

Automated Driving Toolbox™

User's Guide



MATLAB® & SIMULINK®

R2019b



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

© COPYRIGHT 2017–2019 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

Trademarks

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

Patents

MathWorks products are protected by one or more U.S. patents. Please see www.mathworks.com/patents for more information.

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)

Sensor Configuration and Coordinate System Transformations

1

Coordinate Systems in Automated Driving Toolbox	1-2
World Coordinate System	1-2
Vehicle Coordinate System	1-2
Sensor Coordinate System	1-4
Spatial Coordinate System	1-7
Pattern Coordinate System	1-7
Calibrate a Monocular Camera	1-9
Estimate Intrinsic Parameters	1-9
Place Checkerboard for Extrinsic Parameter Estimation	1-9
Estimate Extrinsic Parameters	1-12
Configure Camera Using Intrinsic and Extrinsic Parameters	1-13

Ground Truth Labeling and Verification

2

Get Started with the Ground Truth Labeler	2-2
Load Unlabeled Data	2-2
Set Time Interval to Label	2-4
Create Label Definitions	2-4
Label Ground Truth	2-14
Export Labeled Ground Truth	2-18
Save App Session	2-22
Keyboard Shortcuts and Mouse Actions for Ground Truth Labeler	2-24
Label Definitions	2-24

Frame Navigation and Time Interval Settings	2-24
Labeling Window	2-25
Polyline Drawing	2-26
Polygon Drawing	2-26
Zooming	2-27
App Sessions	2-27

Tracking and Sensor Fusion

3

Visualize Sensor Data and Tracks in Bird's-Eye Scope	3-2
Open Model and Scope	3-2
Find Signals	3-2
Run Simulation	3-6
Organize Signal Groups (Optional)	3-8
Update Model and Rerun Simulation	3-9
Save and Close Model	3-9
Linear Kalman Filters	3-12
State Equations	3-12
Measurement Models	3-14
Linear Kalman Filter Equations	3-14
Filter Loop	3-15
Constant Velocity Model	3-16
Constant Acceleration Model	3-17
Extended Kalman Filters	3-19
State Update Model	3-19
Measurement Model	3-20
Extended Kalman Filter Loop	3-20
Predefined Extended Kalman Filter Functions	3-21

Planning, Mapping, and Control

4

Display Data on OpenStreetMap Basemap	4-2
--	------------

Access HERE HD Live Map Data	4-8
Step 1: Enter Credentials	4-8
Step 2: Create Reader Configuration	4-9
Step 3: Create Reader	4-10
Step 4: Read and Visualize Data	4-11
Enter HERE HD Live Map Credentials	4-15
Create Configuration for HERE HD Live Map Reader	4-17
Create Configuration for Specific Catalog	4-18
Create Configuration for Specific Version	4-21
Configure Reader	4-21
Create HERE HD Live Map Reader	4-23
Create Reader from Specified Driving Route	4-23
Create Reader from Specified Map Tile IDs	4-26
Read and Visualize Data Using HERE HD Live Map Reader .	4-27
Create Reader	4-28
Read Map Layer Data	4-30
Visualize Map Layer Data	4-36
HERE HD Live Map Layers	4-40
Road Centerline Model	4-41
HD Lane Model	4-43
HD Localization Model	4-45
Control Vehicle Velocity	4-46
Velocity Profile of Straight Path	4-49
Velocity Profile of Path with Curve and Direction Change ...	4-55

Driving Scenario Generation and Sensor Models

5

Build a Driving Scenario and Generate Synthetic Detections	5-2
Create a New Driving Scenario	5-2
Add a Road	5-2

Add Lanes	5-6
Add Vehicles	5-7
Add a Pedestrian	5-9
Add Sensors	5-11
Generate Synthetic Detections	5-14
Save Scenario	5-16
Prebuilt Driving Scenarios in Driving Scenario Designer ...	5-18
Choose a Prebuilt Scenario	5-18
Modify Scenario	5-37
Generate Synthetic Detections	5-38
Save Scenario	5-39
Euro NCAP Driving Scenarios in Driving Scenario Designer	
.....	5-41
Choose a Euro NCAP Scenario	5-41
Modify Scenario	5-57
Generate Synthetic Detections	5-58
Save Scenario	5-59
Import OpenDRIVE Roads into Driving Scenario	5-61
Import OpenDRIVE File	5-61
Inspect Roads	5-62
Add Actors and Sensors to Scenario	5-68
Generate Synthetic Detections	5-69
Save Scenario	5-70
Create Driving Scenario Variations Programmatically	5-73
Generate Sensor Detection Blocks Using Driving Scenario Designer	5-81
Test Open-Loop ADAS Algorithm Using Driving Scenario ...	5-93
Test Closed-Loop ADAS Algorithm Using Driving Scenario ..	5-99

3D Simulation for Automated Driving	6-2
3D Simulation Blocks	6-2
Algorithm Testing and Visualization	6-6
3D Simulation Environment Requirements and Limitations	6-8
Software Requirements	6-8
Minimum Hardware Requirements	6-8
Limitations	6-8
How 3D Simulation for Automated Driving Works	6-10
Communication with 3D Simulation Environment	6-10
Block Execution Order	6-10
Coordinate Systems	6-12
Coordinate Systems for 3D Simulation in Automated Driving Toolbox	6-13
World Coordinate System	6-13
Vehicle Coordinate System	6-16
Choose a Sensor for 3D Simulation	6-19
Simulate a Simple Driving Scenario and Sensor in 3D Environment	6-25
Visualize Depth and Semantic Segmentation Data in 3D Environment	6-35

Sensor Configuration and Coordinate System Transformations

- “Coordinate Systems in Automated Driving Toolbox” on page 1-2
- “Calibrate a Monocular Camera” on page 1-9

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 6-13.

World Coordinate System

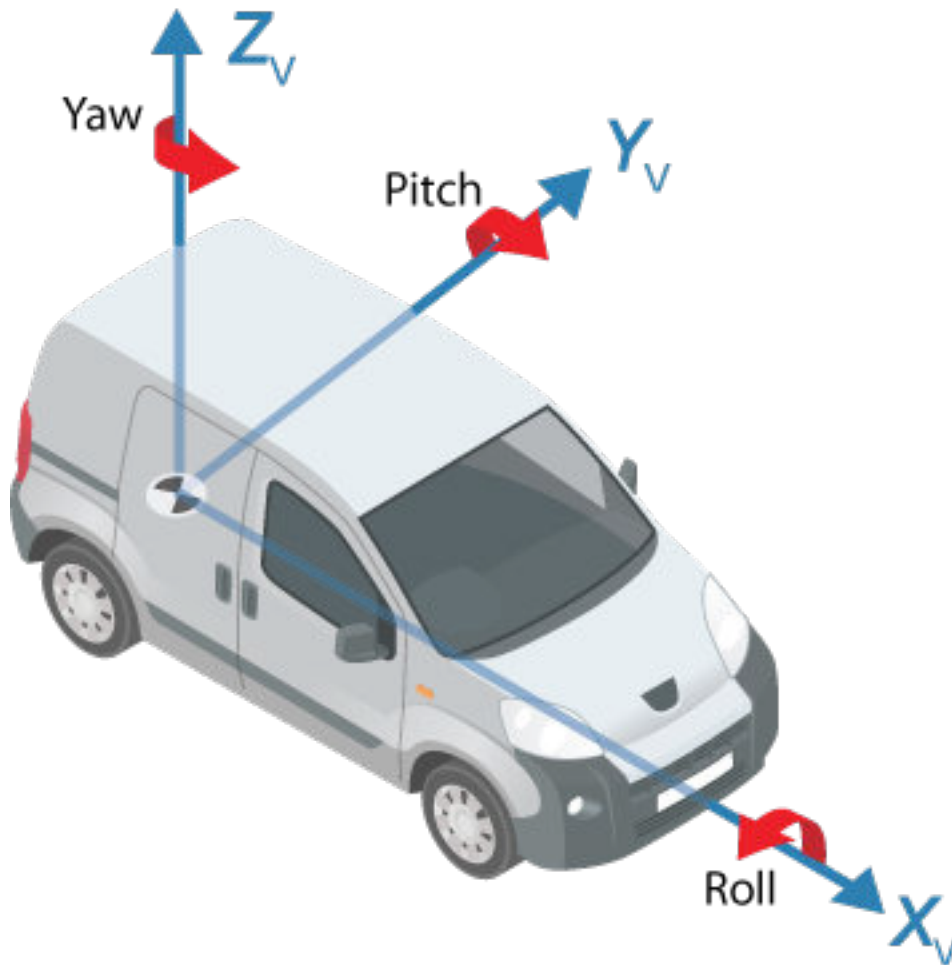
All vehicles, sensors, and their related coordinate systems are placed in the world coordinate system. A world coordinate system is important in global path planning, localization, mapping, and driving scenario simulation. Automated Driving Toolbox uses the right-handed Cartesian world coordinate system 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.



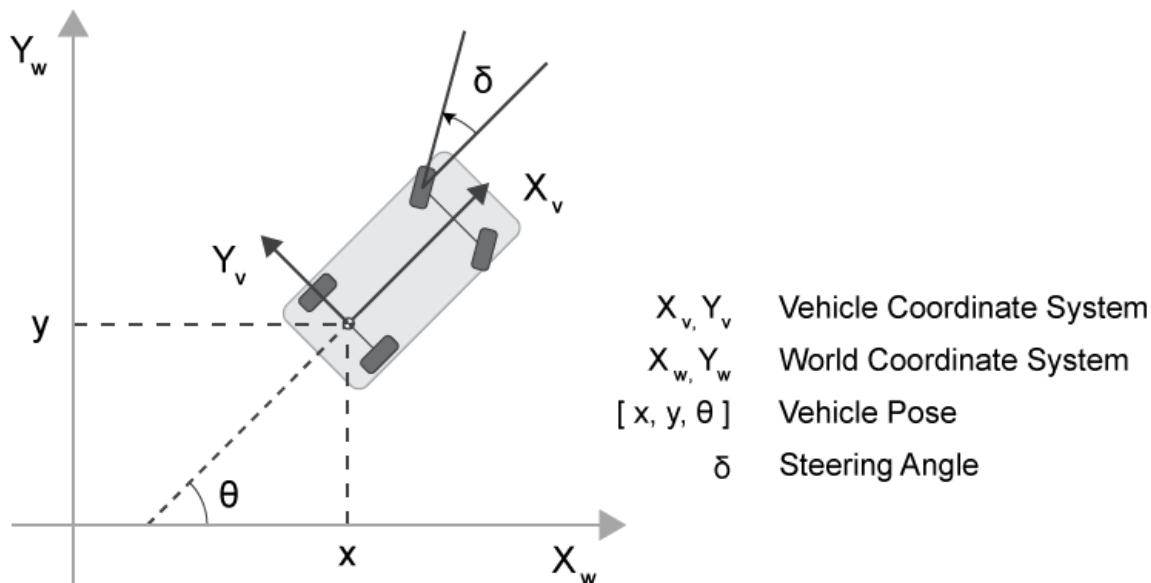
In most Automated Driving Toolbox functionality, such as cuboid driving scenario simulations and visual perception algorithms, the origin of the vehicle coordinate system is on the ground, below the midpoint of the rear axle. In 3D driving scenario simulations, the origin is on ground, below the longitudinal and lateral center of the vehicle. For more

details, see “Coordinate Systems for 3D Simulation in Automated Driving Toolbox” on page 6-13.

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

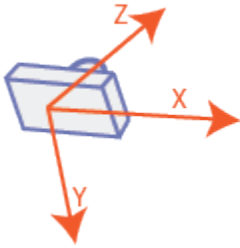
Values returned by individual sensors are transformed into the vehicle coordinate system so that they can be placed in a unified frame of reference.

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.

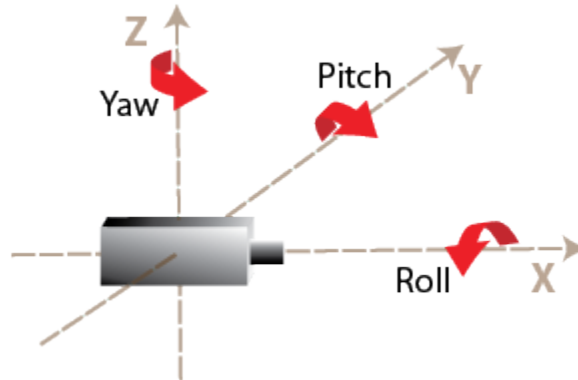


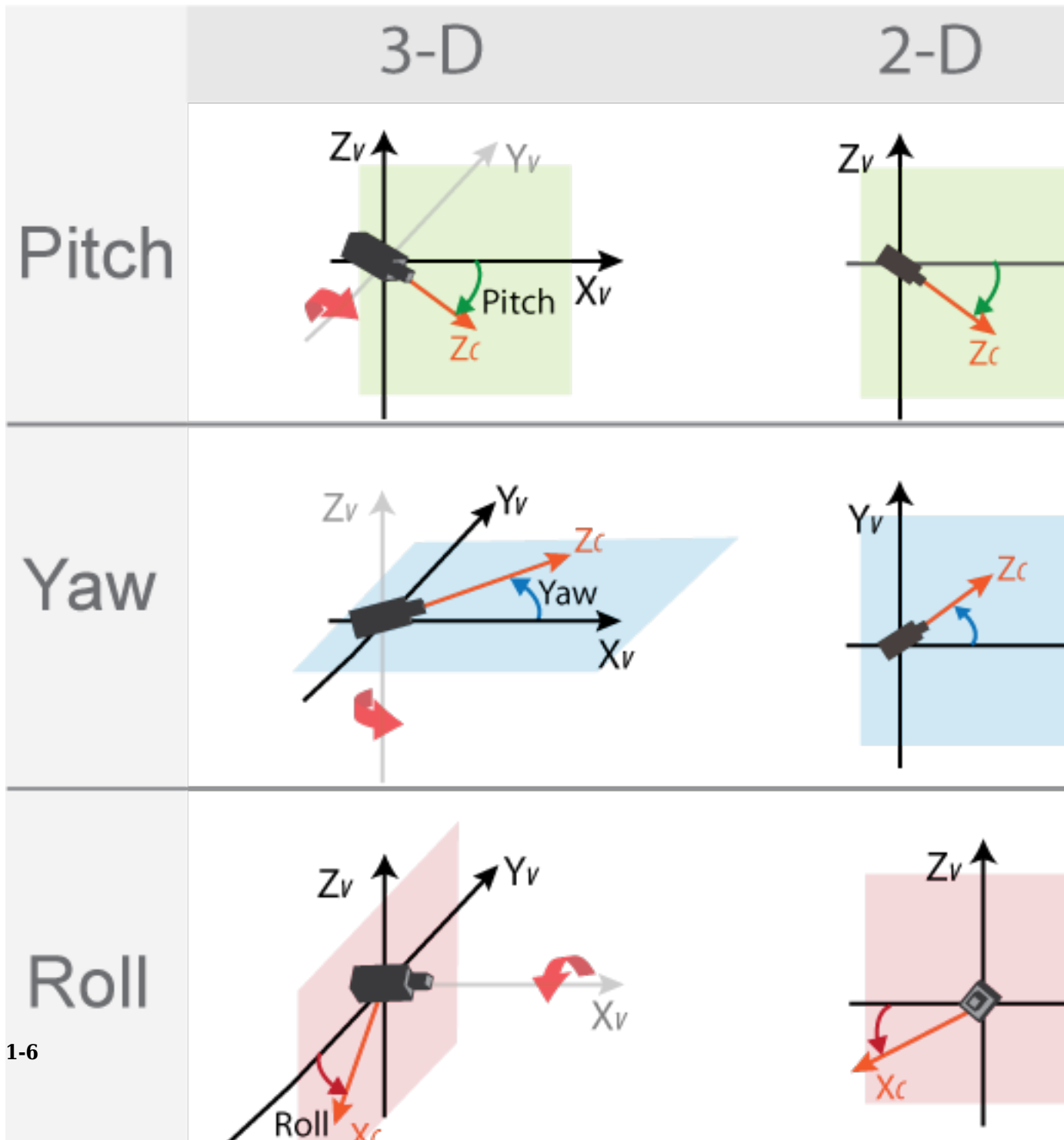
Sensor Coordinate System

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.



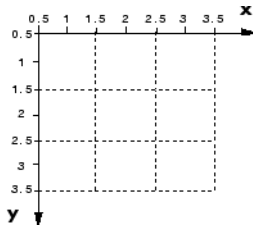
The yaw, pitch, and roll angles of sensors follow an ISO convention. These angles have positive clockwise directions when looking in the positive direction of the Z-, Y-, and X-axes, respectively.





Spatial Coordinate System

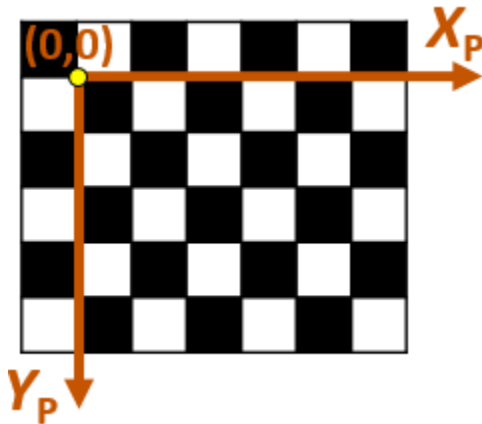
Spatial coordinates enable you to specify a location in an image with greater granularity than pixel coordinates. In the pixel coordinate system, a pixel is treated as a discrete unit, uniquely identified by an integer row and column pair, such as (3, 4). In the spatial coordinate system, locations in 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 6-13
- “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.

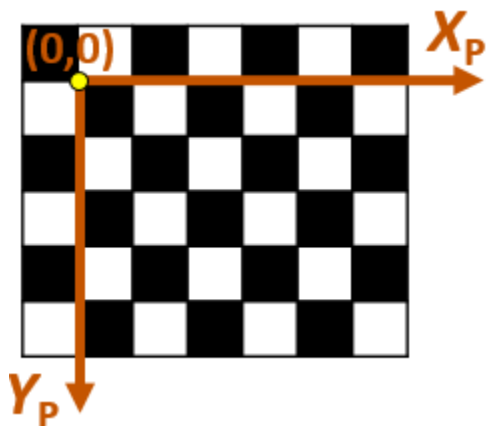
Alternatively, to better visualize the results, use the **Camera Calibrator** app. For information on setting up the camera, preparing the checkerboard pattern, and calibration techniques, see “Single Camera Calibrator App” (Computer Vision Toolbox).

Place Checkerboard for Extrinsic Parameter Estimation

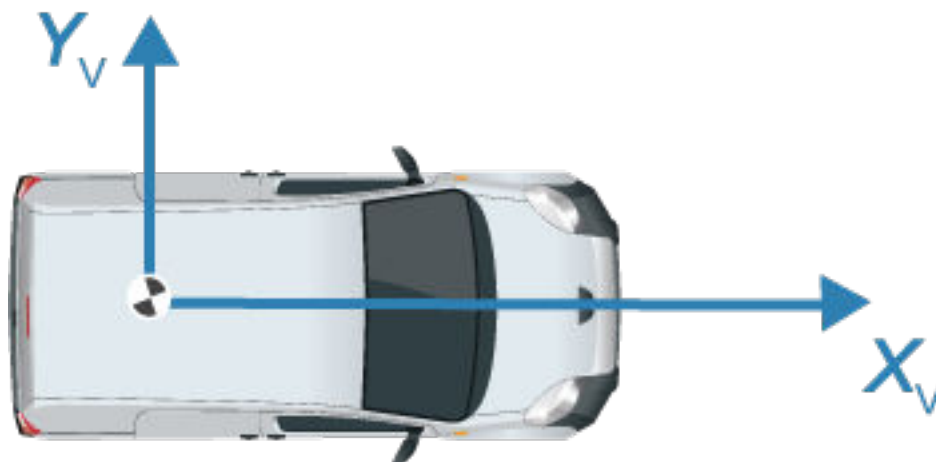
For a monocular camera mounted on a vehicle, the *extrinsic parameters* define the mounting position of that camera. These parameters include the rotation angles of the camera with respect to the vehicle coordinate system, and the height of the camera above the ground.

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.



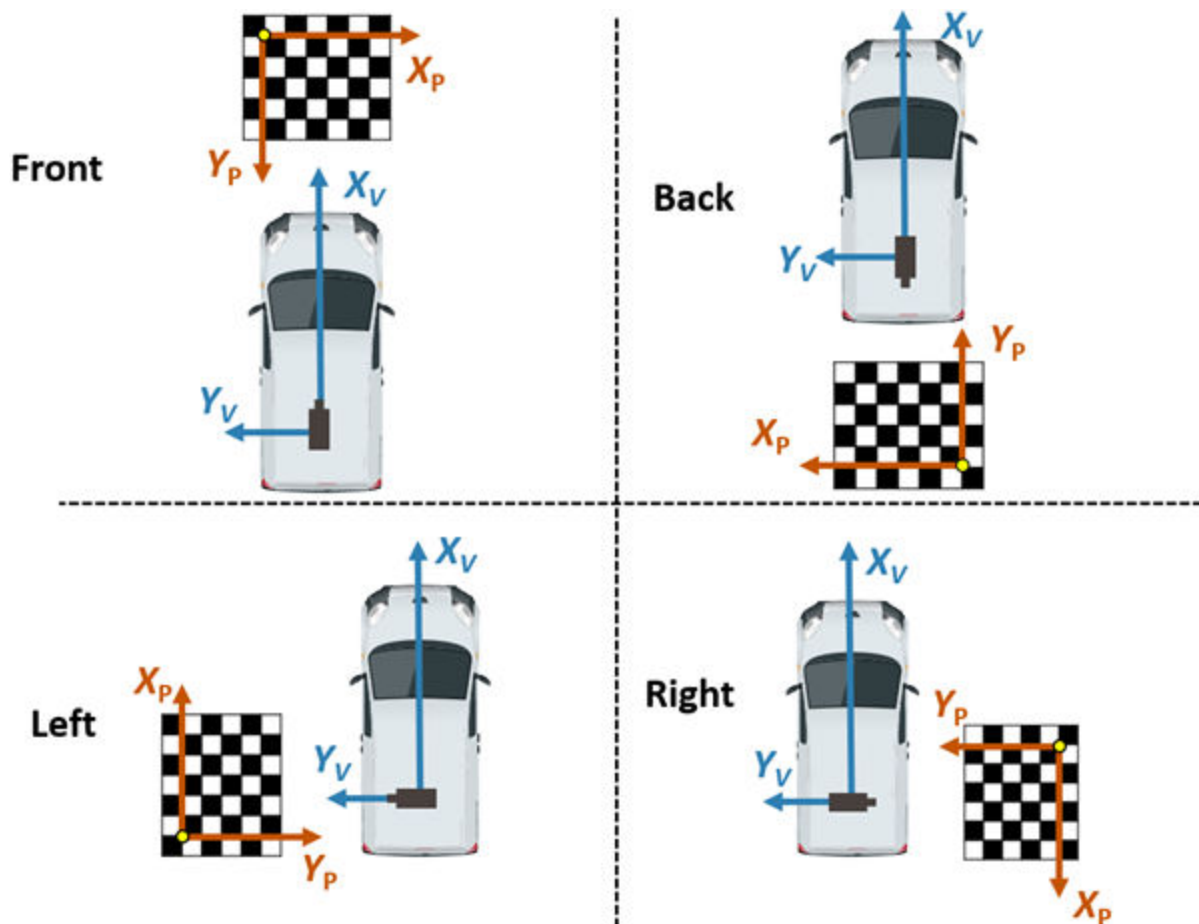
When placing the checkerboard pattern in relation to the vehicle, the X_P - and Y_P -axes must align with the X_V - and Y_V -axes of the vehicle. In the vehicle coordinate system, the X_V -axis points forward from the vehicle and the Y_V -axis points to the left, as viewed when facing forward. The origin 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.

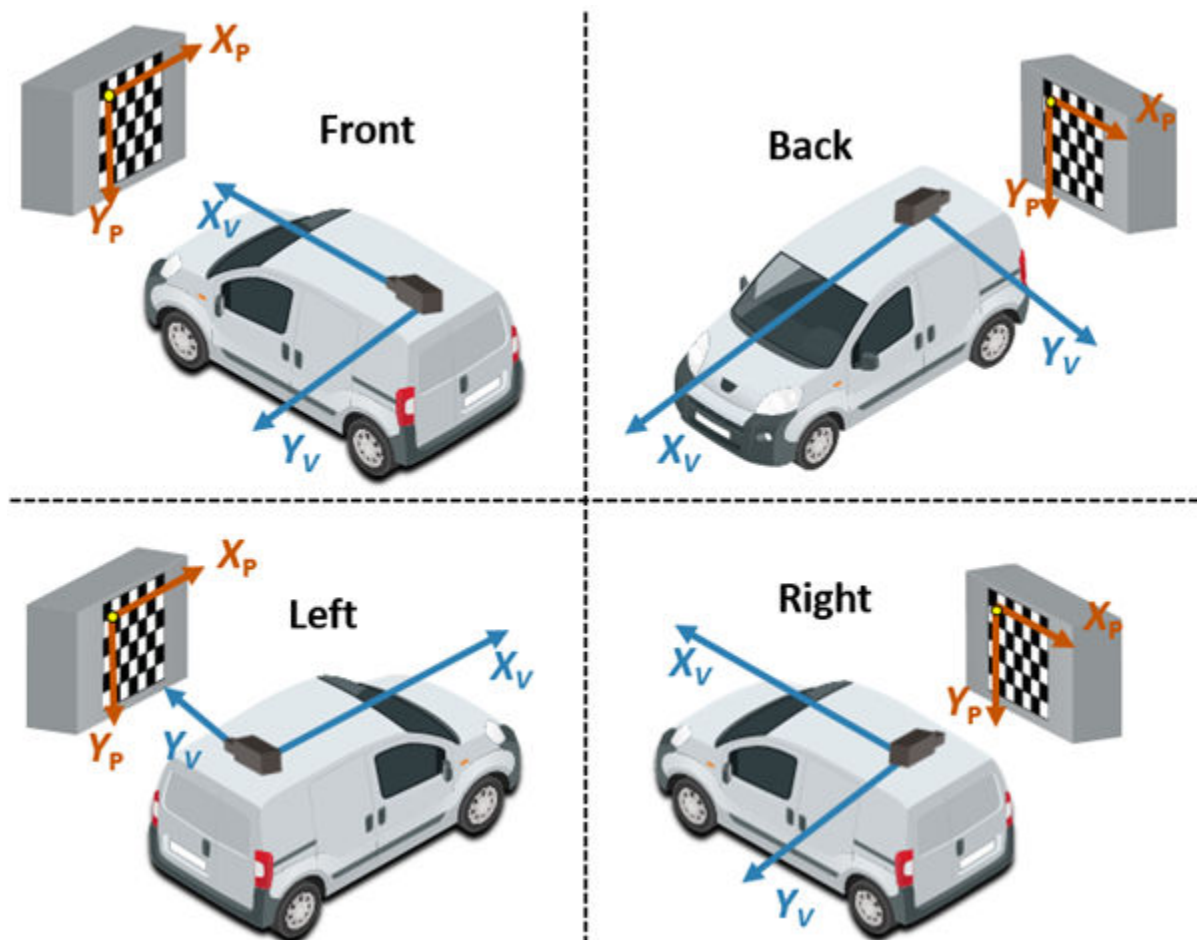
Horizontal Orientation

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 or 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)

Ground Truth Labeling and Verification

- “Get Started with the Ground Truth Labeler” on page 2-2
- “Keyboard Shortcuts and Mouse Actions for Ground Truth Labeler” on page 2-24

Get Started with the Ground Truth Labeler

The **Ground Truth Labeler** app provides an easy way to mark rectangular region of interest (ROI) labels, polyline ROI labels, pixel ROI labels, and scene labels in a video or image sequence. This example gets you started using the app by showing you how to:

- Manually label an image frame from a video.
- Automatically label across image frames using an automation algorithm.
- Export the labeled ground truth data.

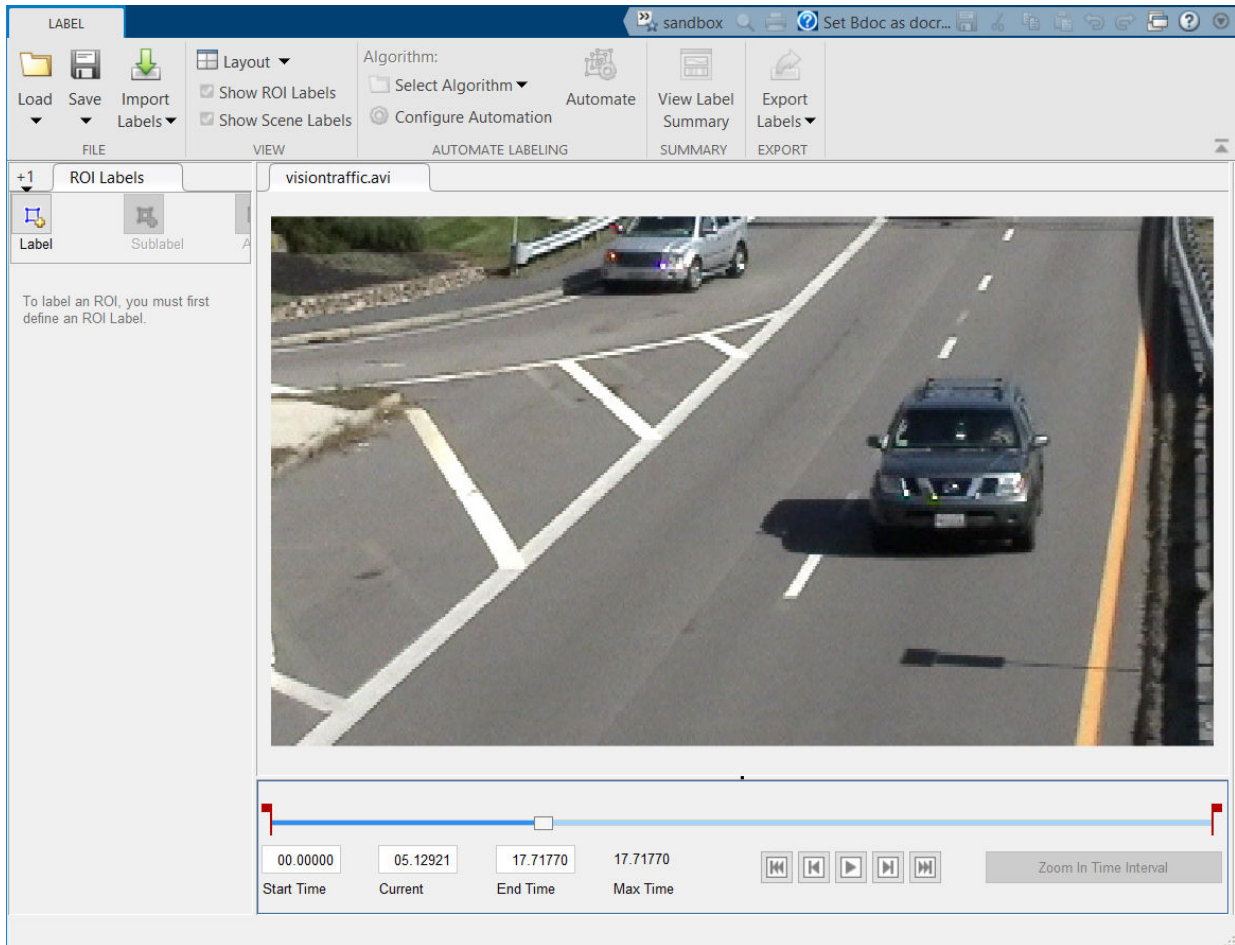
Load Unlabeled Data

Open the app and load a video of vehicles driving on a highway. Videos must be in a file format readable by `VideoReader`.

```
groundTruthLabeler('visiontraffic.avi')
```

Alternatively, open the app from the **Apps** tab, under **Automotive**. Then, from the **Load** menu, load a video data source.

Explore the video. Click the Play button  to play the entire video, or use the slider  to navigate between frames.



The app also enables you to load image sequences, with corresponding timestamps, by selecting **Load > Image Sequence**. The images must be readable by `imread`.

To load a custom data source that is readable by `VideoReader` or `imread`, see “Use Custom Data Source Reader for Ground Truth Labeling” (Computer Vision Toolbox).

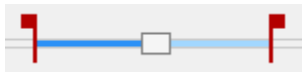
Set Time Interval to Label

You can label the entire video or start with a portion of the video. In this example, you label a five-second time interval within the loaded video. In the text boxes below the video, enter these times in seconds:

- 1 In the **Current Time** box, type 5 and press **Enter**.
- 2 In the **Start Time** box, type 5 so that the slider is at the start of the time interval.
- 3 In the **End Time** box, type 10.

05.00000	05.00000	10.00000
Start Time	Current	End Time

Optionally, to make adjustments to the time interval, click and drag the red interval flags.



The entire app is now set up to focus on this specific time interval. The video plays only within this interval, and labeling and automation algorithms apply only to this interval. You can change the interval at any time by moving the flags.



To expand the time interval to fill the entire playback section, click **Zoom in Time Interval**.


Create Label Definitions

Define the labels you intend to draw. In this example, you define labels directly within the app. To define labels from the MATLAB® command line instead, use the `labelDefinitionCreator`.

Create ROI Labels

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

ROI Label	Description	Example: Driving Scene
<p>Rectangle</p>	<p>Draw rectangular ROI labels (bounding boxes) around objects.</p>	<p>Vehicles, pedestrians, road signs</p> 
<p>Line</p>	<p>Draw linear ROI labels to represent lines. To draw a polyline ROI, use two or more points.</p>	<p>Lane boundaries, guard rails, road curbs</p> 

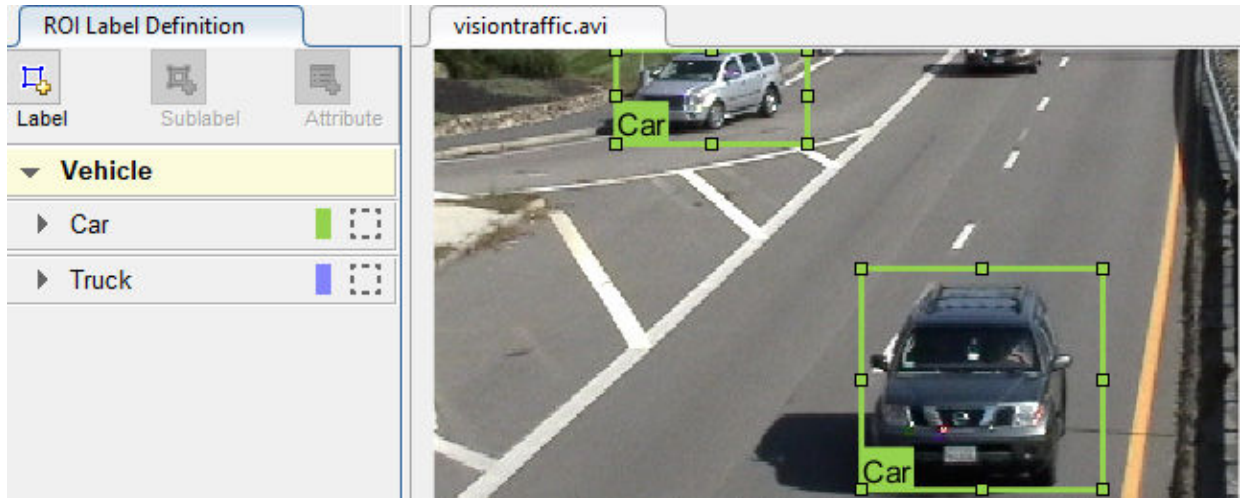
ROI Label	Description	Example: Driving Scene
Pixel label	Assign labels to pixels for semantic segmentation. You can label pixels manually using polygons, brushes, or flood fill. See “Label Pixels for Semantic Segmentation” (Computer Vision Toolbox).	Vehicles, road surface, trees, pavement 

In this example, you define a **vehicle** group for labeling types of vehicles, and then create a **Rectangle** ROI label for a **Car** and a **Truck**.

- 1 In the **ROI Label Definition** pane on the left, click **Label**.
- 2 Create a **Rectangle** label named **Car**.
- 3 From the **Group** drop-down menu, select **New Group** and name the group **Vehicle**
- 4 Click **OK**.

The **Vehicle** group name appears in the **ROI Label Definition** pane with the label **Car** created. You can move a labels to a different position or group by left-clicking and dragging the label.

- 5 Add a second label. Click **Label**. Name the label **Truck** and make sure the **Vehicle** group is selected. Click **OK**.
- 6 In the first video frame within the time interval, use the mouse to draw rectangular **Car** ROIs around the two vehicles.



Create Sublabels

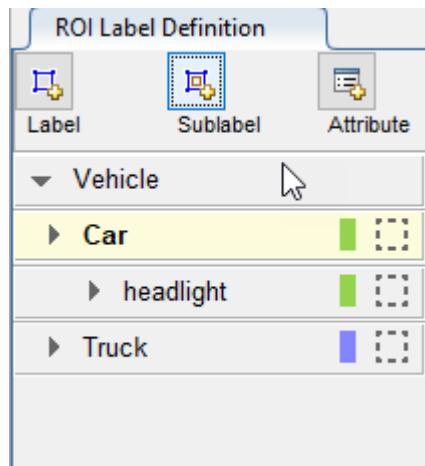
A sublabel is a type of ROI label that corresponds to a parent ROI label. Each sublabel must belong to, or be a child of, a specific label defined in the **ROI Label Definition** pane. For example, in a driving scene, a vehicle label might have sublabels for headlights, license plates, or wheels.

Define a sublabel for headlights.

- 1 In the **ROI Label Definition** pane on the left, click the **Car** label.
- 2 Click **Sublabel**.
- 3 Create a **Rectangle** sublabel named **headlight** and optionally write a description. Click **OK**.

The **headlight** sublabel appears in the **ROI Label Definition** pane. The sublabel is nested under the selected ROI label, **Car**, and has the same color as its parent label.

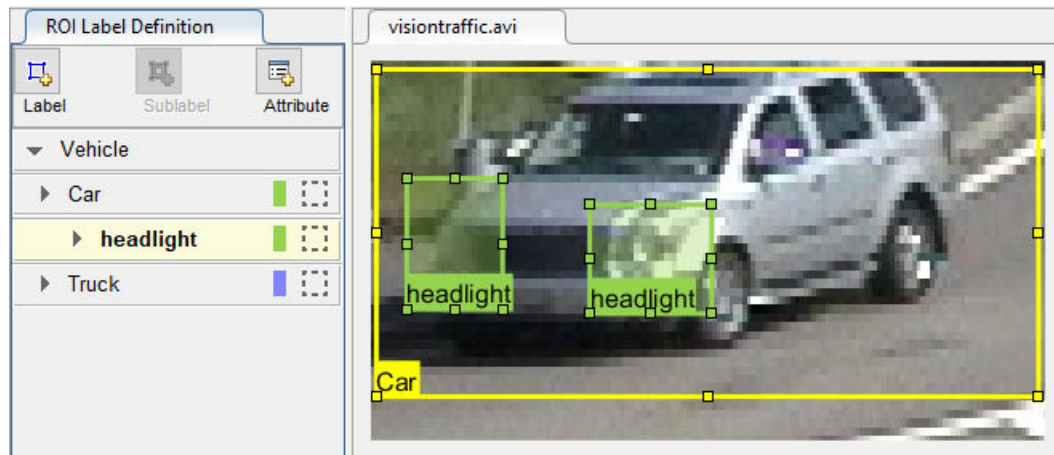
You can add multiple sublabels under a label. You can also drag-and-drop the sublabels to reorder them in the list. Right-click any label for additional edits.



- 4 In the **ROI Label Definition** pane, select the **headlight** sublabel.
- 5 In the video frame, select the **Car** label. The label turns yellow when selected. You must select the **Car** label (parent ROI) before you can add a sublabel to it.

Draw **headlight** sublabels for each of the cars.

- 6 Repeat the previous steps to label the headlights of the other car. To draw the labels more precisely, use the **Pan**, **Zoom In**, and **Zoom Out** options available from the toolbar.




Sublabels can only be used with rectangular or polyline ROI labels and cannot have their own sublabels. For more details on working with sublabels, see “Use Sublabels and Attributes to Label Ground Truth Data” (Computer Vision Toolbox).

Create Attributes

An attribute provides further categorization of an ROI label or sublabel. Attributes specify additional information about a drawable label. For example, in a driving scene, attributes might include the type or color of a vehicle.

You can define these types of attributes.

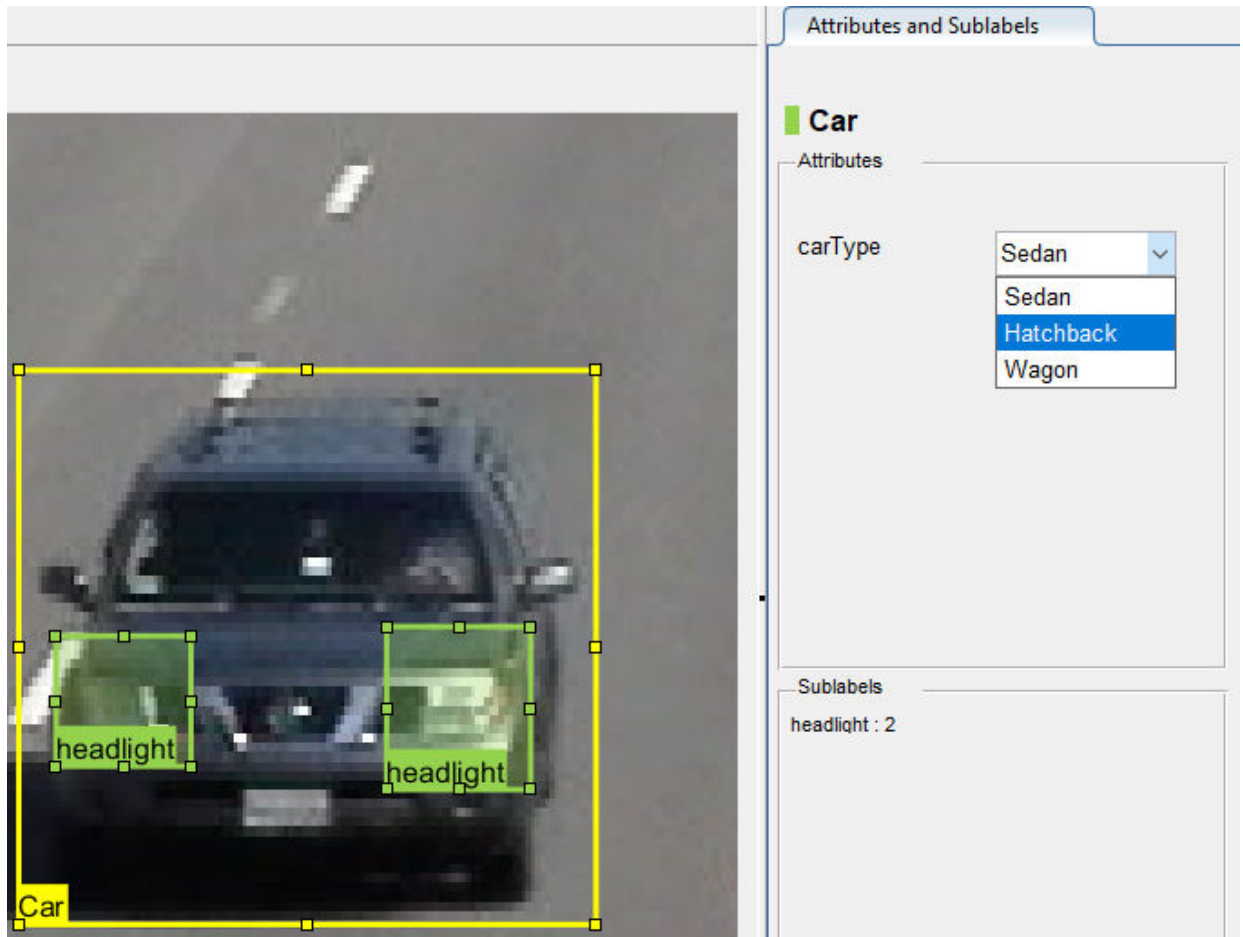
Attribute Type	Sample Attribute Definition	Sample Default Values
Numeric Value	<p>Attribute Name</p> <input type="text" value="numDoors"/> <p>Default Scalar Value (Optional)</p> <input type="text" value="4"/>	
String	<p>Attribute Name</p> <input type="text" value="color"/> <p>Default Value (Optional)</p> <input type="text"/>	<p>String</p> <input type="text"/>
Logical	<p>Attribute Name</p> <input type="text" value="inMotion"/> <p>Default Value (Optional)</p> <input type="text" value="True"/>	<p>Logical</p> <input type="text"/>

Attribute Type	Sample Attribute Definition	Sample Default Values
List	<p>Attribute Name</p> <input data-bbox="635 388 946 427" type="text" value="carType"/> <p>List Items (Each item must appear)</p> <input data-bbox="635 493 946 649" type="text" value="Sedan
Hatchback
Wagon"/>	<p>Attributes and Sublabels</p> <p>Car</p> <p>Attributes</p> <p>carMake <input data-bbox="1228 583 1426 621" type="text" value="Nissan"/></p> <p>inMotion <input data-bbox="1228 666 1426 704" type="text" value="True"/> ▼</p> <p>color <input data-bbox="1228 749 1426 788" type="text" value="Blue"/></p> <p>numDoors <input data-bbox="1228 833 1426 871" type="text" value="4"/></p> <p>carType <input data-bbox="1228 916 1426 1079" type="text" value="Sedan"/> ▼</p> <p>Sedan</p> <p>Hatchback</p> <p>Wagon</p>

Add an attribute for the vehicle type.

- 1 In the **ROI Label Definition** pane on the left, select the **Car** label and click **Attribute**.
- 2 In the **Attribute Name** box, type carType. Set the attribute type to List.
- 3 In the **List Items** section, type different types of cars, such as Sedan, Hatchback, and Wagon, each on its own line. Optionally give the attribute a description, and click **OK**.

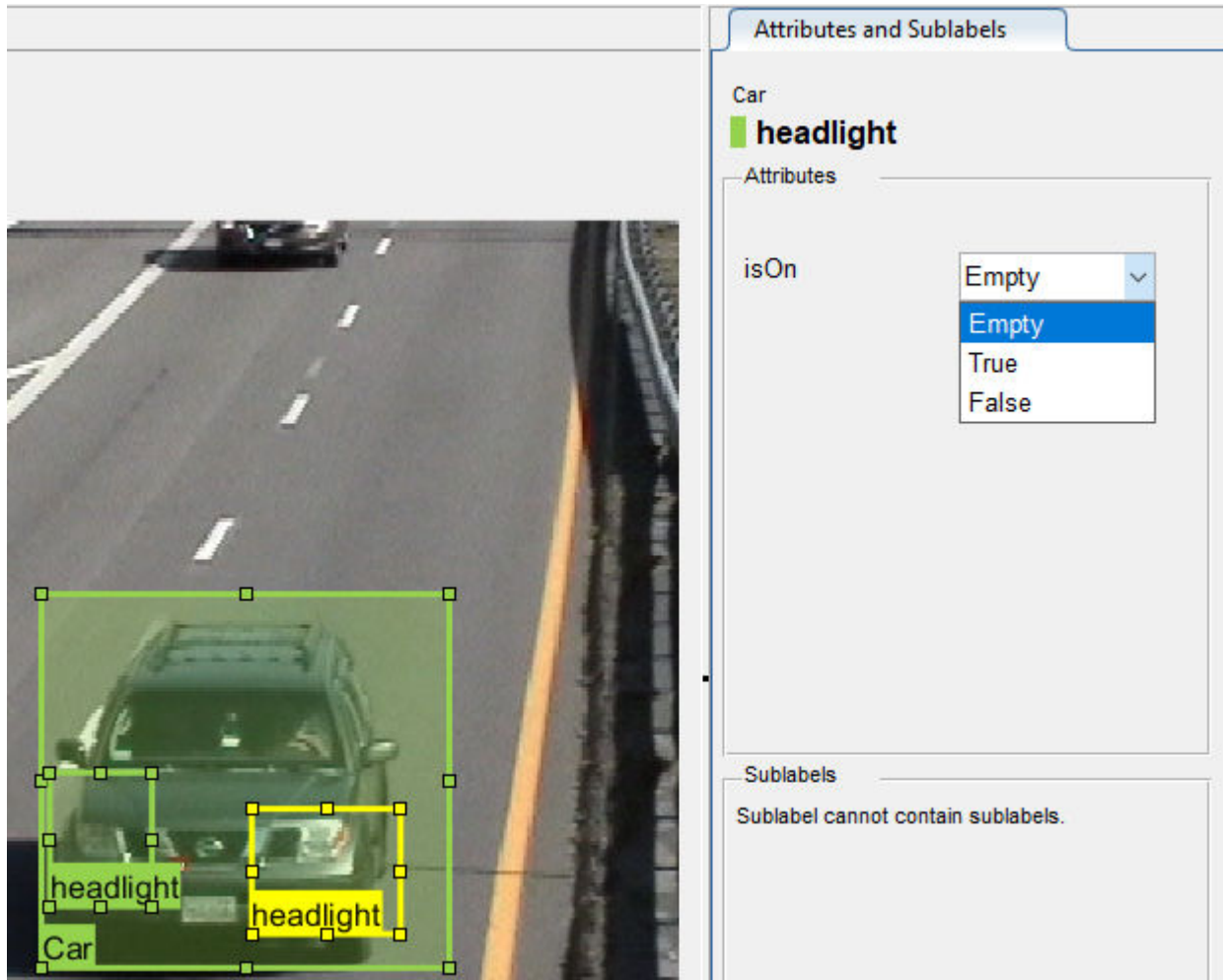
- 4 In the first frame of the video, select a **Car** ROI label. In the **Attributes and Sublabels** pane, select the appropriate **carType** attribute value for that vehicle.
- 5 Repeat the previous step to assign a **carType** attribute to the other vehicle.



