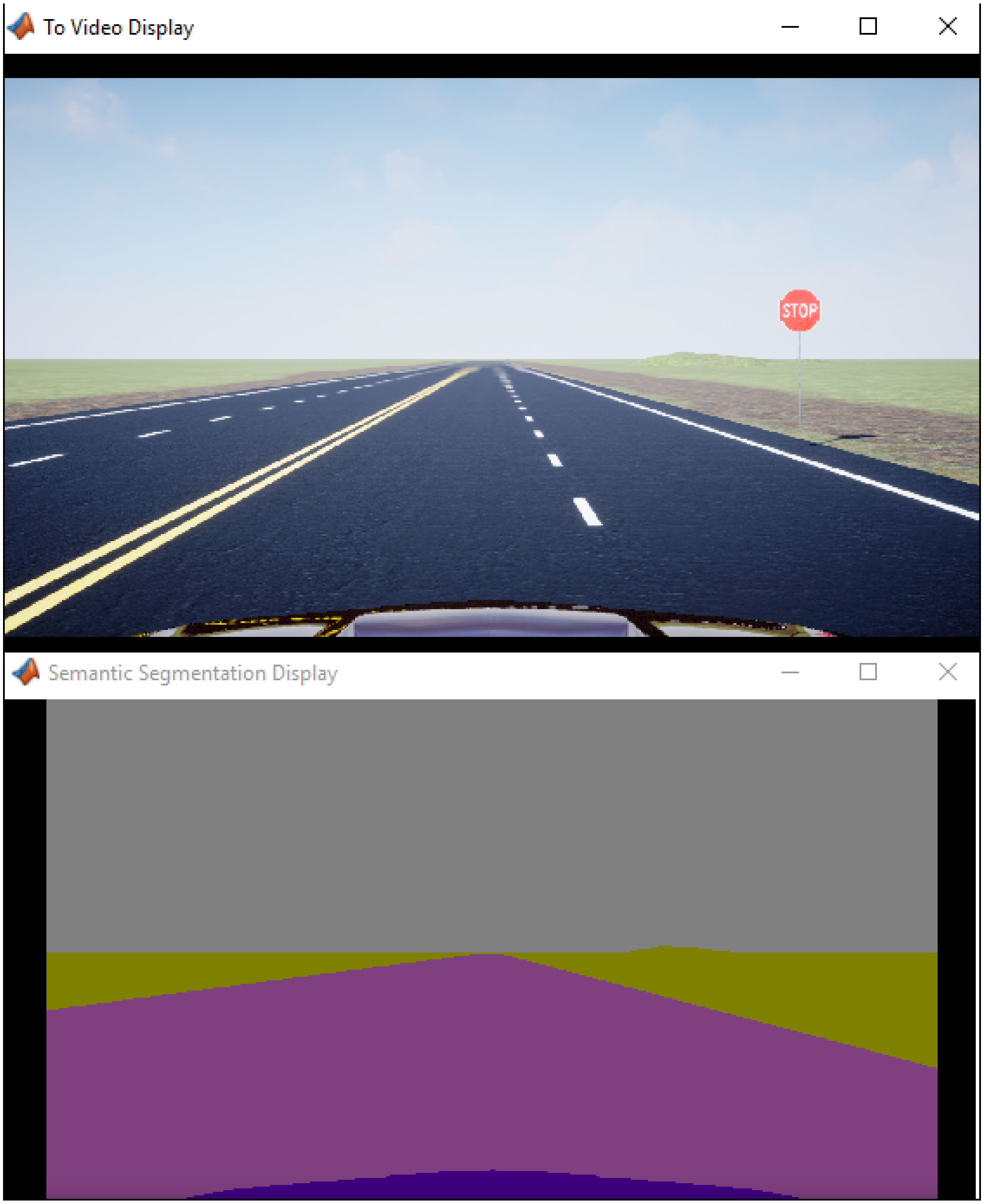
See Also

Simulation 3D Scene Configuration

Apply Semantic Segmentation Labels to Custom Scenes

The Simulation 3D Camera block provides an option to output semantic segmentation data from a scene. If you add new scene elements, or assets (such as traffic signs or roads), to a custom scene, then in the Unreal Editor, you must apply the correct ID to that element. This ID is known as a stencil ID. Without the correct stencil ID applied, the Simulation 3D Camera block does not recognize the scene element and does not display semantic segmentation data for it.

