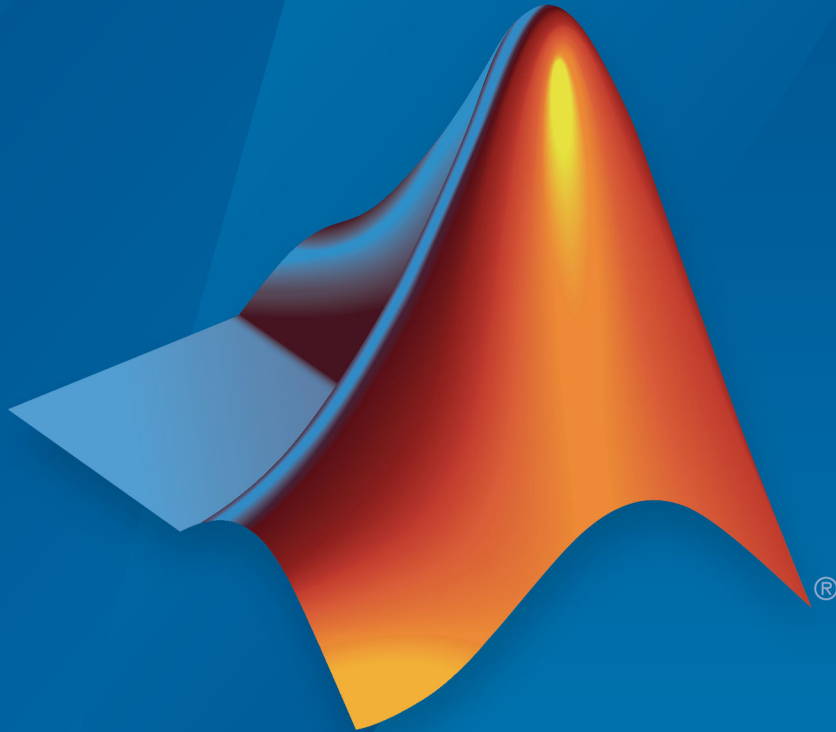


# Automated Driving System Toolbox™

## User's Guide



# MATLAB®

R2018b

 MathWorks®

# How to Contact MathWorks



Latest news: [www.mathworks.com](http://www.mathworks.com)  
Sales and services: [www.mathworks.com/sales\\_and\\_services](http://www.mathworks.com/sales_and_services)  
User community: [www.mathworks.com/matlabcentral](http://www.mathworks.com/matlabcentral)  
Technical support: [www.mathworks.com/support/contact\\_us](http://www.mathworks.com/support/contact_us)



Phone: 508-647-7000



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

## *Automated Driving System Toolbox™ User's Guide*

© COPYRIGHT 2017–2018 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](http://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](http://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)

## Sensor Configuration and Coordinate System Transformations

### 1

<b>Coordinate Systems in Automated Driving System Toolbox . . .</b>	<b>1-2</b>
World Coordinate System . . . . .	1-2
Vehicle Coordinate System . . . . .	1-2
Sensor Coordinate System . . . . .	1-4
Spatial Coordinate System . . . . .	1-7
<b>Calibrate a Monocular Camera . . . . .</b>	<b>1-8</b>
Estimate Intrinsic Parameters . . . . .	1-8
Place Checkerboard for Extrinsic Parameter Estimation . . . . .	1-8
Estimate Extrinsic Parameters . . . . .	1-11
Configure Camera Using Intrinsic and Extrinsic Parameters . . . . .	1-12

## Ground Truth Labeling and Verification

### 2

<b>Get Started with the Ground Truth Labeler . . . . .</b>	<b>2-2</b>
Load Unlabeled Data . . . . .	2-2
Set Time Interval to Label . . . . .	2-3
Create Label Definitions . . . . .	2-4
Label Ground Truth . . . . .	2-14
Export Labeled Ground Truth . . . . .	2-17
Save App Session . . . . .	2-21
<b>Use Custom Data Source Reader for Ground Truth Labeling . . . . .</b>	<b>2-23</b>
Import Data Source Using Custom Reader Dialog Box . . . . .	2-23
Import Data Source Using Custom Reader Function . . . . .	2-24

### 3

<b>Visualize Sensor Data and Tracks in Bird's-Eye Scope</b> . . . . .	<b>3-2</b>
Open Model and Scope . . . . .	3-2
Find Signals . . . . .	3-3
Run Simulation . . . . .	3-6
Organize Signal Groups (Optional) . . . . .	3-8
Update Model and Rerun Simulation . . . . .	3-8
Save and Close Model . . . . .	3-8
<b>Linear Kalman Filters</b> . . . . .	<b>3-11</b>
State Equations . . . . .	3-11
Measurement Models . . . . .	3-13
Linear Kalman Filter Equations . . . . .	3-13
Filter Loop . . . . .	3-14
Constant Velocity Model . . . . .	3-16
Constant Acceleration Model . . . . .	3-17
<b>Extended Kalman Filters</b> . . . . .	<b>3-19</b>
State Update Model . . . . .	3-19
Measurement Model . . . . .	3-20
Extended Kalman Filter Loop . . . . .	3-20
Predefined Extended Kalman Filter Functions . . . . .	3-21

## Driving Scenario Generation and Sensor Models

### 4

<b>Build a Driving Scenario and Generate Synthetic</b>	
<b>Detections</b> . . . . .	<b>4-2</b>
Create a New Driving Scenario . . . . .	4-2
Add a Road . . . . .	4-2
Add Lanes . . . . .	4-6
Add Vehicles . . . . .	4-8
Add a Pedestrian . . . . .	4-10
Add Sensors . . . . .	4-12
Generate Sensor Detections . . . . .	4-15
Save Session . . . . .	4-17