You can also add attributes to sublabels. Add an attribute for the **headlight** sublabel that tells whether the headlight is on.

- 1 In the **ROI Label Definition** pane on the left, select the **headlight** sublabel and click **Attribute**.

- 2 In the **Attribute Name** box, type `isOn`. Set the attribute type to `Logical`. Leave the **Default Value** set to `Empty`, optionally write a description, and click **OK**.
- 3 Select a headlight in the video frame. Set the appropriate **isOn** attribute value, or leave the attribute value set to `Empty`.
- 4 Repeat the previous step to set the **isOn** attribute for the other headlights.



The image shows a video frame of a car on a road. The car is labeled with a green bounding box and the text "Car". Two headlights are labeled with yellow bounding boxes and the text "headlight".

The software interface on the right is titled "Attributes and Sublabels". It shows the following information:

- Car
- headlight
- Attributes
 - isOn: A dropdown menu with options: Empty (selected), Empty, True, False.
- Sublabels
 - Sublabel cannot contain sublabels.

To delete an attribute, right-click an ROI label or sublabel, and select the attribute to delete. Deleting the attribute removes attribute information from all previously created ROI label annotations.

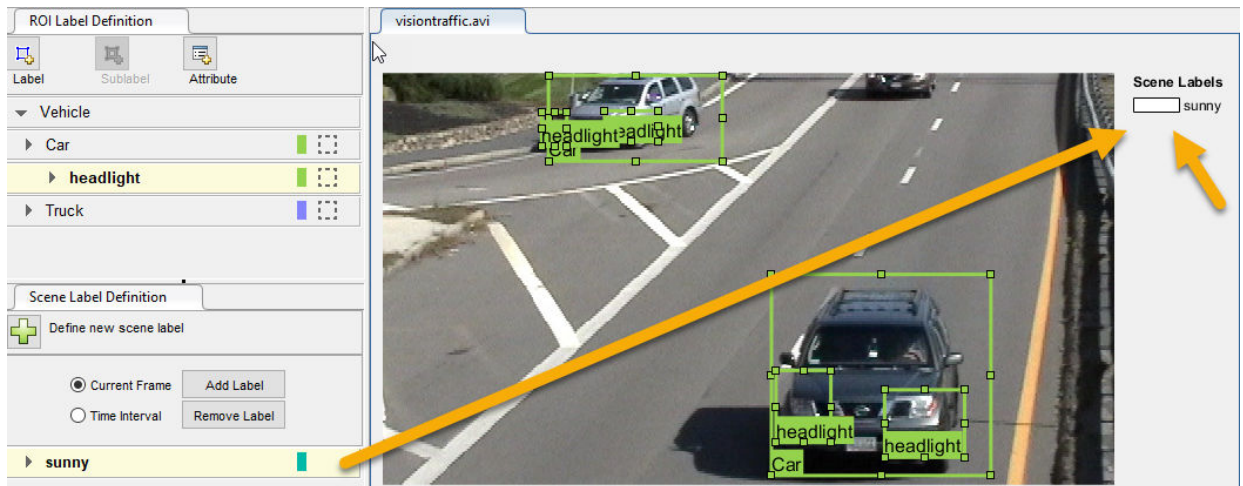
Create Scene Labels

A scene label defines additional information for the entire scene. Use scene labels to describe conditions, such as lighting and weather, or events, such as lane changes.

Create a scene label to use in the video.

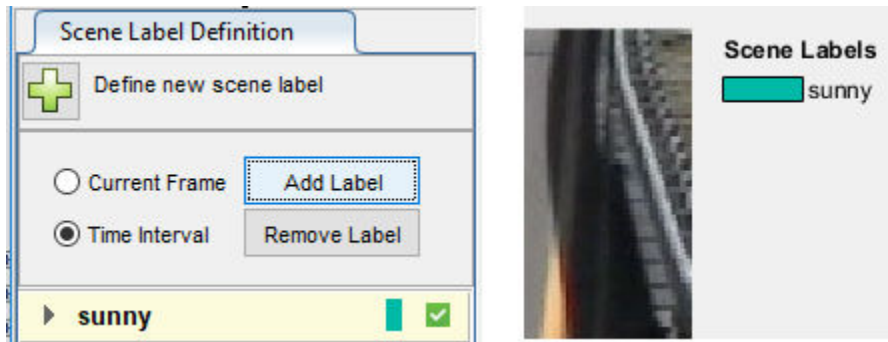
- 1 In the **Scene Label Definition** pane on the left, click the **Define new scene label** button, and create a scene label named **sunny**. Make sure **Group** is set to **None**. Click **OK**.

The **Scene Label Definition** pane shows the scene label definition. The scene labels that are applied to the current frame appear in the **Scene Labels** pane on the right. The **sunny** scene label is empty (white), because the scene label has not yet been applied to the frame.



- 2 The entire scene is sunny, so specify to apply the **sunny** scene label over the entire time interval. With the **sunny** scene label definition still selected in the **Scene Label Definition** pane, select **Time Interval**.
- 3 Click **Add Label**.

The **sunny** label now applies to all frames in the time interval.



Label Ground Truth

So far, you have labeled only one frame in the video. To label the remaining frames, choose one of these options.

Label Ground Truth Manually

When you click the right arrow key to advance to the next frame, the ROI labels from the previous frame do not carry over. Only the **sunny** scene label applies to each frame, because this label was applied over the entire time interval.

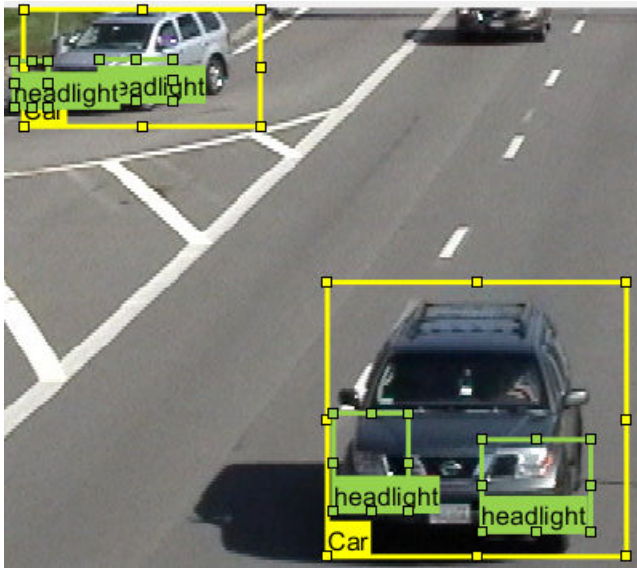
Advance frame by frame and draw the label and sublabel ROIs manually. Also update the attribute information for these ROIs.

Label Ground Truth Using Automation Algorithm

To speed up the labeling process, you can use an automation algorithm within the app. You can either define your own automation algorithm, see “Create Automation Algorithm for Labeling” (Computer Vision Toolbox) and “Temporal Automation Algorithms” (Computer Vision Toolbox), or use a built-in automation algorithm. In this example, you label the ground truth using a built-in point tracking algorithm.

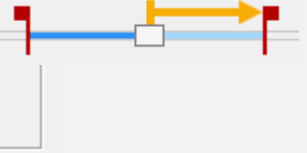
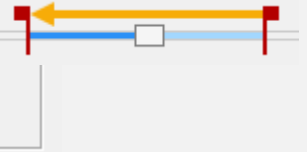
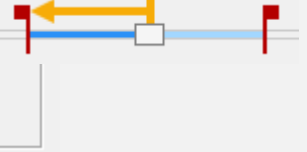
In this example, you automate the labeling of only the **Car** ROI labels. The built-in automation algorithms do not support sublabel and attribute automation.

- 1 Select the labels you want to automate. In the first frame of the video, press **Ctrl** and click to select the two **Car** label annotations. The labels are highlighted in yellow.

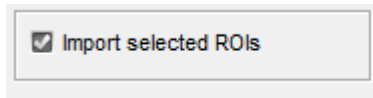


- 2 From the app toolstrip, select **Select Algorithm > Point Tracker**. This algorithm tracks one or more rectangle ROIs over short intervals using the Kanade-Lucas-Tomasi (KLT) algorithm.
- 3 (optional) Configure the automation settings. Click **Configure Automation**. By default, the automation algorithm applies labels from the start of the time interval to the end. To change the direction and start time of the algorithm, choose one of the options shown in this table.

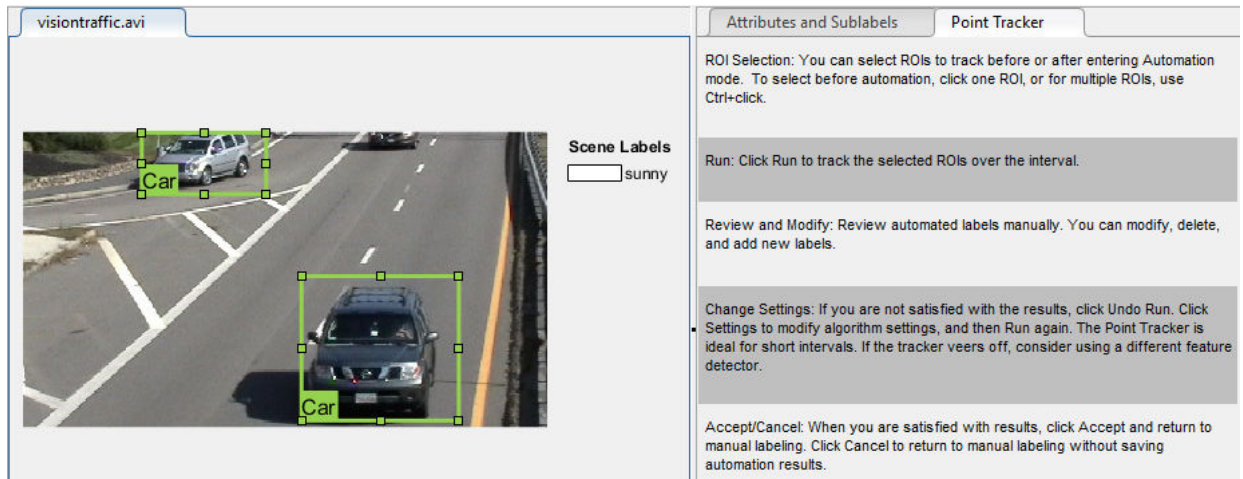
Direction of automation	Run automation from	Example

Direction of automation	Run automation from	Example
	Run automation from: <input type="radio"/> End time to Start time <input checked="" type="radio"/> Current time to Start time	
Direction of automation: <input type="radio"/> Forward <input checked="" type="radio"/> Reverse	Run automation from: <input checked="" type="radio"/> Start time to End time <input type="radio"/> Current time to End time	
	Run automation from: <input type="radio"/> End time to Start time <input checked="" type="radio"/> Current time to Start time	

The **Import selected ROIs** must be selected so that the **Car** labels you selected are imported into the automation session.



- 4 Click **Automate** to open an automation session. The algorithm instructions appear in the right pane, and the selected labels are available to automate.



- 5 Click **Run** to track the selected ROIs over the interval.
- 6 Examine the results of running the algorithm.

The vehicles that enter the scene later are unlabeled. The unlabeled vehicles did not have an initial ROI label, so the algorithm did not track them. Click **Undo Run**. Use the slider to find the frames where each vehicle first appears. Draw **vehicle** ROIs around each vehicle, and then click **Run** again.

- 7 Advance frame by frame and manually move, resize, delete, or add ROIs to improve the results of the automation algorithm.

When you are satisfied with the algorithm results, click **Accept**. Alternatively, to discard labels generated during the session and label manually instead, click **Cancel**. The **Cancel** button cancels only the algorithm session, not the app session.

Optionally, you can now manually label the remaining frames with sublabel and attribute information.

To further evaluate your labels, you can view a visual summary of the labeled ground truth. From the app toolbar, select **View Label Summary**. Use this summary to compare the frames, frequency of labels, and scene conditions. For more details, see “View Summary of Ground Truth Labels” (Computer Vision Toolbox). This summary does not support sublabels or attributes.

Export Labeled Ground Truth

You can export the labeled ground truth to a MAT-file or to a variable in the MATLAB workspace. In both cases, the labeled ground truth is stored as a `groundTruth` object. You can use this object to train a deep-learning-based computer vision algorithm. For more details, see “Training Data for Object Detection and Semantic Segmentation” (Computer Vision Toolbox).

Note If you export pixel data, the pixel label data and ground truth data are saved in separate files but in the same folder. For considerations when working with exported pixel labels, see “How Labeler Apps Store Exported Pixel Labels” (Computer Vision Toolbox).

In this example, you export the labeled ground truth to the MATLAB workspace. From the app toolstrip, select **Export Labels > To Workspace**. The exported MATLAB variable, `gTruth`, is a `groundTruth` object.

Display the properties of the exported `groundTruth` object. The information in your exported object might differ from the information shown here.

```
gTruth
```

```
gTruth =
```

```
groundTruth with properties:
```

```
DataSource: [1x1 groundTruthDataSource]  
LabelDefinitions: [3x5 table]  
LabelData: [531x3 timetable]
```

Data Source

`DataSource` is a `groundTruthDataSource` object containing the path to the video and the video timestamps. Display the properties of this object.

```
gTruth.DataSource
```

```
ans =
```

```
groundTruthDataSource for a video file with properties
```

```
Source: ...matlab\toolbox\vision\visiondata\visiontraffic.avi  
TimeStamps: [531x1 duration]
```

Label Definitions

`LabelDefinitions` is a table containing information about the label definitions. This table does not contain information about the labels that are drawn on the video frames. To save the label definitions in their own MAT-file, from the app toolstrip, select **Save > Label Definitions**. You can then import these label definitions into another app session by selecting **Import Files**.

Display the label definitions table. Each row contains information about an ROI label definition or a scene label definition. If you exported pixel label data, the `LabelDefinitions` table also includes a `PixelLabelID` column containing the ID numbers for each pixel label definition.

```
gTruth.LabelDefinitions
```

```
ans =
```

```
3x5 table
```

Name	Type	Group	Description	Hierarchy
'Car'	Rectangle	'Vehicle'	''	[1x1 struct]
'Truck'	Rectangle	'Vehicle'	''	[]
'sunny'	Scene	'None'	''	[]

Within `LabelDefinitions`, 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.

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

```
ans =
```

```
struct with fields:
```

```
    carType: [1x1 struct]
    headlight: [1x1 struct]
    Type: Rectangle
    Description: ''
```

Display information about the `headlight` sublabel.

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

```
ans =  
  
  struct with fields:  
  
    Type: Rectangle  
  Description: ''  
    isOn: [1x1 struct]
```

Display information about the `carType` attribute.

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

```
ans =  
  
  struct with fields:  
  
    ListItems: {3x1 cell}  
  Description: ''
```

Label Data

`LabelData` is a timetable containing information about the ROI labels drawn at each timestamp, across the entire video. The timetable contains one column per label.

Display the first few rows of the timetable. The first few timestamps indicate that no vehicles were detected and that the `sunny` scene label is `false`. These results are because this portion of the video was not labeled. Only the time interval of 5-10 seconds was labeled.

```
labelData = gTruth.labelData;  
head(labelData)
```

```
ans =  
  
  8x3 timetable  
  
      Time           Car           Truck           sunny  
      -----  
  5.005 sec   [1x2 struct]   [1x0 struct]   true  
  5.0384 sec  [1x2 struct]   [1x0 struct]   true  
  5.0717 sec  [1x2 struct]   [1x0 struct]   true  
  5.1051 sec  [1x2 struct]   [1x0 struct]   true  
  5.1385 sec  [1x2 struct]   [1x0 struct]   true  
  5.1718 sec  [1x2 struct]   [1x0 struct]   true
```

```

5.2052 sec    [1x2 struct]    [1x0 struct]    true
5.2386 sec    [1x2 struct]    [1x0 struct]    true

```

Display the first few timetable rows from the 5-10 second interval that contains labels.

```

gTruthInterval = labelData(timerange('00:00:05','00:00:10'),:);
head(gTruthInterval)

```

```
ans =
```

```
8x3 timetable
```

Time	Car	Truck	sunny
5.005 sec	[1x2 struct]	[1x0 struct]	true
5.0384 sec	[1x2 struct]	[1x0 struct]	true
5.0717 sec	[1x2 struct]	[1x0 struct]	true
5.1051 sec	[1x2 struct]	[1x0 struct]	true
5.1385 sec	[1x2 struct]	[1x0 struct]	true
5.1718 sec	[1x2 struct]	[1x0 struct]	true
5.2052 sec	[1x2 struct]	[1x0 struct]	true
5.2386 sec	[1x2 struct]	[1x0 struct]	true

For each Car label, the structure includes the position of the bounding box and information about its sublabels and attributes.

Display the bounding box positions for the vehicles at the start of the time interval. Your bounding box positions might differ from the ones shown here.

```
gTruthInterval(1,:).Car{1}.Position % [x y width height], in pixels
```

```
ans =
```

```
1x4 single row vector
```

```
415.8962    82.4737   130.8474   129.3805
```

```
ans =
```

```
1x4 single row vector
```

```
235.2182     1.0000   117.0611    55.3500
```

Save App Session

From 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. To change layout options, select **Layout**.

The app session MAT-file is separate from the ground truth MAT-file that is exported when you select **Export > From File**. To share labeled ground truth data, as a best practice, share the ground truth MAT-file containing the `groundTruth` object, not the app session MAT-file. For more details, see “Share and Store Labeled Ground Truth Data” (Computer Vision Toolbox).

See Also

Apps

Ground Truth Labeler

Objects

`driving.connector.Connector` | `groundTruth` | `groundTruthDataSource` | `labelDefinitionCreator` | `vision.labeler.AutomationAlgorithm` | `vision.labeler.mixin.Temporal`

Related Examples

- “Automate Ground Truth Labeling of Lane Boundaries”
- “Automate Ground Truth Labeling for Semantic Segmentation”
- “Automate Attributes of Labeled Objects”
- “Evaluate Lane Boundary Detections Against Ground Truth Data”
- “Evaluate and Visualize Lane Boundary Detections Against Ground Truth”

More About

- “Use Custom Data Source Reader for Ground Truth Labeling” (Computer Vision Toolbox)
- “Keyboard Shortcuts and Mouse Actions for Ground Truth Labeler” on page 2-24
- “Use Sublabels and Attributes to Label Ground Truth Data” (Computer Vision Toolbox)

- “Label Pixels for Semantic Segmentation” (Computer Vision Toolbox)
- “Create Automation Algorithm for Labeling” (Computer Vision Toolbox)
- “View Summary of Ground Truth Labels” (Computer Vision Toolbox)
- “Share and Store Labeled Ground Truth Data” (Computer Vision Toolbox)
- “Training Data for Object Detection and Semantic Segmentation” (Computer Vision Toolbox)

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	<ul style="list-style-type: none"> • PC: End • Mac: Hold Fn and press the right arrow
Go to the first frame	<ul style="list-style-type: none"> • PC: Home • Mac: Hold Fn and press the left arrow
Navigate through time interval boxes and frame navigation buttons	Tab

Task	Action
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
Cut selected rectangle and line ROIs	Ctrl+X
Copy selected rectangle and line ROIs to clipboard	Ctrl+C
Paste copied rectangle and line ROIs <ul style="list-style-type: none"> • 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

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

Task	Action
Cancel drawing and delete the entire polygon	Escape

Zooming

Task	Action
Zoom in or out of frame	Move the scroll wheel up (zoom in) or down (zoom out) The scroll wheel works in Zoom In or Zoom Out mode but not Label or Pan modes.
Zoom in on specific section of frame	From the app toolbar, under Modes , select Zoom In . Then, draw a box around the section of the frame you want to zoom in on.

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

Tracking and Sensor Fusion

- “Visualize Sensor Data and Tracks in Bird's-Eye Scope” on page 3-2
- “Linear Kalman Filters” on page 3-12
- “Extended Kalman Filters” on page 3-19

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:

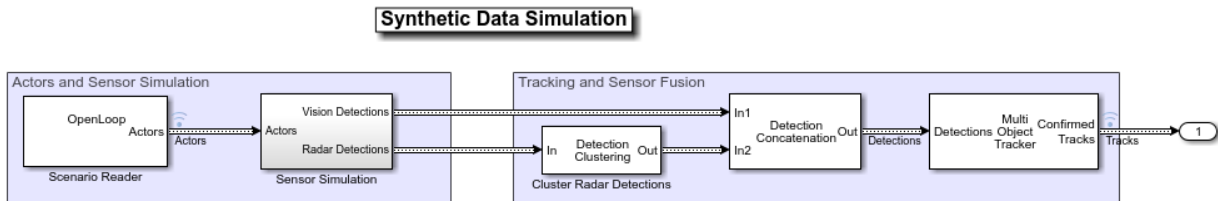
- Sensor coverages of vision and radar sensors
- Sensor detections of actors and lane boundaries
- Tracks of moving objects in the scenario

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')
```

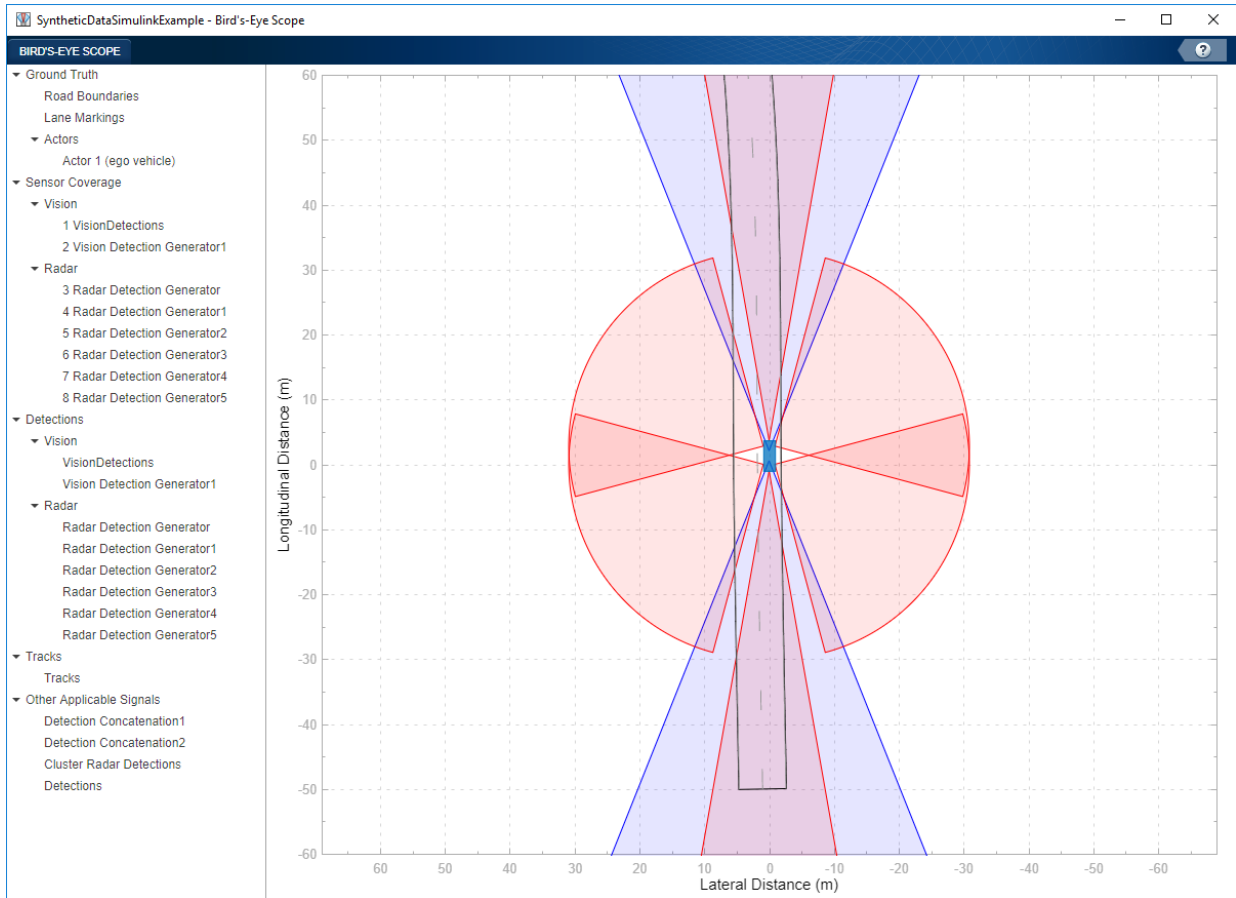


Open the scope from the Simulink toolstrip. Under **Review Results**, click **Bird's-Eye Scope**.

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

toolbar, click **Find Signals**. The scope updates the block diagram and automatically finds the signals in the model.



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
<p>Ground Truth</p>	<p>Road boundaries, lane markings, and actors in the scenario, including the ego vehicle</p> <p>You cannot modify this group or any of the signals within it.</p> <p>To inspect large road networks or to view actors that are located away from the ego vehicle, use the World Coordinates View window. See “Vehicle and World Coordinate Views”.</p>	<ul style="list-style-type: none"> • 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 block defaults. • The profile of the ego vehicle is always set to the block defaults.
<p>Sensor Coverage</p>	<p>Coverage areas of your vision and radar sensors, sorted into Vision and Radar subgroups</p> <p>You can move or modify these subgroups and their signals. You cannot move or modify the top-level Sensor Coverage group.</p>	<ul style="list-style-type: none"> • Vision Detection Generator block • Radar Detection Generator block

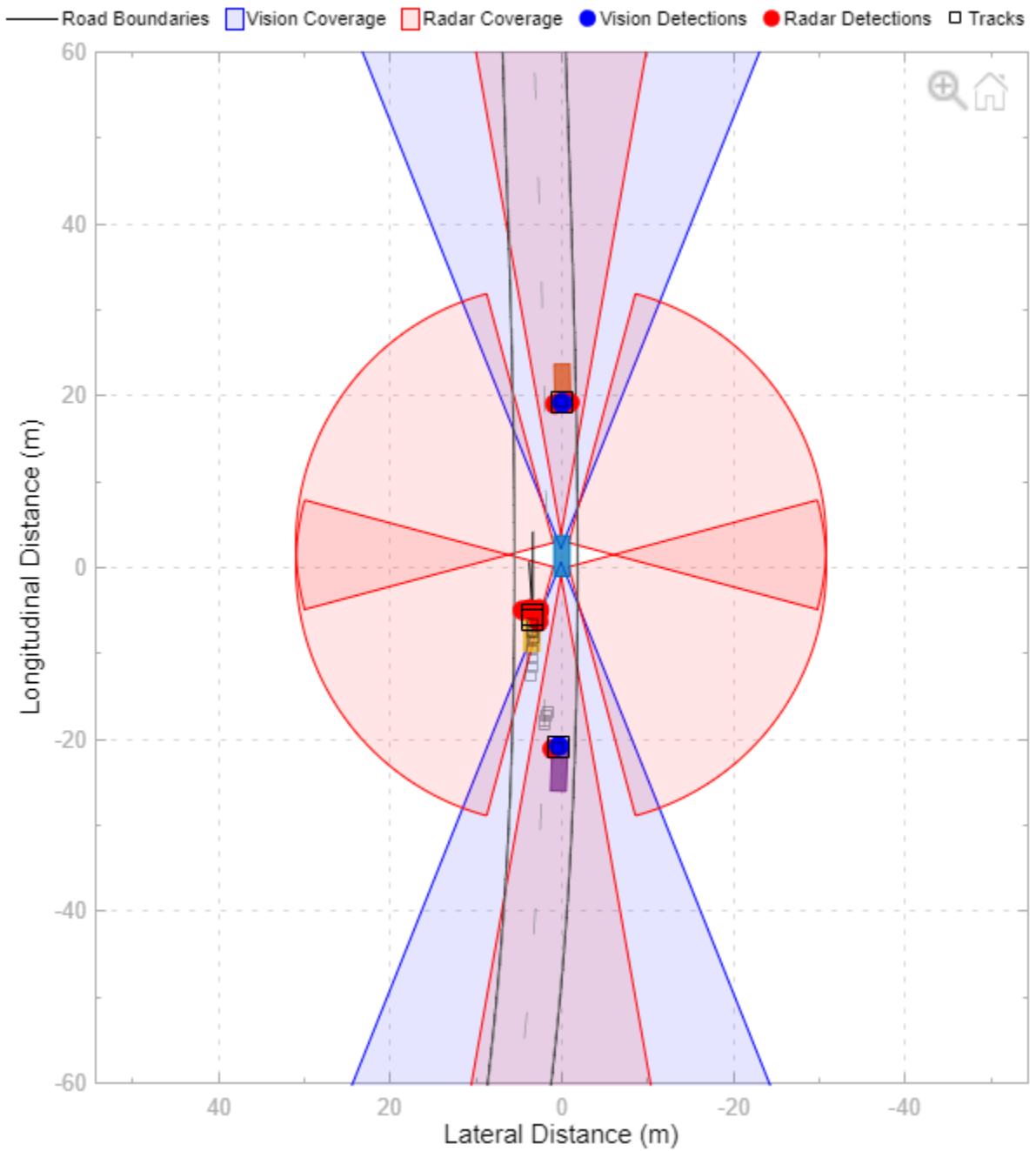
Signal Group	Description	Signal Sources
Detections	<p>Detections obtained from your vision and radar sensors, sorted into Vision and Radar subgroups</p> <p>You can move or modify these subgroups and their signals. You cannot move or modify the top-level Detections group.</p>	<ul style="list-style-type: none"> • Vision Detection Generator block • Radar Detection Generator block • Simulation 3D Probabilistic Radar block • When you first click Find Signals, detection signals from these blocks appear under Other Applicable Signals. To display the detections, move the signals to the Detections group. • The Bird's-Eye Scope does not display sensor coverage areas from these blocks.
Tracks	<p>Tracks of objects in the scenario</p>	<ul style="list-style-type: none"> • Multi Object Tracker block
Other Applicable Signals	<p>Signals that the scope cannot automatically group, such as ones that combine information from multiple sensors</p> <p>Signals in this group do not display during simulation.</p>	<ul style="list-style-type: none"> • Blocks that combine or cluster signals (such as the Detection Concatenation block) • Nonvirtual Simulink buses containing position and velocity information for detections and tracks

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 delete or modify the top-level groups in the left pane, but you can modify or delete any subgroup. If you delete a subgroup, its signals are moved automatically to the group that contained that subgroup.

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.

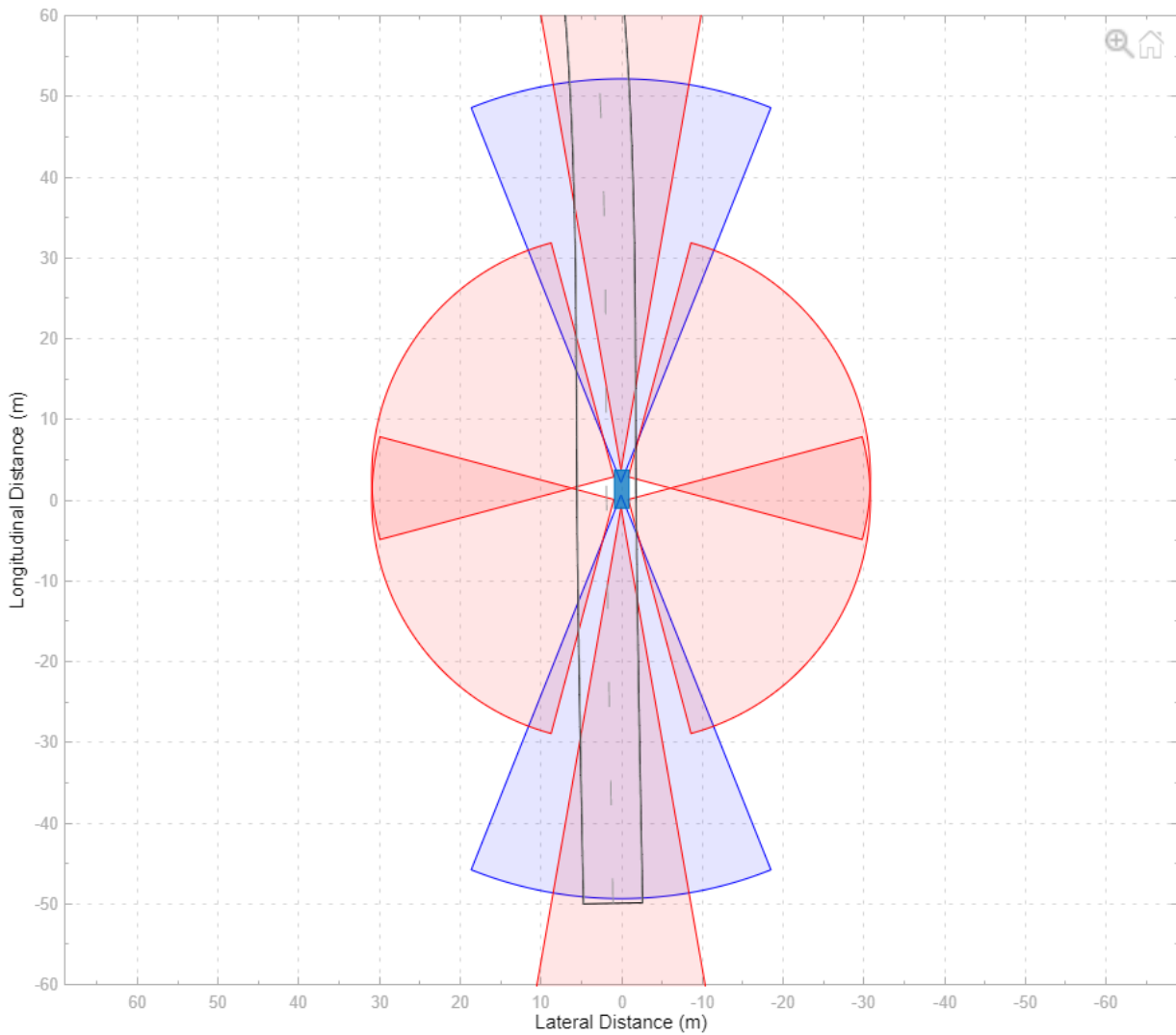
When you modify block parameters, you can rerun the simulation without having to find signals again. If you add or remove blocks, ports, or signal lines, then you must click **Find Signals** again before rerunning the simulation.

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 simulating the model, remove the model file folder from the MATLAB search path.

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


See Also

Bird's-Eye Scope | Detection Concatenation | Multi Object Tracker | Radar Detection Generator | Scenario Reader | Vision Detection Generator

Related Examples

- “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 5-93
- “Test Closed-Loop ADAS Algorithm Using Driving Scenario” on page 5-99

Linear Kalman Filters

In this section...

“State Equations” on page 3-12

“Measurement Models” on page 3-14

“Linear Kalman Filter Equations” on page 3-14

“Filter Loop” on page 3-15

“Constant Velocity Model” on page 3-16

“Constant Acceleration Model” on page 3-17

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 difference 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

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

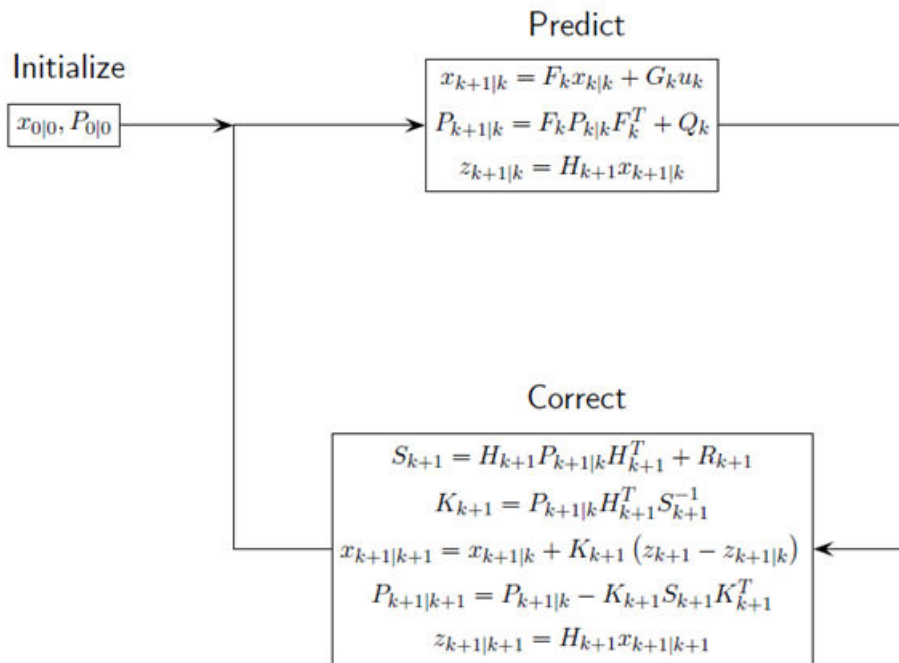
where K_{k+1} is the Kalman gain. The corrected state is often called the *a posteriori* estimate of the state because it is derived after the measurement is included.

Correct the state covariance matrix

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

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.



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.

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

See Also

Objects
trackingKF

Extended Kalman Filters

In this section...

“State Update Model” on page 3-19

“Measurement Model” on page 3-20

“Extended Kalman Filter Loop” on page 3-20

“Predefined Extended Kalman Filter Functions” on page 3-21

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:

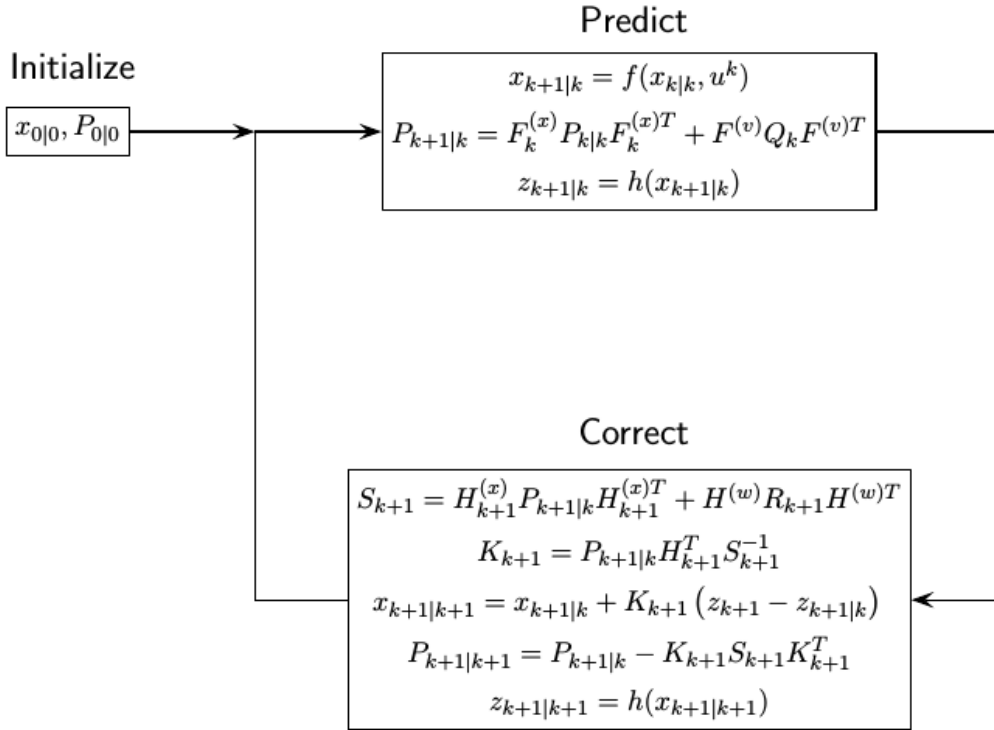
$$z_k = h(x_k, t) + v_k$$

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.



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

Motion Model	Function Name	Function Purpose
	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” on page 4-2
- “Access HERE HD Live Map Data” on page 4-8
- “Enter HERE HD Live Map Credentials” on page 4-15
- “Create Configuration for HERE HD Live Map Reader” on page 4-17
- “Create HERE HD Live Map Reader” on page 4-23
- “Read and Visualize Data Using HERE HD Live Map Reader” on page 4-27
- “HERE HD Live Map Layers” on page 4-40
- “Control Vehicle Velocity” on page 4-46
- “Velocity Profile of Straight Path” on page 4-49
- “Velocity Profile of Path with Curve and Direction Change” on page 4-55

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.

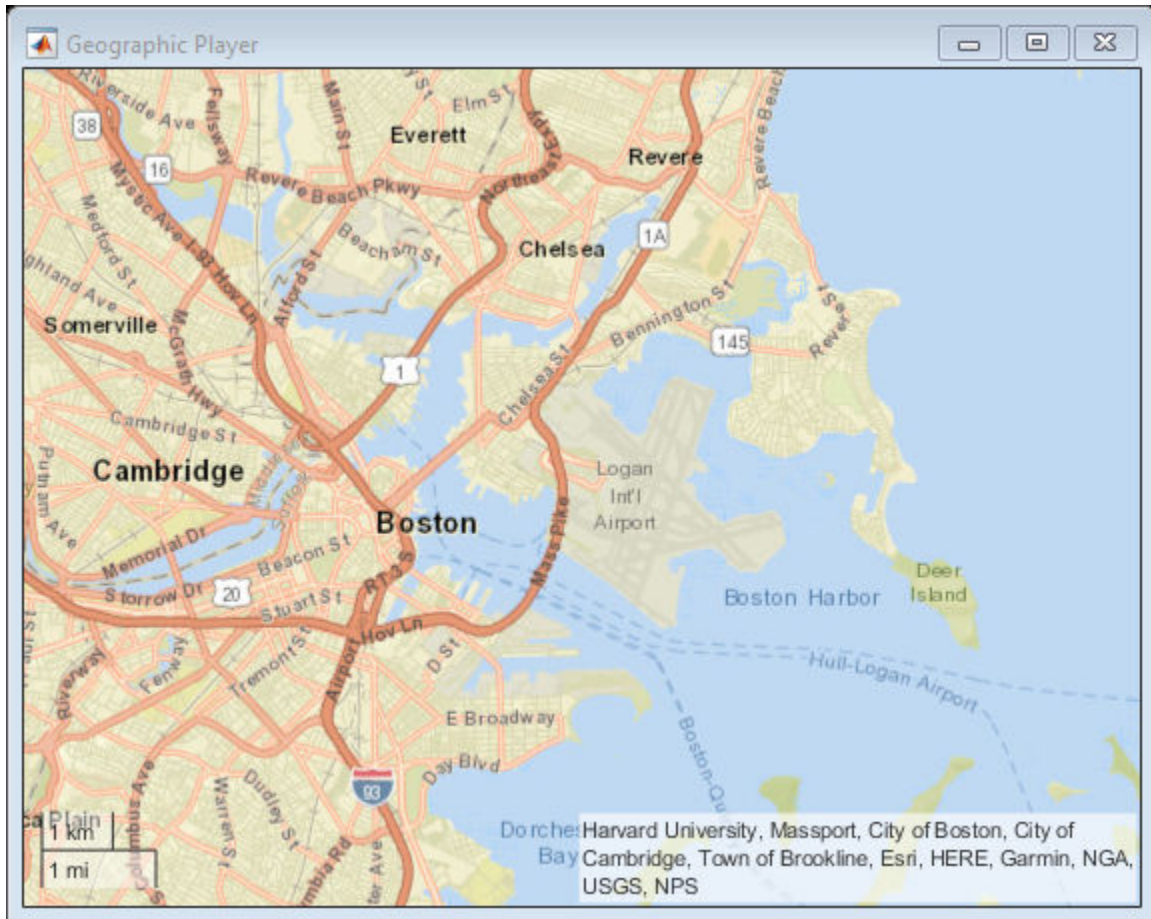
```
name = 'openstreetmap';  
url = 'https://a.tile.openstreetmap.org/{z}/{x}/{y}.png';  
copyright = char(uint8(169));  
attribution = copyright + "OpenStreetMap contributors";  
addCustomBasemap(name,url,'Attribution',attribution)
```

Load a sequence of latitude and longitude coordinates.

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

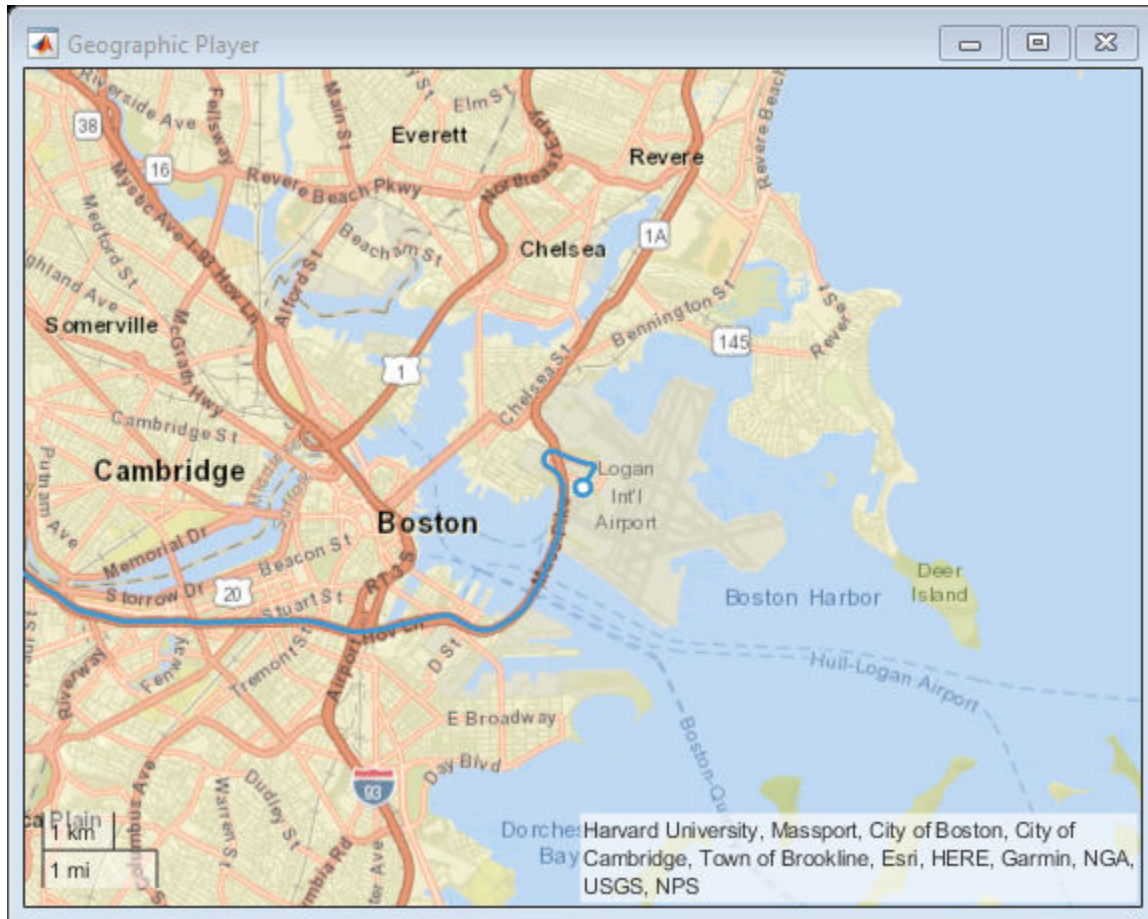
Create a geographic player. Center the geographic player on the first position of the driving route and set the zoom level to 12.

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



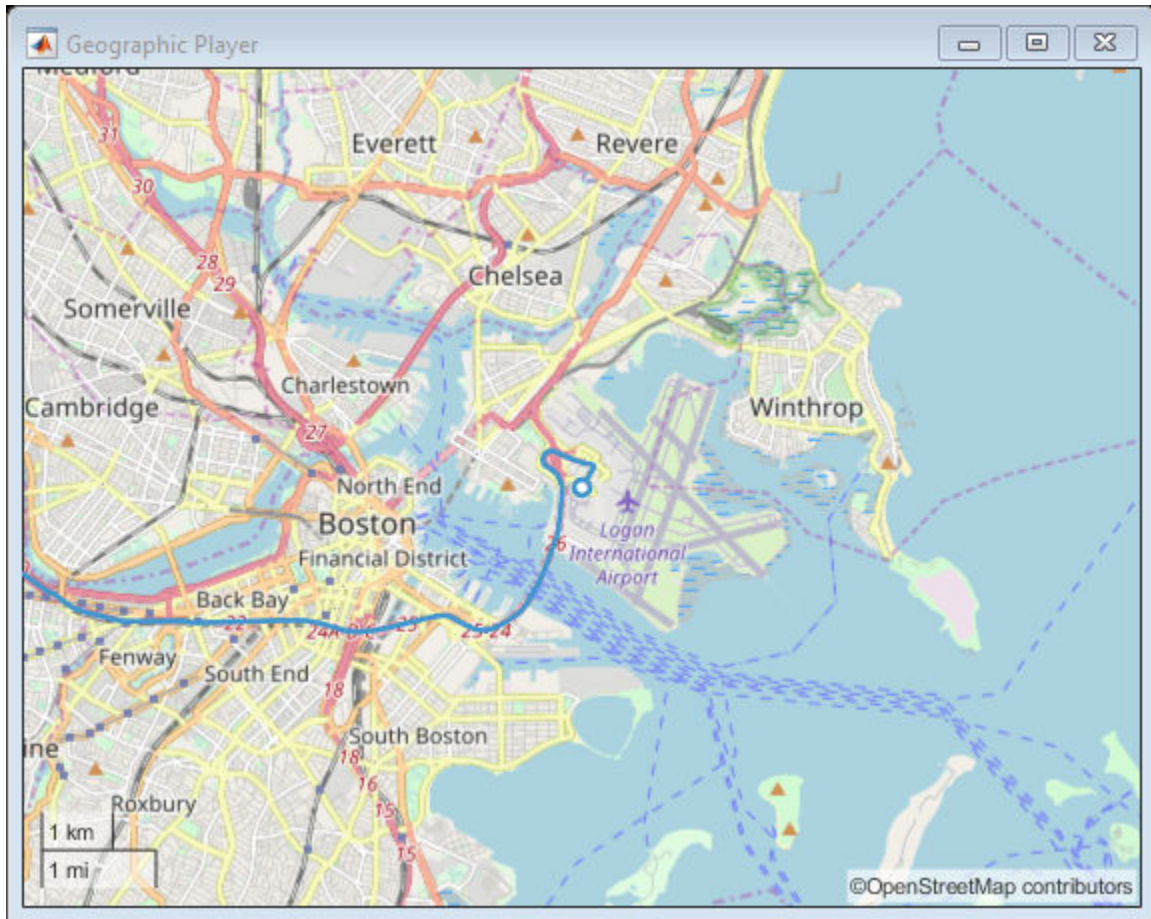
Display the full route.

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



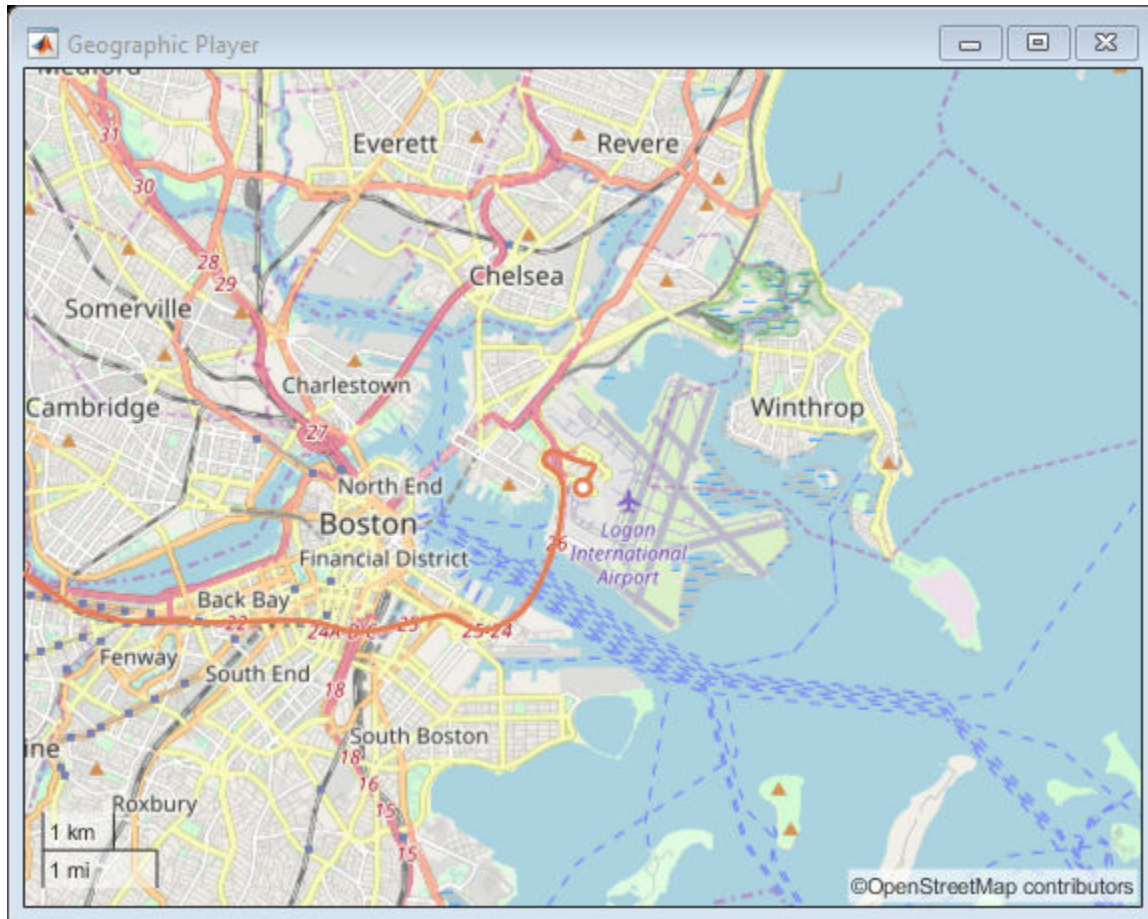
By default, the geographic player uses the World Street Map basemap ('streets') provided by Esri®. Update the geographic player to use the added OpenStreetMap basemap instead.

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

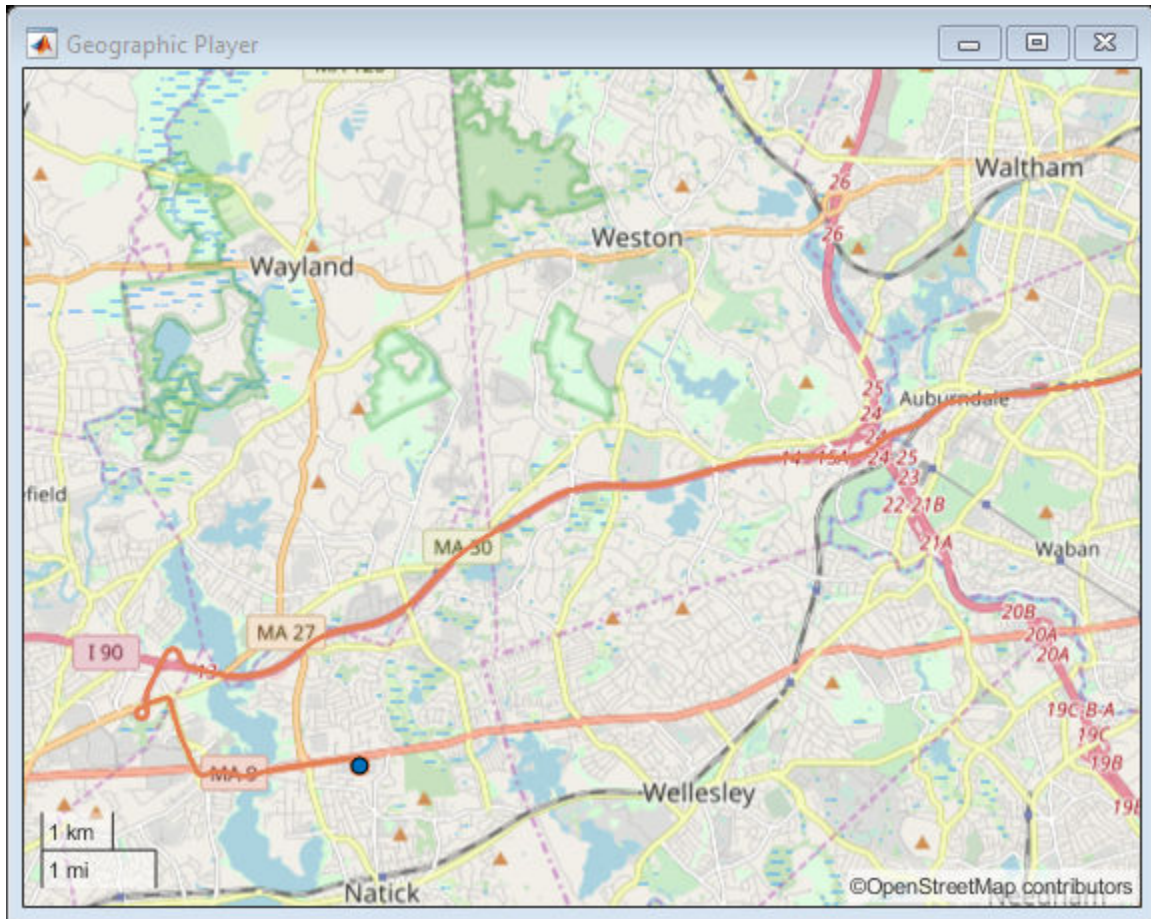
Display the route again.