For example, this To Video Display window shows a stop sign that was added to a custom scene. The Semantic Segmentation Display window does not display the stop sign, because the stop sign is missing a stencil ID.



To apply a stencil ID label to a scene element, follow these steps:

1. Open the Unreal Editor from a Simulink model that is configured to simulate in the 3D environment. For more details, see “Customize Scenes Using Simulink and Unreal Editor” on page 6-47.
2. In the editor window, select the scene element with the missing stencil ID.
3. On the **Details** pane on the right, in the **Rendering** section, select **Render CustomDepth Pass**.

If you do not see this option, click the **Show Advanced** expander

to show all rendering options.

4. In the **CustomDepth Stencil Value** box, enter the stencil ID that corresponds to the asset. If you are adding an asset to a scene from the Automated Driving Toolbox Interface for Unreal Engine 4 Projects support package, then enter the stencil ID corresponding to that asset type, as shown in the table. If you are adding assets other than the ones shown, then you can assign them to unused IDs. If you do not assign a stencil ID to an asset, then the Unreal Editor assigns that asset an ID of `0`.

Note The Simulation 3D Camera block does not support the output of semantic segmentation data for lane markings. Even if you assign a stencil ID to lane markings, the block ignores this setting.

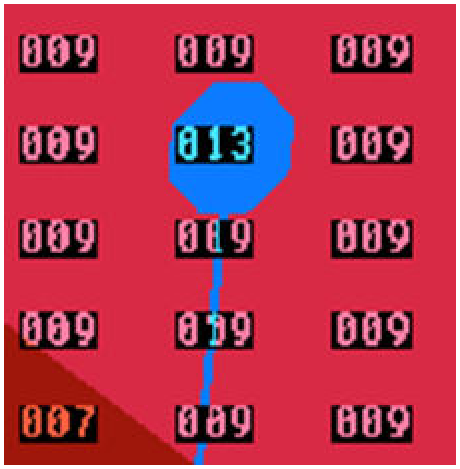
ID	Type
`0`	None/default
`1`	Building
`2`	*Not used*
`3`	Other
`4`	*Not used*
`5`	Pole
`6`	*Not used*
`7`	Road
`8`	Sidewalk
`9`	Vegetation
`10`	Vehicle
`11`	*Not used*
`12`	Generic traffic sign
`13`	Stop sign
`14`	Yield sign
`15`	Speed limit sign
`16`	Weight limit sign
`17-18`	*Not used*
`19`	Left and right arrow warning sign
`20`	Left chevron warning sign
`21`	Right chevron warning sign
`22`	*Not used*
`23`	Right one-way sign
`24`	*Not used*

ID	Type
25	School bus only sign
26-38	*Not used*
39	Crosswalk sign
40	*Not used*
41	Traffic signal
42	Curve right warning sign
43	Curve left warning sign
44	Up right arrow warning sign
45-47	*Not used*
48	Railroad crossing sign
49	Street sign
50	Roundabout warning sign
51	Fire hydrant
52	Exit sign
53	Bike lane sign
54-56	*Not used*
57	Sky
58	Curb
59	Flyover ramp
60	Road guard rail
61-66	*Not used*
67	Deer
68-70	*Not used*
71	Barricade
72	Motorcycle
73-255	*Not used*

For example, for a stop sign that is missing a stencil ID, enter 13.

Tip If you are adding stencil ID for scene elements of the same type, you can copy **(Ctrl+C)** and paste **(Ctrl+V)** the element with the added stencil ID. The copied scene element includes the stencil ID.