<b>Generate Synthetic Detections from a Prebuilt Driving Scenario</b> .....	<b>4-18</b>
Choose a Prebuilt Scenario .....	<b>4-18</b>
Modify Scenario .....	<b>4-37</b>
Generate Synthetic Detections .....	<b>4-38</b>
Save Scenario .....	<b>4-39</b>
<b>Generate Synthetic Detections from a Euro NCAP Scenario</b> .....	<b>4-40</b>
Choose a Euro NCAP Scenario .....	<b>4-40</b>
Modify Scenario .....	<b>4-56</b>
Generate Synthetic Detections .....	<b>4-57</b>
Save Scenario .....	<b>4-58</b>
<b>Add OpenDRIVE Roads to Driving Scenario</b> .....	<b>4-60</b>
Import OpenDRIVE File .....	<b>4-60</b>
Inspect Roads .....	<b>4-61</b>
Add Actors and Sensors to Scenario .....	<b>4-67</b>
Generate Synthetic Detections .....	<b>4-68</b>
Save Session .....	<b>4-69</b>



# Sensor Configuration and Coordinate System Transformations

---

- “Coordinate Systems in Automated Driving System Toolbox” on page 1-2
- “Calibrate a Monocular Camera” on page 1-8

# Coordinate Systems in Automated Driving System Toolbox

Automated Driving System 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.

## 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 generation.

## Vehicle Coordinate System

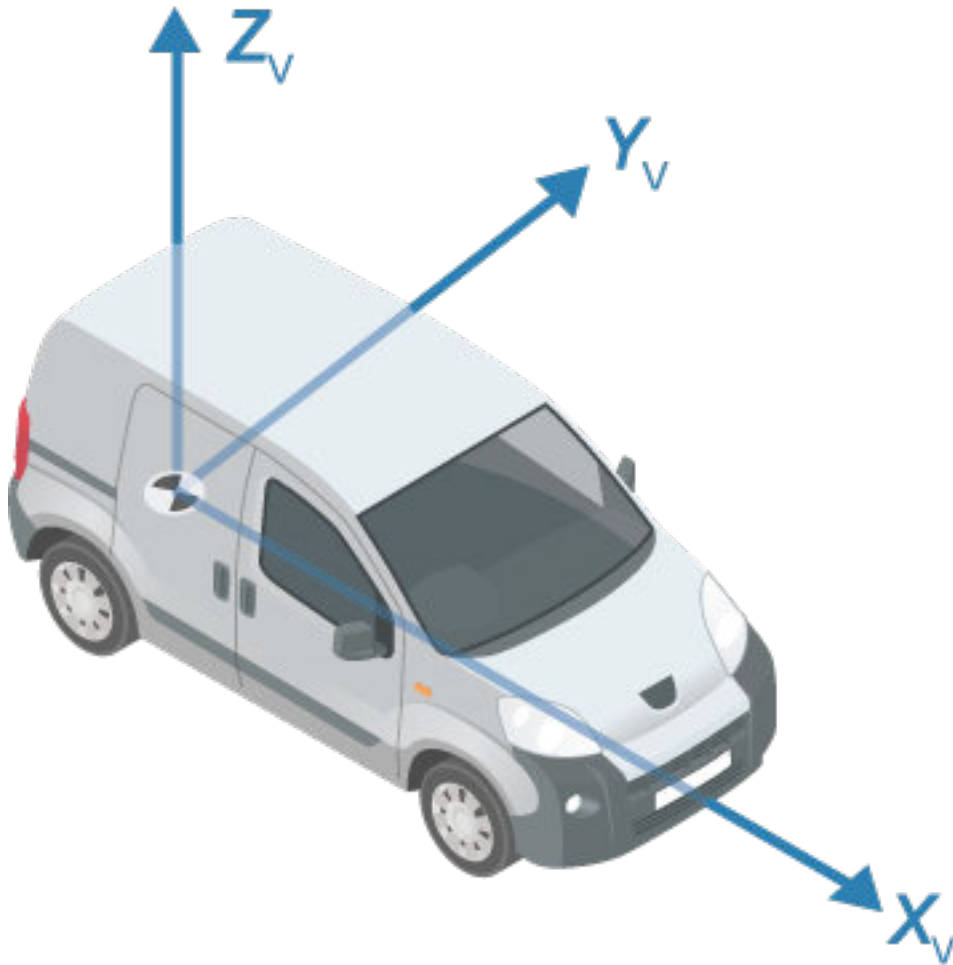
The vehicle coordinate system ( $X_V$ ,  $Y_V$ ,  $Z_V$ ) used by Automated Driving System Toolbox is anchored to the ego vehicle. The term ego vehicle refers to the vehicle that contains the sensors that perceive the environment around the vehicle.

- 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.