```
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 Map¹ (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.

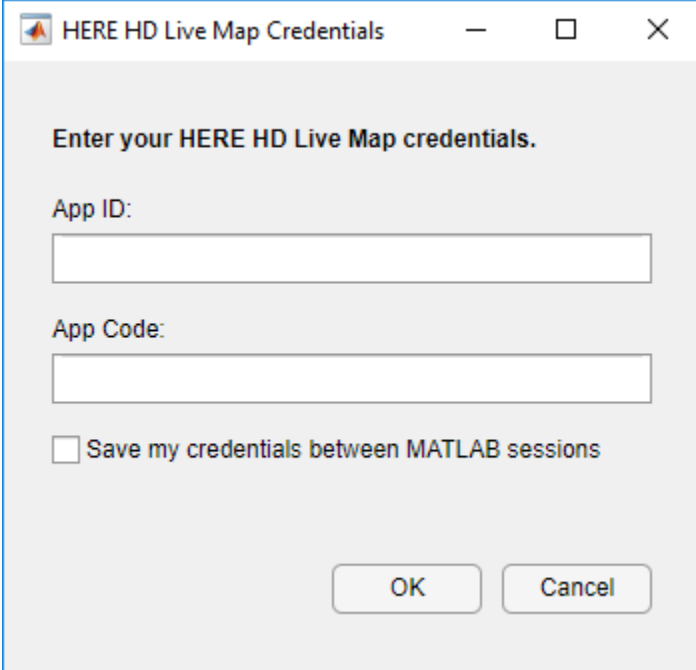
Using Automated Driving Toolbox functions and objects, you can create a HERE HDLM reader, read map data from the HERE HDLM web service, and then visualize the data from certain layers.

Step 1: Enter Credentials

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

`hereHDLMCredentials` [setup](#)

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.

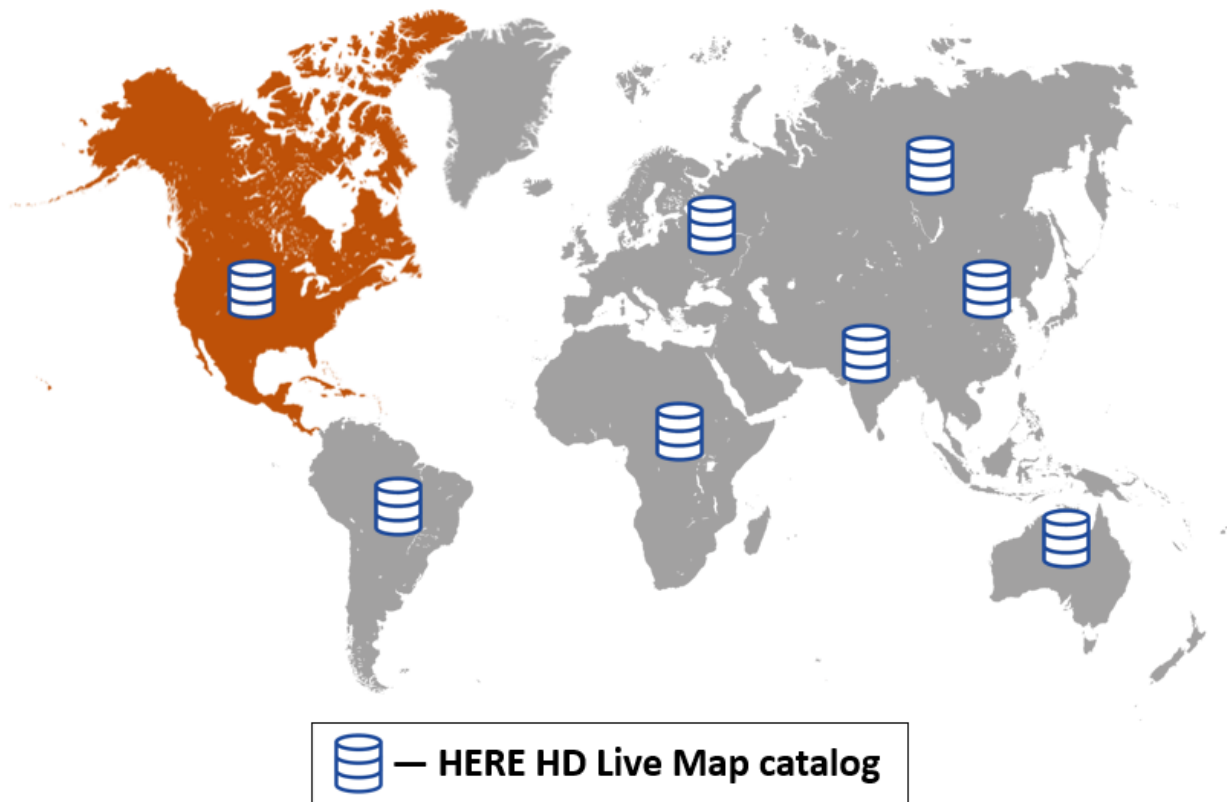
A screenshot of a MATLAB dialog box titled "HERE HD Live Map Credentials". The dialog box has a standard Windows-style title bar with a minimize button, a maximize button, and a close button. The main content area is light gray and contains the following elements: a bold instruction "Enter your HERE HD Live Map credentials.", a label "App ID:" followed by a white text input field, a label "App Code:" followed by another white text input field, a checkbox labeled "Save my credentials between MATLAB sessions" which is currently unchecked, and two buttons at the bottom: "OK" and "Cancel".

For more details, see “Enter HERE HD Live Map Credentials” on page 4-15.

Step 2: Create Reader Configuration

Optionally, to speed up performance, create a `hereHDLConfiguration` 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 = hereHDLConfiguration('North America');
```

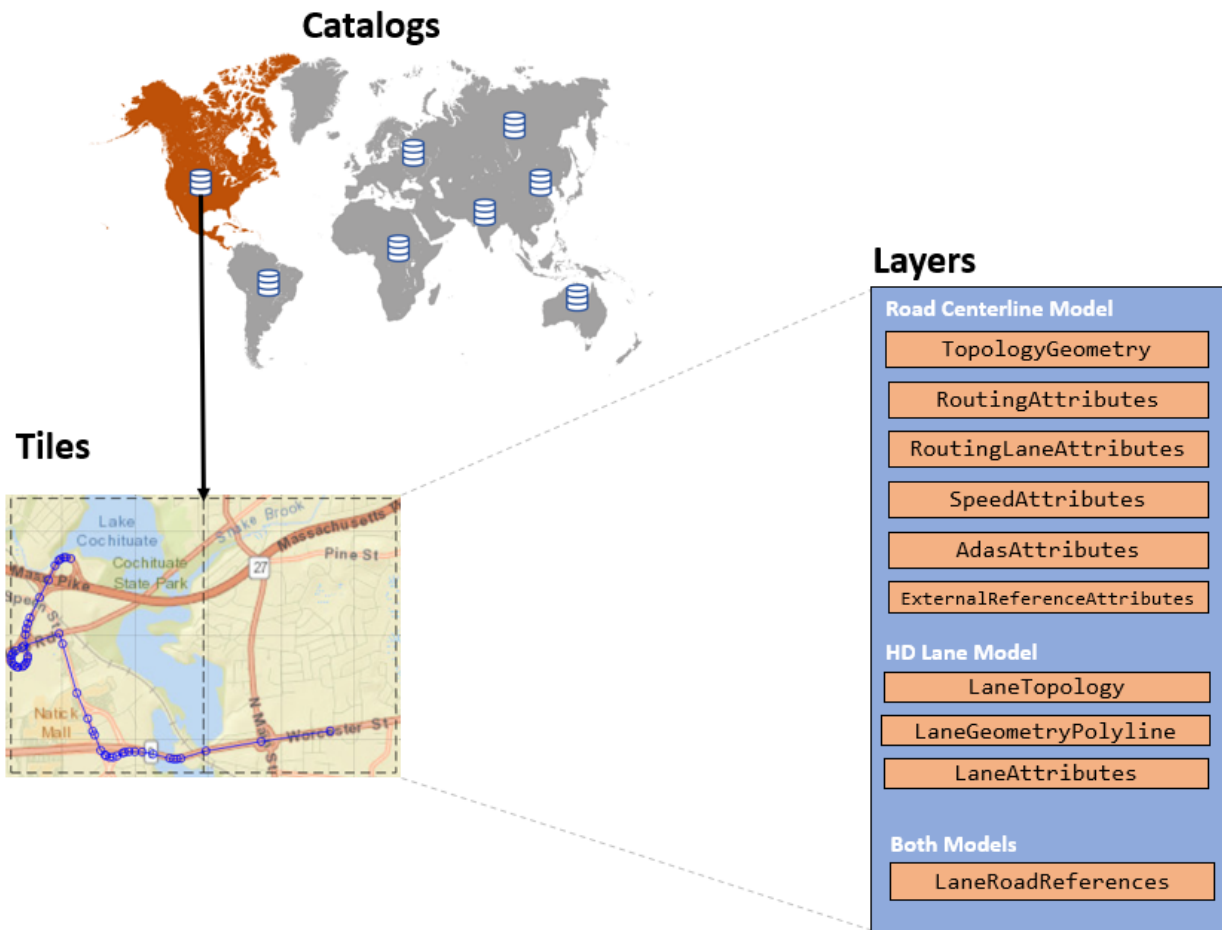



For more details, see “Create Configuration for HERE HD Live Map Reader” on page 4-17.

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 as 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);
```



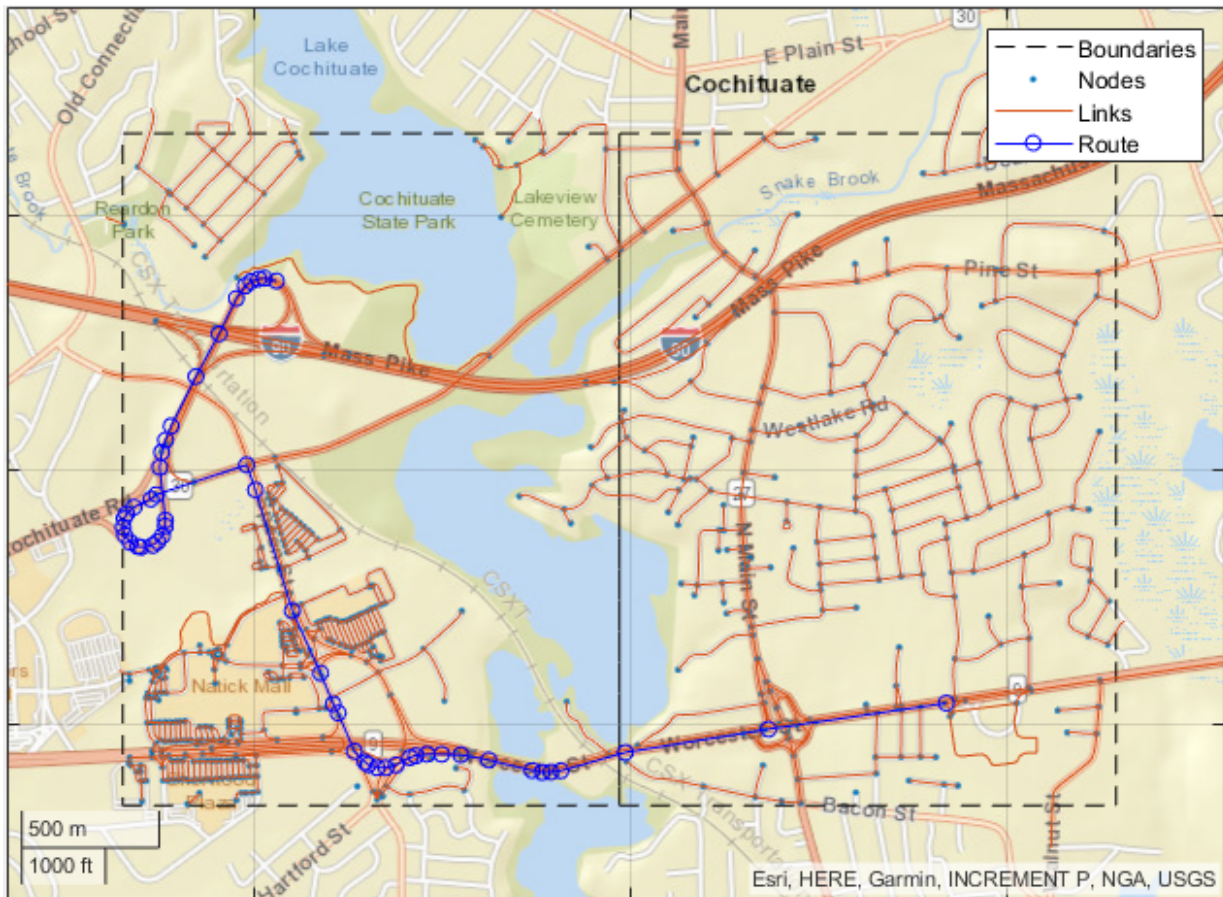
For more details, see “Create HERE HD Live Map Reader” on page 4-23.

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 =  
  
    2x1 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 4-27.

See Also

[hereHDLConfiguration](#) | [hereHDLMCredentials](#) | [hereHDLMReader](#) | [plot](#) | [read](#)

More About

- “Enter HERE HD Live Map Credentials” on page 4-15

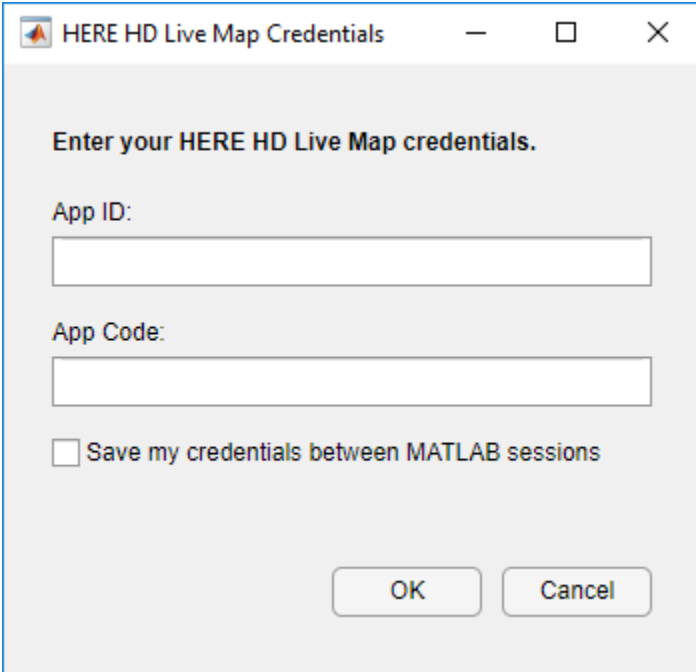
- “Create Configuration for HERE HD Live Map Reader” on page 4-17
- “Create HERE HD Live Map Reader” on page 4-23
- “Read and Visualize Data Using HERE HD Live Map Reader” on page 4-27
- “HERE HD Live Map Layers” on page 4-40
- “Use HERE HD Live Map Data to Verify Lane Configurations”

External Websites

- HD Live Map Data Specification

Enter HERE HD Live Map Credentials

To access the HERE HD Live Map² (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

OK Cancel

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.

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.

`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 4-17. Alternatively, you can create the reader without specifying a configuration. See “Create HERE HD Live Map Reader” on page 4-23.

See Also

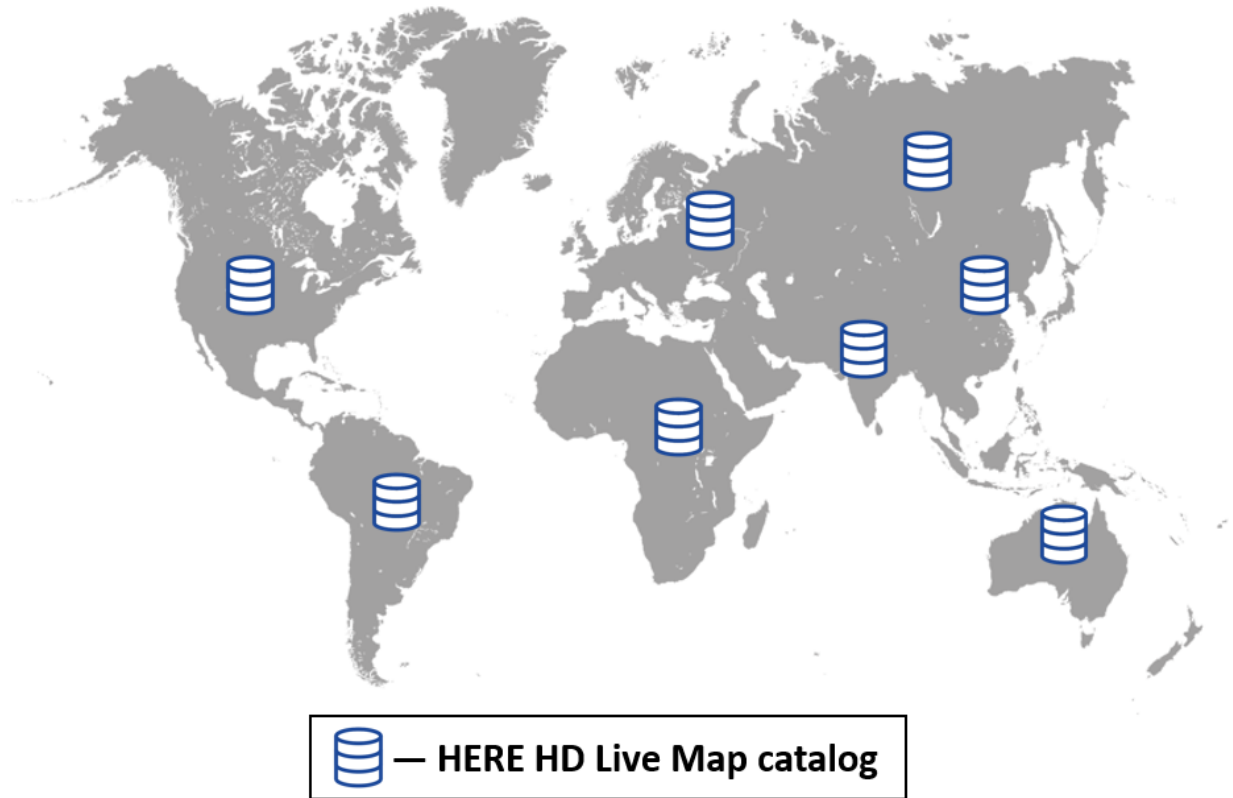
`hereHDLMConfiguration` | `hereHDLMCredentials` | `hereHDLMReader`

More About

- “Create Configuration for HERE HD Live Map Reader” on page 4-17
- “Create HERE HD Live Map Reader” on page 4-23

Create Configuration for HERE HD Live Map Reader

In the HERE HD Live Map³ (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 `hereHDLConfiguration` 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

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.

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.

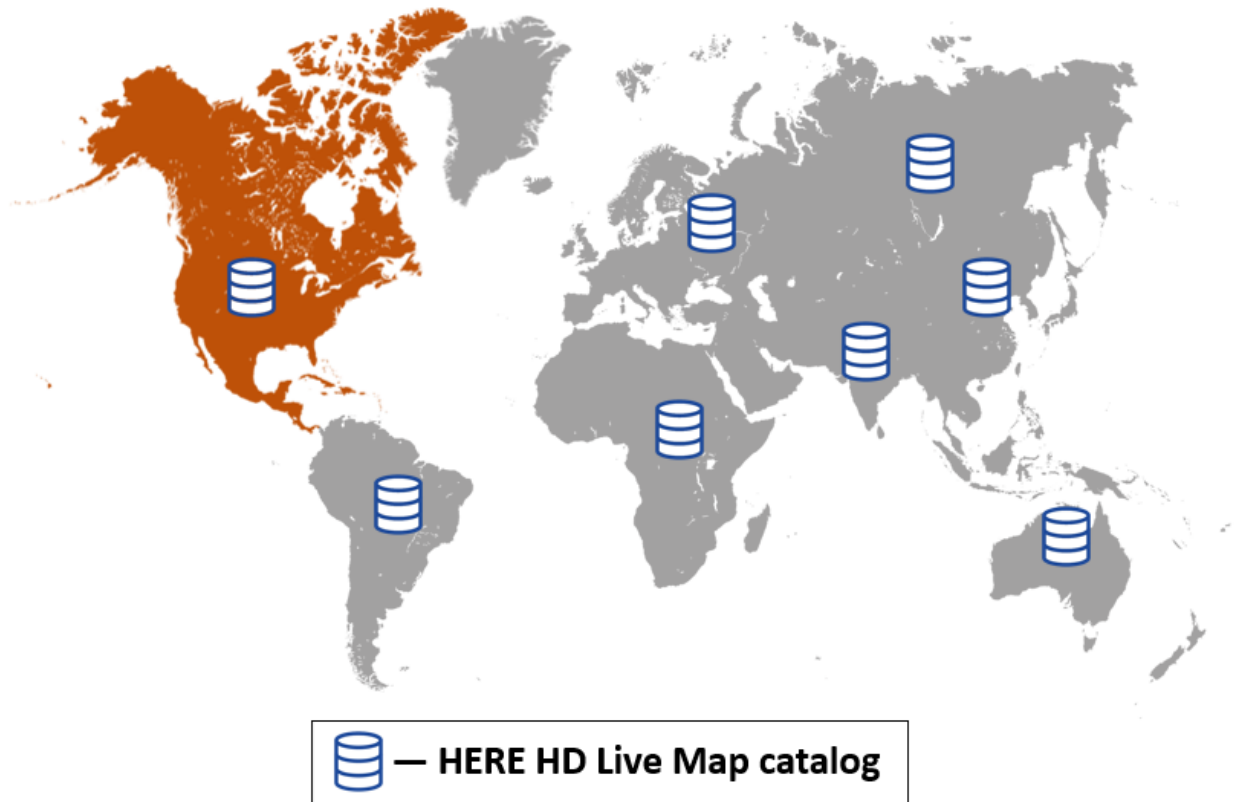
Consider a driving route located in North America.

```
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'

Region	Catalog
'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 = hereHDLConfiguration('Western Europe');
previousVersion = configLatest.CatalogVersion - 1;
configPrevious = hereHDLConfiguration('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 `hereHDLConfiguration` object returns an error.

Configure Reader

To configure a HERE HDLM reader, specify the configuration object when you create the `hereHDLReader` 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 4-18 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 = hereHDLReader(lat,lon,'Configuration',config);
reader.Configuration
```

`hereHDLConfiguration` 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 4-23.

See Also

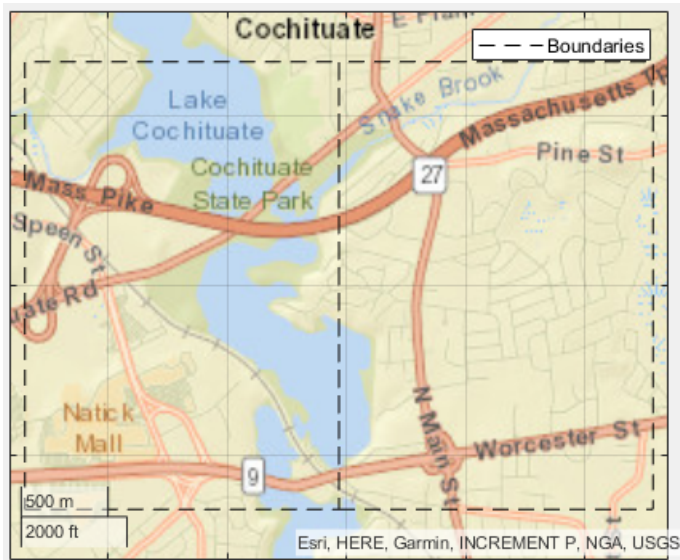
[hereHDLConfiguration](#) | [hereHDLReader](#)

More About

- “Create HERE HD Live Map Reader” on page 4-23

Create HERE HD Live Map Reader

A `hereHDLMReader` object reads HERE HD Live Map⁴ (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.

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.

```

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

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

```



Create a `hereHDLConfiguration` 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 4-17. If you have not previously set up HERE HDLM credentials, a dialog box prompts you to enter them.

```

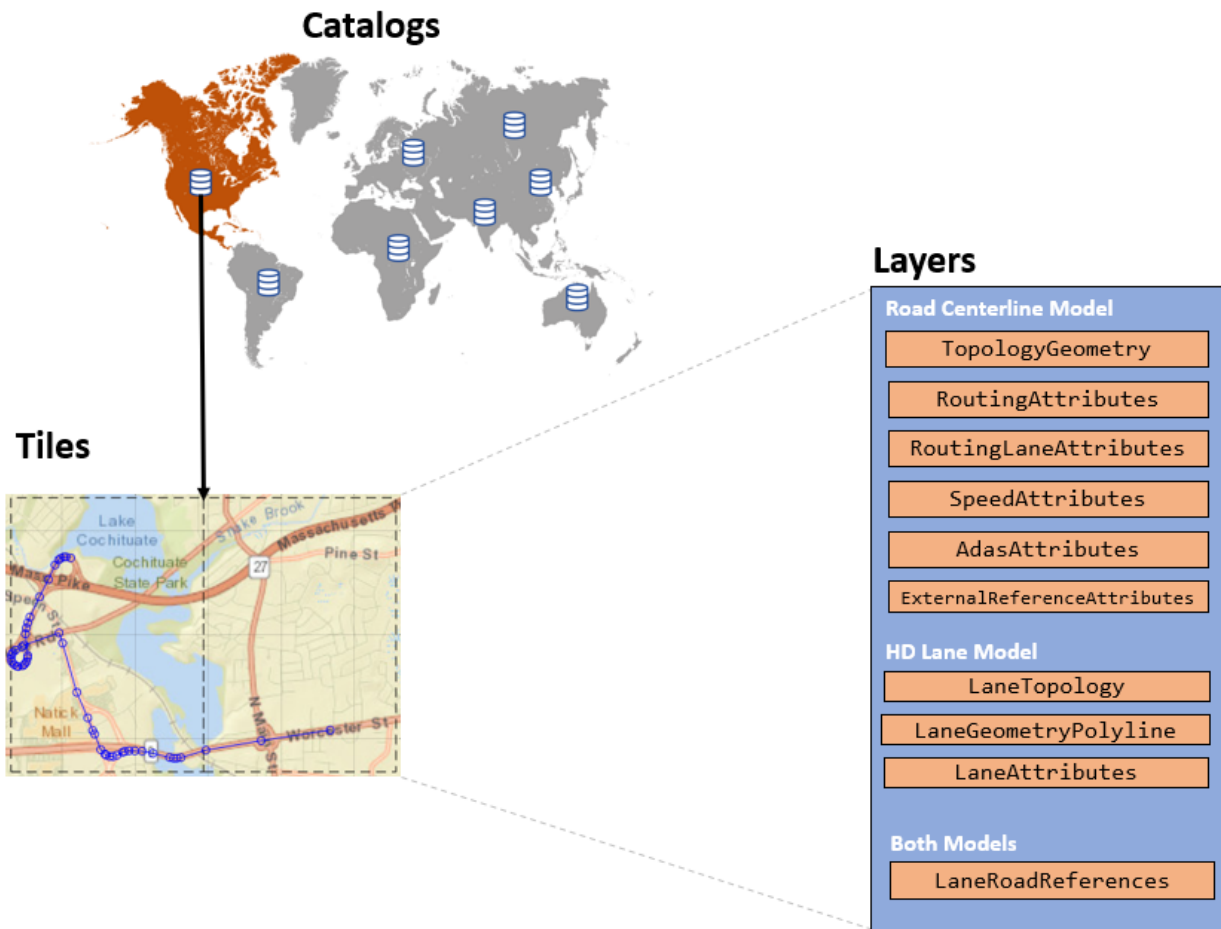
config = hereHDLConfiguration('North America');

```

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 4-40.



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 4-27.

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 `hereHDLMLReader` object, you can specify the map tile IDs directly. Specify the map tile IDs as an array of unsigned 32-bit integers.