5 Visually verify that the correct stencil ID shows by using the custom stencil view. In the top-left corner of the editor window, click Lit and select **Buffer Visualization** > **Custom Stencil**. The scene displays the stencil IDs specified for each scene element. For example, if you added the correct stencil ID to a stop sign (13) then the editor window, the stop sign displays a stencil ID value of 13.

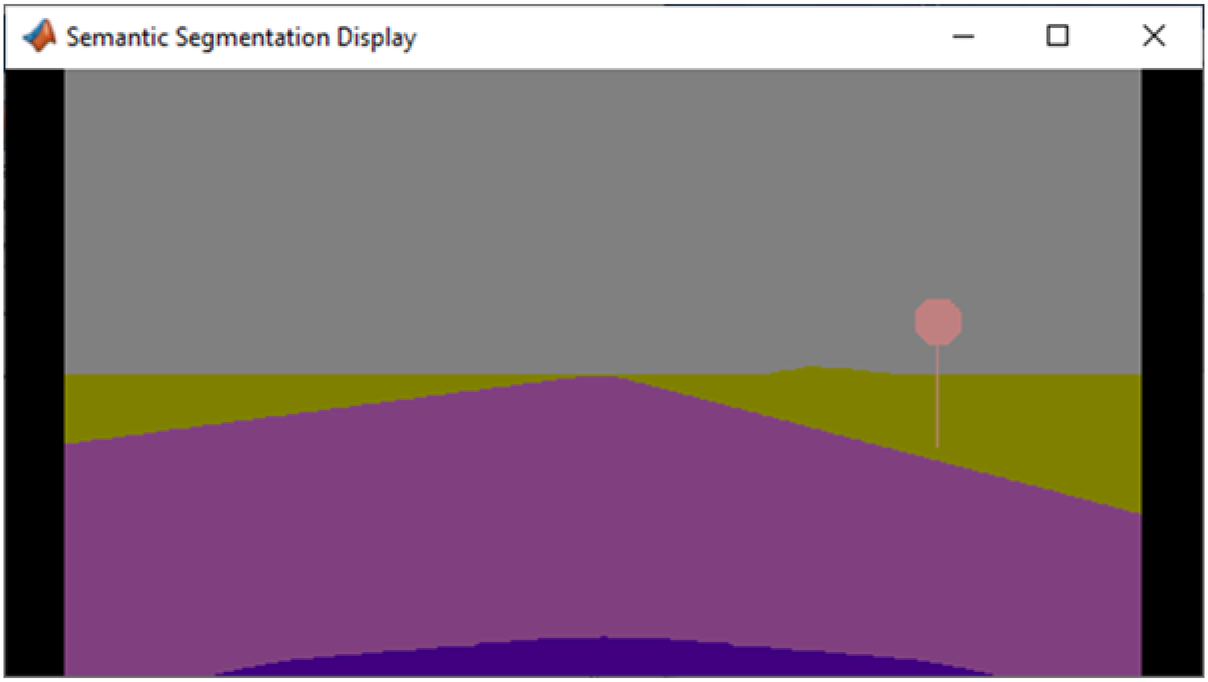


- If you did not set a stencil ID value for a scene element, then the element appears in black and displays no stencil ID.
- If you did not select **CustomDepth Stencil Value**, then the scene element does not appear at all in this view.

6 Turn off the custom stencil ID view. In the top-left corner of the editor window, click **Buffer Visualization** and then select **Lit**.

7 If you have not already done so, set up your Simulink model to display semantic segmentation data from a Simulation 3D Camera block. For an example setup, see “Visualize Depth and Semantic Segmentation Data in 3D Environment” on page 6-30.

8 Run the simulation and verify that the Simulation 3D Camera block outputs the correct data. For example, here is the Semantic Segmentation Display window with the correct stencil ID applied to a stop sign.



See Also

Simulation 3D Camera | Simulation 3D Scene Configuration

More About

- “Visualize Depth and Semantic Segmentation Data in 3D Environment” on page 6-30
- “Customize Scenes Using Simulink and Unreal Editor” on page 6-47