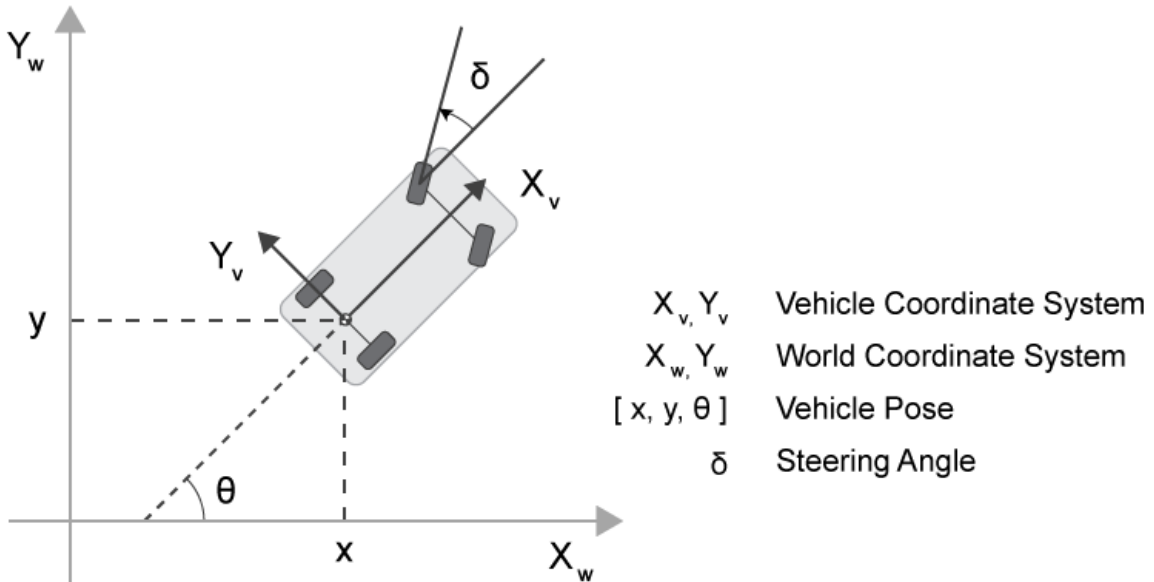
Typically, the origin of the vehicle coordinate system is placed directly on the ground below the midpoint of the rear axle. Locations in this coordinate system are expressed in world units, such as meters.

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



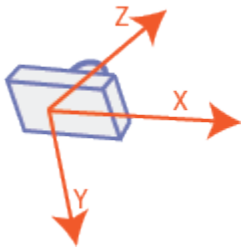


For global path planning, localization, mapping, and driving scenario generation, the state of the vehicle can be described using the pose of the vehicle. The steering angle of the vehicle is positive in the counterclockwise direction.