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

```
tileIds = uint32([321884279 321884450]);  
reader = hereHDLMLReader(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 4-27.

See Also

[hereHDLMLConfiguration](#) | [hereHDLMLReader](#) | [read](#)

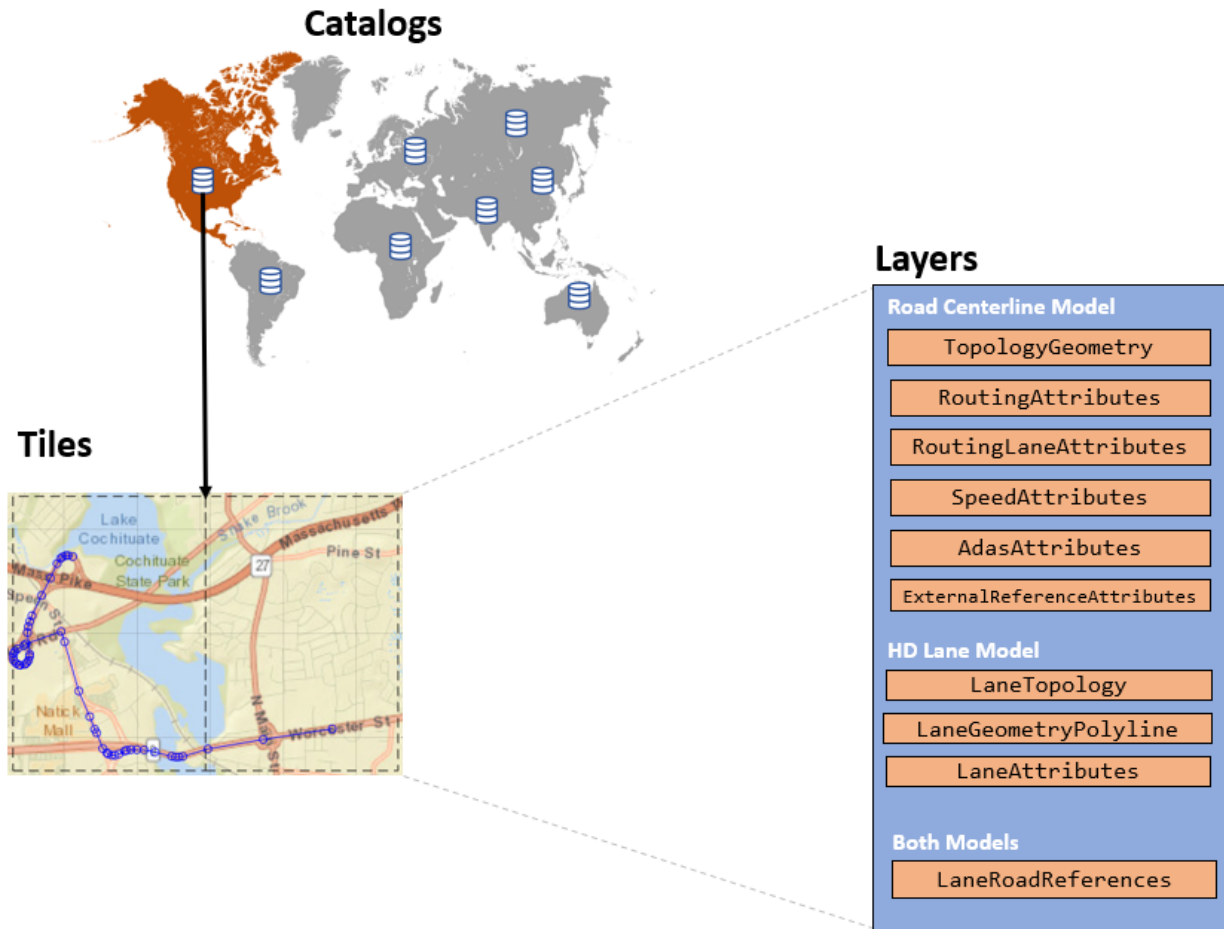
More About

- “Read and Visualize Data Using HERE HD Live Map Reader” on page 4-27
- “HERE HD Live Map Layers” on page 4-40

Read and Visualize Data Using HERE HD Live Map Reader

You can read map tile data from the HERE HD Live Map⁵ (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.

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.



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 `hereHDLConfiguration` 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 = hereHDLConfiguration('North America');
reader = hereHDLReader(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 4-17. For more details about creating a reader, see “Create HERE HD Live Map Reader” on page 4-23.

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: [38x1 struct]
  LinksStartingInTile: [490x1 struct]
    NodesInTile: [336x1 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
<code>here_tile_id</code>	<code>HereTileId</code>
<code>tile_center_here_2d_coordinate</code>	<code>TileCenterHere2dCoordinate</code>
<code>nodes_in_tile</code>	<code>NodesInTile</code>

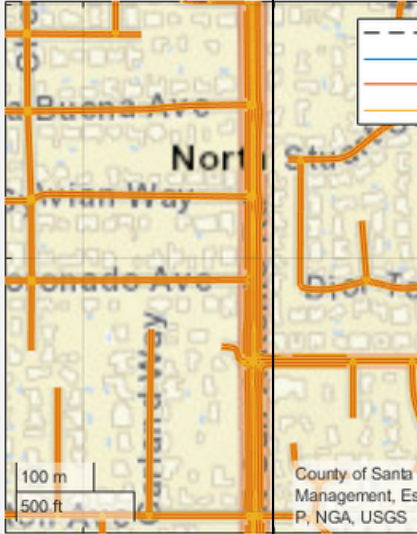
The layer objects are MATLAB structures whose properties correspond to structure fields. To access data from these fields, use dot notation. For example, this code selects the `NodeId` subfield from the `NodeAttribution` field of a layer:

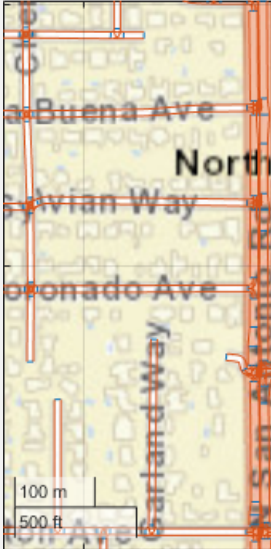
```
layerData.NodeAttribution.NodeId
```

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

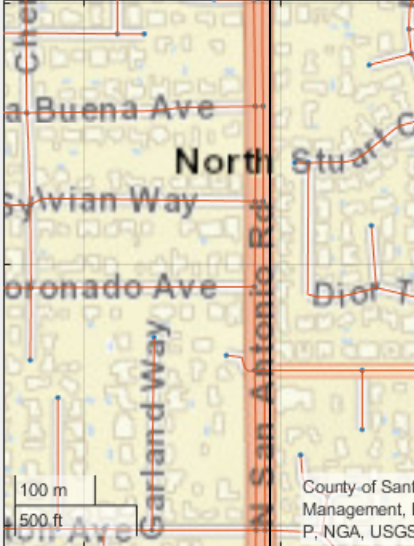
Road Centerline Model and HD Lane Model. For an overview of HERE HDLM layers and the models that they belong to, see “HERE HD Live Map Layers” on page 4-40. For a full description of the fields, see HD Live Map Data Specification on the HERE Technologies website.

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).	<ul style="list-style-type: none"> • HereTileId • LinkAttribution • NodeAttribution 	Not available
ExternalReferenceAttributes	References to external map links, nodes, and topologies for other HERE maps.	<ul style="list-style-type: none"> • HereTileId • LinkAttribution • NodeAttribution 	Not available
LaneAttributes	Lane-level attributes, such as direction of travel and lane type.	<ul style="list-style-type: none"> • HereTileId • LaneGroupAttribution 	Not available

Layer Object	Description	Top-Level Data Fields (Layer Object Properties)	Plot Support
LaneGeometryPolyline	3-D lane geometry composed of a set of 3-D points joined into polylines.	<ul style="list-style-type: none"> • HereTileId • TileCenterHere3dCoordinate • LaneGroupGeometries 	Available — Use the plot function. 
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.	<ul style="list-style-type: none"> • HereTileId • LaneGroupLinkReferences • LinkLaneGroupReferences 	Not available

Layer Object	Description	Top-Level Data Fields (Layer Object Properties)	Plot Support
LaneTopology	<p>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.</p>	<ul style="list-style-type: none"> • HereTileId • TileCenterHere2dCoordinate • LaneGroupsStartingInTile • LaneGroupConnectorsInTile • IntersectingLaneGroupRefs 	<p>Available — Use the plot function.</p> 
RoutingAttributes	<p>Road attributes related to navigation and conditions. These attributes are mapped parametrically to the 2-D polyline geometry in the topology layer.</p>	<ul style="list-style-type: none"> • HereTileId • LinkAttribution • NodeAttribution • StrandAttribution • AttributionGroupList 	<p>Not available</p>

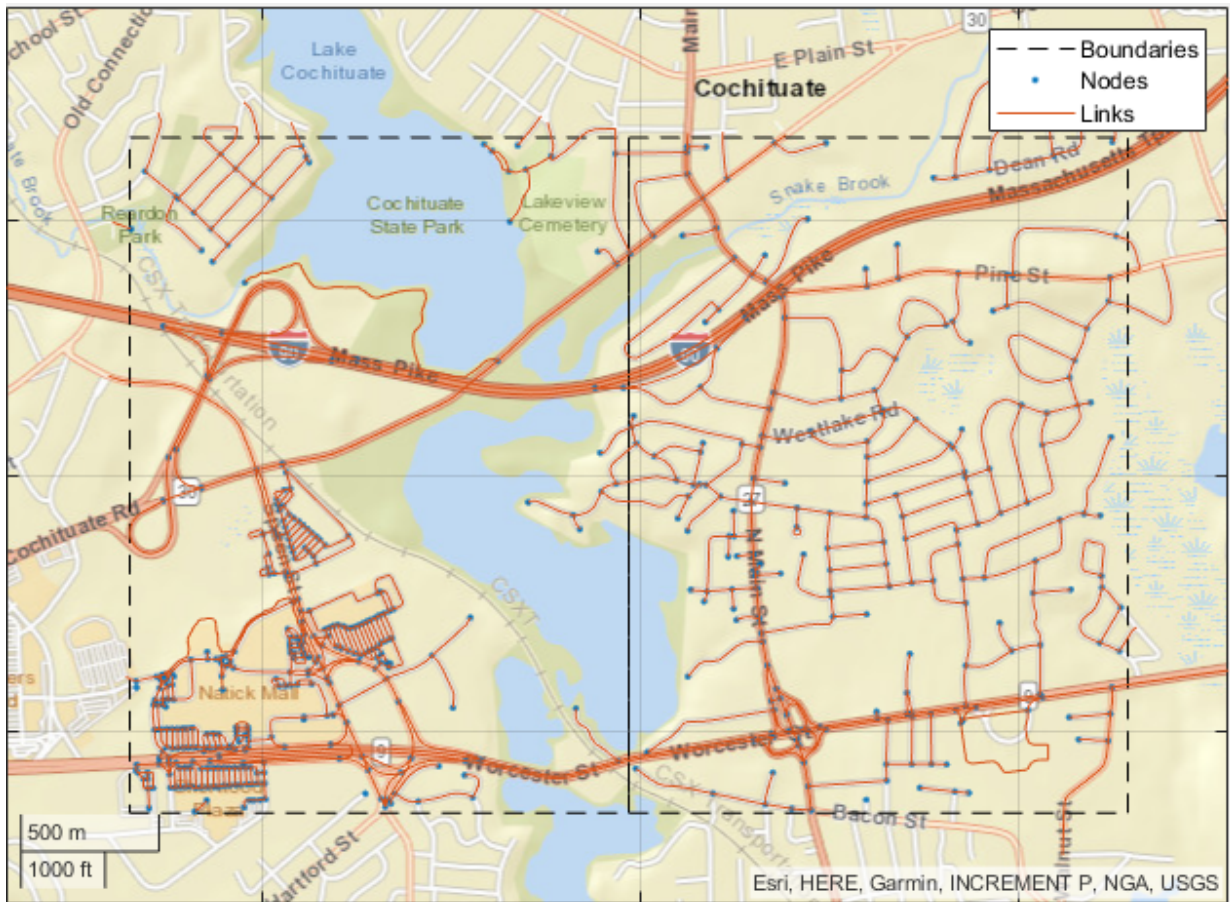
Layer Object	Description	Top-Level Data Fields (Layer Object Properties)	Plot Support
RoutingLaneAttributes	Core navigation lane attributes and conditions, such as the number of lanes in a road. These values are mapped parametrically to 2-D polylines along the road links.	<ul style="list-style-type: none">• 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.	<ul style="list-style-type: none">• 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.	<ul style="list-style-type: none"> • HereTileId • TileCenterHere2dCoordinate • NodesInTile • LinksStartingInTile • IntersectingLinkRefs 	Available — Use the plot function. 

Visualize Map Layer Data

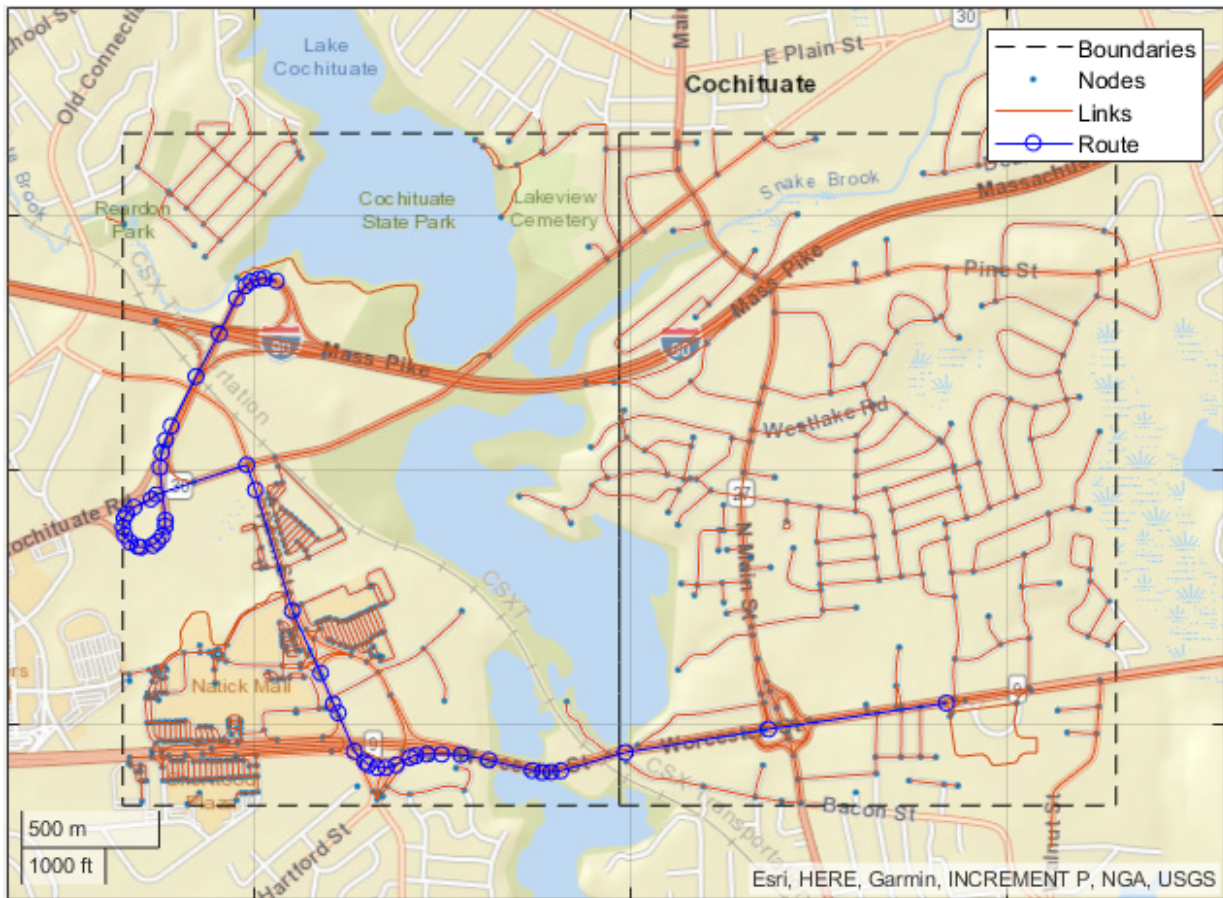
You can visualize the data of certain map layers. To visualize these layers, use the `plot` function. Plot the topology geometry of the returned map layers. The plot shows the boundaries, nodes (intersections and dead-ends), and links (streets) within the map tiles. If a link extends past the tile boundary,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

[hereHDLReader](#) | [plot](#) | [read](#)

More About

- “HERE HD Live Map Layers” on page 4-40
- “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 Map⁶ (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.

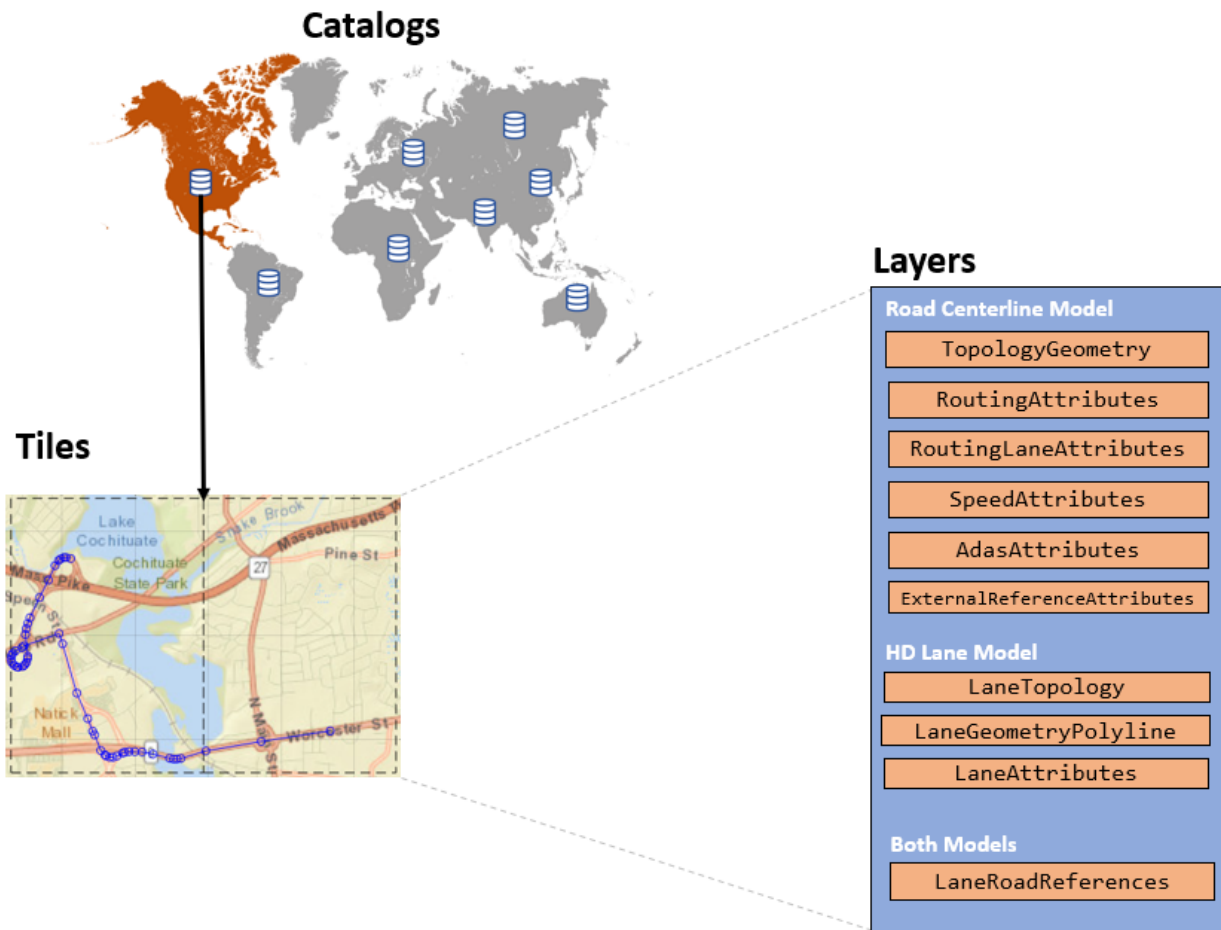
To access layer data for a selection of map tiles, use a `hereHDLMReader` object. For information on the `hereHDLMReader` workflow, see “Access HERE HD Live Map Data” on page 4-8.

The layers are grouped into these models:

- “Road Centerline Model” on page 4-41 — Provides road topology, shape geometry, and other road-level attributes
- “HD Lane Model” on page 4-43 — Contains lane topology, highly accurate geometry, and lane-level attributes
- “HD Localization Model” on page 4-45 — 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.

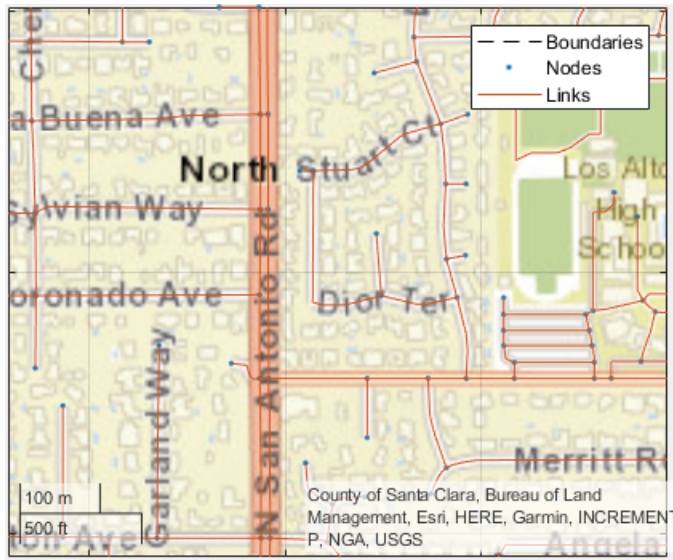
6. You need to enter into a separate agreement with HERE in order to gain access to the HDLM services and to get the required credentials (`app_id` and `app_code`) for using the HERE Service.



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.

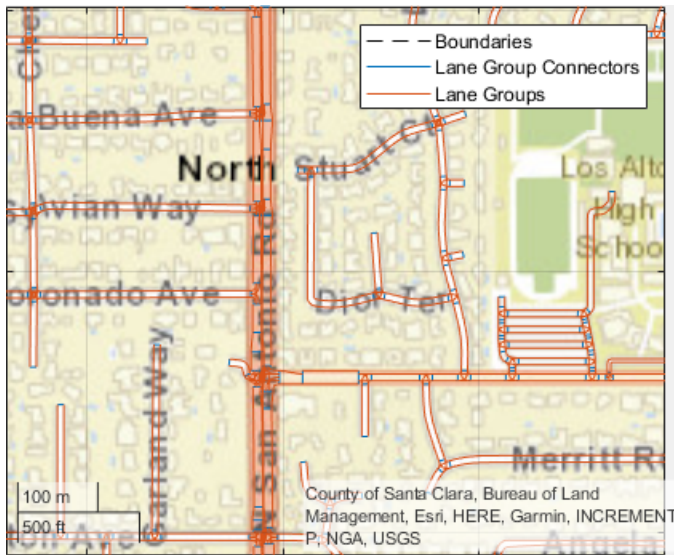
Road Centerline Model Layers	Description
TopologyGeometry	Topology and 2-D line geometry of the road. This layer also contains definitions of the links (streets) and nodes (intersections and dead-ends) in the map tile.
RoutingAttributes	Road attributes related to navigation and conditions. These attributes are mapped parametrically to the 2-D polyline geometry in the topology layer.

Road Centerline Model Layers	Description
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.
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.



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.
LaneAttributes	Lane-level attributes, such as direction of travel and lane type.

HD Lane Model Layers	Description
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 4-8
- “Use HERE HD Live Map Data to Verify Lane Configurations”

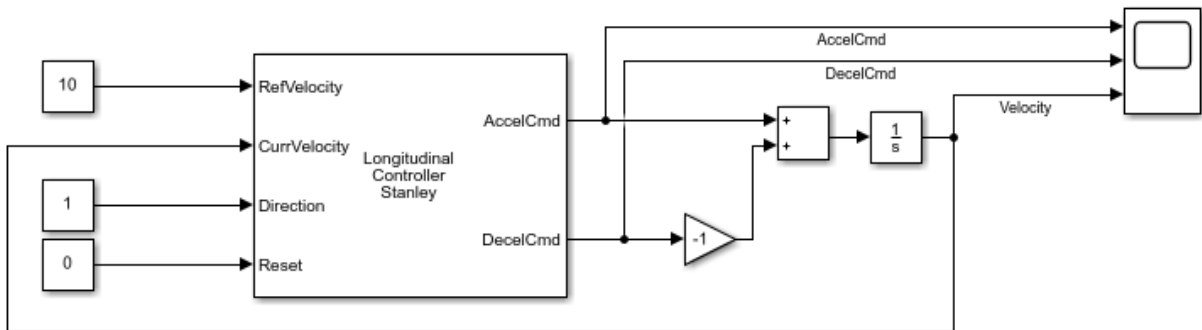
External Websites

- HD Live Map Data Specification

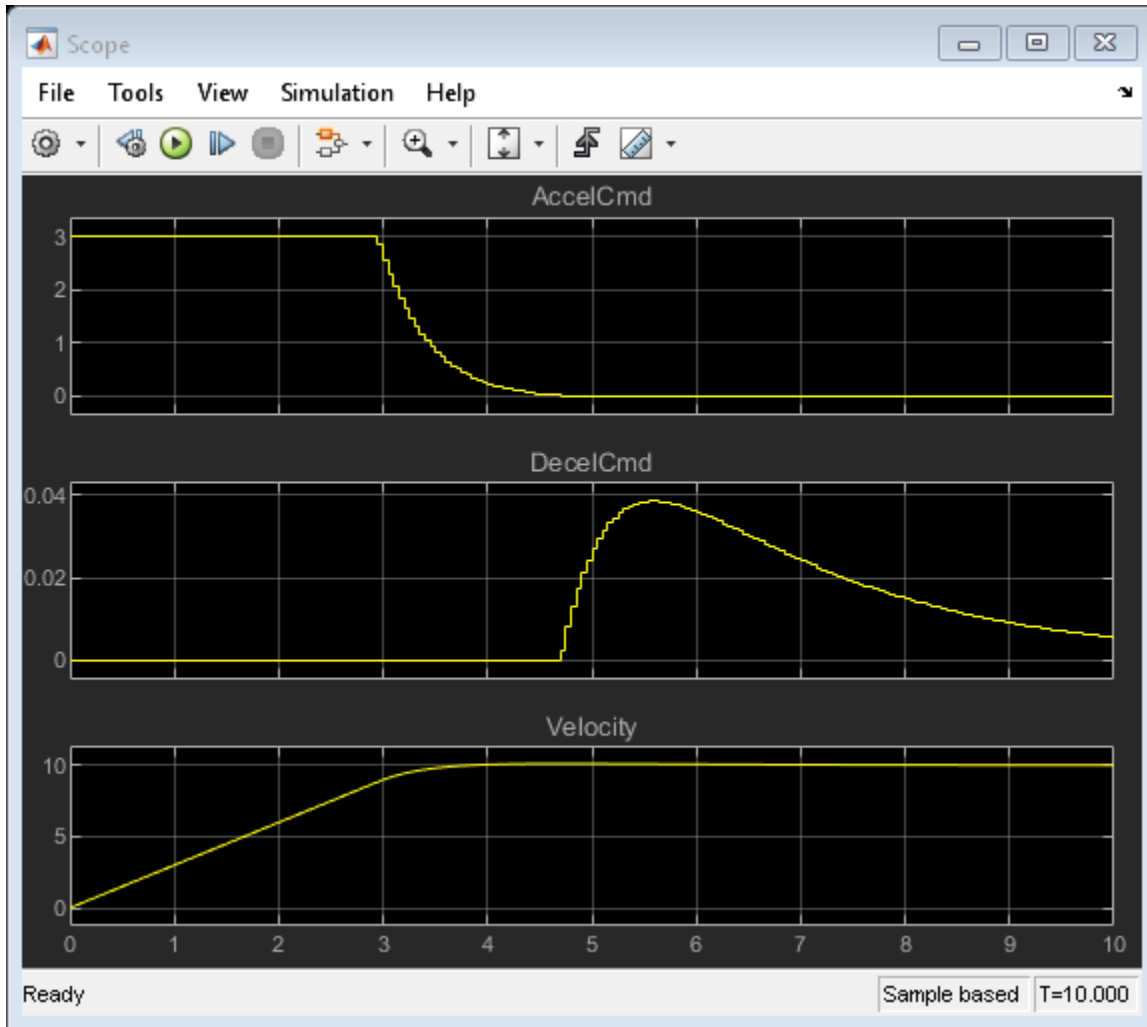
Control Vehicle Velocity

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/s²)** 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, K_p** and **Integral gain, K_i** parameters, respectively.

See Also

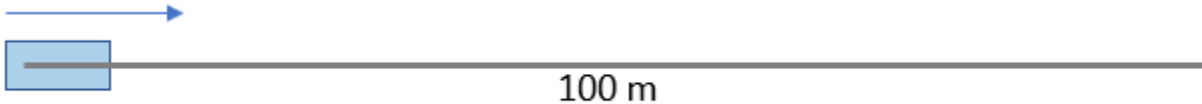
Lateral Controller Stanley | Longitudinal Controller Stanley

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.



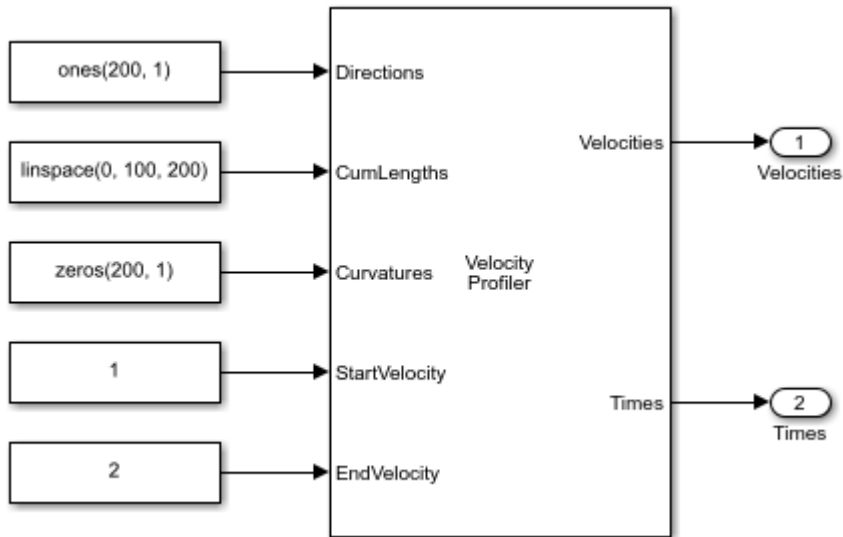
The Velocity Profiler block generates velocity profiles based on the speed, acceleration, and jerk constraints that you specify using parameters. You can use the generated velocity profile as the input reference velocities of a vehicle controller.

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.

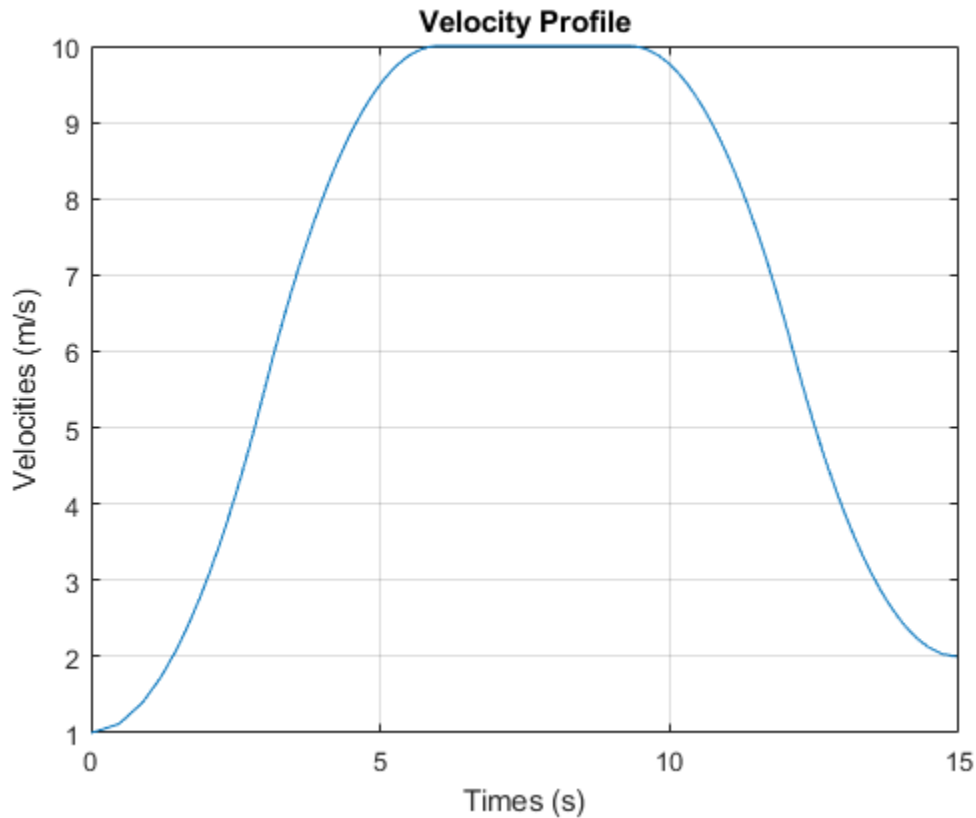
The block also outputs the times at which the vehicle arrives at each point along the path. You can use this output to visualize the velocities over time.

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:

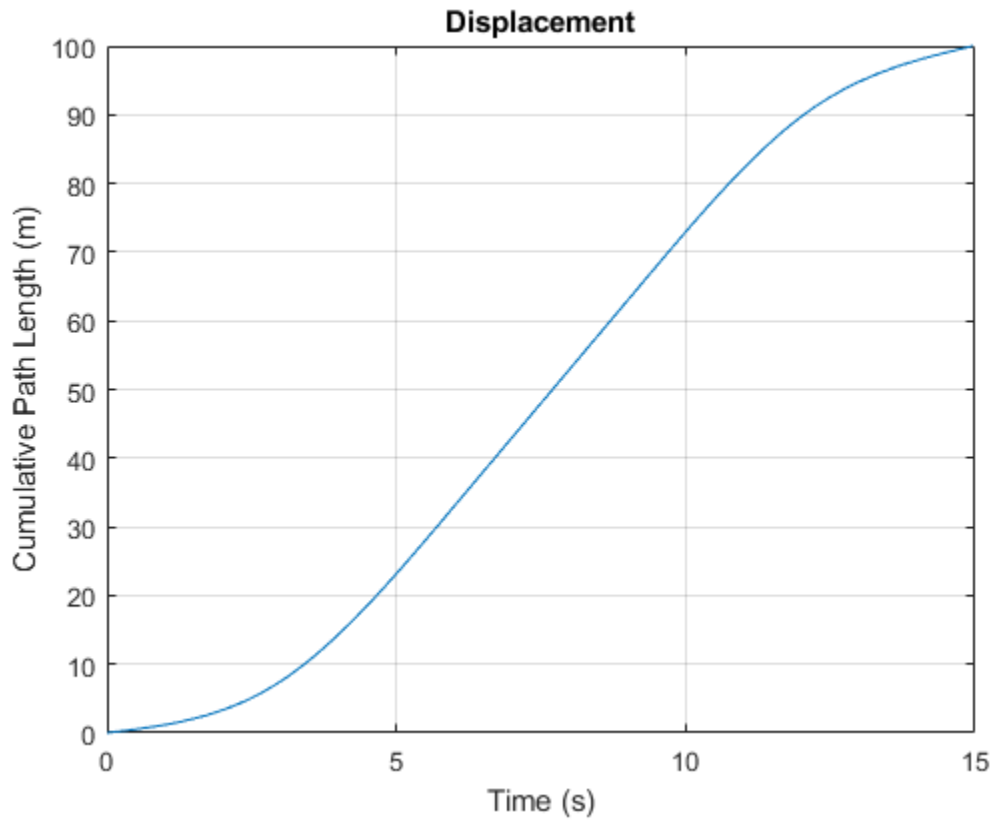
- 1 Starts at a velocity of 1 meter per second
- 2 Accelerates to a maximum speed of 10 meters per second, as specified by the **Maximum allowable speed (m/s)** parameter of the Velocity Profiler block
- 3 Decelerates to its ending velocity of 2 meters per second

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

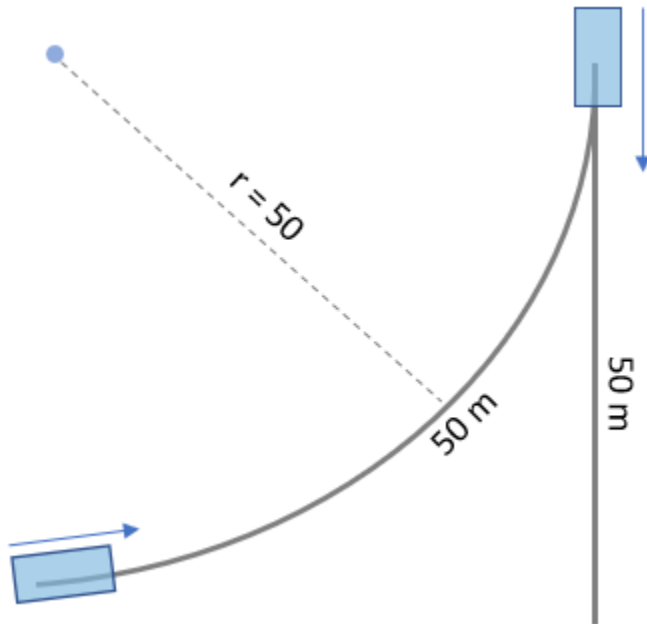
Path Smoother Spline | Velocity Profiler

More About

- “Velocity Profile of Path with Curve and Direction Change” on page 4-55
- “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.



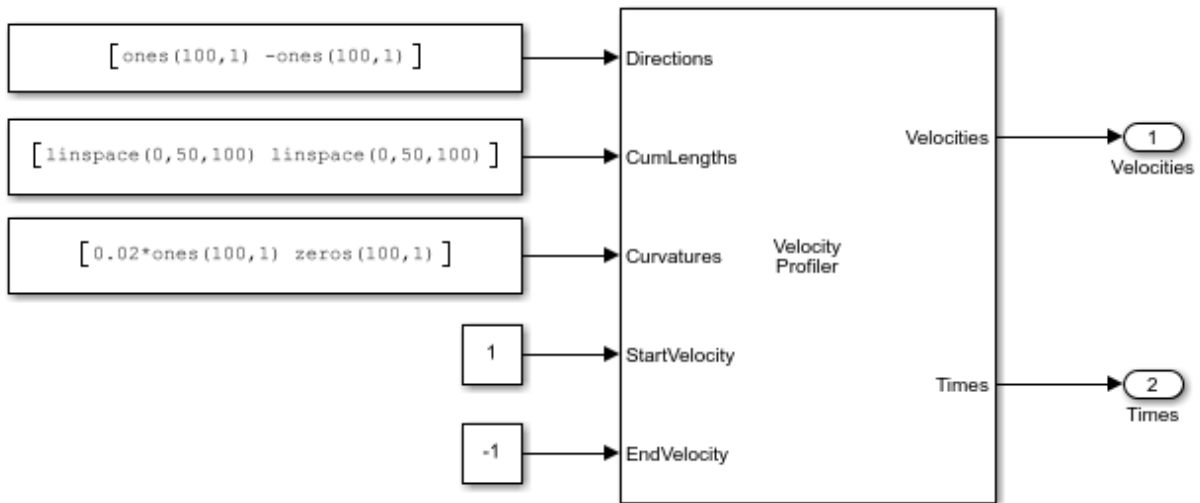
The Velocity Profiler block generates velocity profiles based on the speed, acceleration, and jerk constraints that you specify using parameters. You can use the generated velocity profile as the input reference velocities of a vehicle controller.

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.

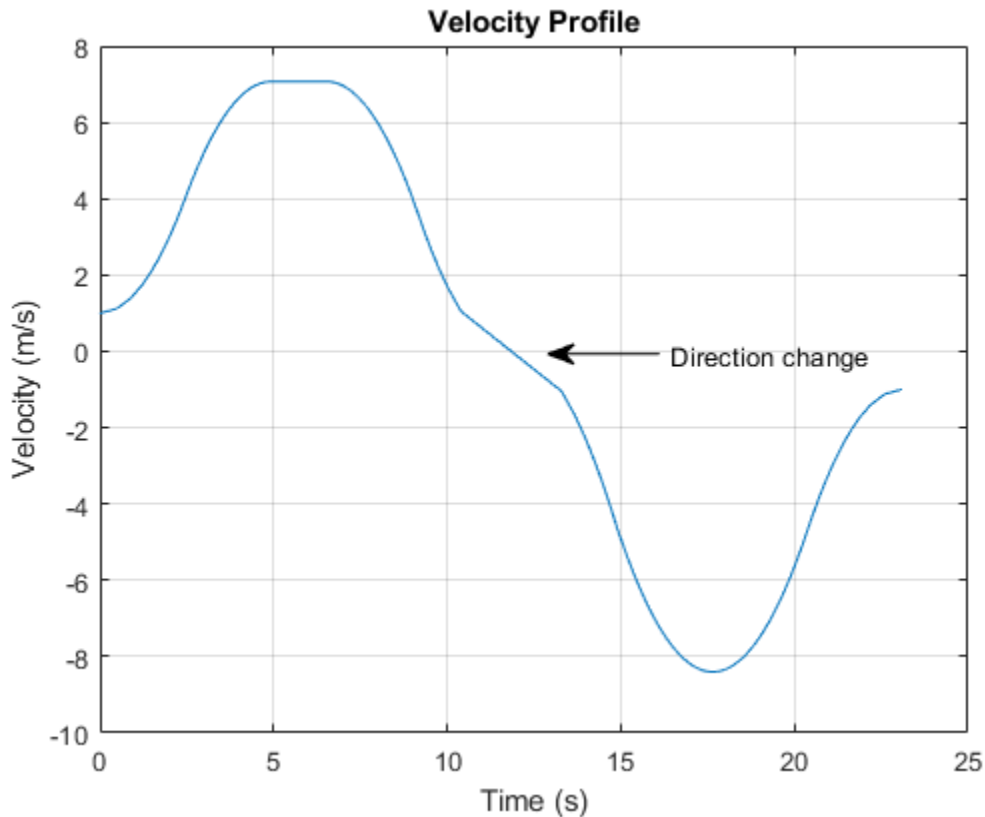
The block also outputs the times at which the vehicle arrives at each point along the path. You can use this output to visualize the velocities over time.

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 4-49
- “Automated Parking Valet in Simulink”

Driving Scenario Generation and Sensor Models

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.

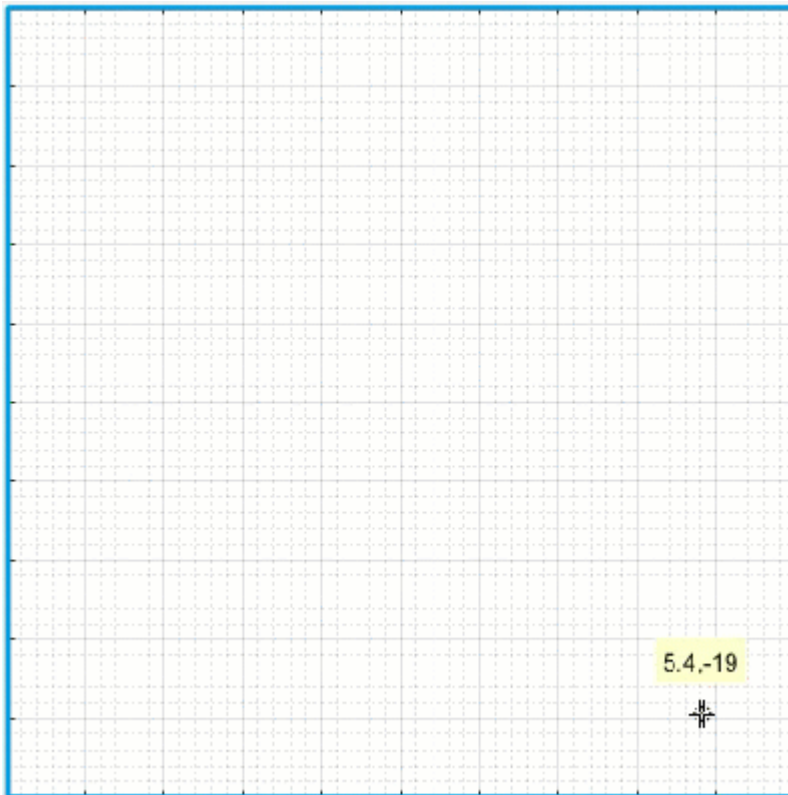
This example covers the entire workflow for creating a scenario and generating synthetic detections. Alternatively, you can generate detections from prebuilt scenarios. For more details, see “Prebuilt Driving Scenarios in Driving Scenario Designer” on page 5-18.

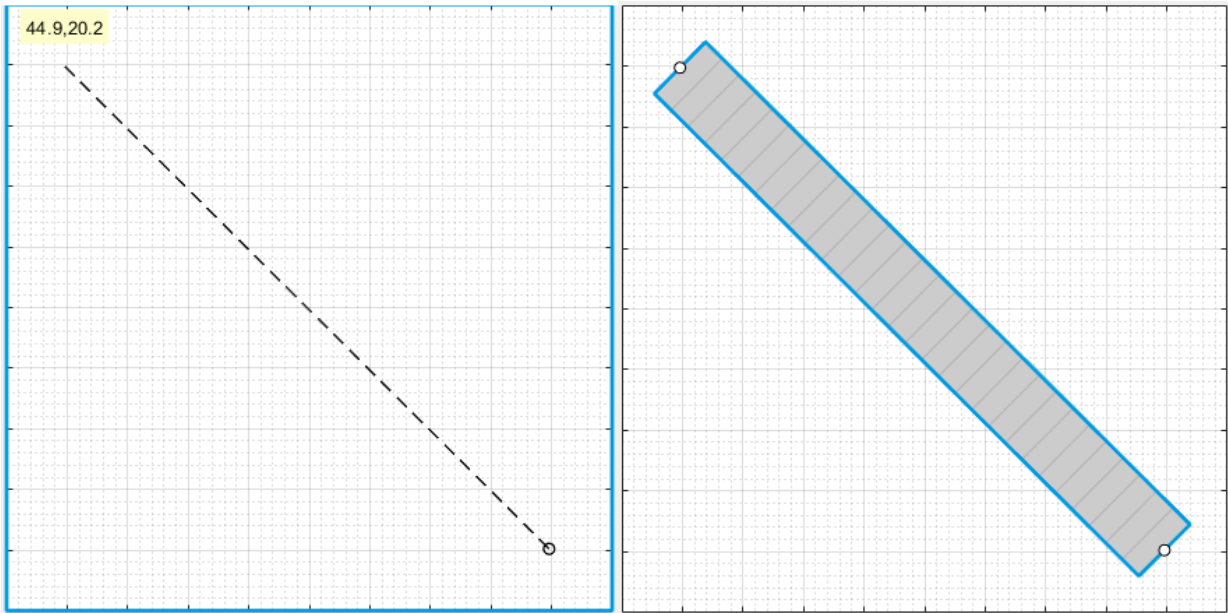
Create a New Driving Scenario

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

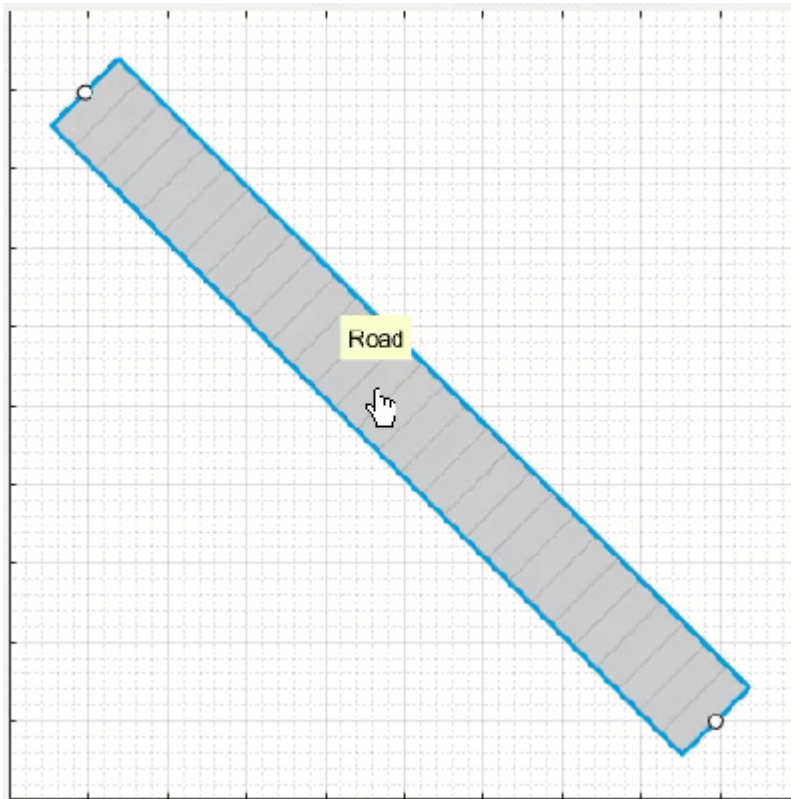
Add a Road

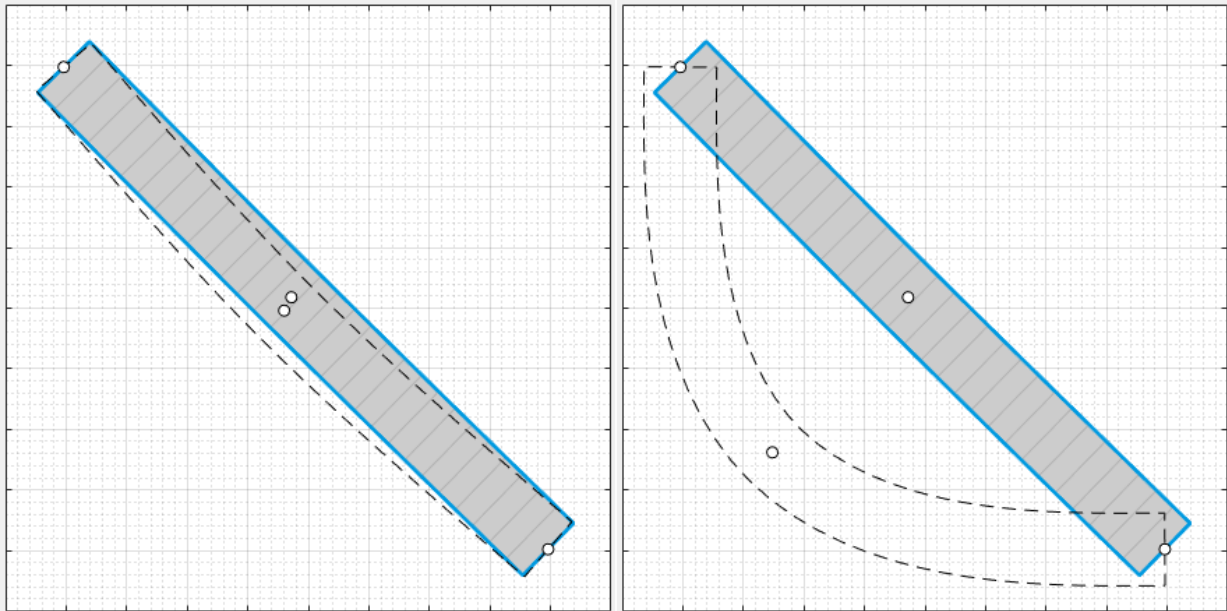
Add a curved road to the scenario canvas. On the app toolbar, click **Add Road**. Then click one corner of the canvas, extend the road to the opposite corner, and double-click to create the road.





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.

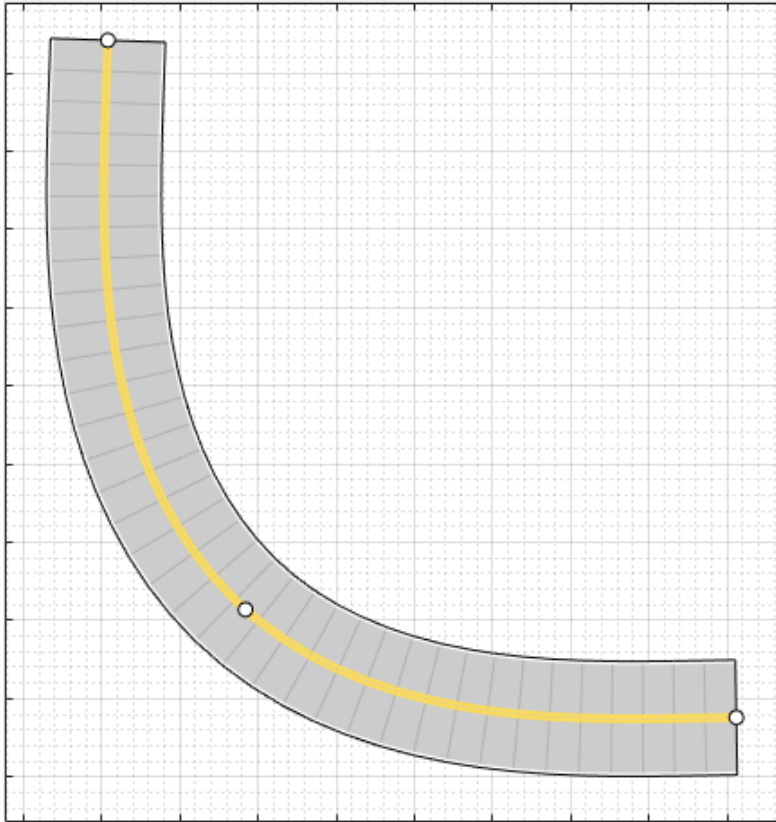




To adjust the road further, you can click and drag any of the road centers. To create more complex curves, add more road centers.

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.



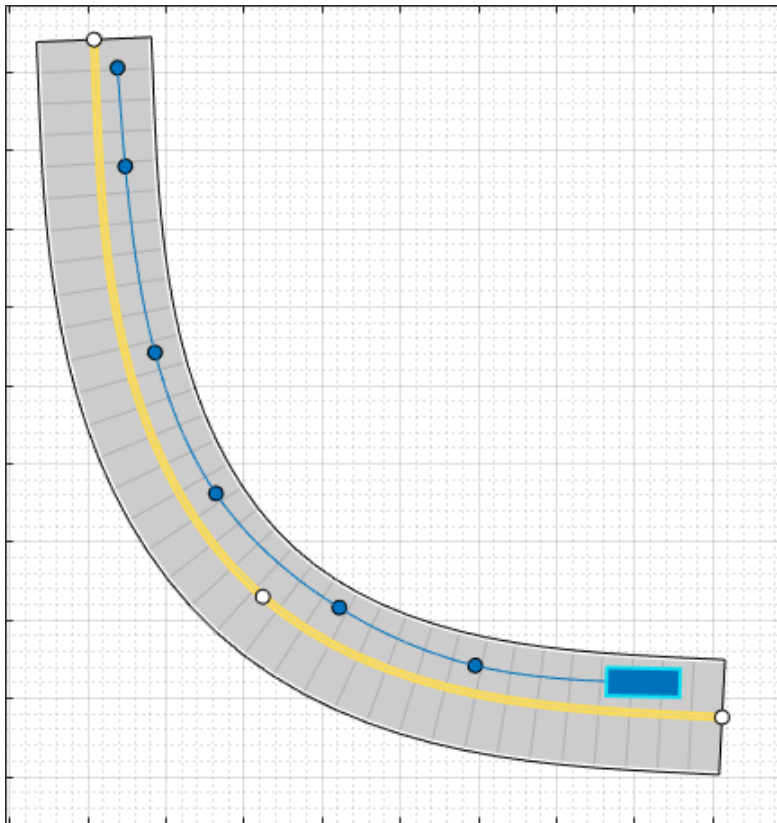
The white, solid lanes markings on either edge of the road indicate the road shoulder. The yellow, double-solid lane marking in the center indicates that the road is two-way. To inspect or modify these lanes, from the **Marking** list, select one of the lanes and modify the lane parameters.

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

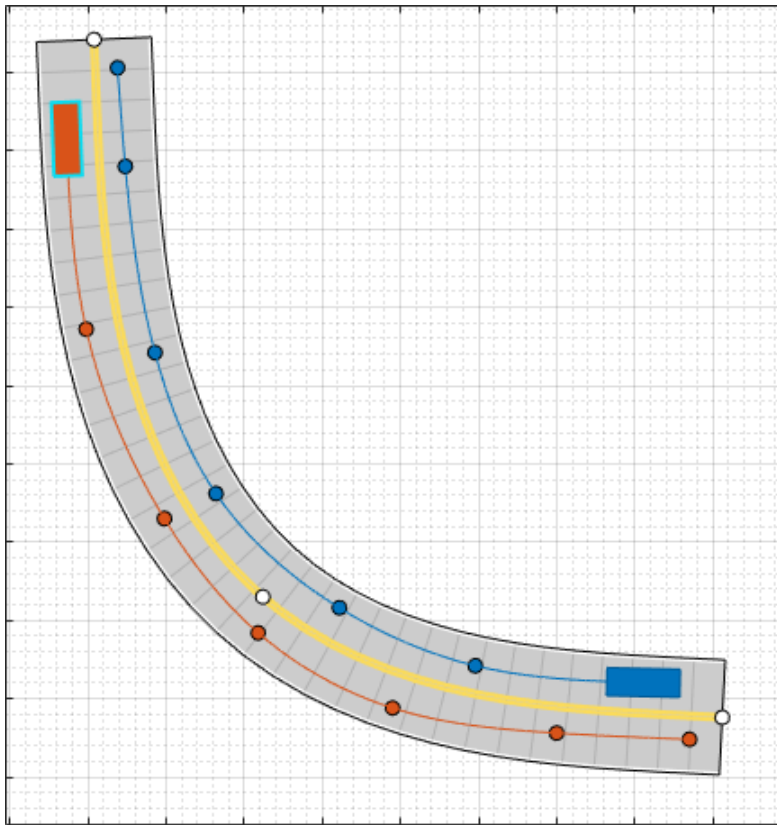
To add the ego vehicle, right-click one end of the road, and select **Add Car**. To specify the trajectory of the car, right-click the car, select **Add Waypoints**, and add waypoints along the road for the car to pass through. After you add the last waypoint along the road, press **Enter**. The car 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.

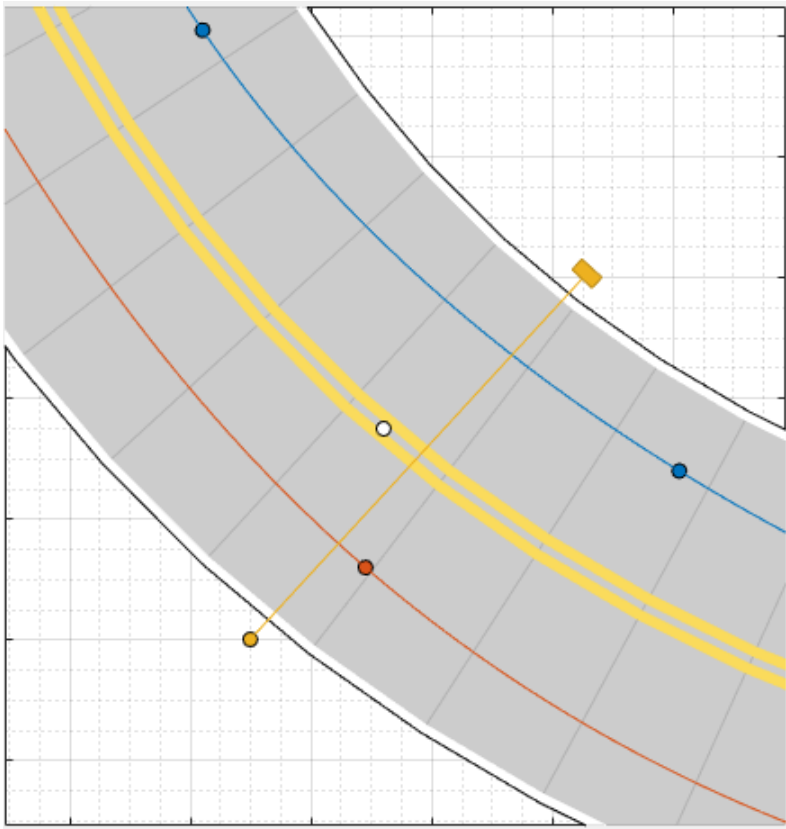
Add Second Car

Add a vehicle for the ego vehicle to detect. On the app toolbar, 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.

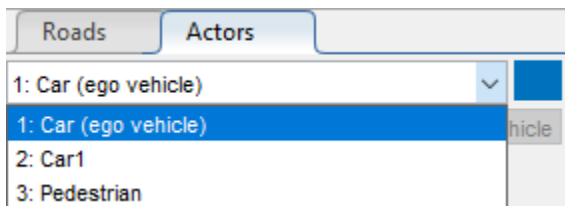


Add a Pedestrian

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.



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

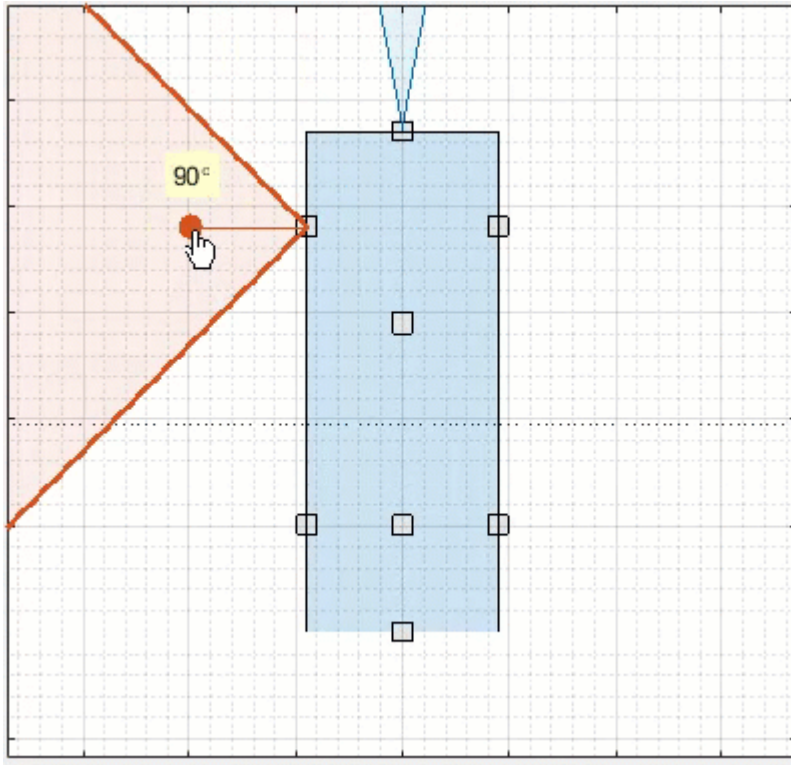
Configure the Sensor Canvas button .

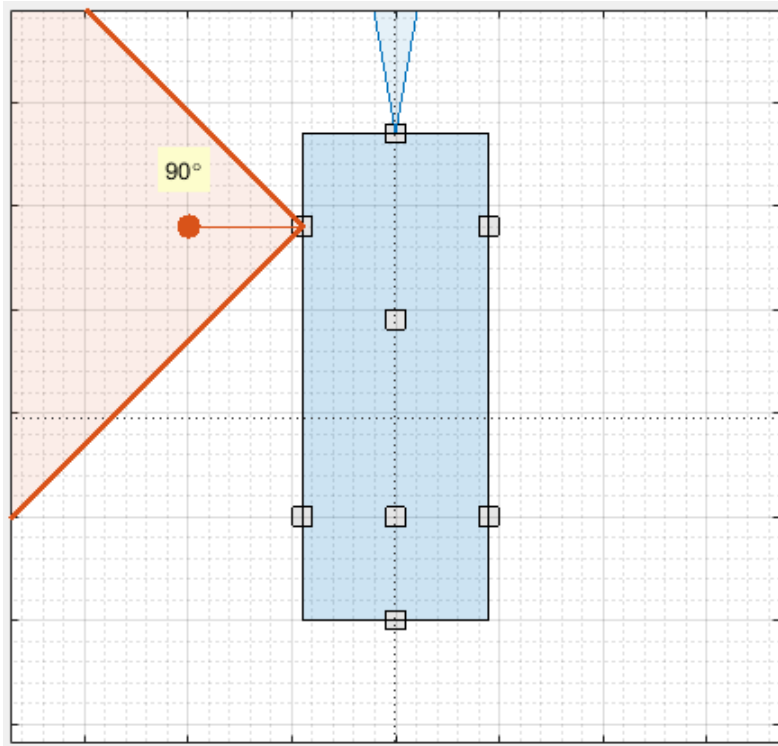
By default, the camera detects only actors and not lanes. To enable lane detections, on the **Sensors** tab in the left pane, expand the **Detection Parameters** section and set **Detection Type** to **Objects & Lanes**. Then expand the **Lane Settings** section and update the settings as needed.

Add Radar

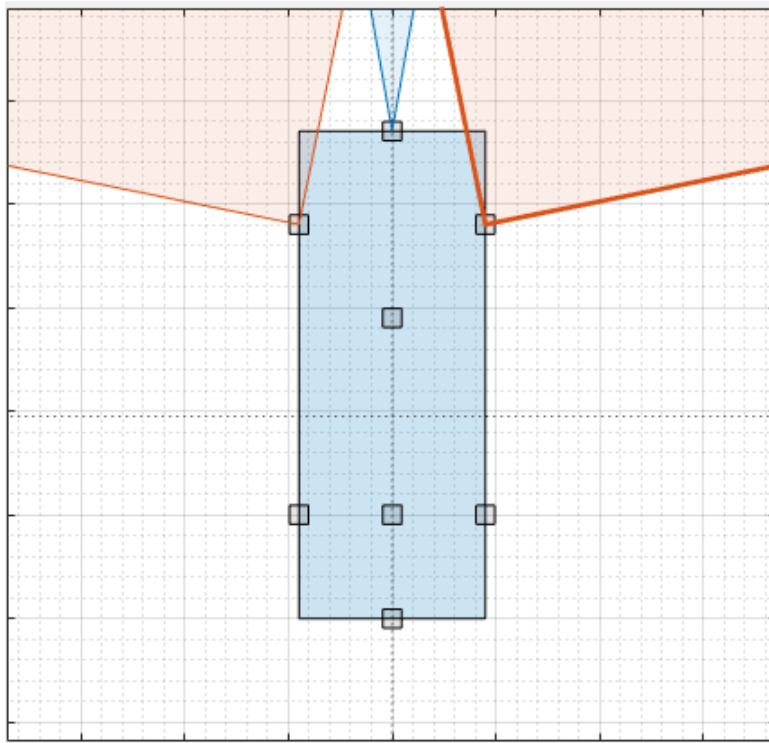
Snap a radar sensor to the front-left wheel. Right-click the predefined sensor location for the wheel and select **Add Radar**. By default, sensors added to the wheels are short range.

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.

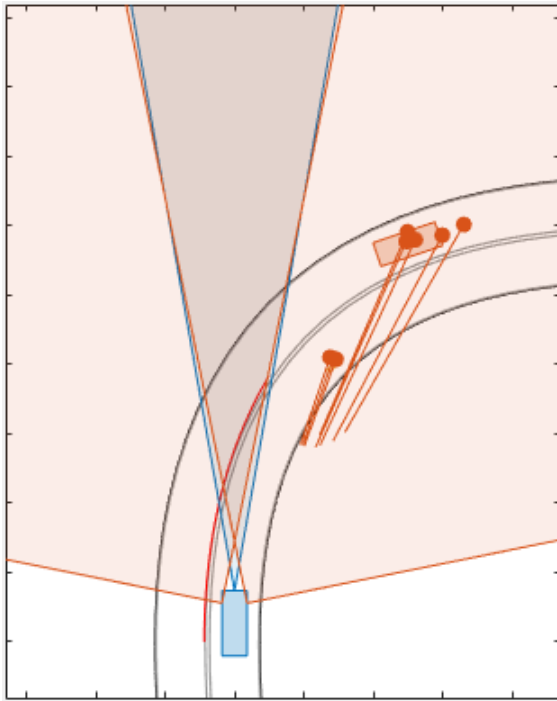



The camera and radar sensors now provide overlapping coverage of the front of the ego vehicle.

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 5-73.

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.