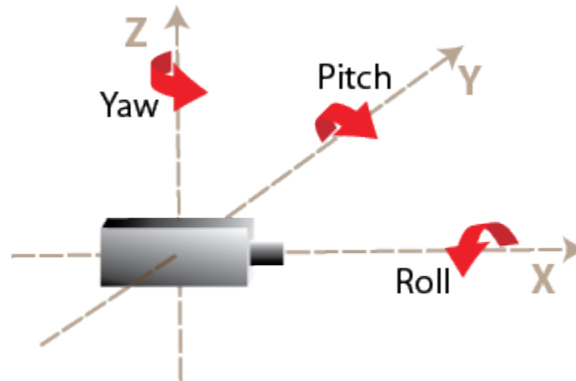


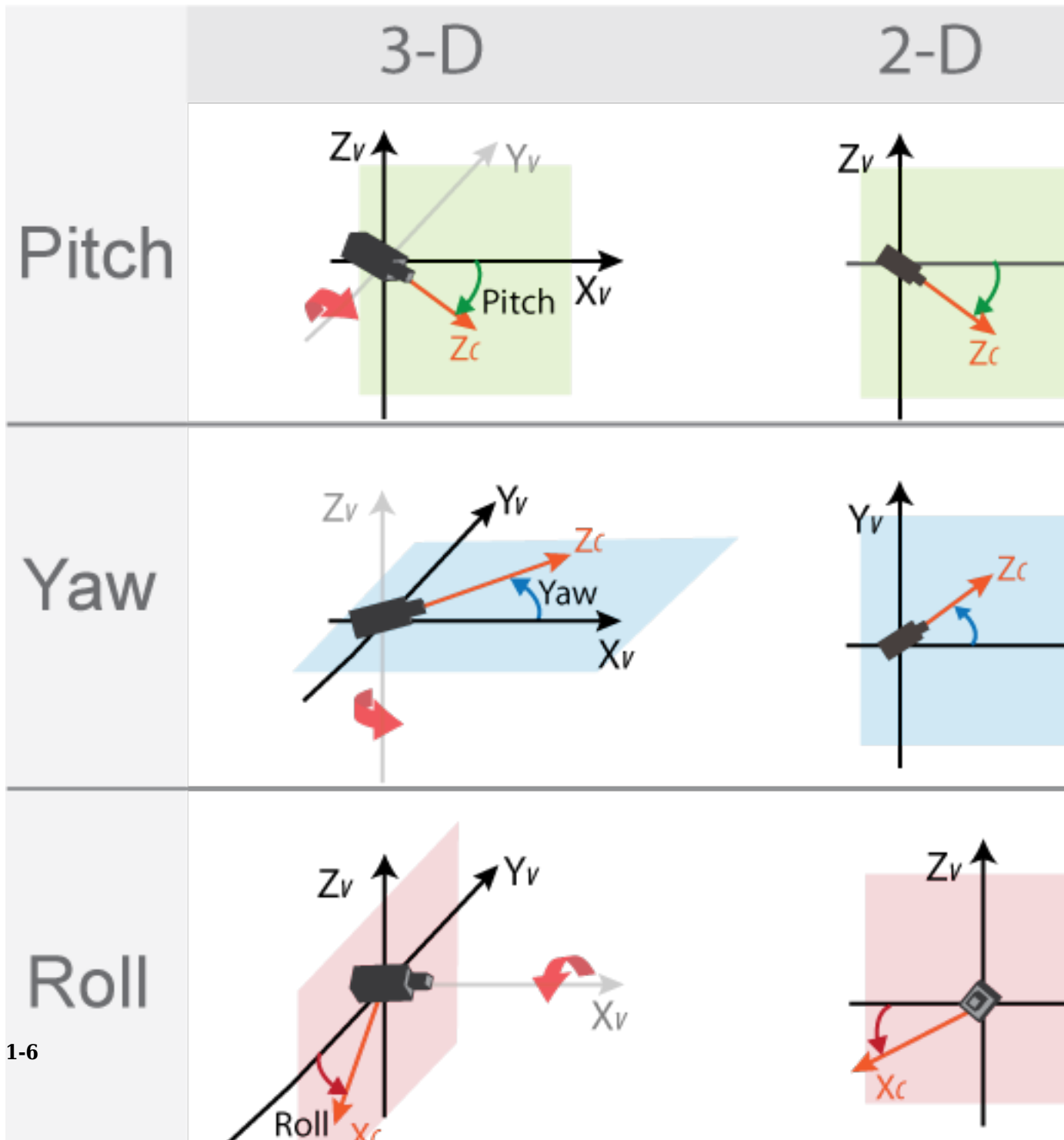
## Sensor Coordinate System

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 system 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.



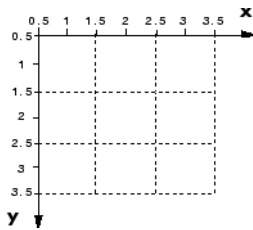
The yaw, pitch, and roll angles of sensors follow an ISO convention. These angles have positive clockwise directions when looking in the positive direction of the  $Z$ -,  $Y$ -, and  $X$ -axes, respectively.





## 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).

# 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 System 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.

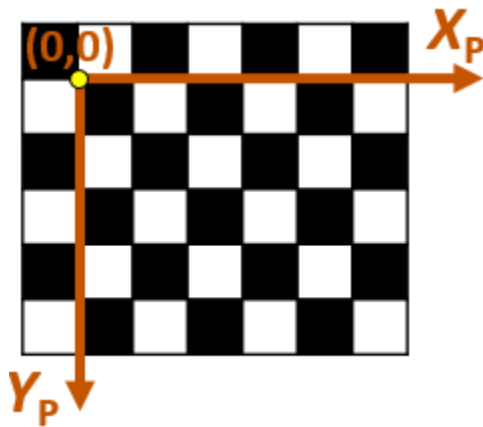
Alternatively, to better visualize the results, use the **Camera Calibrator** app. For information on setting up the camera, preparing the checkerboard pattern, and calibration techniques, see “Single Camera Calibrator App” (Computer Vision System Toolbox).

## Place Checkerboard for Extrinsic Parameter Estimation

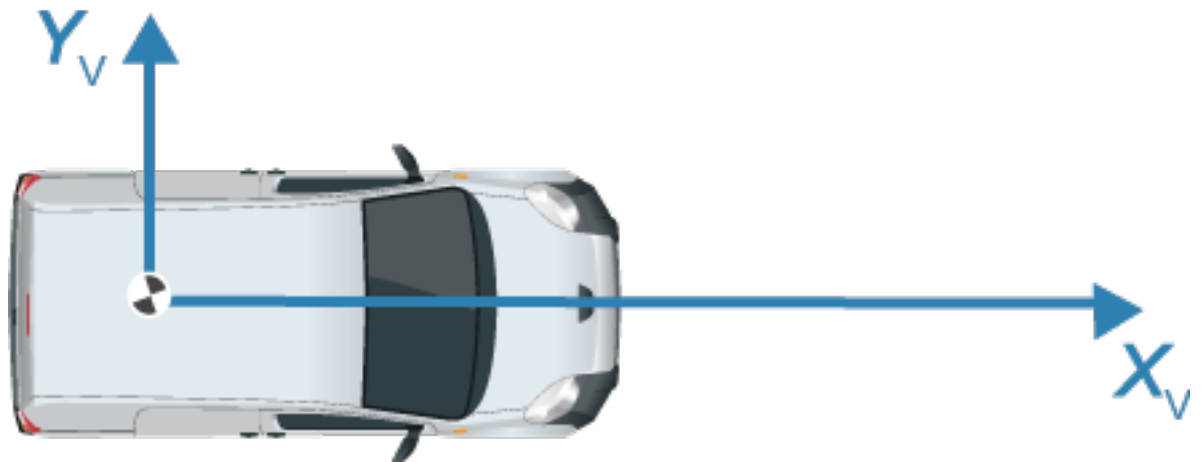
For a monocular camera mounted on a vehicle, the *extrinsic parameters* define the mounting position of that camera. These parameters include the rotation angles of the camera with respect to the vehicle coordinate system, and the height of the camera above the ground.

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.



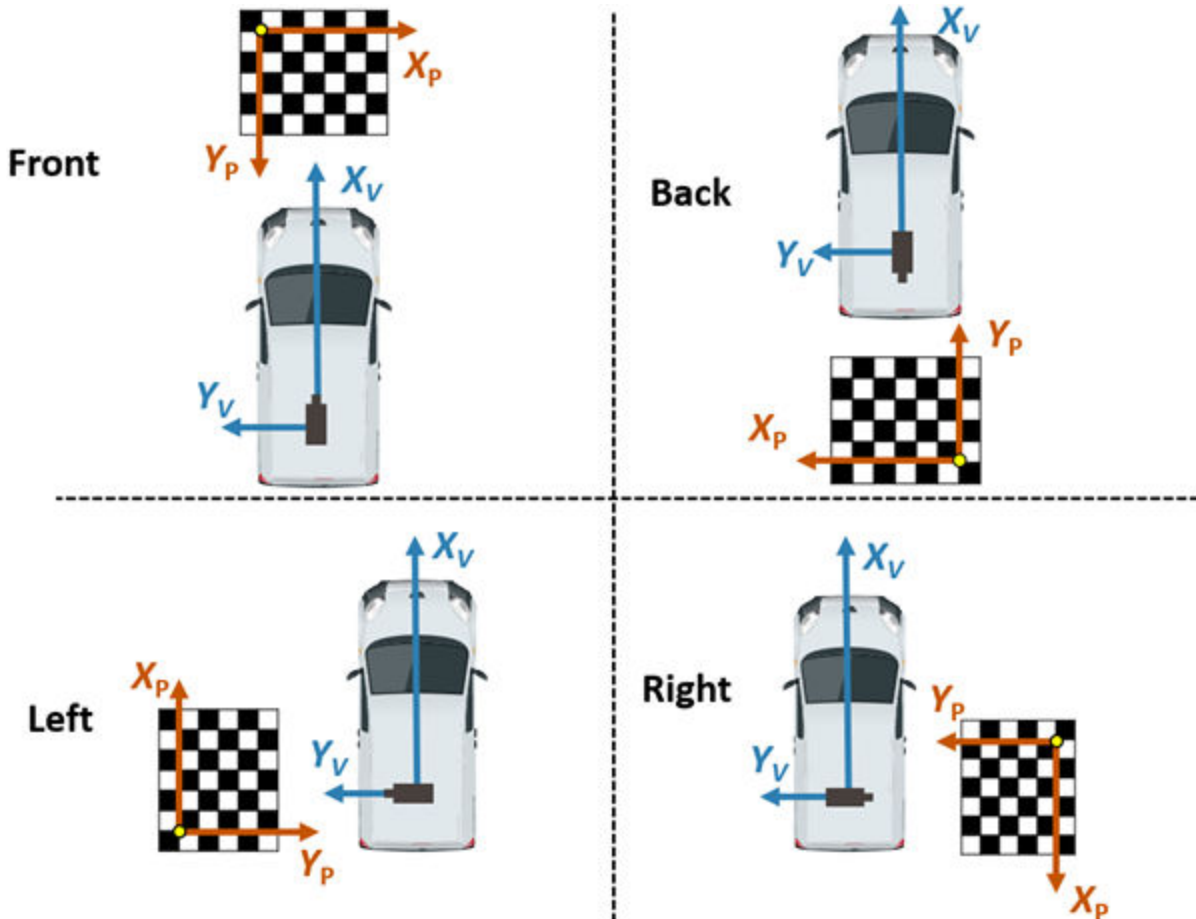
When placing the checkerboard pattern in relation to the vehicle, the  $X_P$ - and  $Y_P$ -axes must align with the  $X_V$ - and  $Y_V$ -axes of the vehicle. In the vehicle coordinate system, the  $X_V$ -axis points forward from the vehicle and the  $Y_V$ -axis points to the left, as viewed when facing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

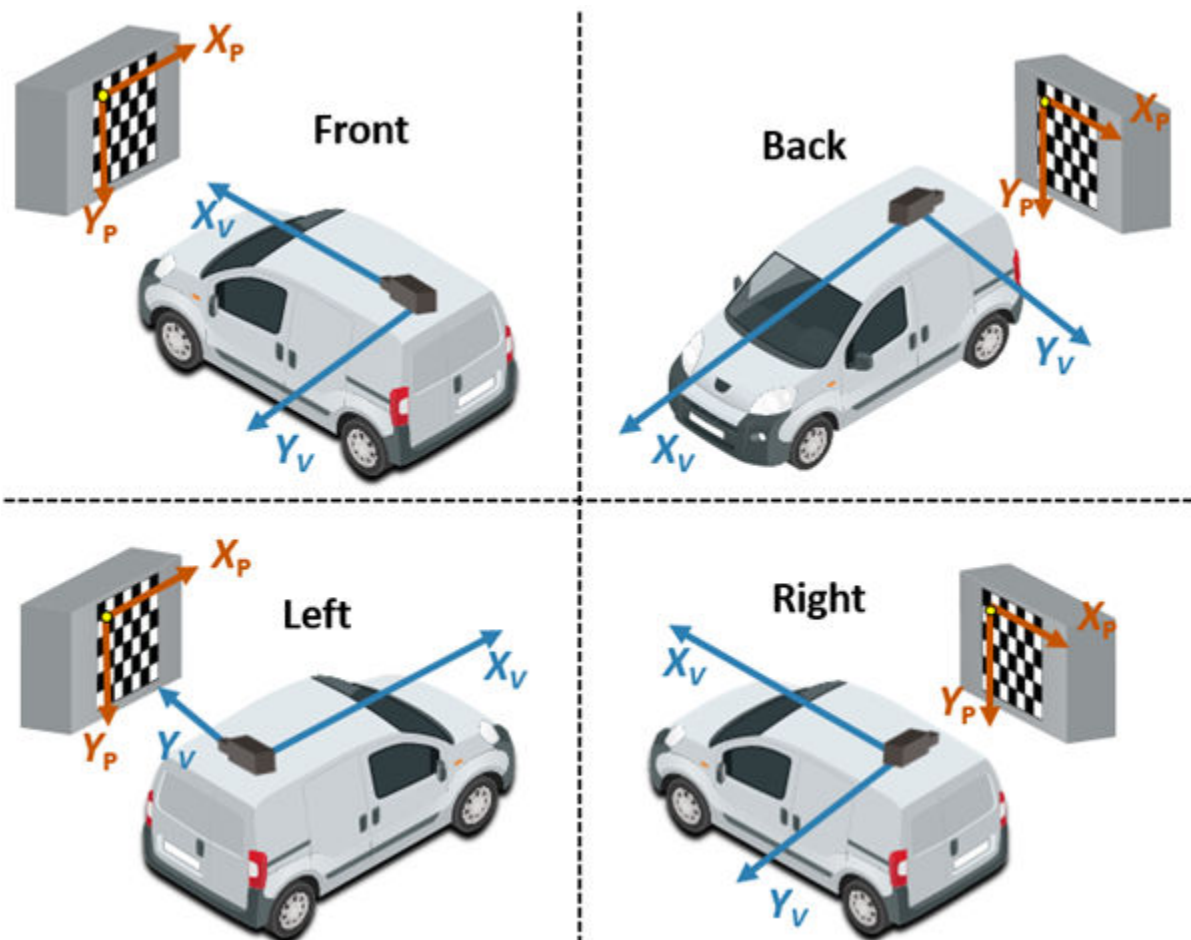
In the horizontal orientation, the checkerboard pattern is either on the ground or parallel to the ground. You can place the pattern in front of the vehicle, in back of the vehicle, or on the left or right side of the vehicle.