```
drivingScenarioDesigner(scenario)
```

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 5-18
- “Euro NCAP Driving Scenarios in Driving Scenario Designer” on page 5-41
- “Import OpenDRIVE Roads into Driving Scenario” on page 5-61
- “Create Driving Scenario Variations Programmatically” on page 5-73
- “Test Open-Loop ADAS Algorithm Using Driving Scenario” on page 5-93
- “Test Closed-Loop ADAS Algorithm Using Driving Scenario” on page 5-99

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.

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.

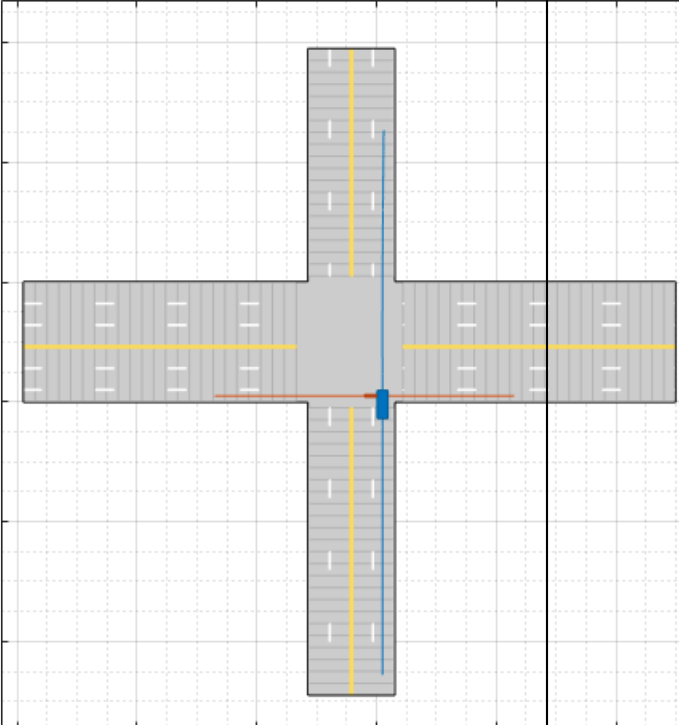
- “Euro NCAP” on page 5-18
- “Intersections” on page 5-18
- “Turns” on page 5-23
- “U-Turns” on page 5-31

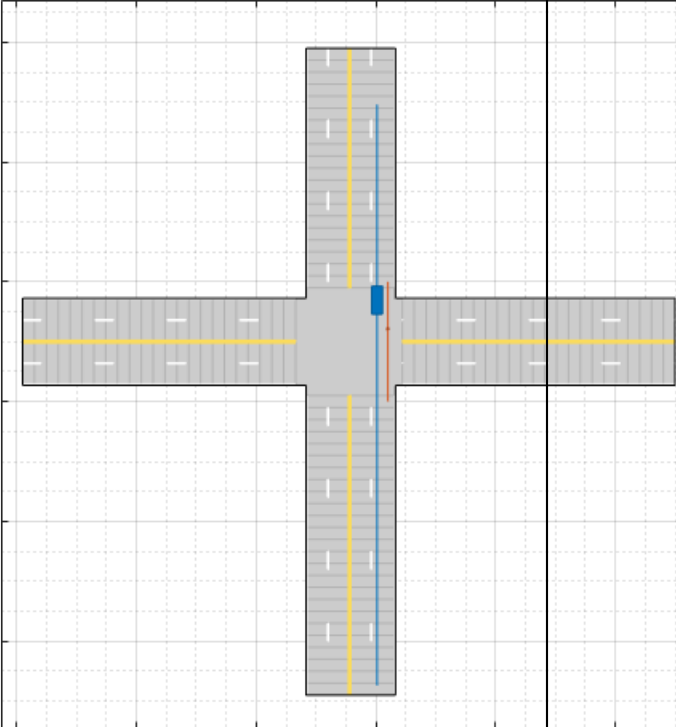
Euro NCAP

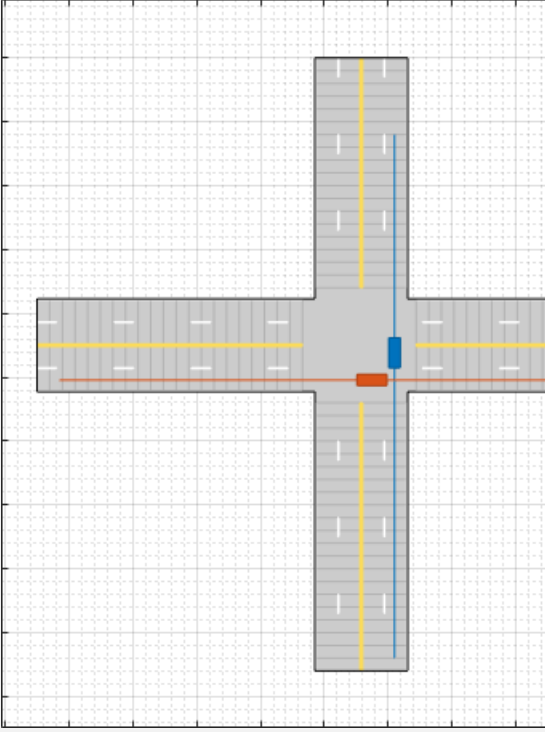
These scenarios represent Euro NCAP test protocols. The app includes scenarios for testing autonomous emergency braking, emergency lane keeping, and lane keep assist systems. For more details, see “Euro NCAP Driving Scenarios in Driving Scenario Designer” on page 5-41.

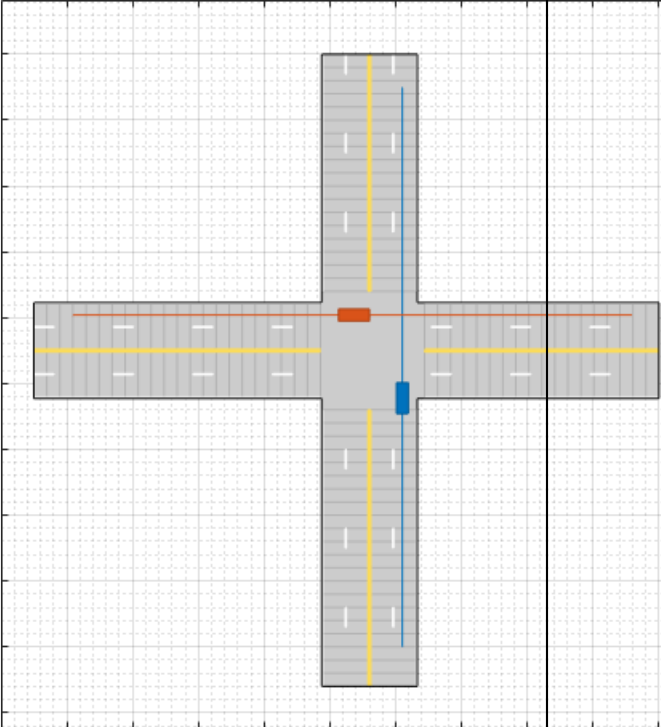
Intersections

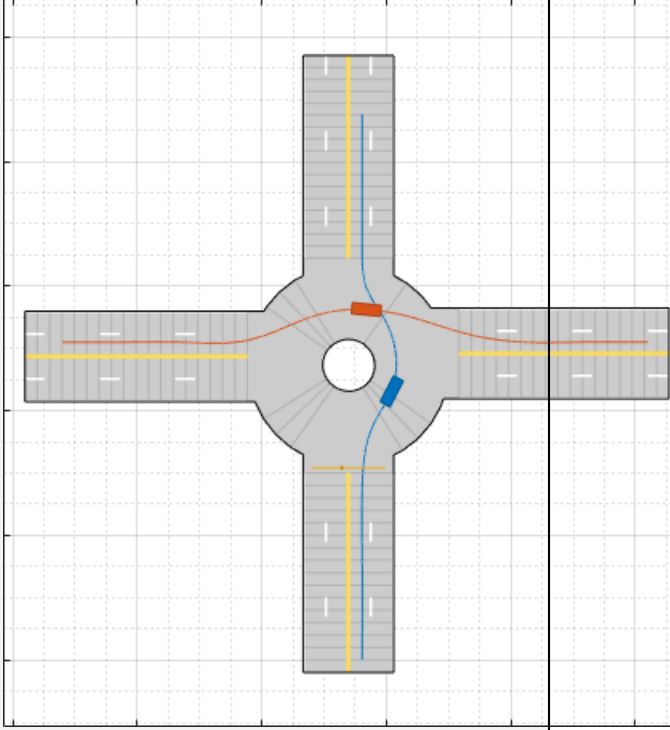
These scenarios involve common traffic patterns at four-way intersections and roundabouts.

File Name	Description
EgoVehicleGoesStraight_BicycleFromLeftGoesStraight_Collision.mat	<p data-bbox="795 305 1323 458">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.</p> 

File Name	Description
<p>EgoVehicleGoesStraight_PedestrianToRightGoesStraight.mat</p>	<p>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.</p> 

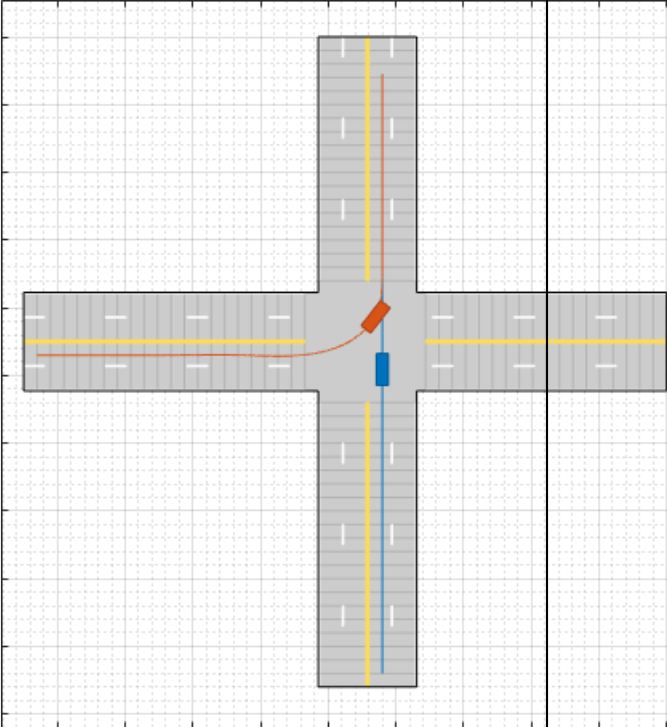
File Name	Description
EgoVehicleGoesStraight_VehicleFromLeftGoesStraight.mat	<p>The ego vehicle travels north and goes straight through an intersection. A vehicle coming from the left side of the intersection also goes straight. The ego vehicle crosses in front of the other vehicle.</p> 

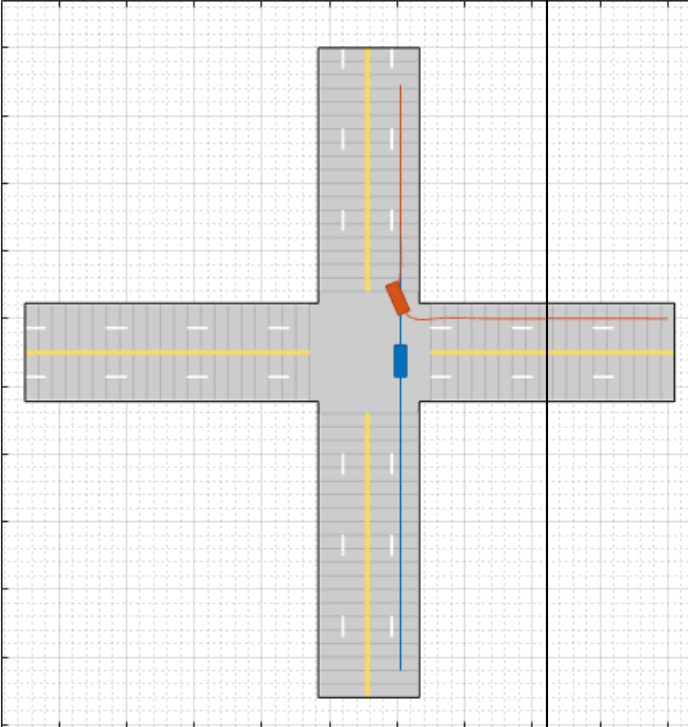
File Name	Description
<p>EgoVehicleGoesStraight_VehicleFromRightGoesStraight.mat</p>	<p>The ego vehicle travels north and goes straight through an intersection. A vehicle coming from the right side of the intersection also goes straight and crosses through the intersection first.</p> 

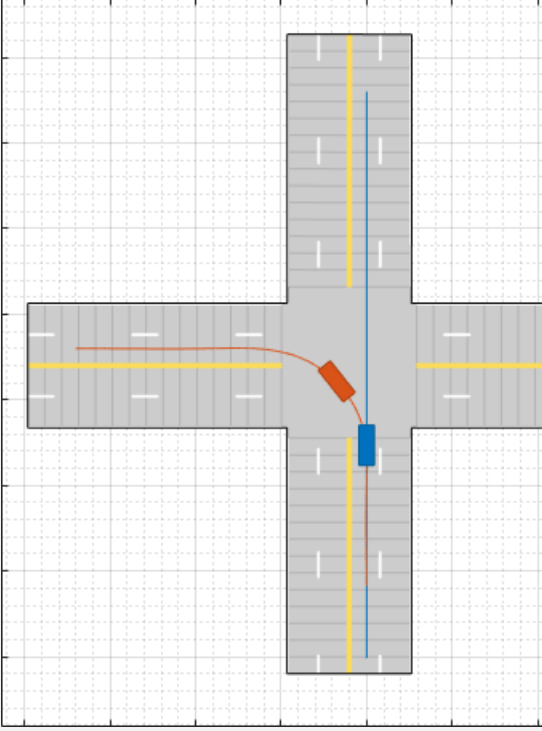
File Name	Description
Roundabout .mat	<p data-bbox="793 303 1329 460">The ego vehicle travels north and crosses the path of a pedestrian while entering a roundabout. The ego vehicle then crosses the path of another vehicle as both vehicles drive through the roundabout.</p>  <p data-bbox="793 491 1461 1223">The diagram illustrates a roundabout intersection on a grid. A central white circle is surrounded by a grey roundabout area. Four roads enter and exit the roundabout from the north, south, east, and west. A blue line represents the ego vehicle's path, starting from the top road and moving clockwise through the roundabout. A red line represents another vehicle's path, starting from the left road and moving clockwise. A blue rectangle represents a pedestrian's path, starting from the top road and moving across the roundabout. A yellow line represents a third vehicle's path, starting from the left road and moving clockwise. The paths are shown as solid lines with dashed lines indicating lane boundaries.</p>

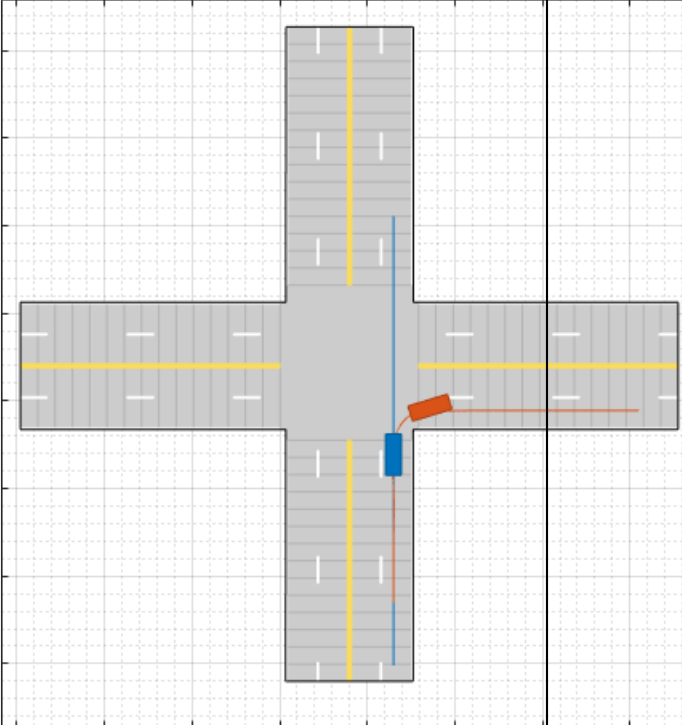
Turns

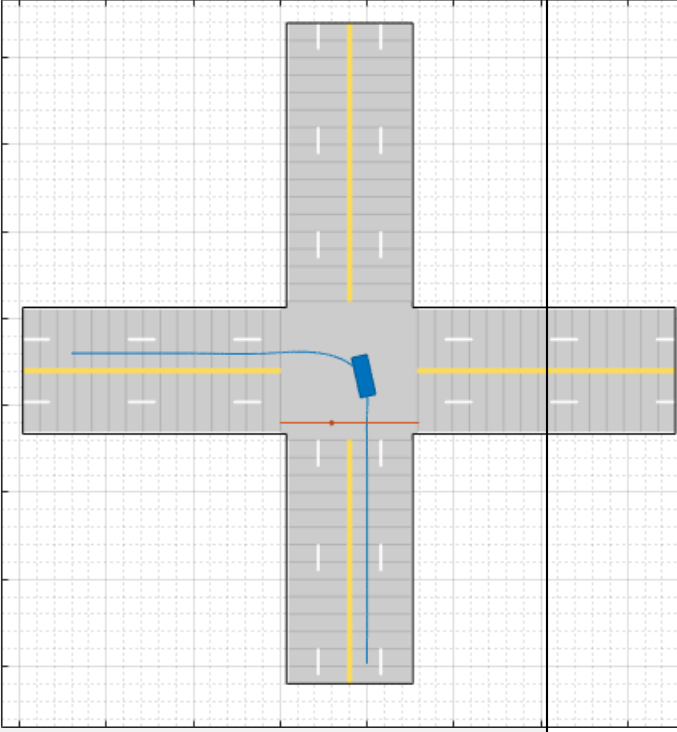
These scenarios involve turns at four-way intersections.

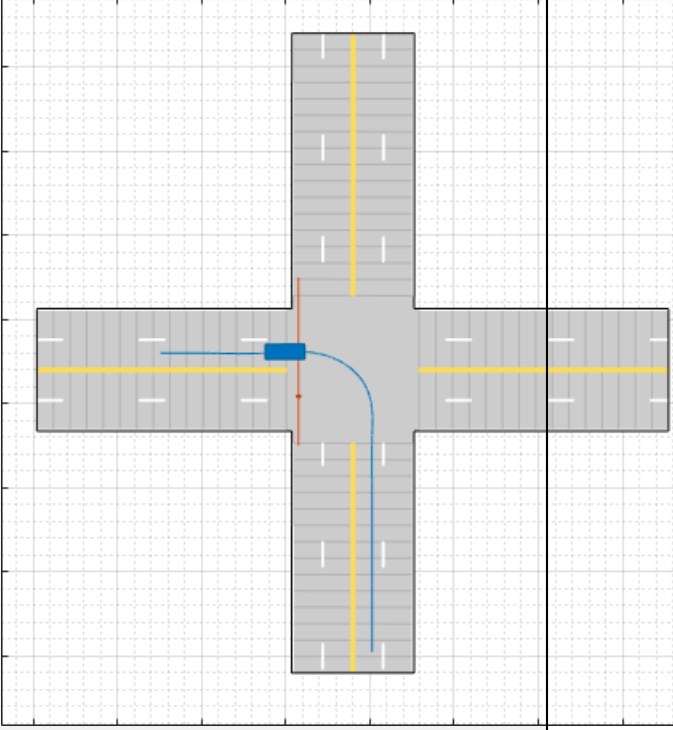
File Name	Description
<p>EgoVehicleGoesStraight_VehicleFromLeftTurnsLeft.mat</p>	<p>The ego vehicle travels north and goes straight through an intersection. A vehicle coming from the left side of the intersection turns left and ends up in front of the ego vehicle.</p> 

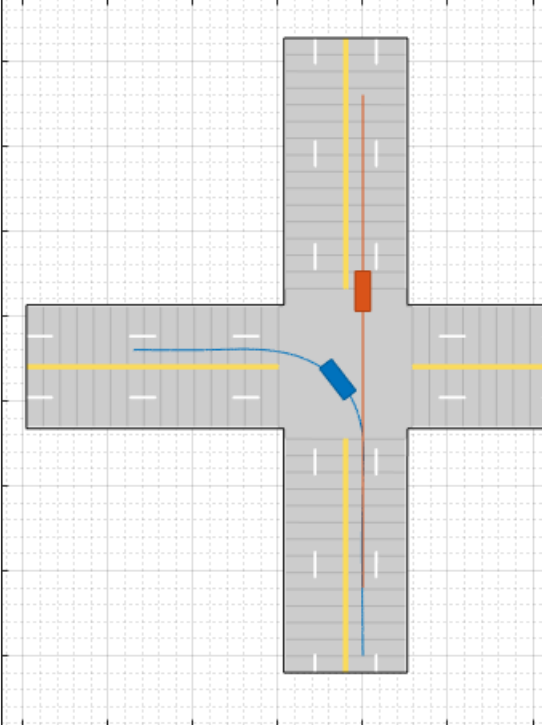
File Name	Description
EgoVehicleGoesStraight_VehicleFromRightTurnsRight.mat	<p>The ego vehicle travels north and goes straight through an intersection. A vehicle coming from the right side of the intersection turns right and ends up in front of the ego vehicle.</p> 

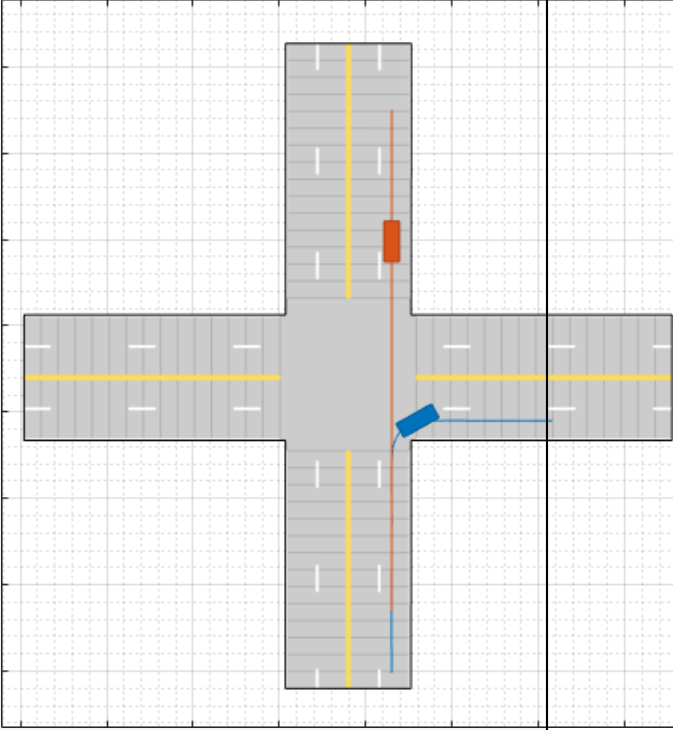
File Name	Description
EgoVehicleGoesStraight_VehicleInFrontTurnsLeft.mat	<p data-bbox="795 305 1317 423">The ego vehicle travels north and goes straight through an intersection. A vehicle in front of the ego vehicle turns left at the intersection.</p> 

File Name	Description
EgoVehicleGoesStraight_VehicleInFrontTurnsRight.mat	<p data-bbox="795 305 1329 423">The ego vehicle travels north and goes straight through an intersection. A vehicle in front of the ego vehicle turns right at the intersection.</p> 

File Name	Description
<p>EgoVehicleTurnsLeft_PedestrianFromLeftGoesStraight.mat</p>	<p>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.</p> 

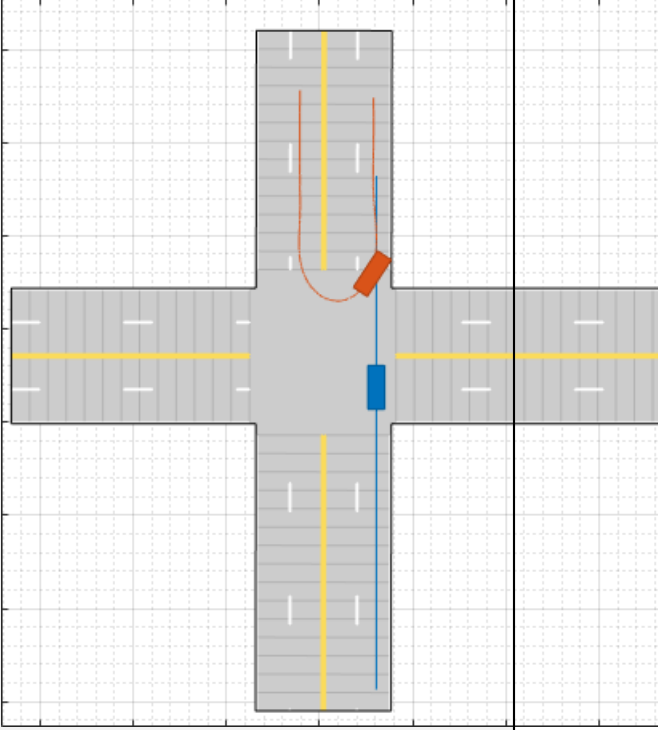
File Name	Description
EgoVehicleTurnsLeft_PedestrianInOppositeLaneGoesStraight.mat	<p data-bbox="795 305 1332 489">The ego vehicle travels north and turns left at an intersection. A pedestrian in the opposite lane goes straight through the intersection. The ego vehicle completes its turn before the pedestrian finishes crossing the intersection.</p> 

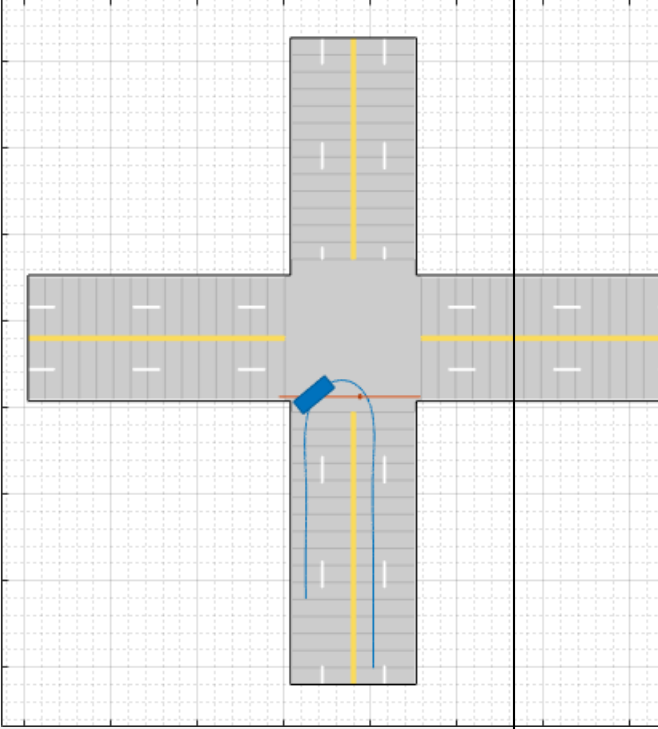
File Name	Description
<p>EgoVehicleTurnsLeft_VehicleInFrontGoesStraight.mat</p>	<p>The ego vehicle travels north and turns left at an intersection. A vehicle in front of the ego vehicle goes straight through the intersection.</p> 

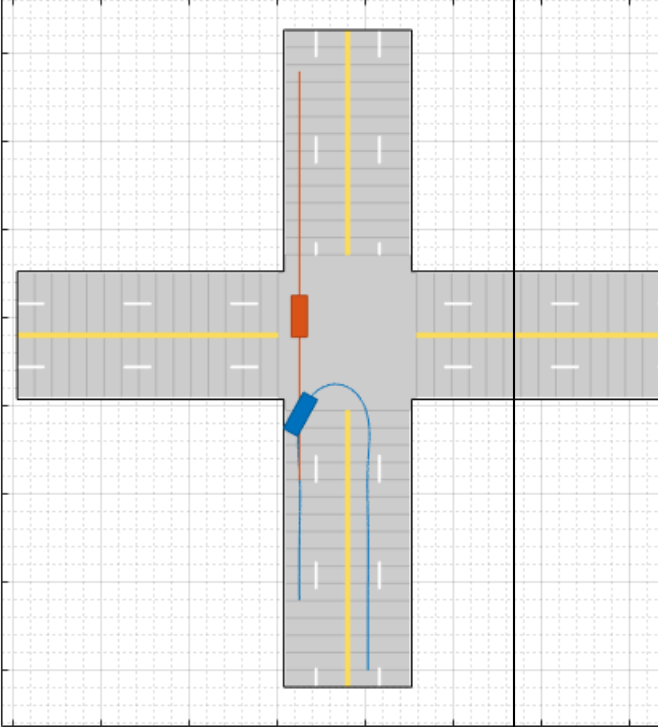
File Name	Description
EgoVehicleTurnsRight_VehicleInFrontGoesStraight.mat	<p data-bbox="795 305 1338 427">The ego vehicle travels north and turns right at an intersection. A vehicle in front of the ego vehicle goes straight through the intersection.</p>  <p data-bbox="795 458 1466 1190">The diagram illustrates a four-way intersection on a grid. A red vehicle is positioned in the northbound lane, moving south and turning right into the eastbound lane. A blue vehicle is positioned in the eastbound lane, moving west through the intersection. Yellow lines indicate the centerlines of the roads. The intersection area is shaded gray.</p>

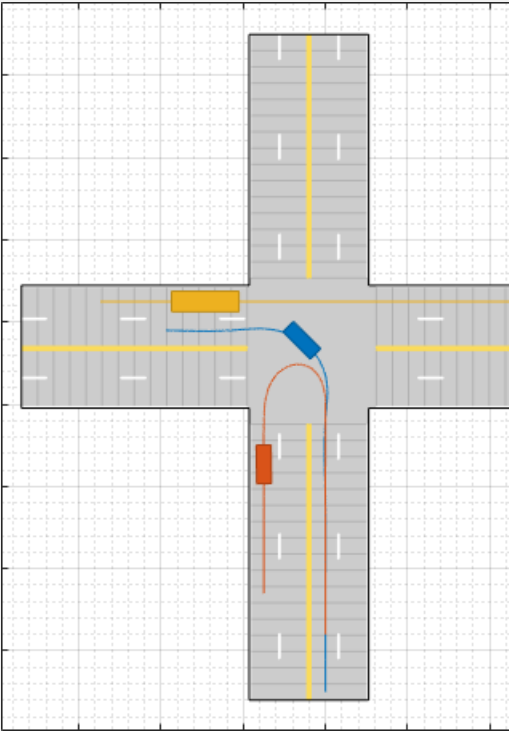
U-Turns

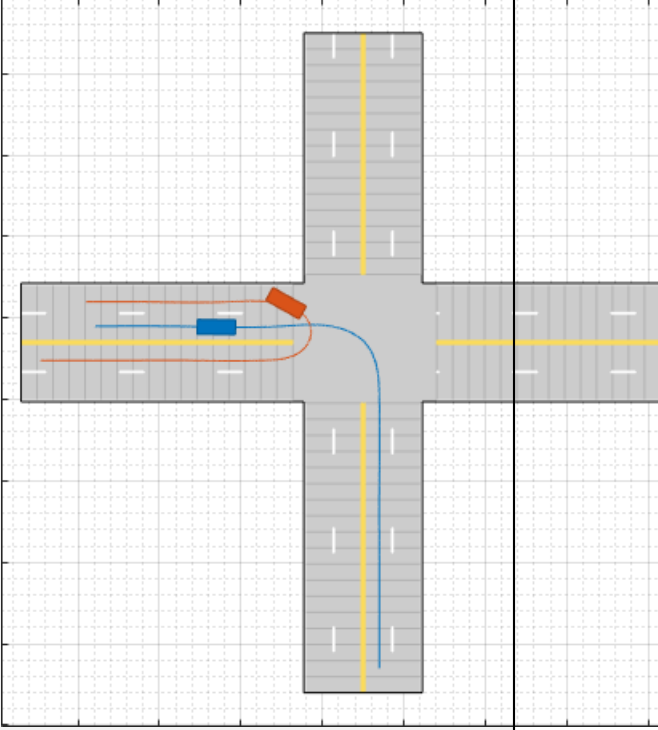
These scenarios involve U-turns at four-way intersections.

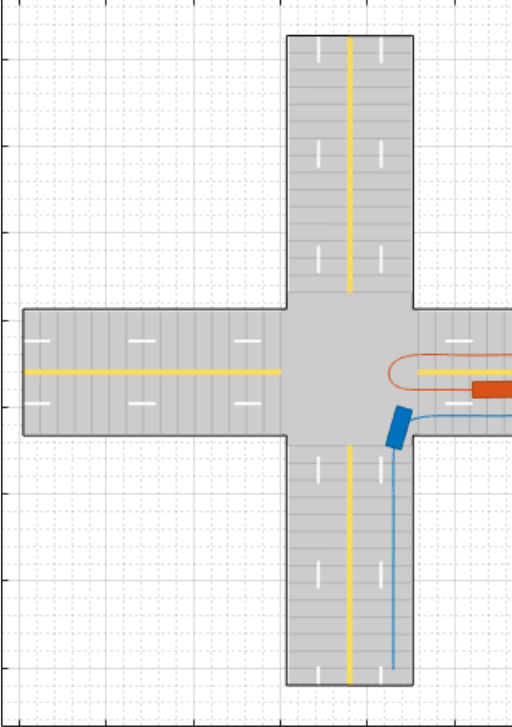
File Name	Description
<p>EgoVehicleGoesStraight_VehicleInOppLaneMakesUTurn.mat</p>	<p>The ego vehicle travels north and goes straight through an intersection. A vehicle in the opposite lane makes a U-turn. The ego vehicle ends up behind the vehicle.</p> 

File Name	Description
EgoVehicleMakesUTurn_PedestrianFromRightGoesStraight.mat	<p data-bbox="828 305 1325 461">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.</p> 

File Name	Description
<p>EgoVehicleMakesUTurn_VehicleInOppLaneGoesStraight.mat</p>	<p>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.</p> 

File Name	Description
EgoVehicleTurnsLeft_Vehicle1MakesUTurn_Vehicle2GoesStraight.mat	<p>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.</p>  <p>The diagram illustrates a driving scenario at a T-junction intersection. The ego vehicle, represented by a blue car, starts in the northbound lane and turns left into the westbound lane. A red car is shown making a U-turn from the northbound lane into the eastbound lane. A yellow truck is shown approaching from the eastbound lane. The ego vehicle ends up in the lane next to the truck. The intersection is marked with yellow lines, and the surrounding area is a grid of gray squares.</p>

File Name	Description
<p>EgoVehicleTurnsLeft_VehicleFromLeft MakesUTurn.mat</p>	<p>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.</p> 

File Name	Description
EgoVehicleTurnsRight_VehicleFromRightMakesUTurn.mat	<p>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.</p> 

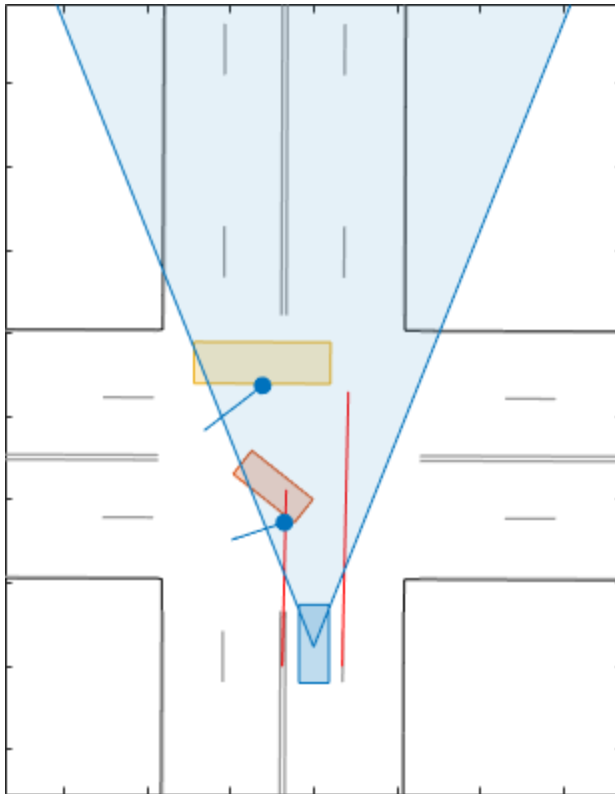
Modify Scenario

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 5-73.

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.