### 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 System Toolbox” on page 1-2
- “Configure Monocular Fisheye Camera”
- “Single Camera Calibrator App” (Computer Vision System Toolbox)



# Ground Truth Labeling and Verification

---

- “Get Started with the Ground Truth Labeler” on page 2-2
- “Use Custom Data Source Reader for Ground Truth Labeling” on page 2-23

# 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.

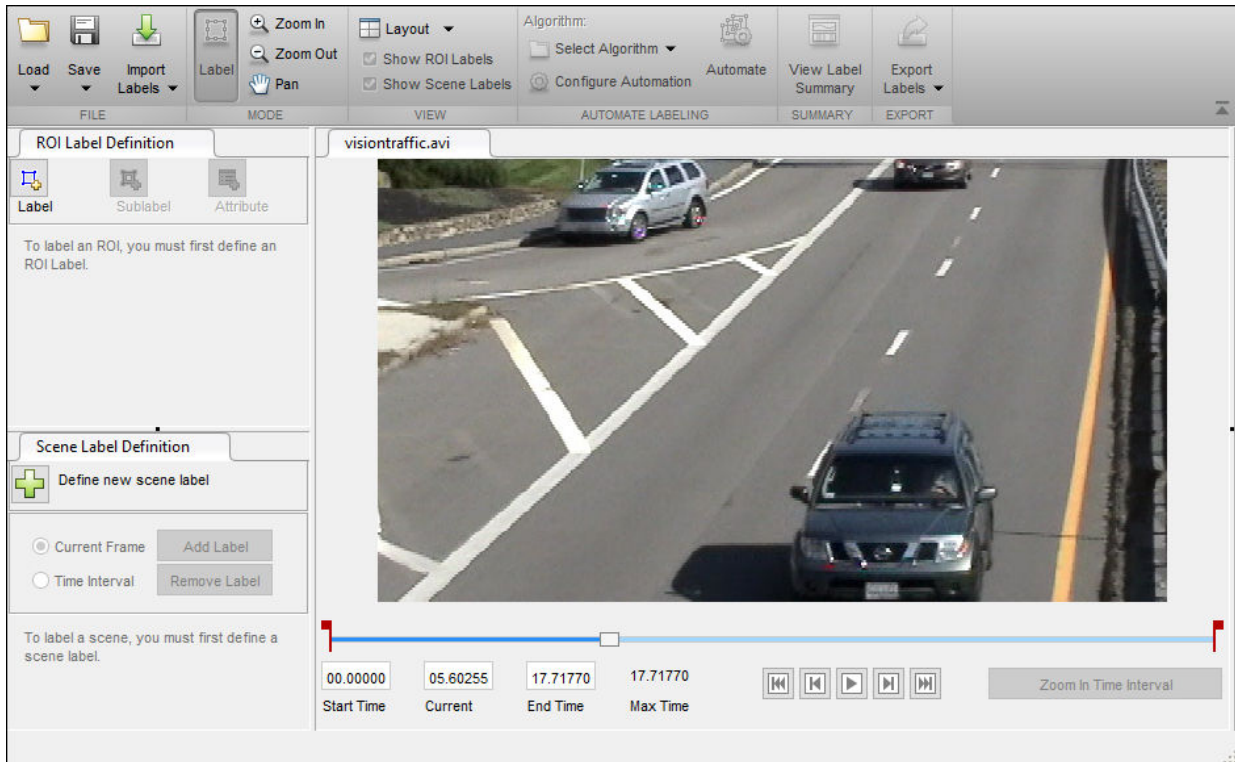
## Load Unlabeled Data

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`.

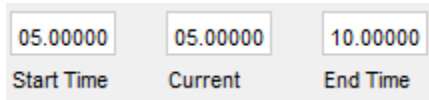
To load a custom data source that is readable by `VideoReader` or `imread`, see “Use Custom Data Source Reader for Ground Truth Labeling” (Computer Vision System Toolbox).

## 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 **Start Time** box, type 5.
- 2 In the **Current Time** box, type 5 so that the slider is at the start of the time interval.

**3** In the **End Time** box, type 10.



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 on the video frames. 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. See “Label Pixels for Semantic Segmentation” (Computer Vision System Toolbox).	Vehicles, road surface, trees, pavement 

In this example, you define a Rectangle ROI label for labeling the vehicles.

- 1 In the **ROI Label Definition** pane on the left, click **Label**.
- 2 Create a Rectangle ROI label named `vehicle` and optionally write a description. Click **OK**.

The **vehicle** label appears in the **ROI Label Definition** pane and is selected by default.

- 3 In the first video frame within the time interval, use the mouse to draw rectangular **vehicle** ROIs around the two vehicles.



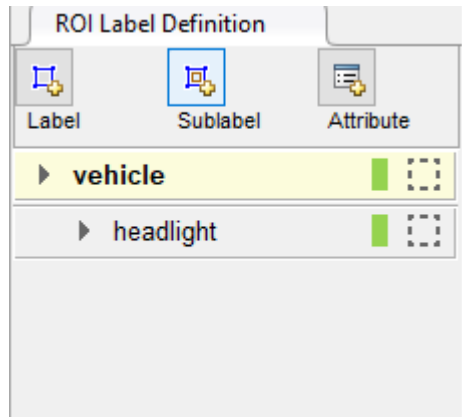
## Create Sublabels

A sublabel is a type of ROI label that corresponds to part of 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.

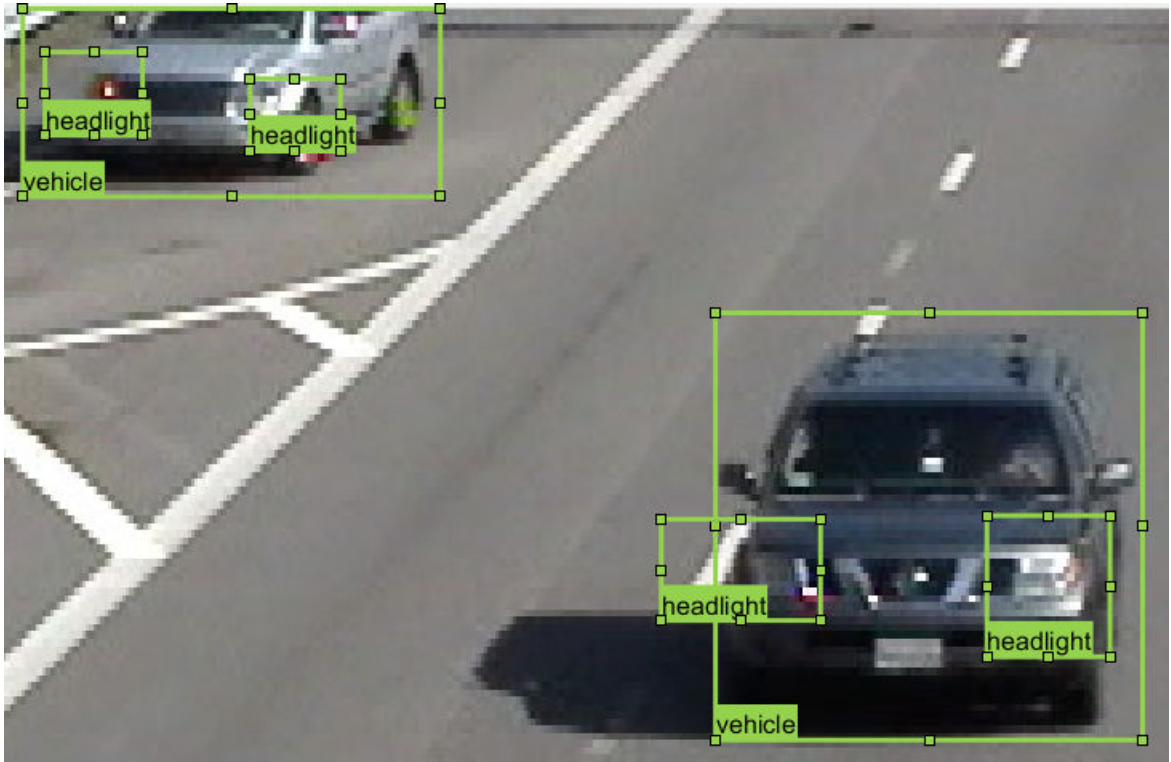
Define a sublabel for the vehicle headlights.

- 1 In the **ROI Label Definition** pane on the left, click **Sublabel**.
- 2 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, **vehicle**, and has the same color as its parent label.



- 3 In the **ROI Label Definition** pane, select the **headlight** sublabel.
- 4 In the video frame, select one of the **vehicle** labels. To draw a sublabel in a video frame, you must always select a parent label first. Draw a **headlight** sublabel around one of the vehicle headlights.
- 5 Select the **vehicle** label again and draw a **headlight** sublabel around the other headlight.
- 6 Repeat the previous steps to label the headlights of the other vehicle. To draw the labels more precisely, use the **Pan**, **Zoom In**, and **Zoom Out** options available from the toolbar.



When you select one of the **vehicle** labels, the **Sublabels** section of the **Attributes and Sublabels** pane displays information about the **headlight** sublabels associated with that label.




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 System 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.