```
drivingScenarioDesigner(scenario)
```

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 | 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 5-41
- “Test Open-Loop ADAS Algorithm Using Driving Scenario” on page 5-93
- “Test Closed-Loop ADAS Algorithm Using Driving Scenario” on page 5-99

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 5-18).

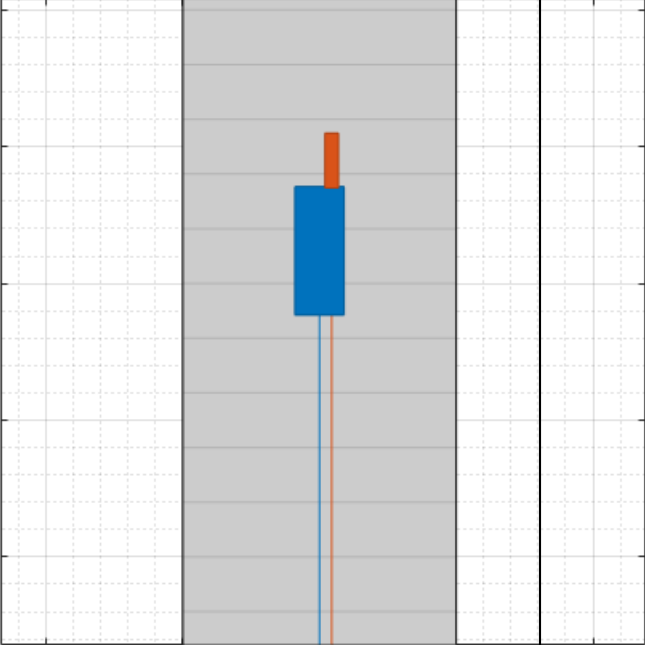
Double-click the **EuroNCAP** folder, and then choose a Euro NCAP scenario from one of these subfolders.

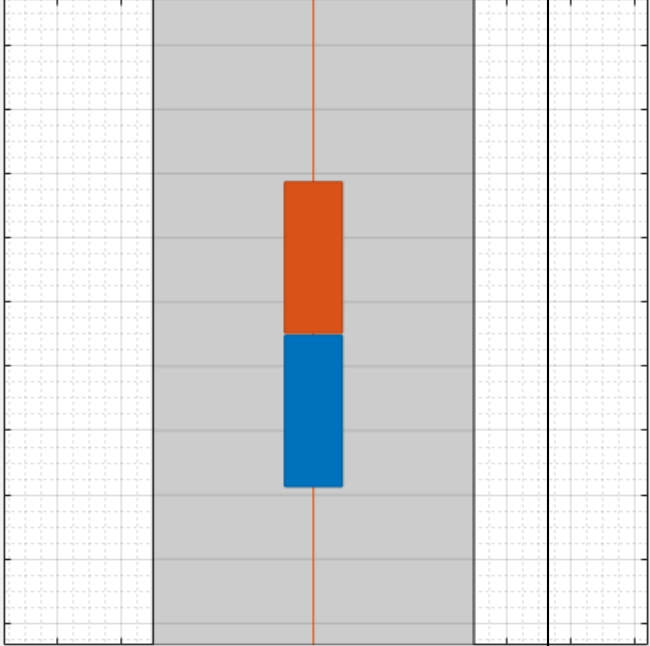
- “Autonomous Emergency Braking” on page 5-41
- “Emergency Lane Keeping” on page 5-47
- “Lane Keep Assist” on page 5-51

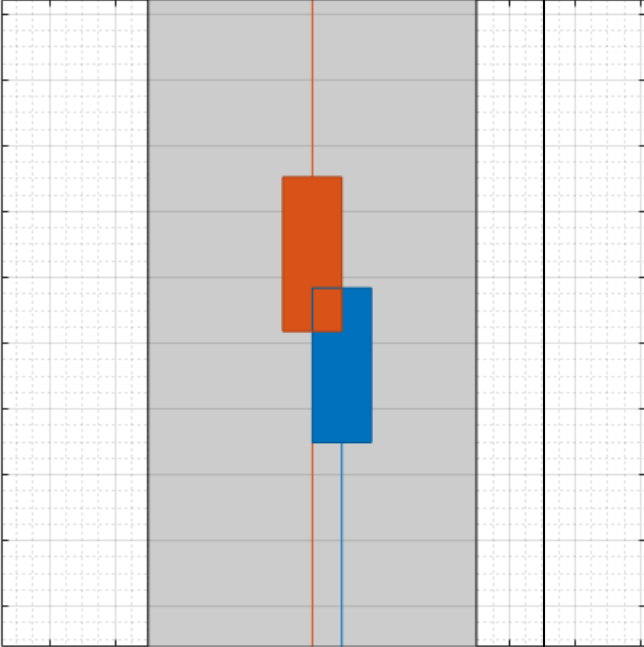
Autonomous Emergency Braking

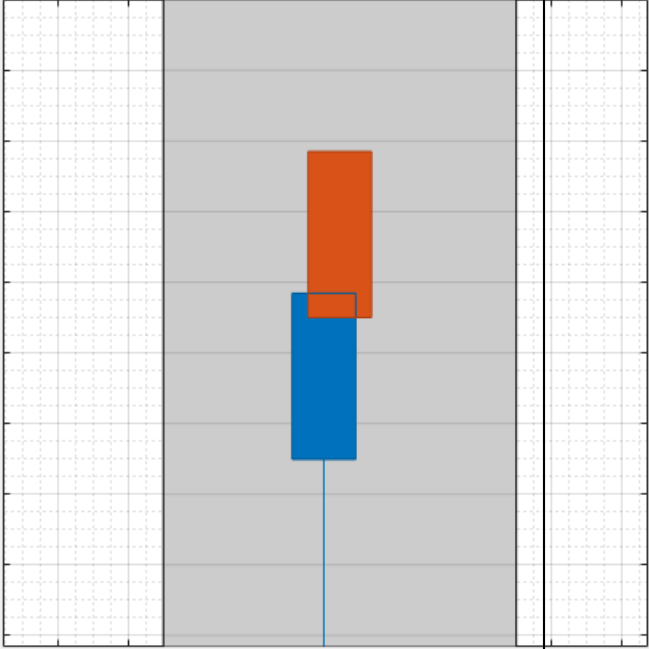
These scenarios are designed to test autonomous emergency braking (AEB) systems. AEB systems warn drivers of impending collisions and automatically apply brakes to prevent collisions or reduce the impact of collisions. Some AEB systems prepare the vehicle and restraint systems for impact.

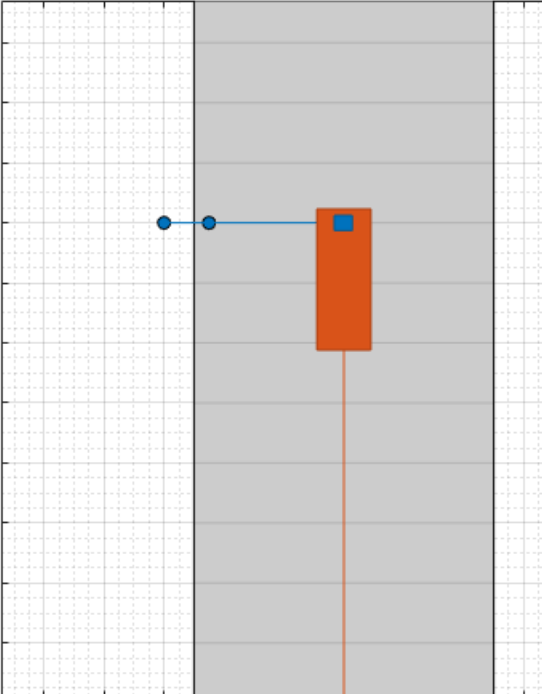
The table lists a subset of the available AEB scenarios. Other AEB scenarios in the folder vary the points of collision, the amount of overlap between vehicles, and the initial gap between vehicles.

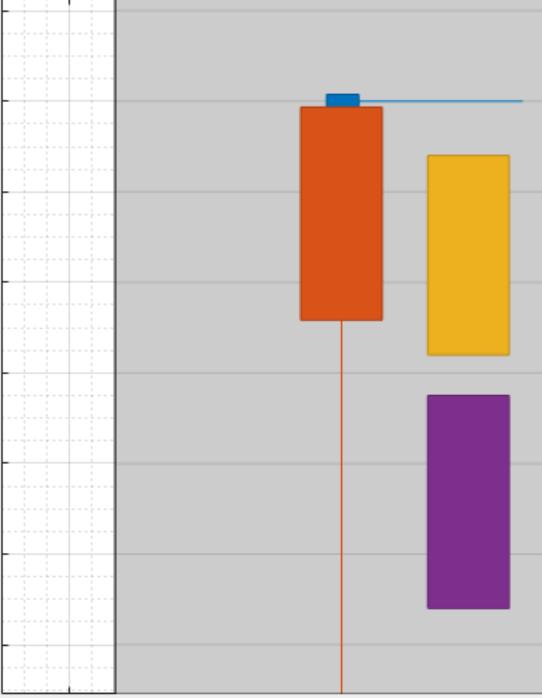
File Name	Description
<p>AEB_Bicyclist_Longitudinal_25width h.mat</p>	<p>The ego vehicle collides with the bicyclist that is in front of it. Before the collision, the bicyclist and ego vehicle are traveling in the same direction along the longitudinal axis. At collision time, the bicycle is 25% of the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

File Name	Description
AEB_CCRb_2_initialGap_12m.mat	<p data-bbox="795 305 1316 487">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.</p> 

File Name	Description
AEB_CCRm_50overlap.mat	<p>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.</p> 

File Name	Description
AEB_CCRs_-75overlap.mat	<p data-bbox="793 303 1329 526">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.</p> 

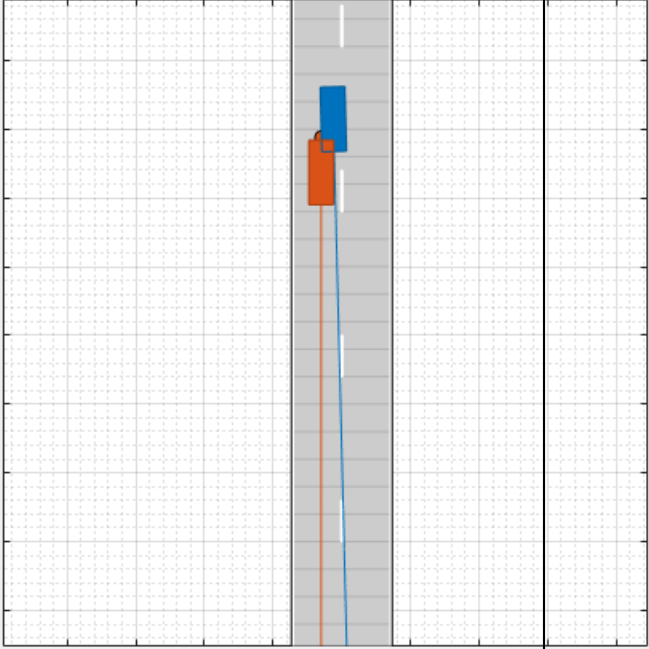
File Name	Description
<p>AEB_Pedestrian_Farside_50width.mat</p>	<p>The ego vehicle collides with a pedestrian who is traveling from the left side of the road, which Euro NCAP test protocols refer to as the far side. These protocols assume that vehicles travel on the right side of the road. Therefore, 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.</p> 

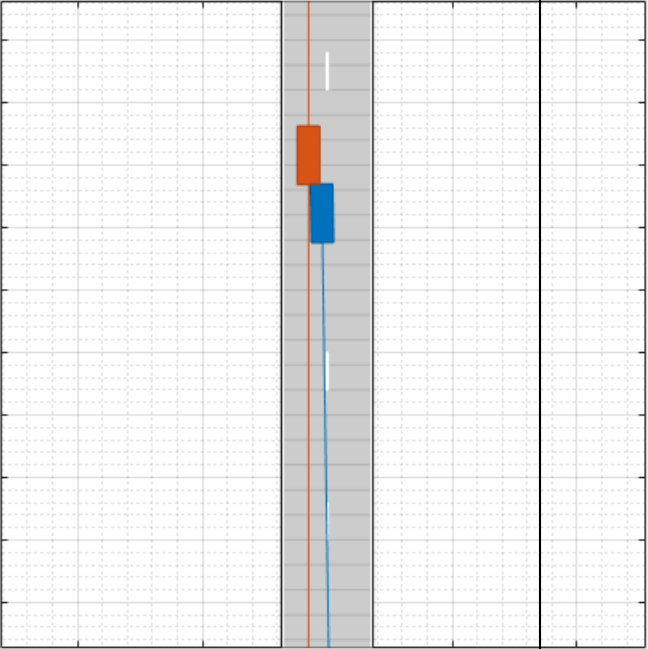
File Name	Description
AEB_PedestrianChild_Nearside_50width.mat	<p>The ego vehicle collides with a pedestrian who is traveling from the right side of the road, which Euro NCAP test protocols refer to as the near side. These protocols assume that vehicles travel on the right side of the road. Therefore, 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.</p> 

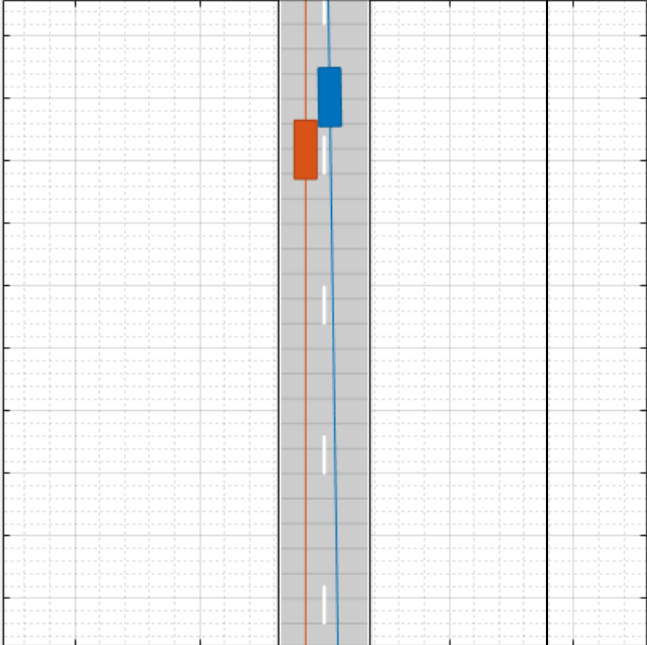
Emergency Lane Keeping

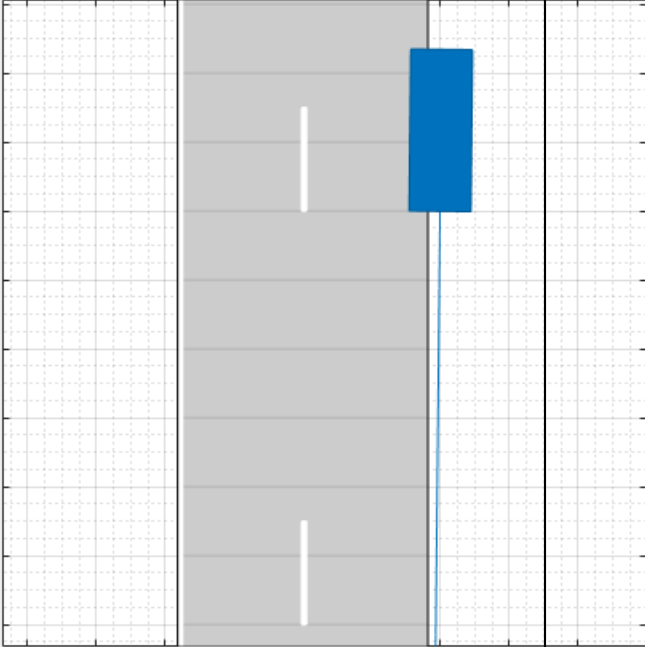
These scenarios are designed to test emergency lane keeping (ELK) systems. ELK systems prevent collisions by warning drivers of impending, unintentional lane departures.

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_Vl at_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	<p>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.</p> 

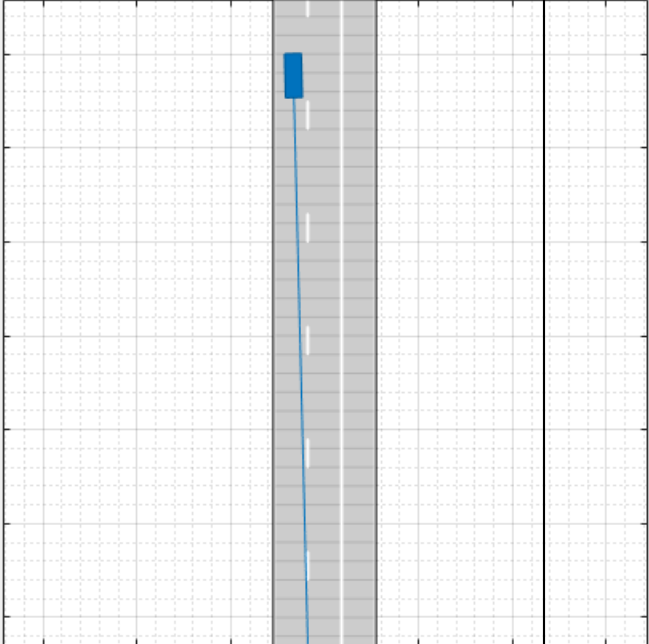
File Name	Description
<p>ELK_OvertakingVeh_Unintent_Vlat_0 .3.mat</p>	<p>The ego vehicle unintentionally changes lanes, overtakes a vehicle in the other lane, and collides with that vehicle. The ego vehicle travels at a lateral velocity of 0.3 m/s.</p> 

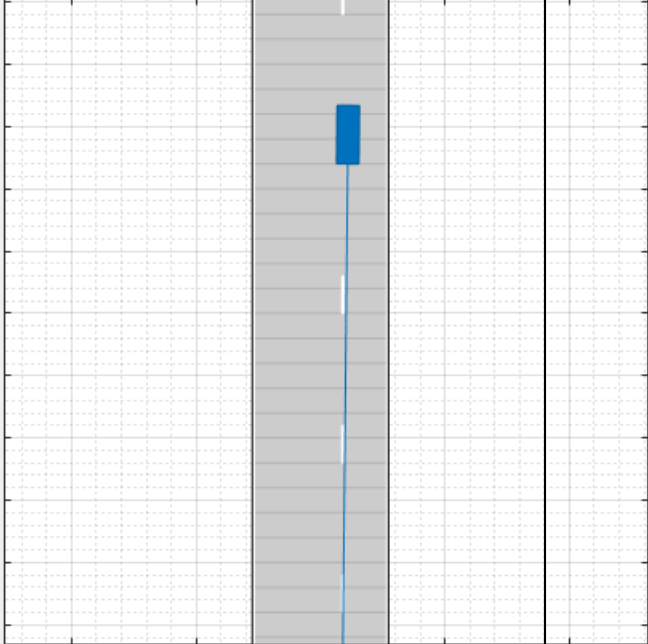
File Name	Description
ELK_RoadEdge_NoBndry_Vlat_0.2.mat	<p>The ego vehicle unintentionally changes lanes and ends up on the road edge. The road edge has no lane boundary markings. The ego vehicle travels at a lateral velocity of 0.2 m/s.</p> 

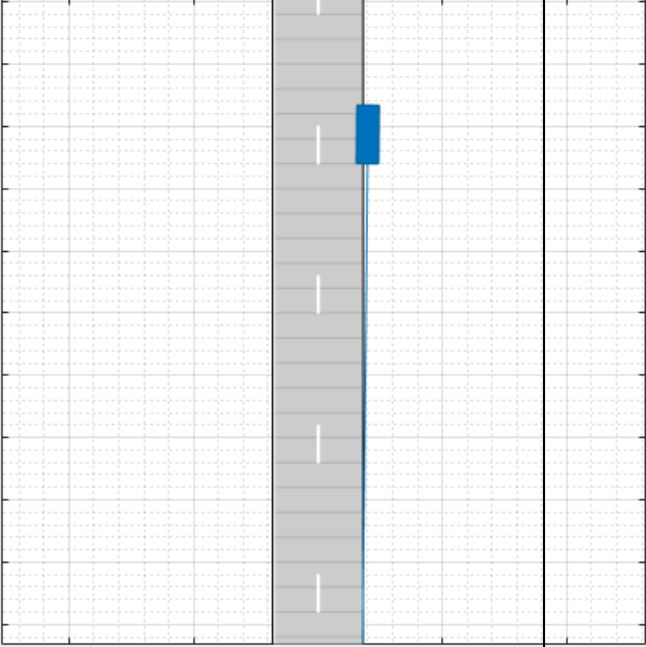
Lane Keep Assist

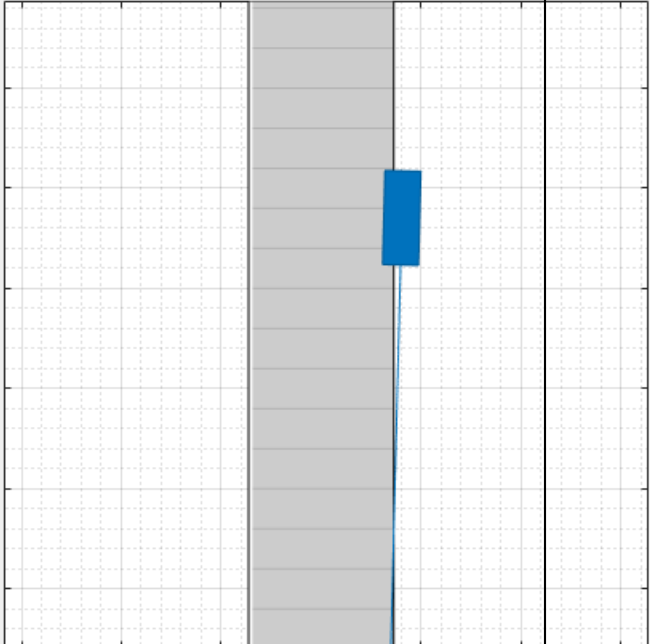
These scenarios are designed to test lane keep assist (LKA) systems. LKA systems detect unintentional lane departures and automatically adjust the steering angle of the vehicle to stay within the lane boundaries.

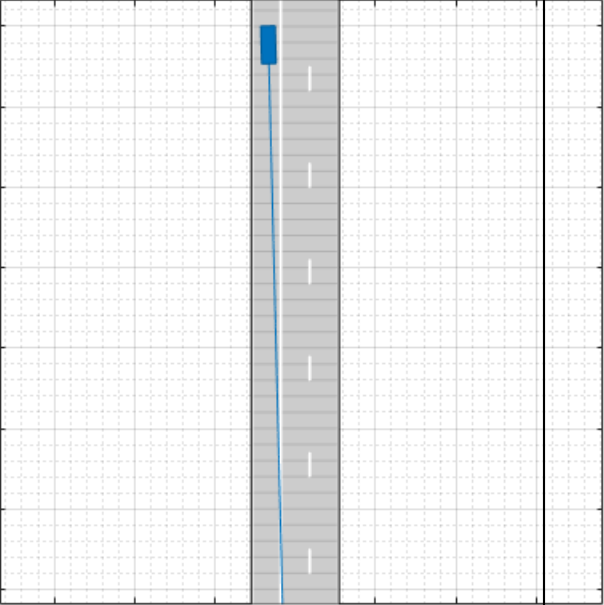
The table lists a subset of the available LKA scenarios. Other LKA scenarios in the folder vary the lateral velocity of the ego vehicle and the lane marking types.

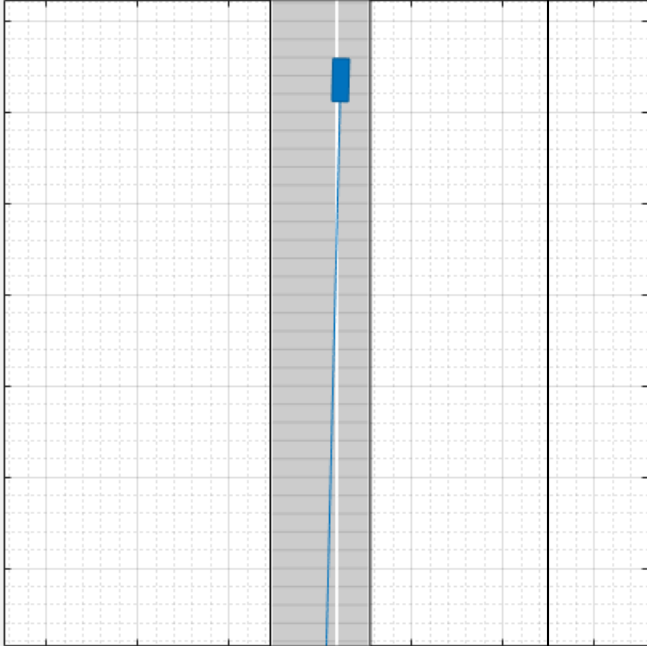
File Name	Description
<p>LKA_DashedLine_Solid_Left_Vlat_0.5.mat</p>	<p>The ego vehicle unintentionally departs from a lane that is dashed on the left and solid on the right. The car departs the lane from the left (dashed) side, traveling at a lateral velocity of 0.5 m/s.</p> 

File Name	Description
LKA_DashedLine_Unmarked_Right_Vla t_0.5.mat	<p data-bbox="795 305 1331 458">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.</p>  <p>The diagram illustrates a lane departure scenario. It features a grid background with a central lane. The lane is bounded by a dashed line on the right and an unmarked line on the left. A blue rectangle representing the ego vehicle is positioned on the dashed side of the lane, indicating it has departed from its intended path. A vertical blue line extends downwards from the vehicle, likely representing its trajectory or a specific parameter.</p>

File Name	Description
LKA_RoadEdge_NoBndry_Vlat_0.5.mat	<p>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.</p> 

File Name	Description
LKA_RoadEdge_NoMarkings_Vlat_0.5.mat	<p data-bbox="793 302 1313 427">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.</p> 

File Name	Description
LKA_SolidLine_Dashed_Left_Vlat_0.5.mat	<p>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.</p> 

File Name	Description
LKA_SolidLine_Unmarked_Right_Vlat_0.5.mat	<p>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.</p> 

Modify Scenario

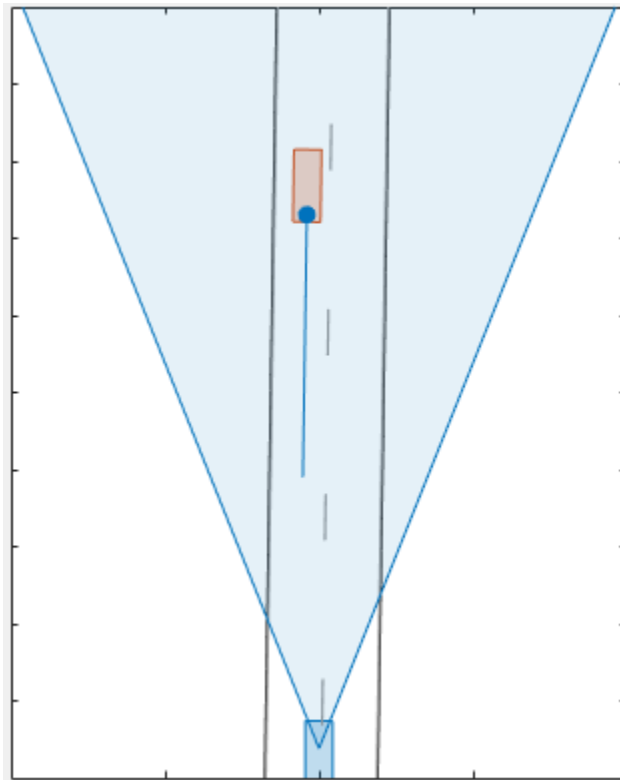
By default, in Euro NCAP scenarios, the ego vehicle does not contain sensors. If you are testing a vehicle sensor, on the app toolstrip, click **Add Camera** or **Add Radar** to add a sensor to the ego vehicle. Then, on the **Sensor** tab, adjust the parameters of the sensors to match your sensor model. 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 5-73.

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 toolbar, 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.

```
drivingScenarioDesigner(scenario)
```

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 5-18
- “Create Driving Scenario Variations Programmatically” on page 5-73
- “Autonomous Emergency Braking with Sensor Fusion”
- “Test Open-Loop ADAS Algorithm Using Driving Scenario” on page 5-93
- “Test Closed-Loop ADAS Algorithm Using Driving Scenario” on page 5-99

External Websites

- Euro NCAP Safety Assist Protocols

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

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

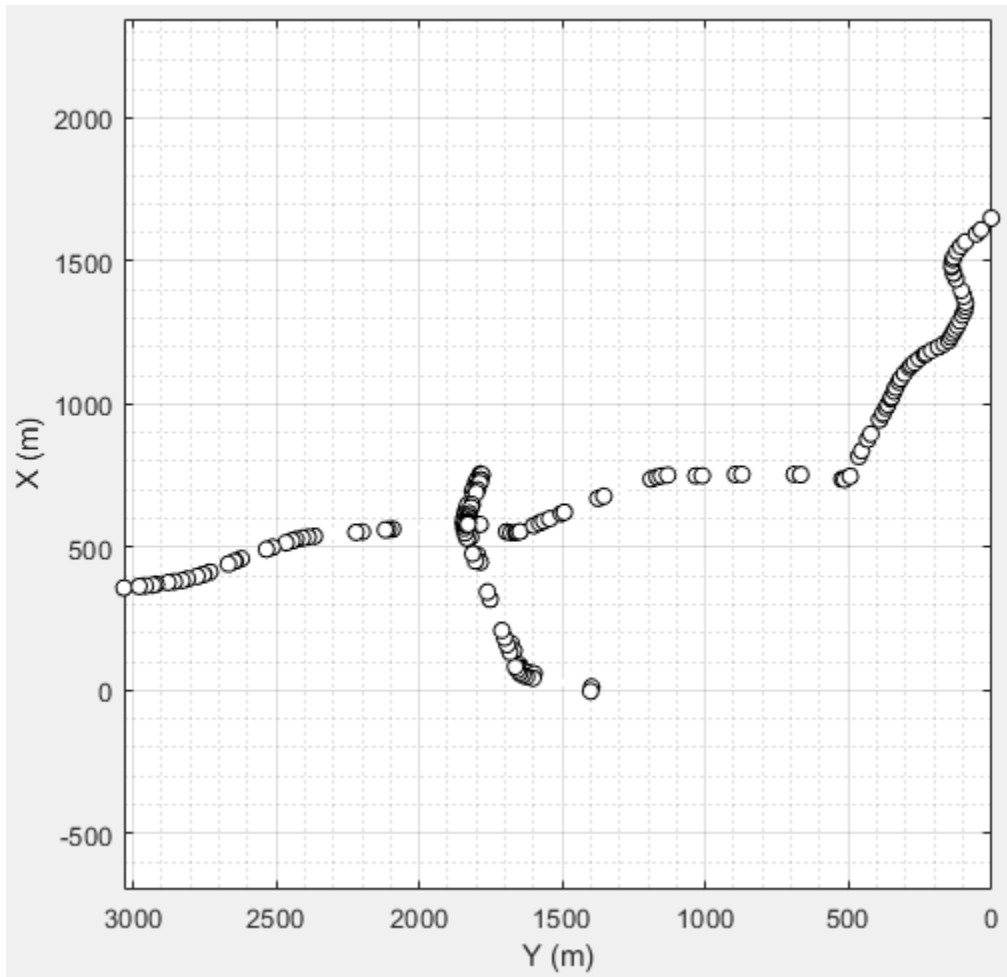
To import an OpenDRIVE file, on the app toolstrip, select **Open > OpenDRIVE Road Network**. The file you select must be a valid OpenDRIVE file of type `.xodr` or `.xml`. In addition, the file must conform with OpenDRIVE format specification version 1.4H.

From your MATLAB root folder, navigate to and open this file:

```
matlabroot/examples/driving/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.

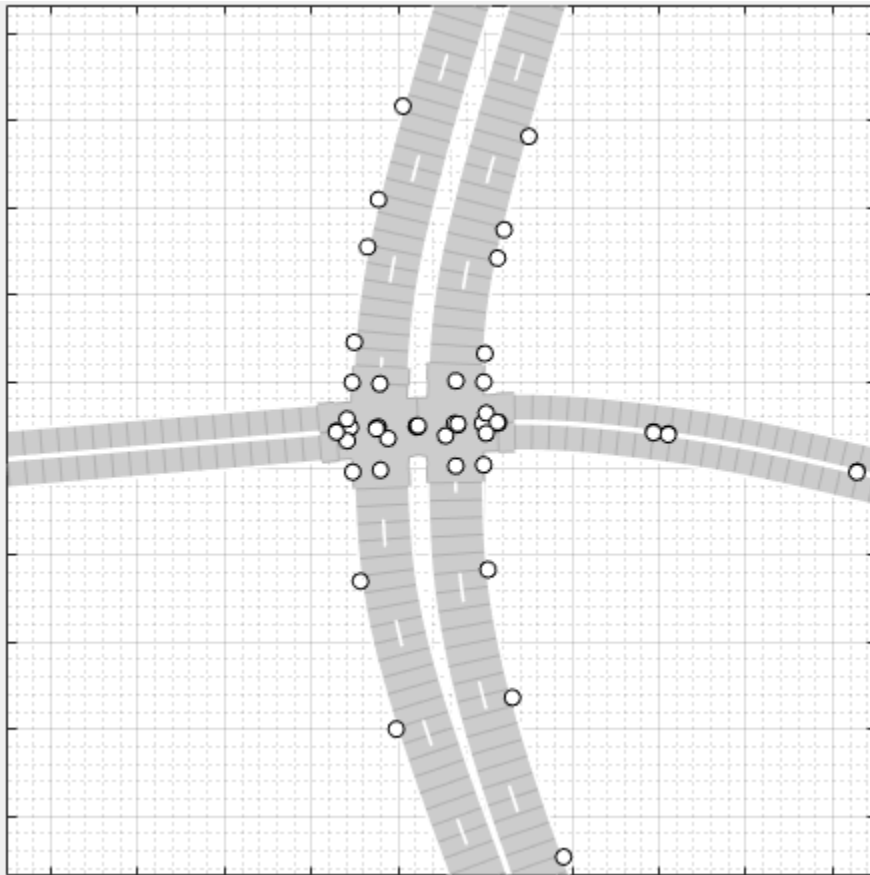
The **Scenario Canvas** of the app displays the imported road network.



The roads in this network are thousands of meters long. You can 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.

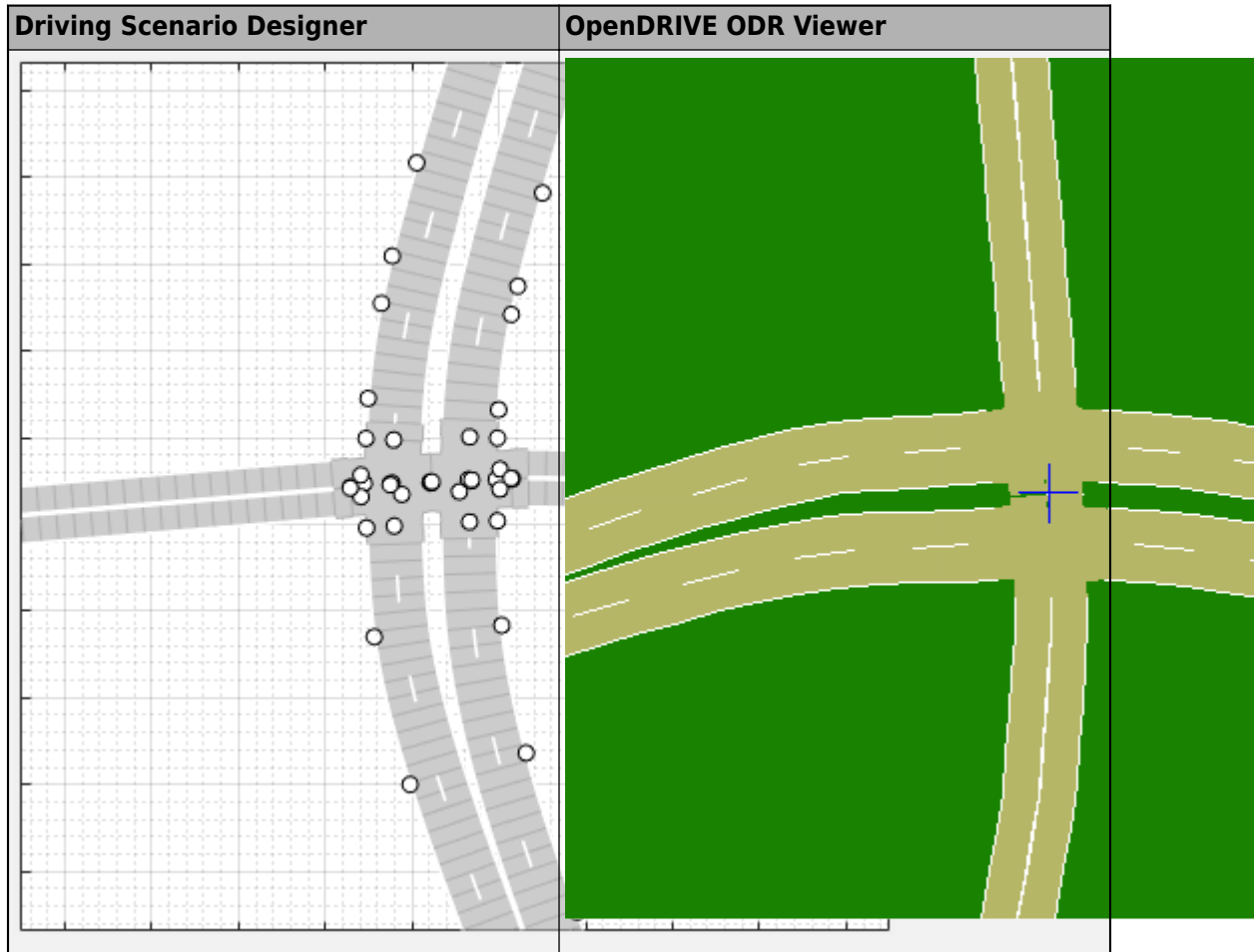
- You can import only lanes, lane type information, and roads. The import of road objects and traffic signals is not supported.
- OpenDRIVE files containing large road networks can take up to several minutes to load. In addition, these road networks can cause slow interactions on the app canvas.

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 are not supported. The app applies the first found marking style to all lanes in the road. For example, if a road has **Dashed** and **Solid** lane markings, the app applies **Dashed** lane markings throughout.
- 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.

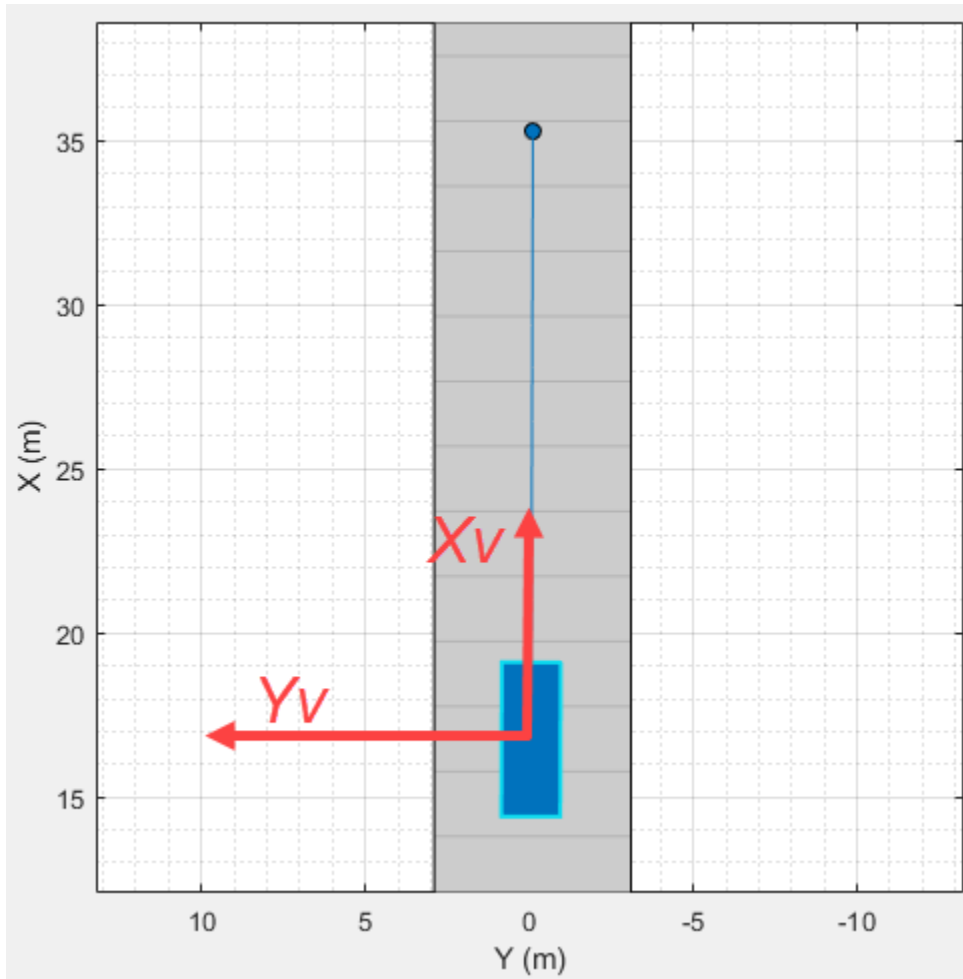


In the OpenDRIVE ODR viewer, the X -axis runs along the bottom of the viewer, and the Y -axis runs along the left side of the viewer.

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.



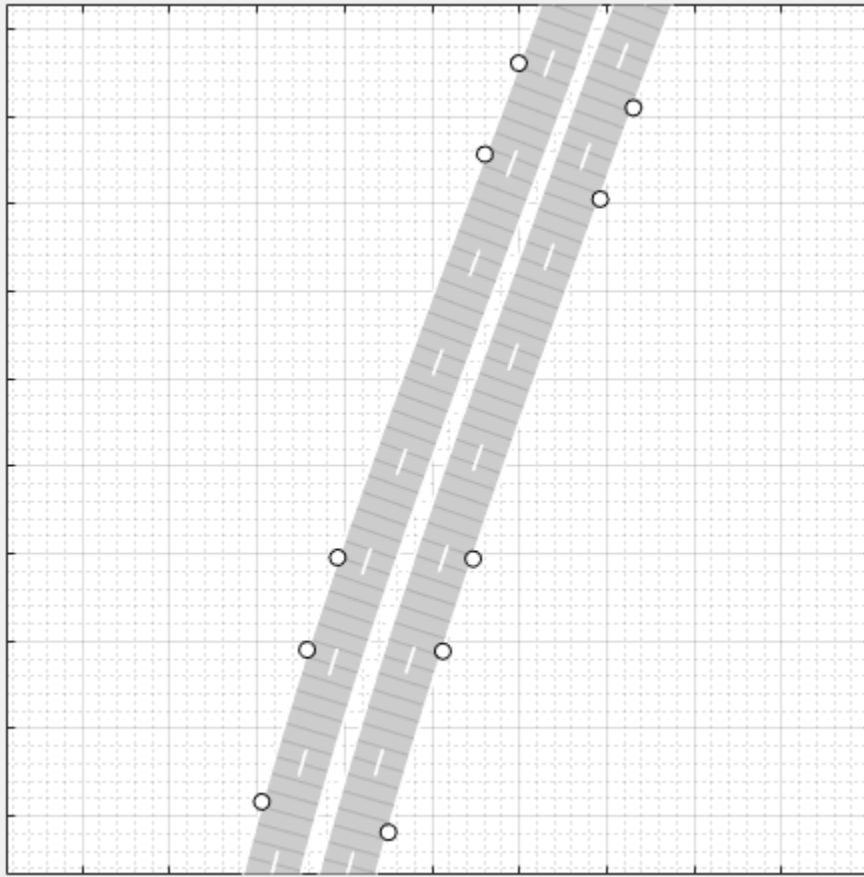
For more details about the coordinate systems, see “Coordinate Systems in Automated Driving Toolbox” on page 1-2.

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.

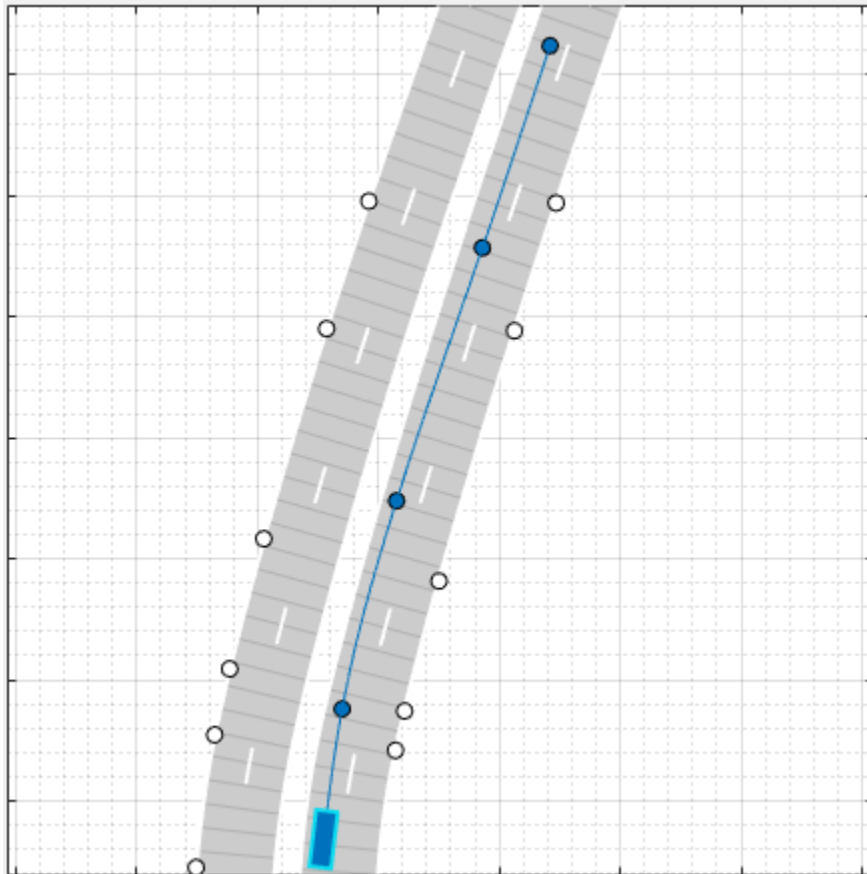
- The lanes on the right side of the highway have their road centers on the right edge.
- The lanes on the left side of the highway have their road centers on the left edge.



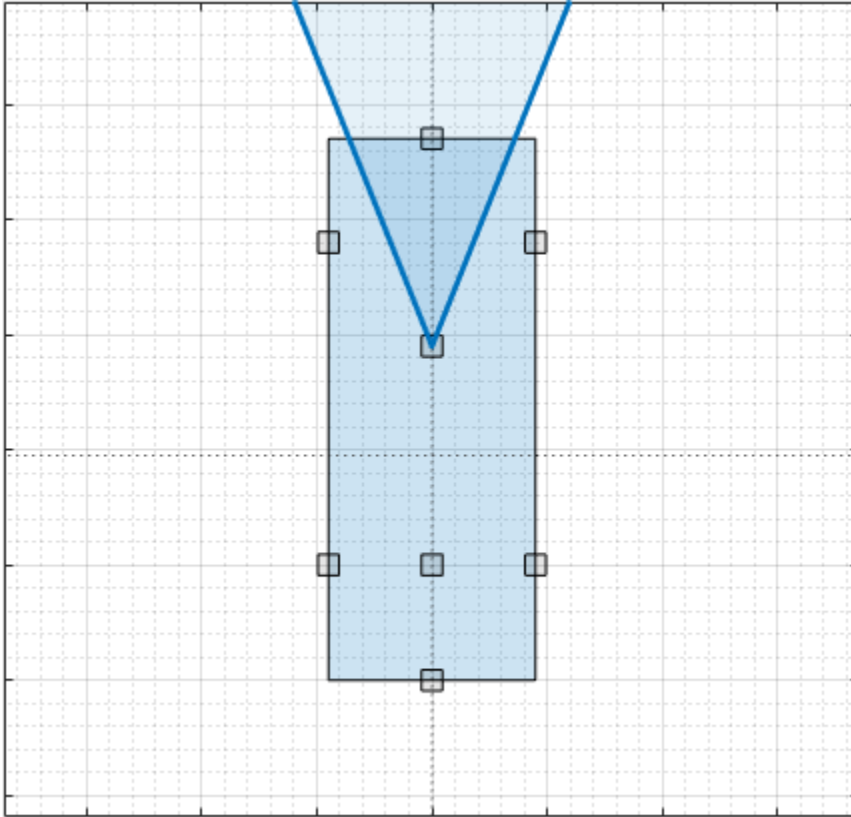
Add Actors and Sensors to Scenario

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.

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.



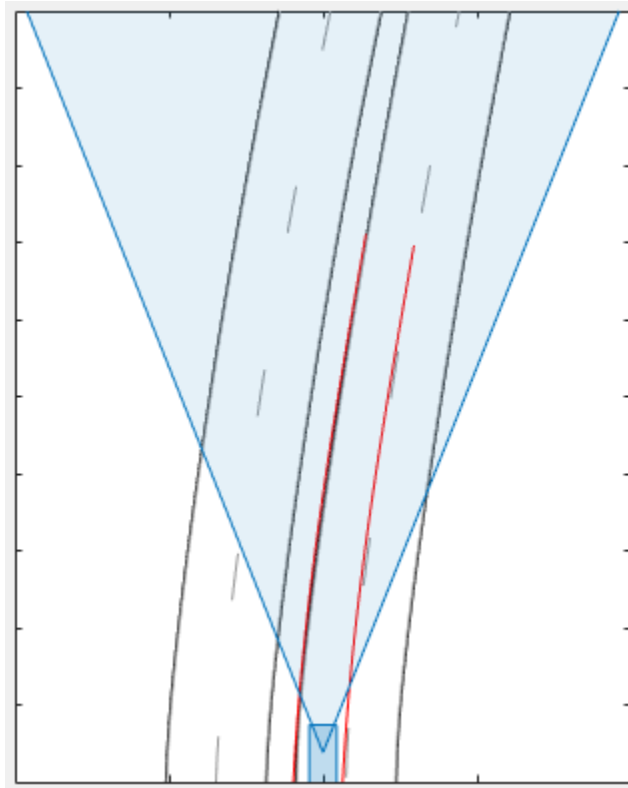
Add a camera sensor to the ego vehicle. On the app toolbar, click **Add Camera**. Then, on the sensor canvas, add the camera to the predefined location representing the front window of the car.



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 toolbar, 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 **ExportExport 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: VIRE 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 5-18
- “Coordinate Systems in Automated Driving Toolbox” on page 1-2

See Also

External Websites

- opendrive.org

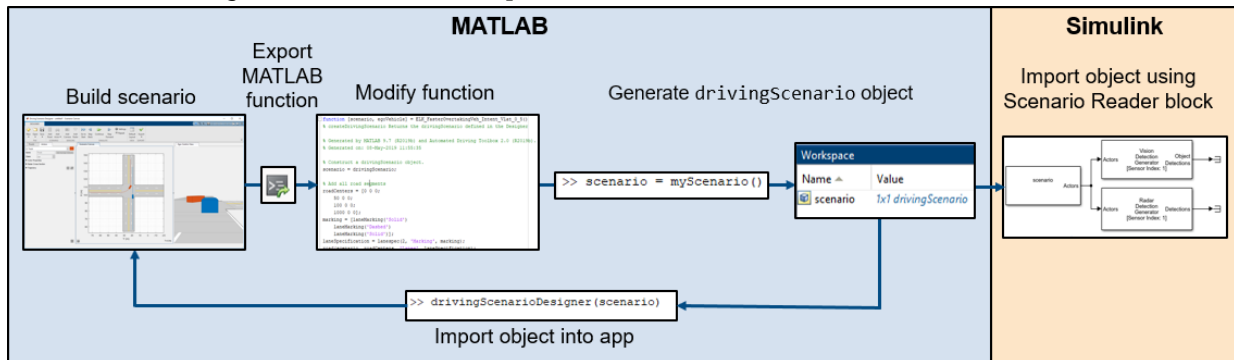
Create Driving Scenario Variations Programmatically

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.



Before beginning this example, add the example file folder to the MATLAB search path.

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

Build Scenario in App

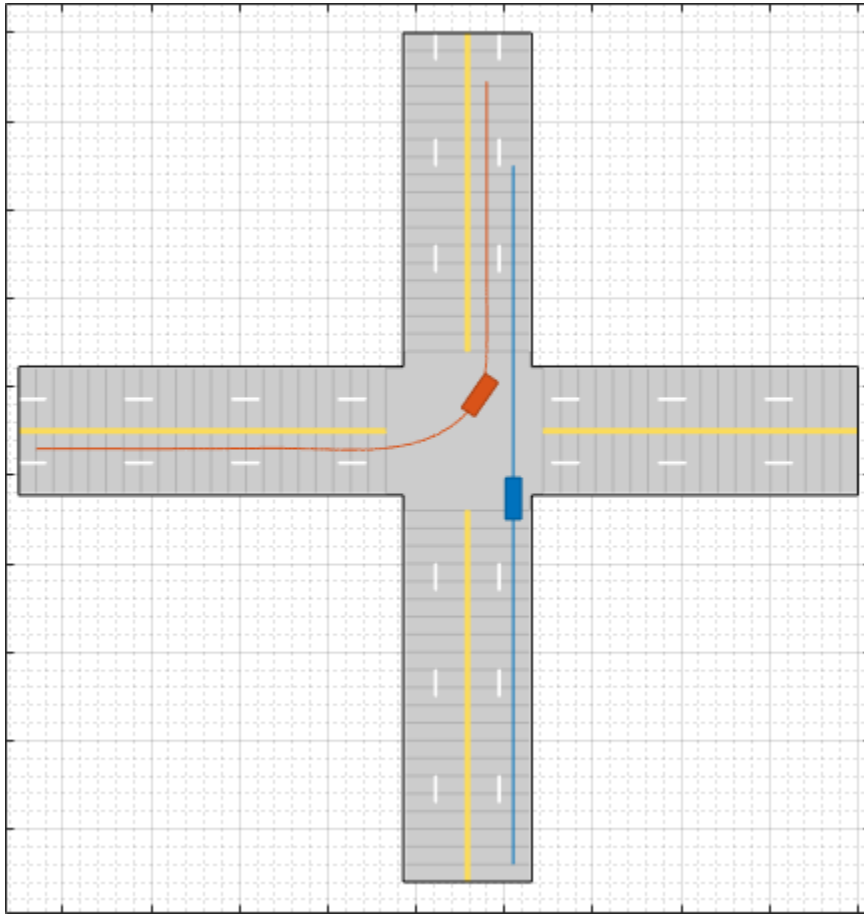
Use the Driving Scenario Designer to interactively build a driving scenario on which to test your algorithms. For more details on building scenarios, see “Build a Driving Scenario and Generate Synthetic Detections” on page 5-2.

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.



This scenario does not include sensors mounted on the ego vehicle. If you have a scenario that includes sensors, you can save them to a scenario file by selecting **Save > Sensors**. Then, in the Driving Scenario Designer app, you can import the sensor file into a programmatic scenario file that you previously generated. The app does not support the direct import of programmatic sensor files.

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 toolbar, 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
```

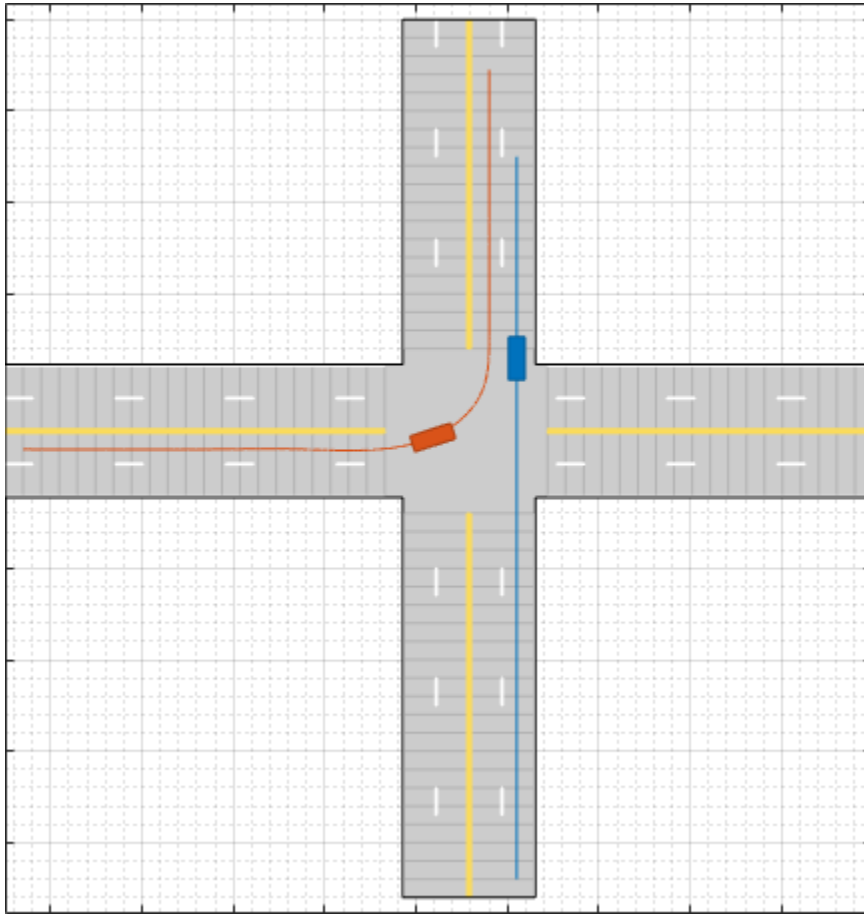
```
scenario =
    drivingScenario with properties:

        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.
- To add sensors previously saved to a scenario file into this scenario, from the app toolbar, select **Open > 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 5-18
- “Euro NCAP Driving Scenarios in Driving Scenario Designer” on page 5-41
- “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.

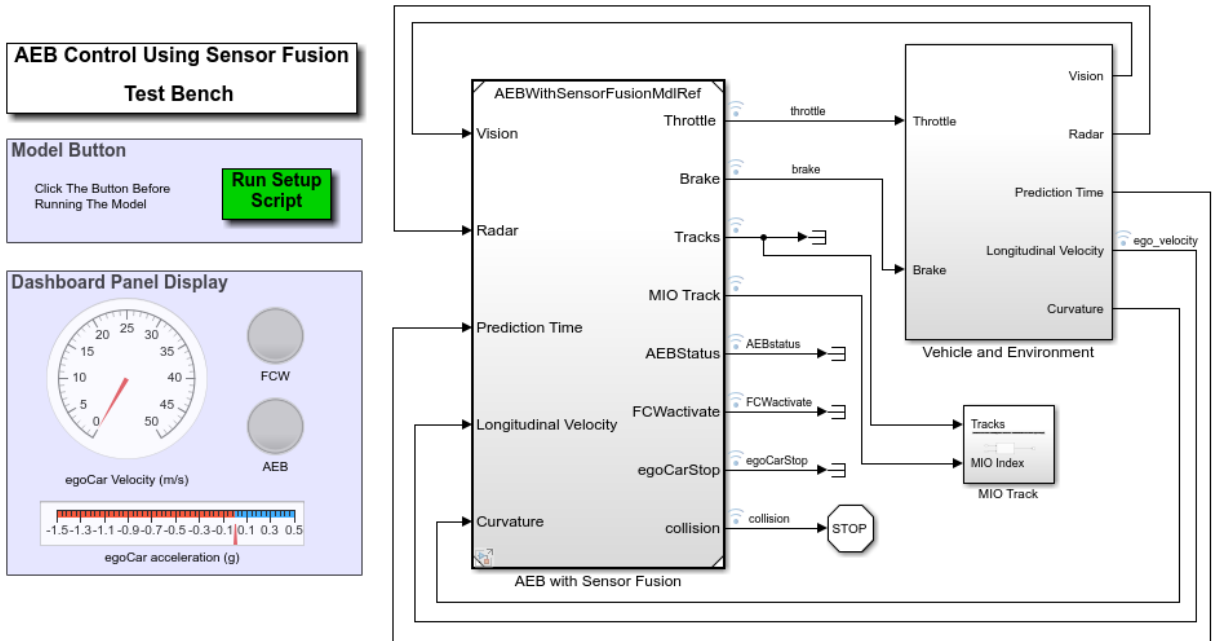
Before beginning this example, add the example file folder to the MATLAB® search path.

```
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.

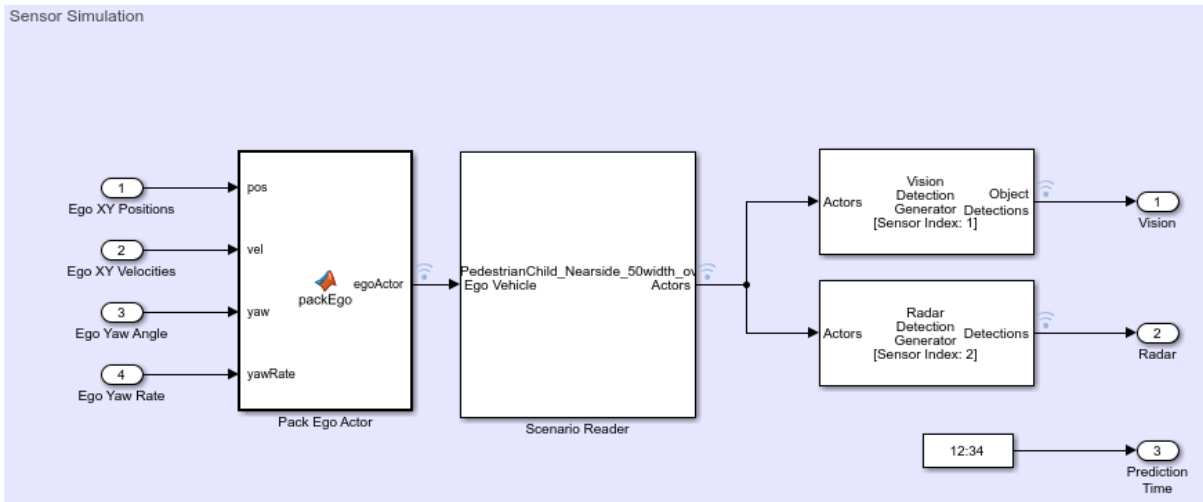
```
open_system('AEBTestBenchExample')
```



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.

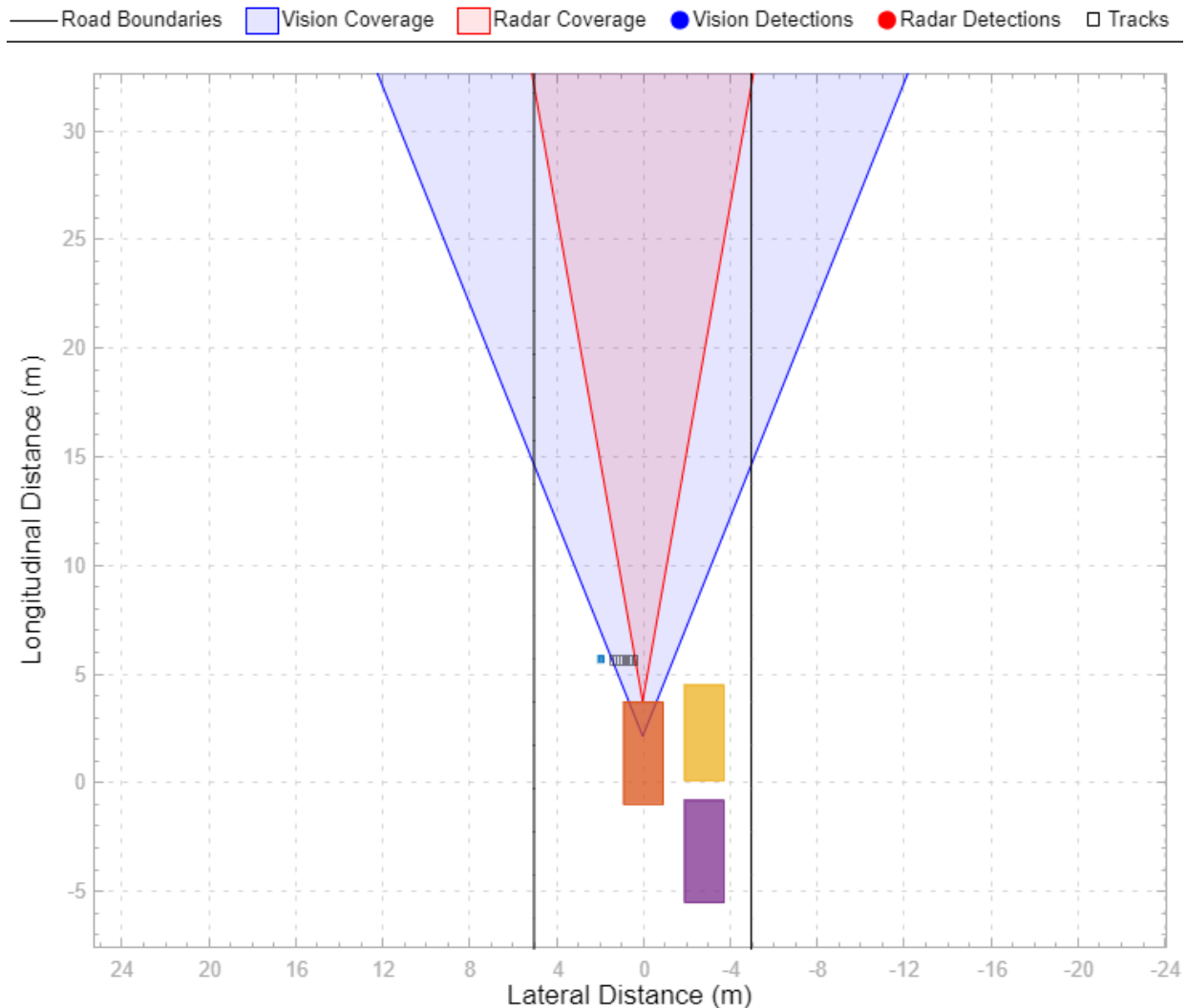
```
open_system('AEBTestBenchExample/Vehicle and Environment/Actors and Sensor Simulation')
```

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 break 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

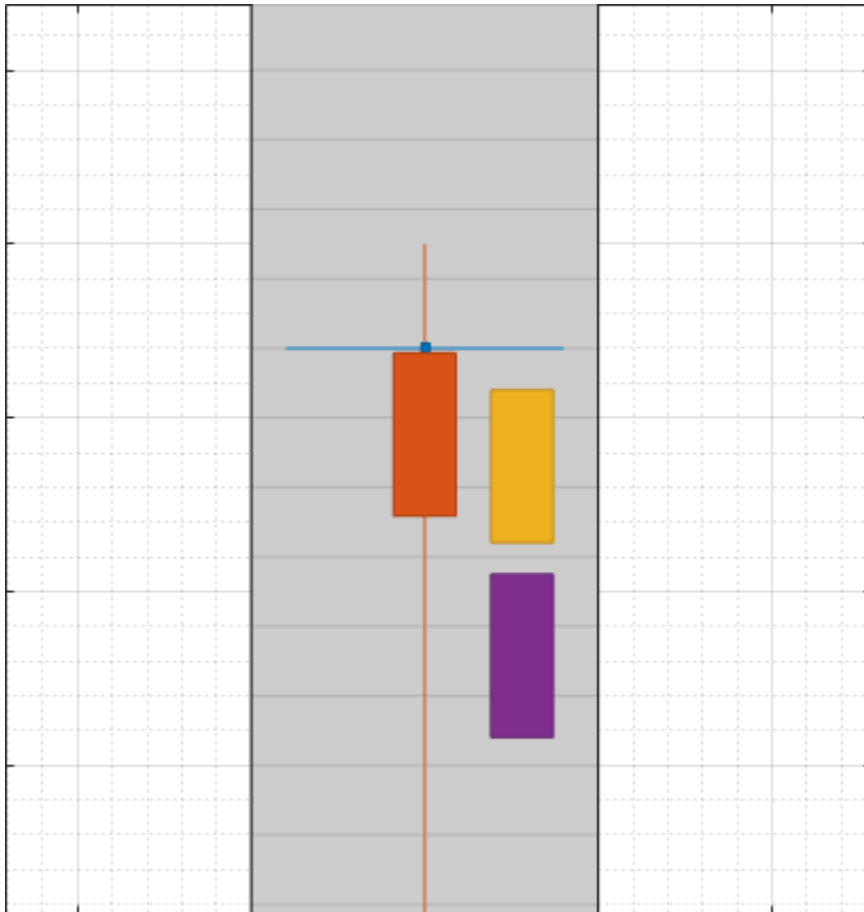
The model uses a driving scenario that is based on one of the prebuilt Euro NCAP test protocol scenarios. You can load these scenarios from the Driving Scenario Designer app.

For more details on these scenarios, see “Euro NCAP Driving Scenarios in Driving Scenario Designer” on page 5-41.

Load the scenario file into the app.

```
drivingScenarioDesigner('AEB_PedestrianChild_Nearside_50width_overrun.mat')
```

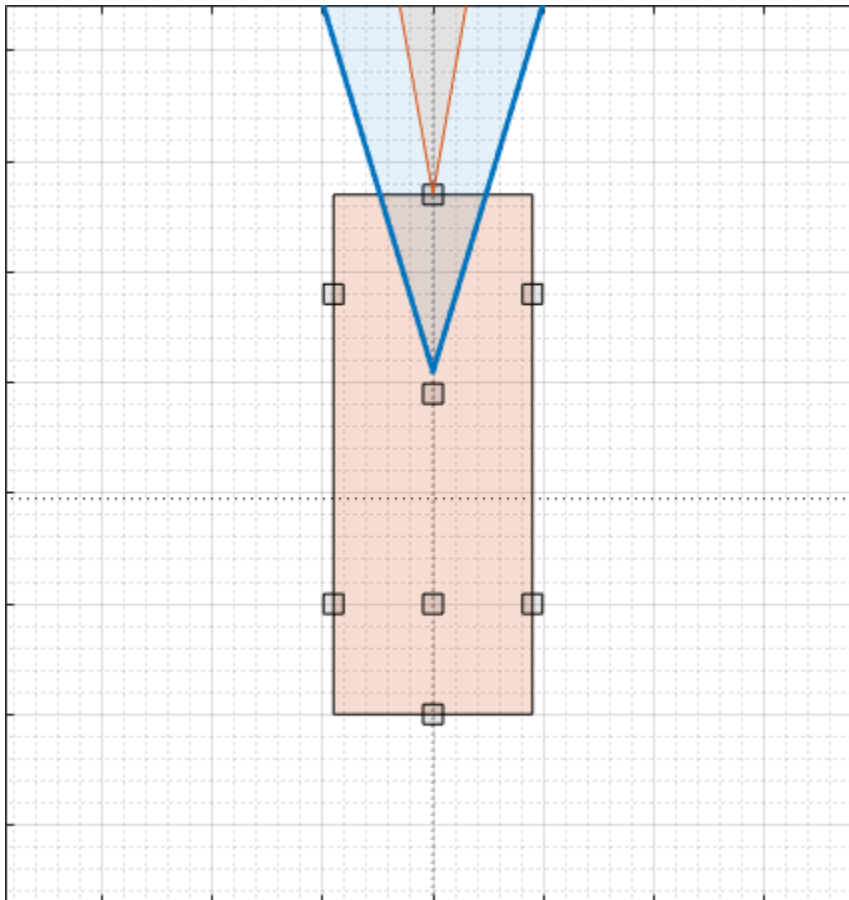
To simulate the scenario in the app, click **Run**. In the app simulation, unlike in the model simulation, the ego vehicle collides with the pedestrian. The app uses a predefined ego vehicle trajectory, whereas the model uses the AEB algorithm to control the trajectory and cause the ego vehicle to brake.



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.



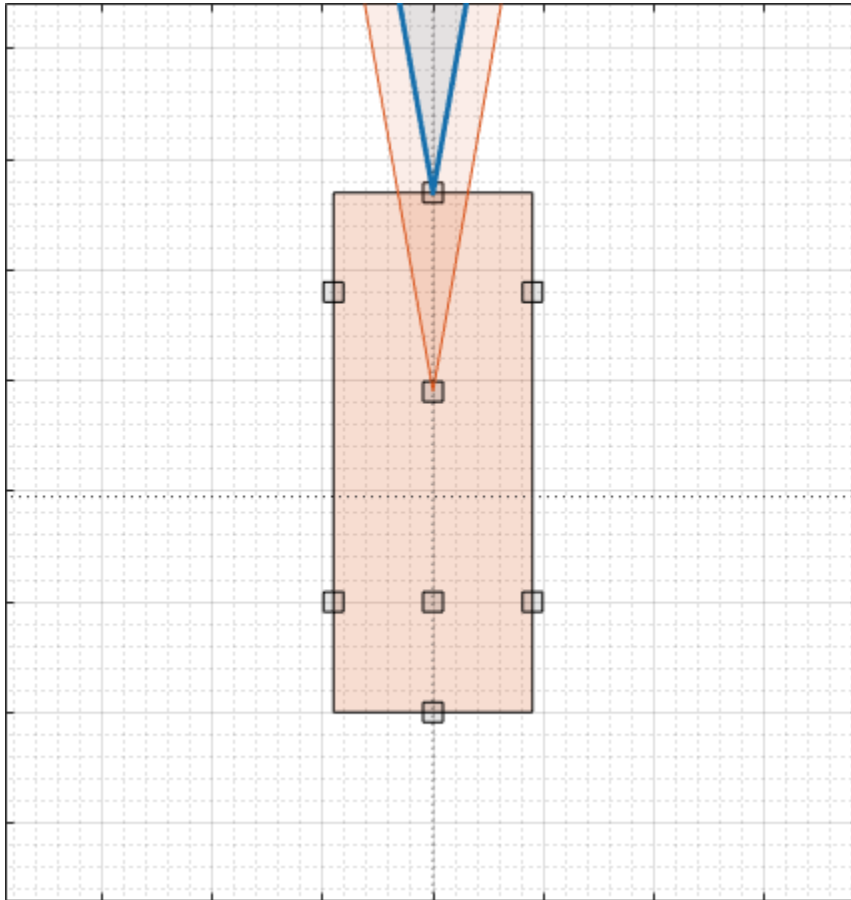
Update Sensors

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.

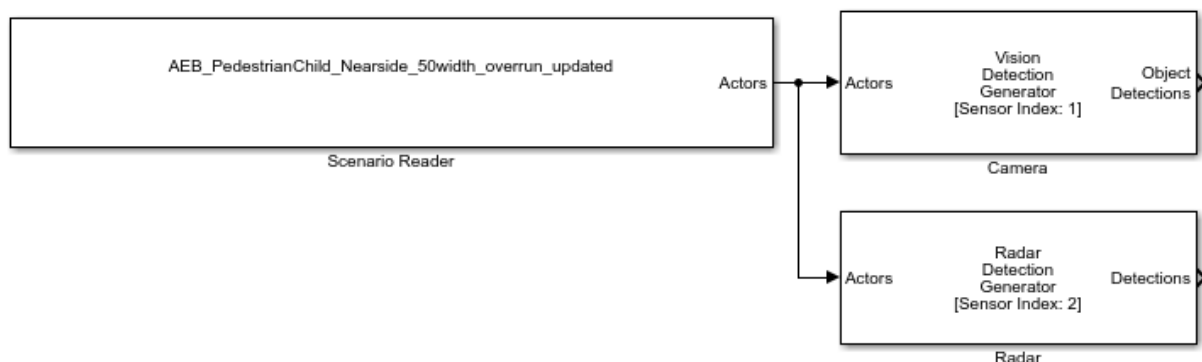
This image shows a sample updated sensor configuration.



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')
```

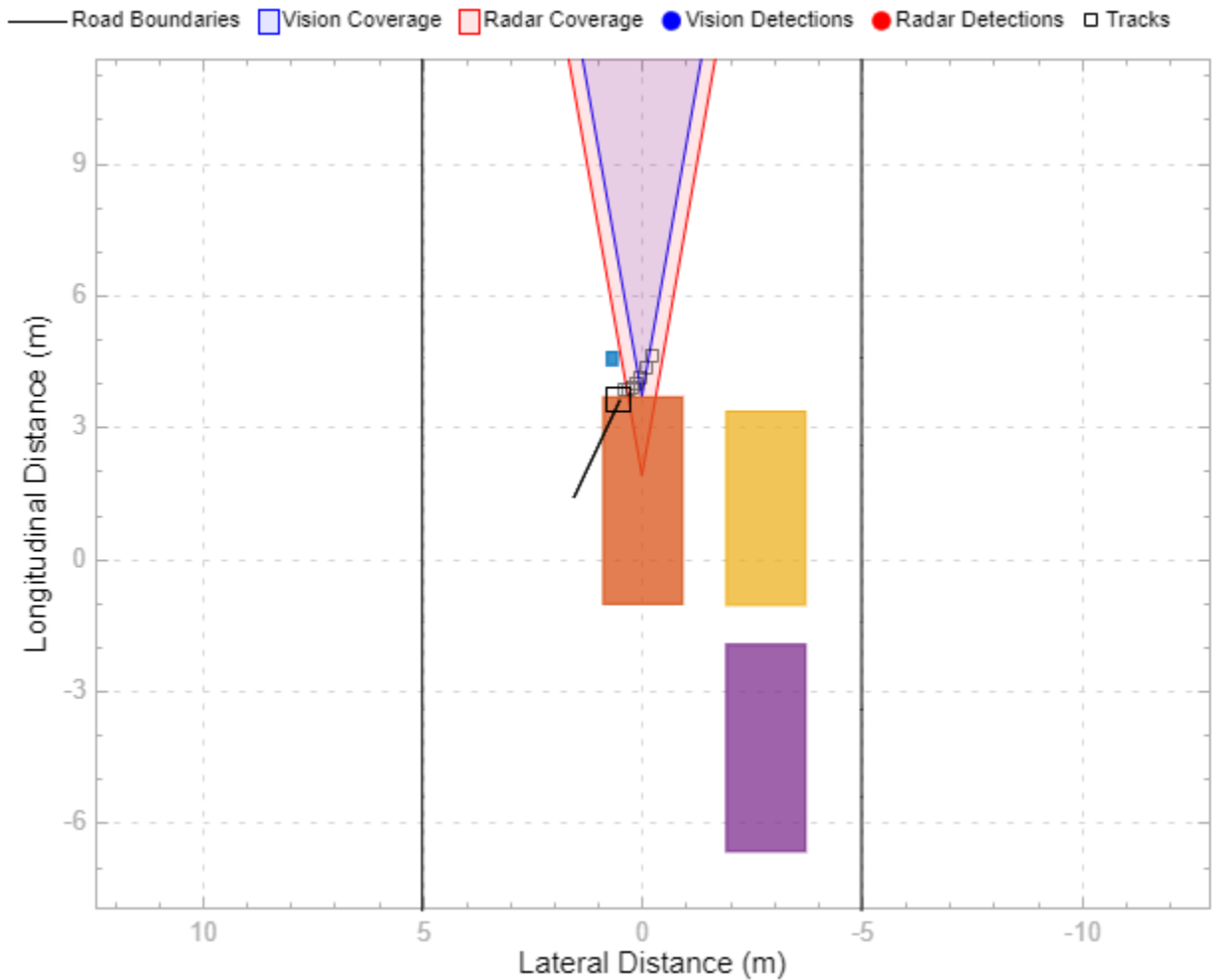
Next, to cope the scenario and sensor blocks with the generated ones, follow these steps:

- 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

Bird's-Eye Scope | **Driving Scenario Designer** | 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 5-41
- “Test Open-Loop ADAS Algorithm Using Driving Scenario” on page 5-93
- “Test Closed-Loop ADAS Algorithm Using Driving Scenario” on page 5-99
- “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')))
```

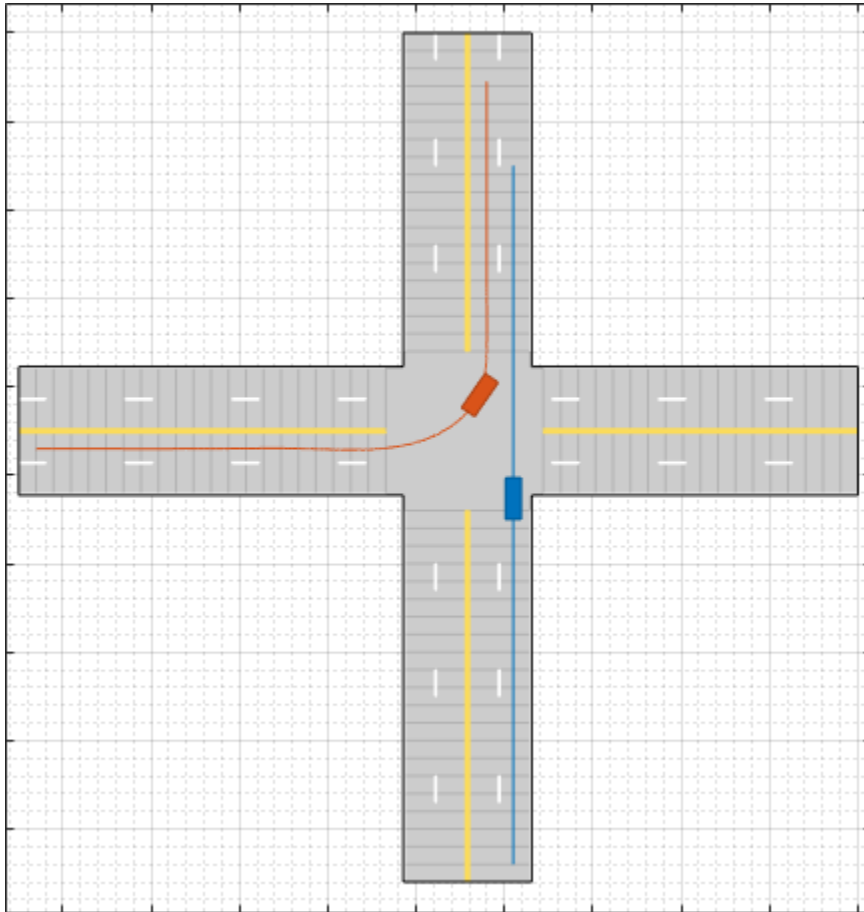
Inspect Driving Scenario

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 5-18.

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.

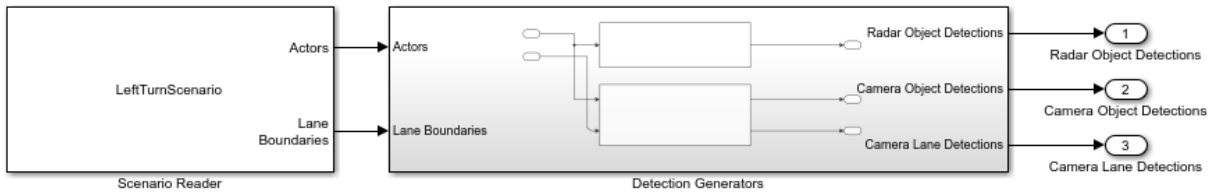


The ego vehicle also includes a front-facing radar sensor and camera sensor for generating detections.

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')
```

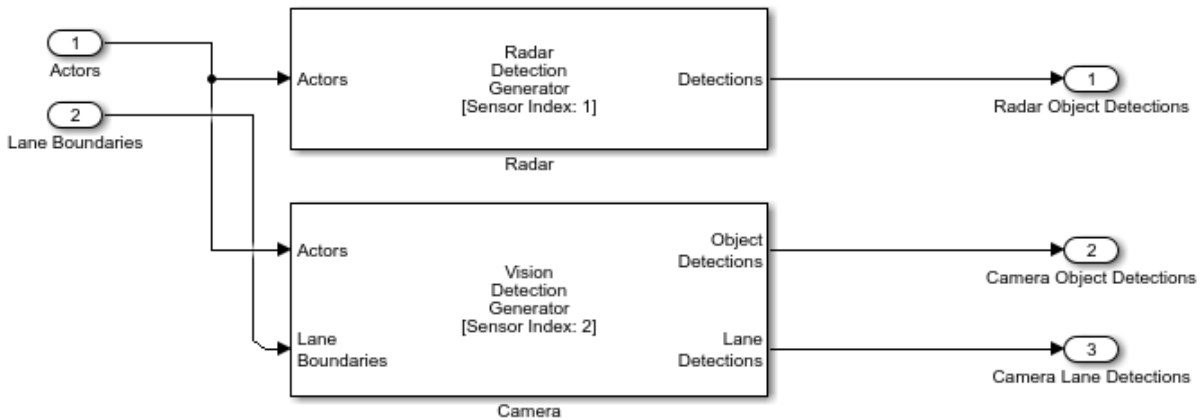


In the Scenario Reader block, the **Driving Scenario Designer file name** parameter specifies the name of the scenario file. You can specify a scenario file that is on the MATLAB search path, such as the scenario file used in this example, or the full path to a scenario file. Alternatively, you can specify 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')
```



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:

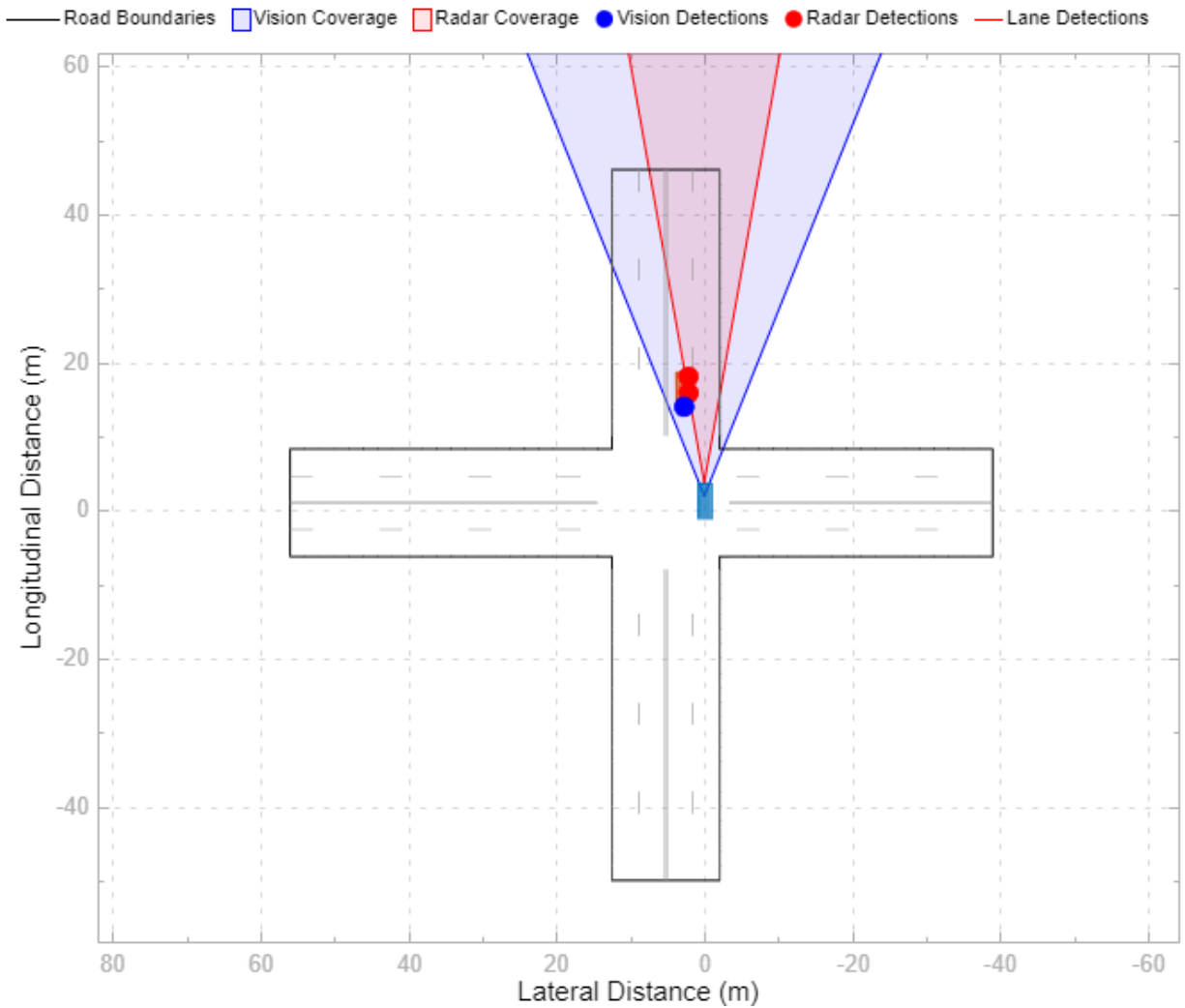
- The X-axis points forward from the ego vehicle.
- The Y-axis points to the left of the ego vehicle.
- The origin is located at the center of the rear axle of the ego vehicle.

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 5-99.

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 vision sensor correctly generates detections for the non-ego actor and the 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

Bird's-Eye Scope | **Driving Scenario Designer** | 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 5-99
- “Create Driving Scenario Variations Programmatically” on page 5-73
- “Generate Sensor Detection Blocks Using Driving Scenario Designer” on page 5-81

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')))
```

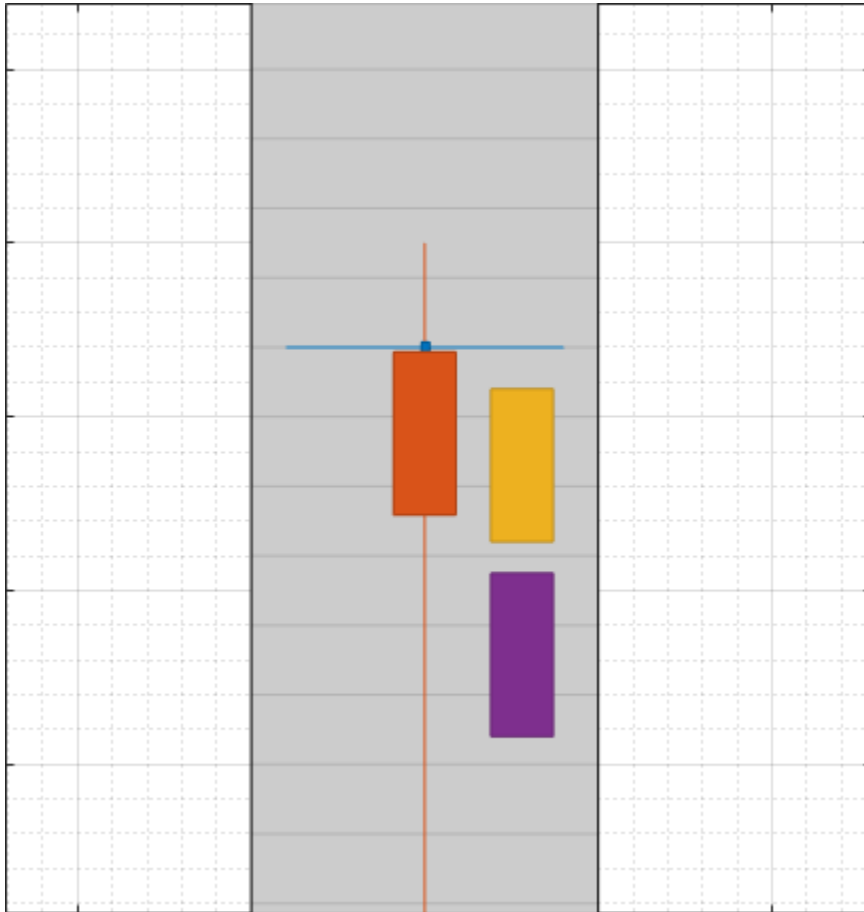
Inspect Driving Scenario

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 5-41.

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.

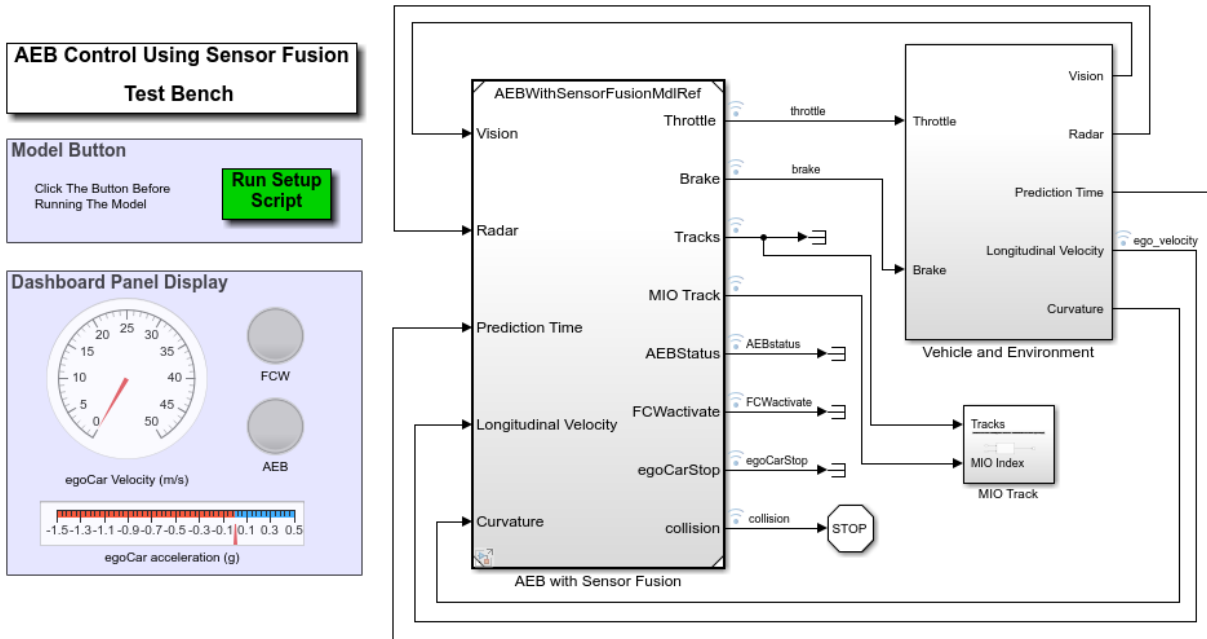


In the model used in this example, you use an AEB sensor fusion algorithm to detect the pedestrian child and test whether the ego vehicle brakes in time to avoid a collision.

Inspect Model

The model implements the AEB algorithm described in the “Autonomous Emergency Braking with Sensor Fusion” example. Open the model.

```
open_system('AEBTestBenchExample')
```

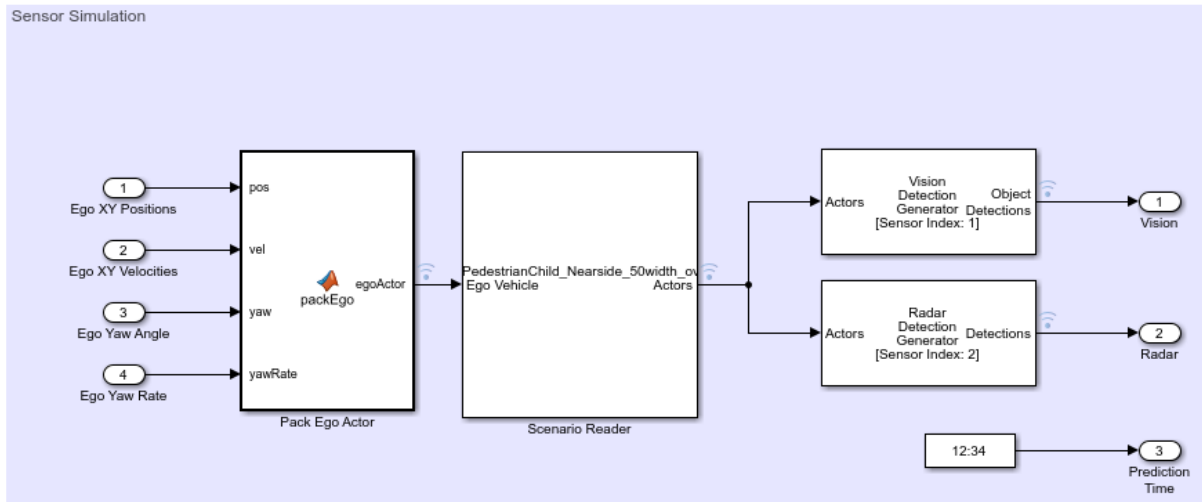


A Scenario Reader block reads the non-ego actors and roads from the specified scenario file and outputs the non-ego actors. The ego vehicle is passed into the block through an input port.

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.

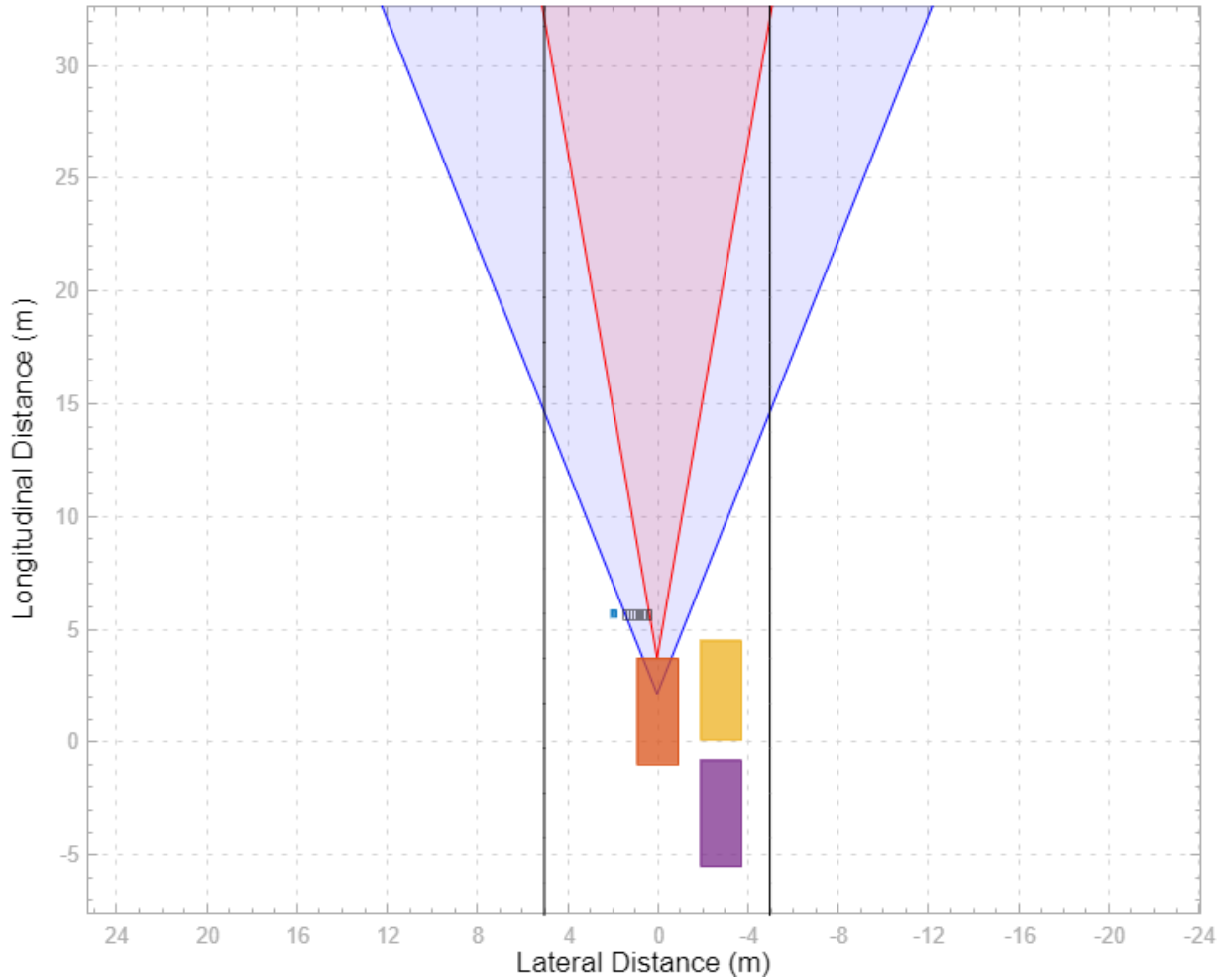
Although this scenario includes a predefined ego vehicle, the Scenario Reader block is configured to ignore this ego vehicle definition. Instead, the ego vehicle is defined in the model and specified as an input to the Scenario Reader block (the **Source of ego vehicle** parameter is set to `Input port`). 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 5-93.

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.

5 Driving Scenario Generation and Sensor Models

— Road Boundaries Vision Coverage Radar Coverage Vision Detections Radar Detections Tracks



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

Bird's-Eye Scope | **Driving Scenario Designer** | Radar Detection Generator | Scenario Reader | Vision Detection Generator

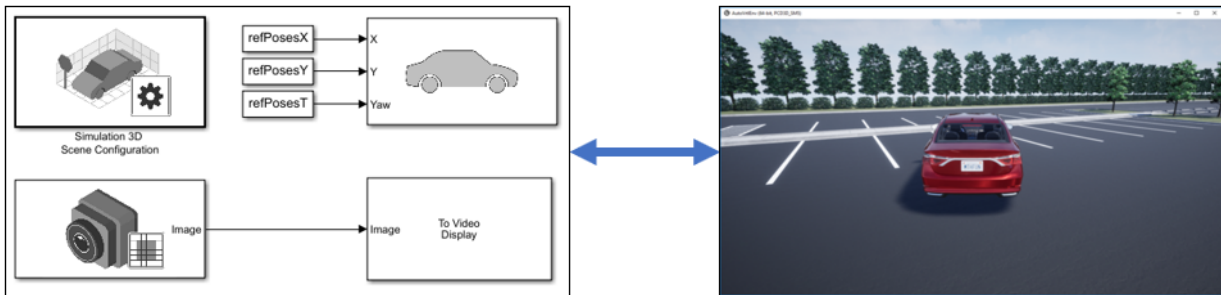
More About

- “Autonomous Emergency Braking with Sensor Fusion”
- “Lateral Control Tutorial”
- “Test Open-Loop ADAS Algorithm Using Driving Scenario” on page 5-93
- “Create Driving Scenario Variations Programmatically” on page 5-73
- “Generate Sensor Detection Blocks Using Driving Scenario Designer” on page 5-81

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.

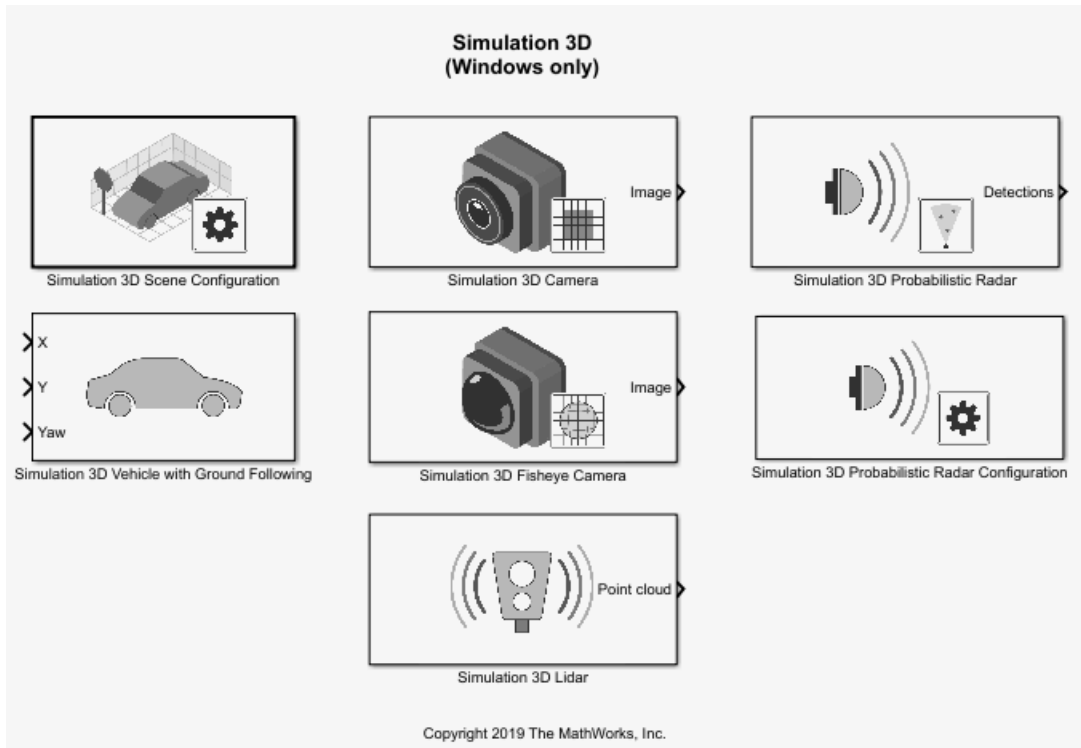
This simulation tool is commonly used to supplement real data when developing, testing, and verifying the performance of automated driving algorithms. In conjunction with a vehicle model, you can use these blocks to perform realistic closed-loop simulations that encompass the entire automated driving stack, from perception to control.

For more details on the simulation environment, see “How 3D Simulation for Automated Driving Works” on page 6-10.

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 multiple intersections
US Highway	Highway with cones, barriers, an animal, traffic lights, and traffic signs

Scene	Description
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

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:

- **Muscle Car**
- **Sedan**
- **Sport Utility Vehicle**
- **Small Pickup Truck**
- **Hatchback**

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.

Block	Description
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 6-19.

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 6-35.

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 that discusses closed-loop simulation in the 3D environment, see “Design of Lane Marker Detector in 3D Simulation Environment”.

See Also

More About

- “3D Simulation Environment Requirements and Limitations” on page 6-8
- “Simulate a Simple Driving Scenario and Sensor in 3D Environment” on page 6-25
- “Coordinate Systems for 3D Simulation in Automated Driving Toolbox” on page 6-13

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 4.19 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
- 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.

Note Mac and Linux® platforms are not supported.

Minimum Hardware Requirements

The 3D simulation environment also requires:

- 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 6-10

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.

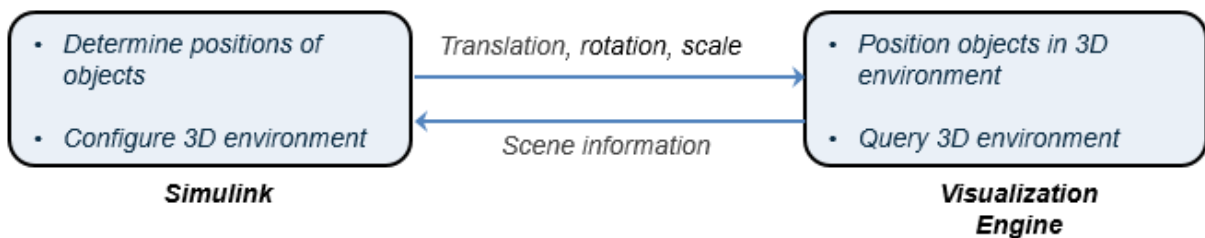
Communication with 3D Simulation Environment

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.



Block Execution Order

During simulation, the 3D simulation blocks follow a specific execution order:

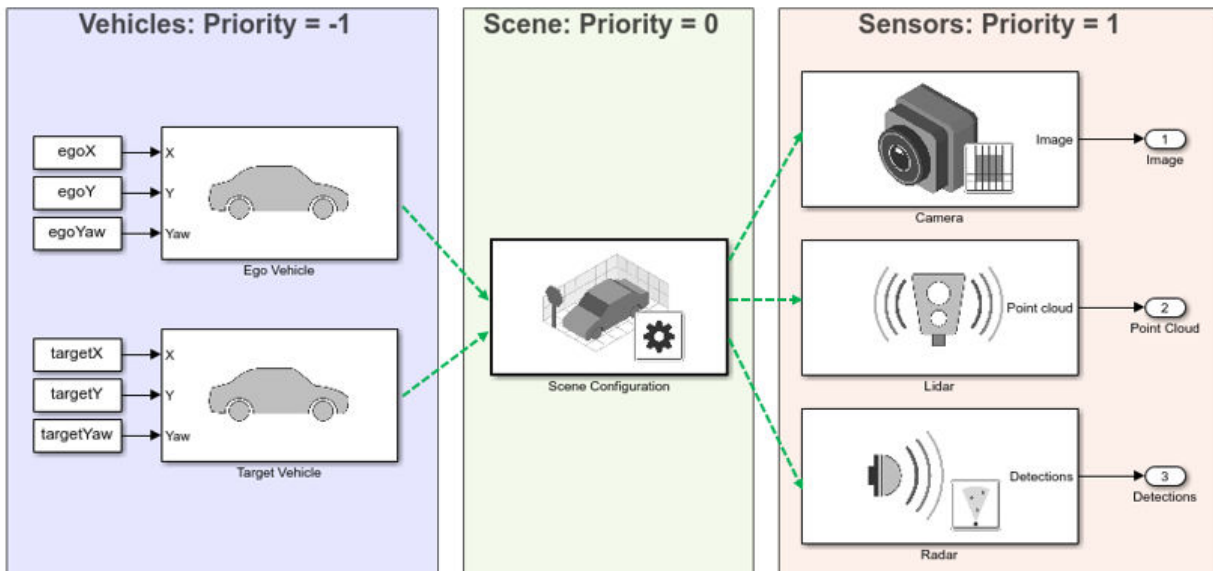
- 1 The Simulation 3D Vehicle with Ground Following blocks initialize the vehicles and send their **X**, **Y**, and **Yaw** signal data to the Simulation 3D Scene Configuration block.

- 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.

Execution Order for 3D Simulation Blocks



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 the 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.

Coordinate Systems

Scenes in the 3D simulation environment use the right-handed Cartesian world coordinate system defined in ISO 8855. In this coordinate system, when looking in the positive *X*-axis direction, the positive *Y*-axis points left. The positive *Z*-axis points up from the ground.

Sensors are mounted on vehicles relative to the vehicle coordinate system. In this system the positive *X*-axis points forward from the vehicle, the positive *Y*-axis points left, and the positive *Z*-axis points up from the ground. The vehicle origin is on the ground, below the longitudinal and lateral center of the vehicle.

These coordinate systems differ from the ones used in the Unreal® Editor. The Unreal Editor uses left-handed Cartesian coordinate systems, where the *Y*-axis points right and the *Z*-axis points down.

For more details on the coordinate systems used for 3D simulation, see “Coordinate Systems for 3D Simulation in Automated Driving Toolbox” on page 6-13.

See Also

More About

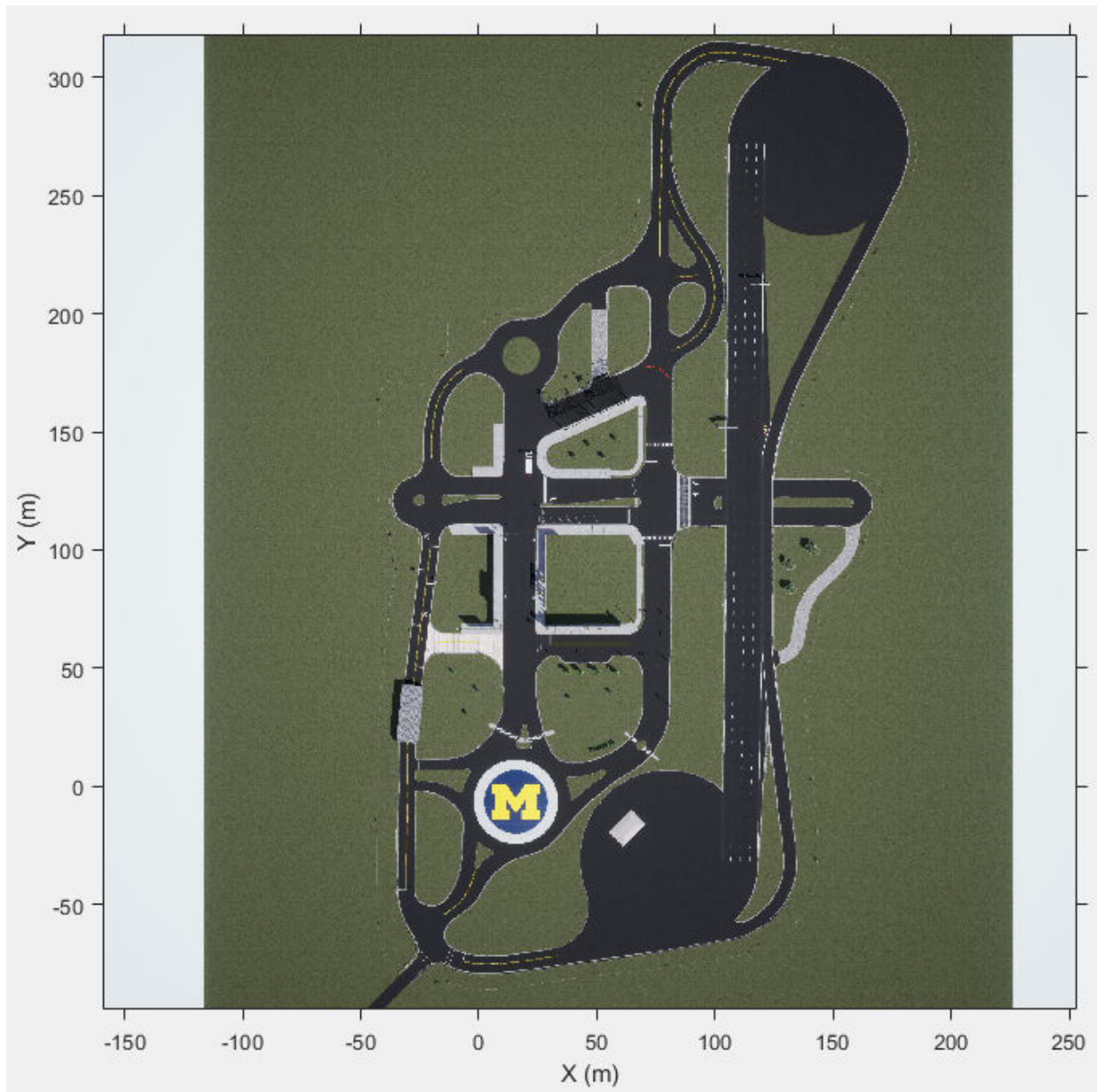
- “3D Simulation for Automated Driving” on page 6-2
- “3D Simulation Environment Requirements and Limitations” on page 6-8
- “Choose a Sensor for 3D Simulation” on page 6-19
- “Simulate a Simple Driving Scenario and Sensor in 3D Environment” on page 6-25

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

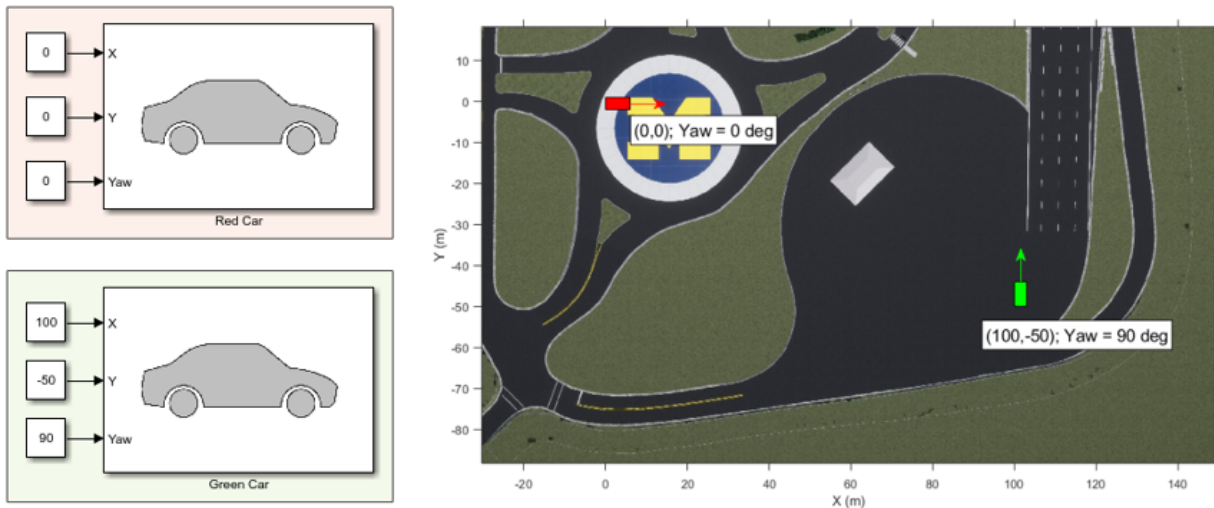
As with other Automated Driving Toolbox functionality, the 3D simulation environment uses the right-handed Cartesian world coordinate system defined in ISO 8855. The following 2D top-view image of the **Virtual Mcity** scene shows the *X*- and *Y*-coordinates of the scene.



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

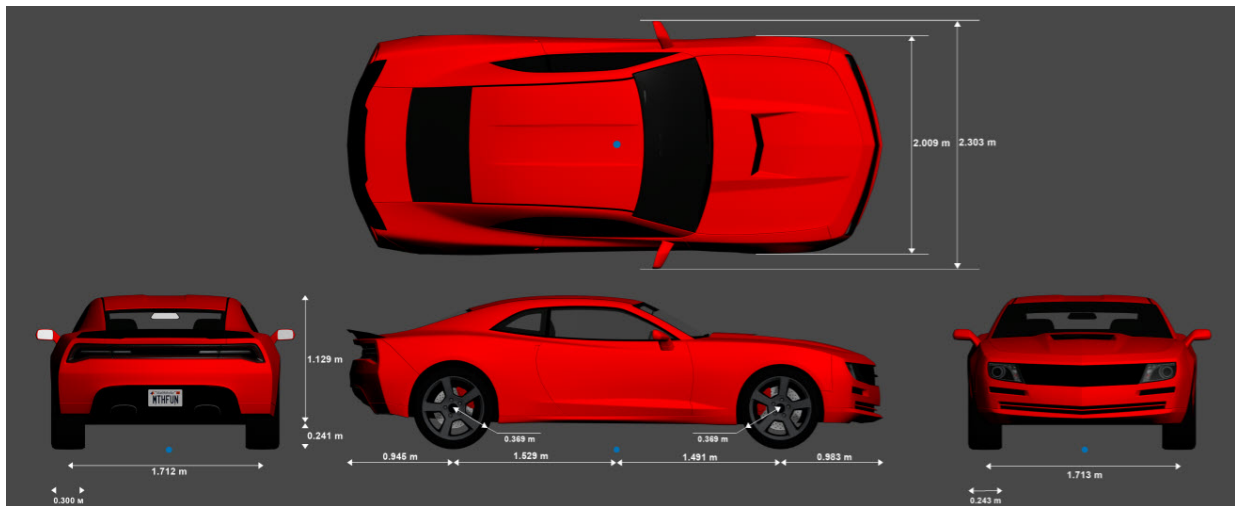
The Unreal Editor uses a left-handed world Cartesian coordinate system in which the positive Y -axis points right and the positive Z -axis points down. If you are converting from the Unreal Editor coordinate system to the coordinate system of the 3D environment, you must flip the sign of the Y -axis, Z -axis, pitch angle, and yaw angle. The X -axis and roll 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.

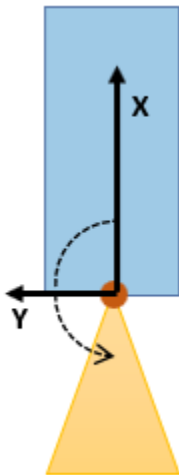
These equations define the vehicle coordinates for a sensor with location (X, Y, Z) and orientation $(Roll, Pitch, Yaw)$:

- $(X, Y, Z) = (X_{mount} + X_{offset}, Y_{mount} + Y_{offset}, Z_{mount} + Z_{offset})$
- $(Roll, Pitch, Yaw) = (Roll_{mount} + Roll_{offset}, Pitch_{mount} + Pitch_{offset}, Yaw_{mount} + Yaw_{offset})$

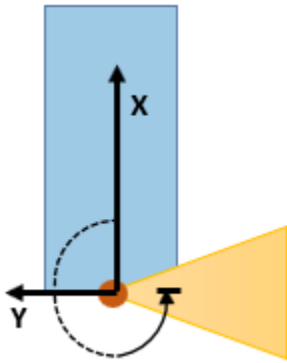
The "mount" variables refer to the predefined mounting locations relative to the vehicle origin. You define these mounting locations in the **Mounting location** parameter of the sensor block.

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. If you are converting sensor positions between coordinate systems, then you need to account for this difference in origin. For an example model that uses such conversions, see “Lane-Following Control with Monocular Camera Perception” (Model Predictive Control Toolbox).

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 and the positive Z-axis points down. 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, Z-axis, pitch angle, and yaw angle. The X-axis and roll angle are the same in both coordinate systems.

See Also

Simulation 3D Vehicle with Ground Following

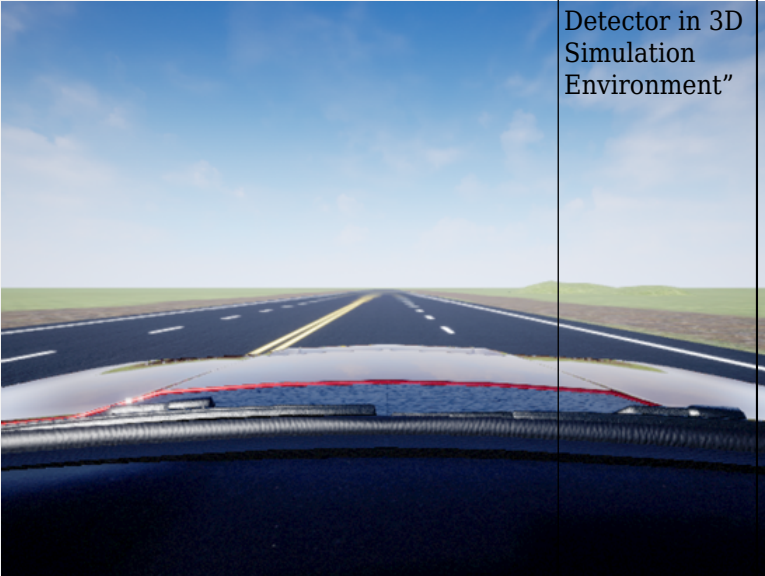
More About

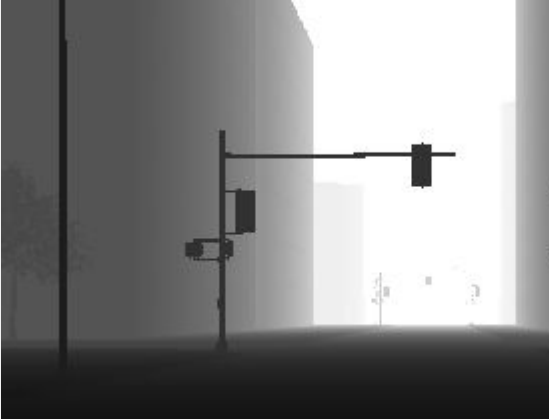
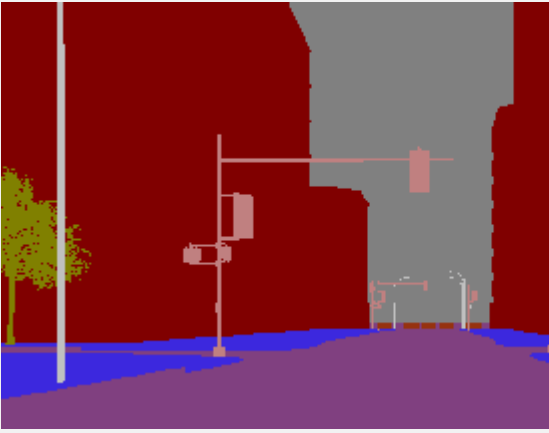
- “How 3D Simulation for Automated Driving Works” on page 6-10
- “Coordinate Systems in Automated Driving Toolbox” on page 1-2
- “Coordinate Systems in Vehicle Dynamics Blockset” (Vehicle Dynamics Blockset)

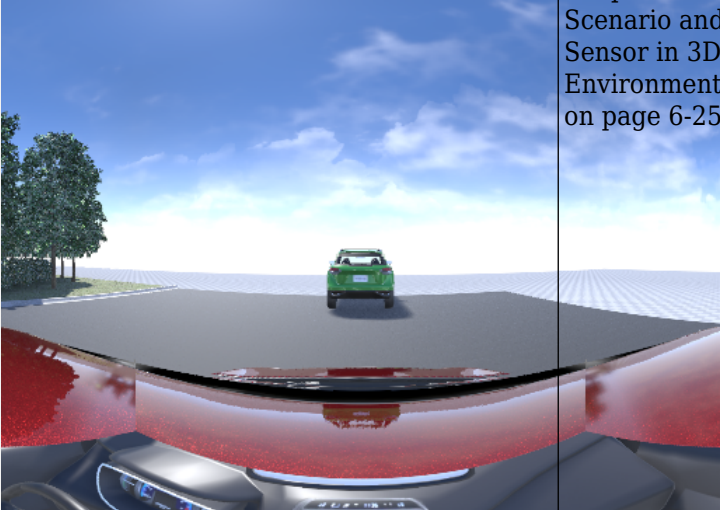
Choose a Sensor for 3D Simulation

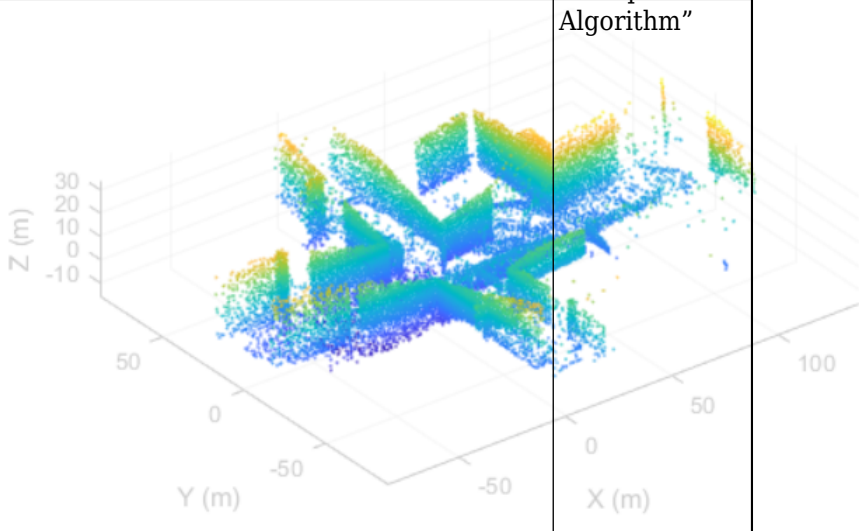
You can use the 3D simulation environment in Automated Driving Toolbox to obtain high-fidelity sensor data. This environment is rendered using the Unreal Engine from Epic Games.

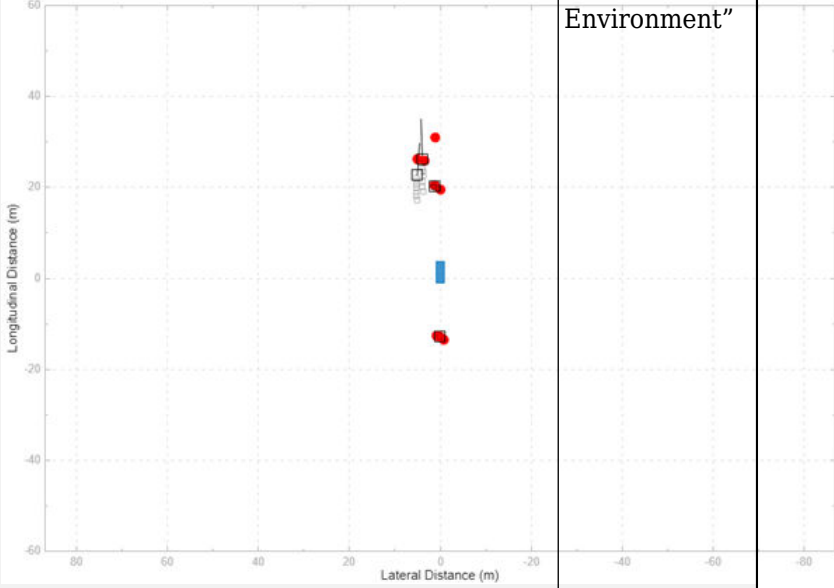
The table summarizes the sensor blocks that you can simulate in this environment.

Sensor Block	Description	Sample Visualization	Example
Simulation 3D Camera	<ul style="list-style-type: none"> • Camera with lens that is based on the ideal pinhole camera. See “What Is Camera Calibration?” (Computer Vision Toolbox) • Includes parameters for 	Camera image using a Video Viewer block: 	“Design of Lane Marker Detector in 3D Simulation Environment”

Sensor Block	Description	Sample Visualization	Example
	<p>image size, focal length, distortion, and skew</p> <ul style="list-style-type: none"> Includes options to output ground truth for depth estimation and semantic segmentation 	<p>Depth map using a To Video Display block:</p> 	<p>“Visualize Depth and Semantic Segmentation Data in 3D Environment” on page 6-35</p>
		<p>Semantic segmentation map using a To Video Display block:</p> 	<p>“Visualize Depth and Semantic Segmentation Data in 3D Environment” on page 6-35</p>

Sensor Block	Description	Sample Visualization	Example
<p>Simulation 3D Fisheye Camera</p>	<ul style="list-style-type: none"> <li data-bbox="446 336 560 991">• Fisheye camera that can be described using the Scaramuzza camera model. See "Fisheye Calibration Basics" <li data-bbox="446 1008 560 1164">(Computer Vision Toolbox) <li data-bbox="446 1182 560 1522">• Includes parameters for distortion center, image size, 	<p>Camera image using a Video Viewer block:</p> 	<p>"Simulate a Simple Driving Scenario and Sensor in 3D Environment" on page 6-25</p>

Sensor Block	Description	Sample Visualization	Example
	and mapping coefficients		
Simulation 3D Lidar	<ul style="list-style-type: none"> Scanning lidar sensor model Includes parameters for detection range, resolution, and fields of view 	Point cloud data using <code>pcplayer</code> within a MATLAB Function block: 	"Simulate Lidar Sensor Perception Algorithm"

Sensor Block	Description	Sample Visualization	Example
Simulation 3D Probabilistic Radar	<ul style="list-style-type: none"> • Probabilistic radar model that returns a list of detections • Includes parameters for radar accuracy, radar bias, detection probability, and detection reporting 	Radar detections using the Bird's-Eye Scope : 	"Simulate Radar Sensors in 3D Environment"

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

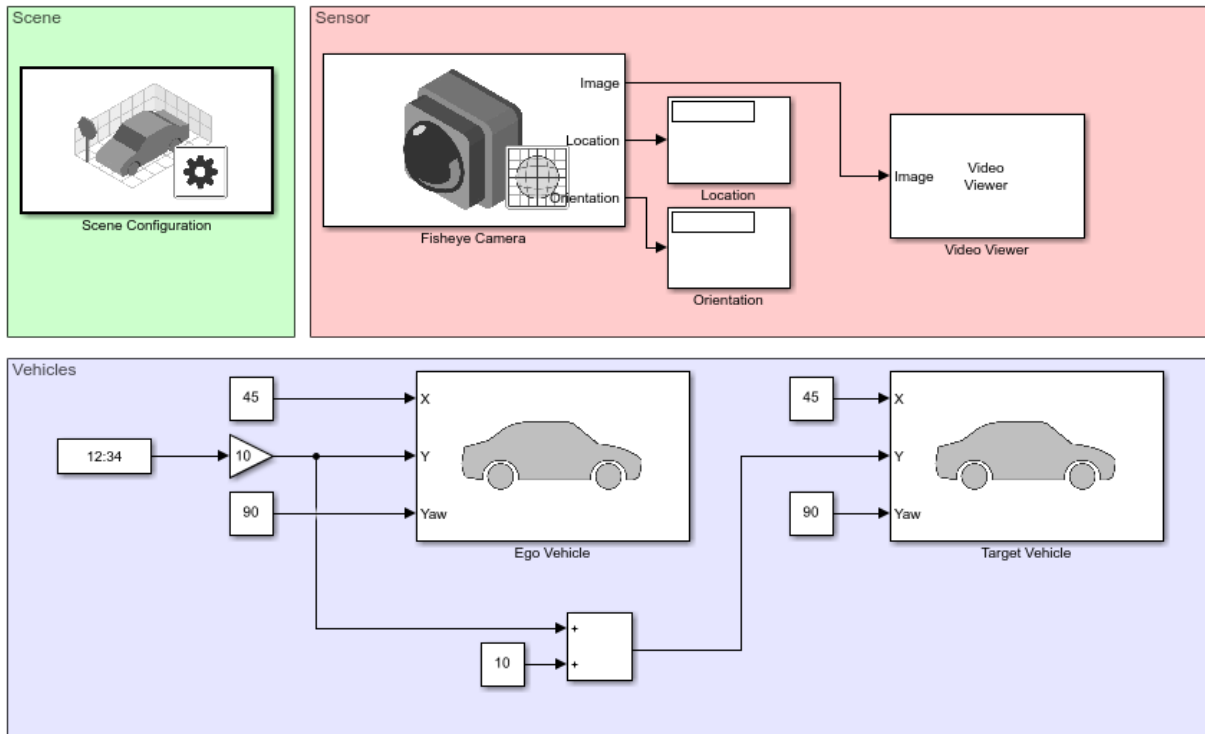
Automated Driving Toolbox™ provides blocks for visualizing sensors in a 3D simulation environment that uses the Unreal Engine® from Epic Games®. This model simulates a simple driving scenario in a prebuilt 3D scene and captures data from the scene using a fisheye camera sensor. Use this model to learn 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.

Simple Driving Scenario and Sensor Model for 3D Simulation



Copyright 2019 The MathWorks, Inc.

Inspect Scene

In the Simulation 3D Scene Configuration block, the **Scene description** 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.

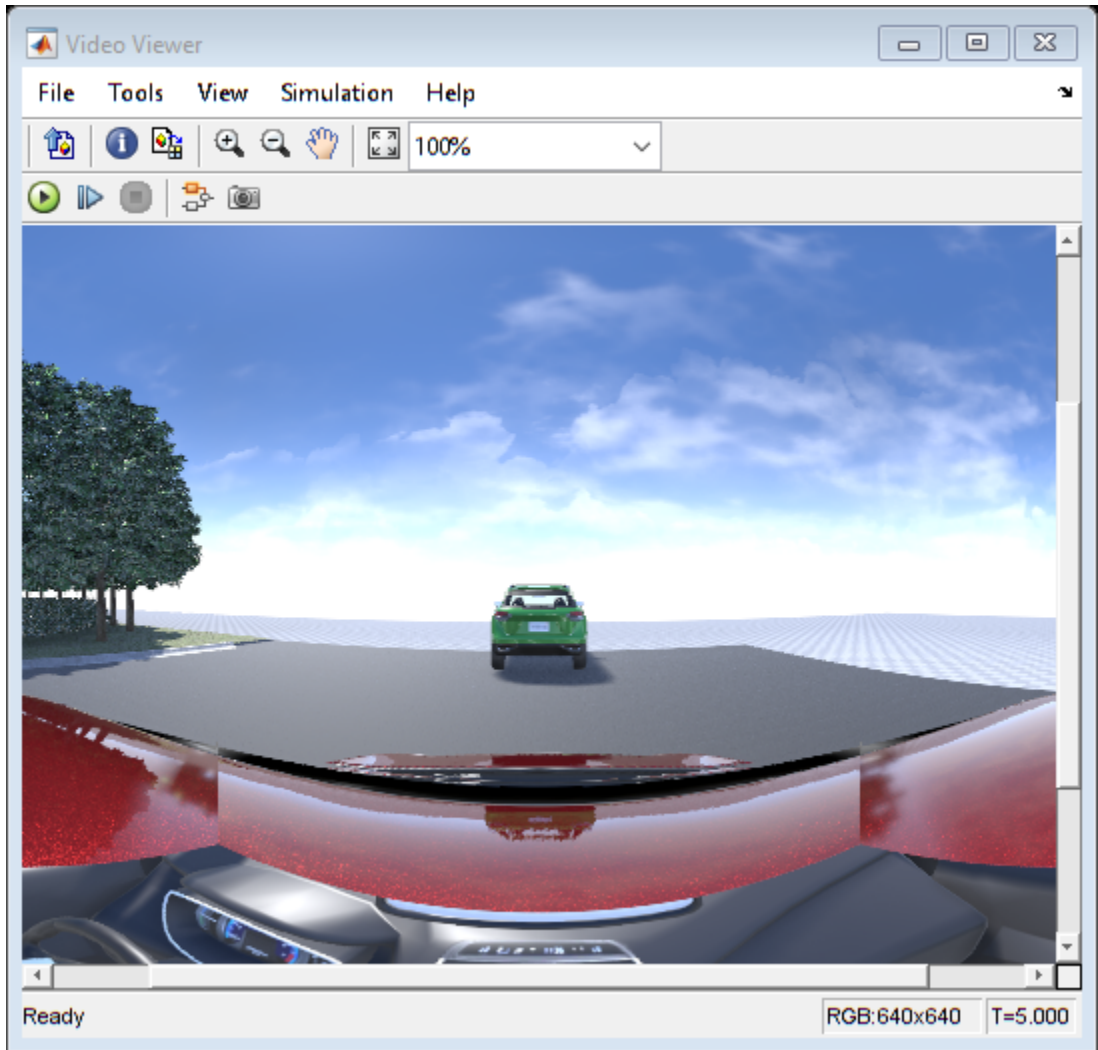
The block outputs images captured from the simulation. During simulation, the Video Viewer block displays these images.

Simulate Model

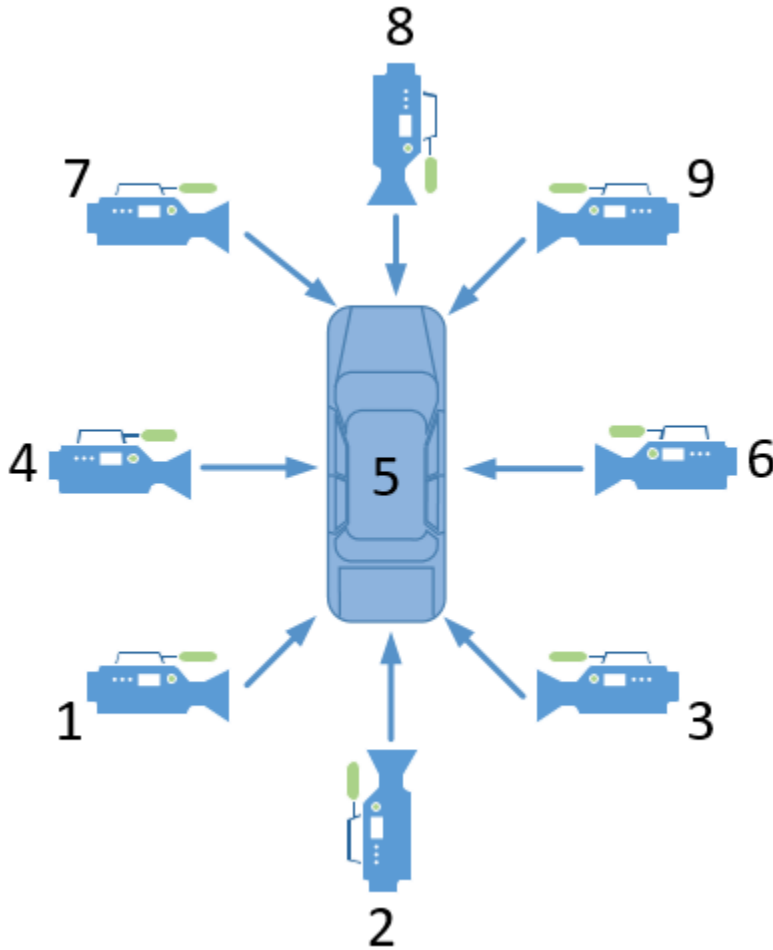
Simulate the model. When the simulation begins, it can take a few seconds for the visualization engine to initialize, especially when you are running it for the first time. The AutoVrtlEnv window shows a view of the scene in the 3D environment.



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 6-19.

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 6-8
- “How 3D Simulation for Automated Driving Works” on page 6-10
- “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.

Before beginning this example, add the example file folder to the MATLAB® search path.

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

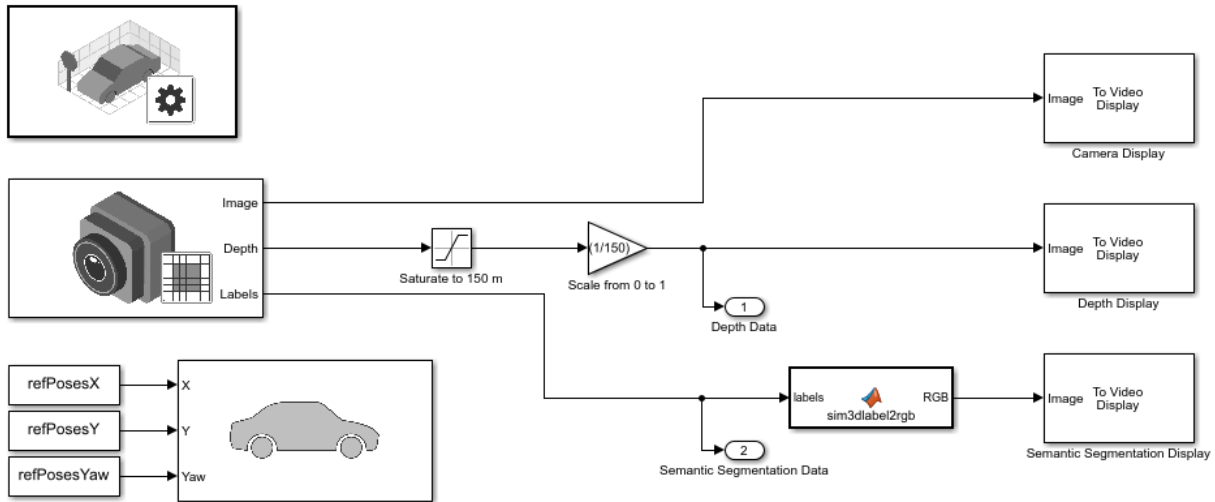
Model Setup

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 then 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 and 2 corresponds to fences. 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 that the 3D simulation blocks use. 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: Buildings
64 64 128   % Label 2: Fences
72 0 90     % Label 3: Other
64 64 0     % Label 4: Pedestrians
192 192 192 % Label 5: Poles
128 64 128  % Label 6: RoadLines
128 64 128  % Label 7: Roads
60 40 222   % Label 8: Sidewalks
128 128 0   % Label 9: Vegetation
64 0 128    % Label 10: Vehicles
128 0 0     % Label 11: Walls
```

Model Simulation

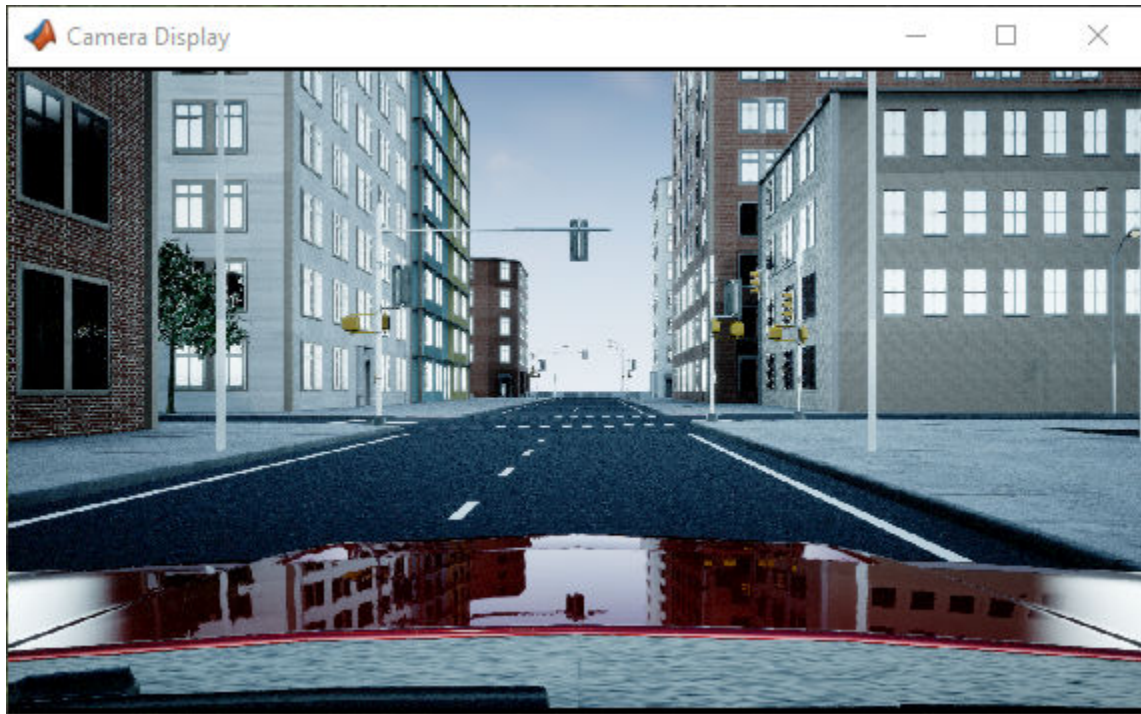
Run the model.

```
sim('DepthSemanticSegmentation.slx');
```

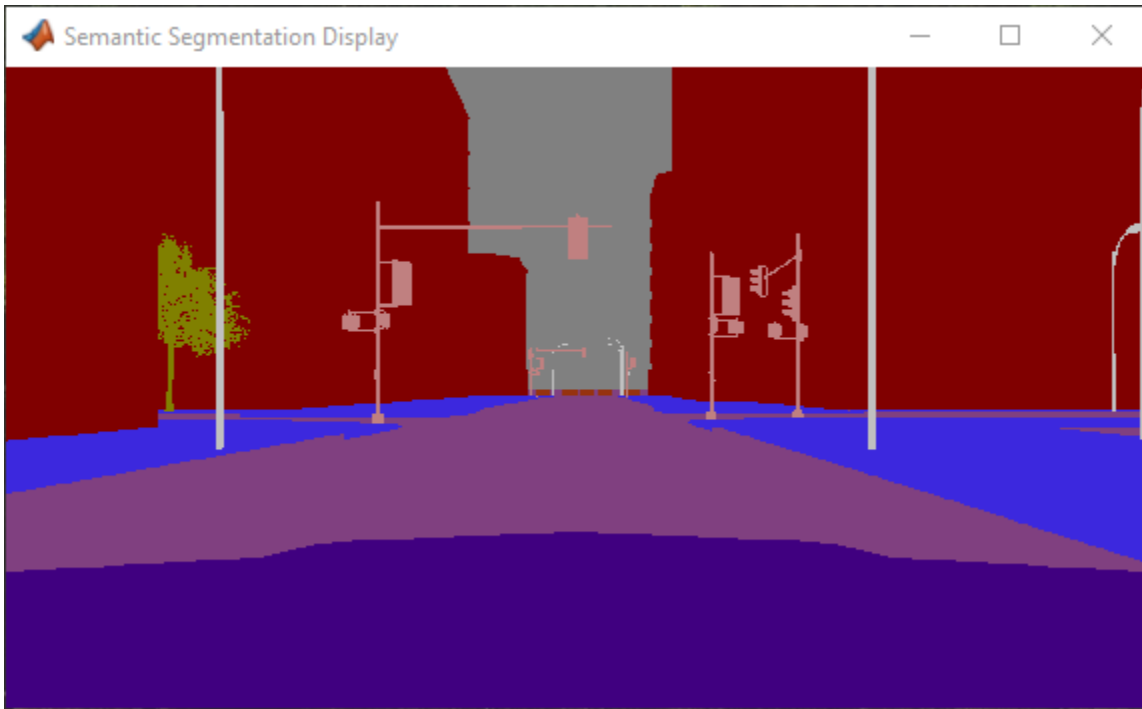
When the simulation begins, it can take a few seconds for the visualization engine to initialize, especially when you are running it for the first time. The `AutoVrtlEnv` window displays the scene from behind the ego vehicle.



The Camera Display, Depth Display, and Semantic Segmentation Display blocks display the outputs from the camera sensor.







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`.

Remove the example file folder from the MATLAB search path.

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

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 6-25
- “Select Waypoints for 3D Simulation”
- “Semantic Segmentation Using Deep Learning” (Computer Vision Toolbox)