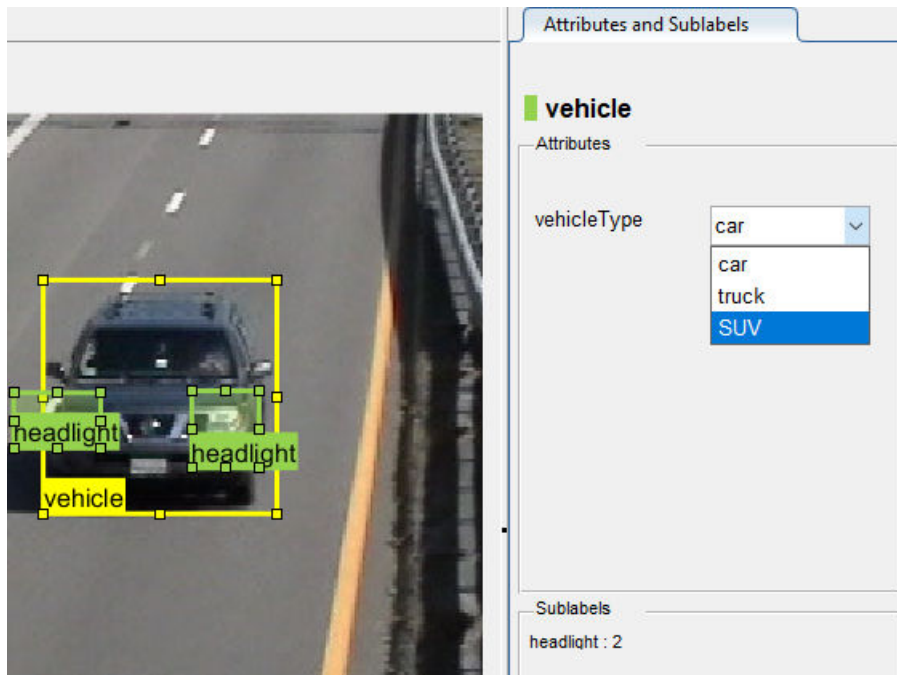
You can define these types of attributes.

Attribute Type	Sample Attribute Definition	Sample Default Values
Numeric Value	Attribute Name <input type="text" value="numWheels"/> Default Scalar Value (Optional) <input type="text" value="4"/>	
String	Attribute Name <input type="text" value="color"/> Default Value (Optional) <input type="text"/>	String
Logical	Attribute Name <input type="text" value="inMotion"/> Default Value (Optional) <input type="text" value="True"/>	Logical
List	Attribute Name <input type="text" value="vehicleType"/> List Items (Each item must appear on a new line) car truck SUV	List True car car truck SUV

Add an attribute for the vehicle type.

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

- 2 In the **Attribute Name** box, type `vehicleType`. Set the attribute type to `List`.
- 3 In the **List Items** section, type different types of vehicles, such as `car`, `truck`, and `SUV`, each on its own line. Optionally give the attribute a description, and click **OK**.
- 4 In the first frame of the video, select a **vehicle** ROI label. In the **Attributes and Sublabels** pane, select the appropriate **vehicleType** attribute value for that vehicle.
- 5 Repeat the previous step to assign a **vehicleType** 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`.



- Repeat the previous step to set the **isOn** attribute for the other headlights.

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.

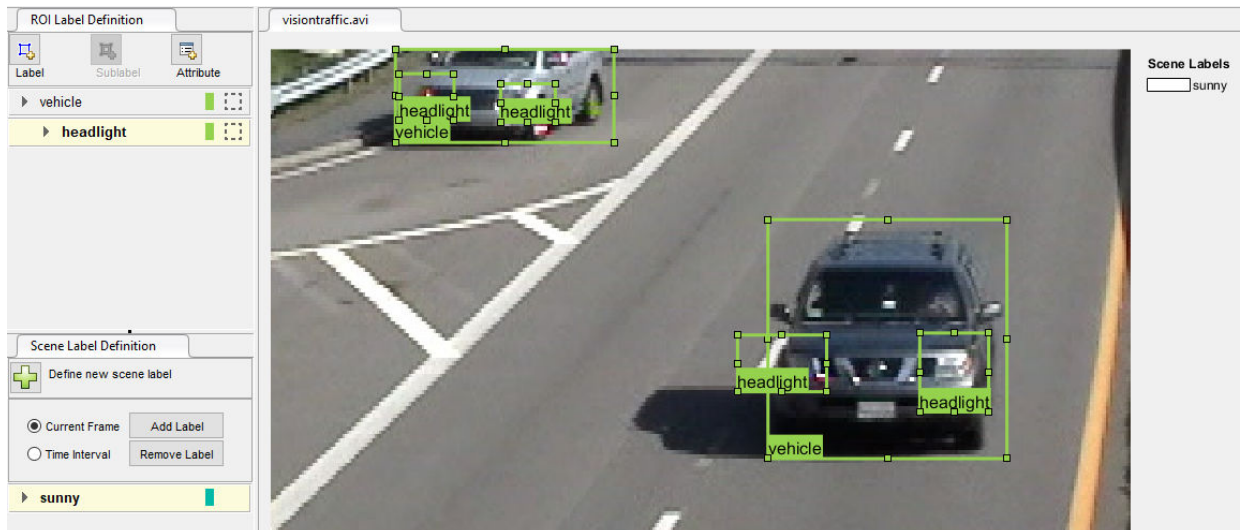
## Create Scene Labels

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.

- In the **Scene Label Definition** pane on the left, click the **Define new scene label** button, and create a scene label named **sunny**. Click **OK**.

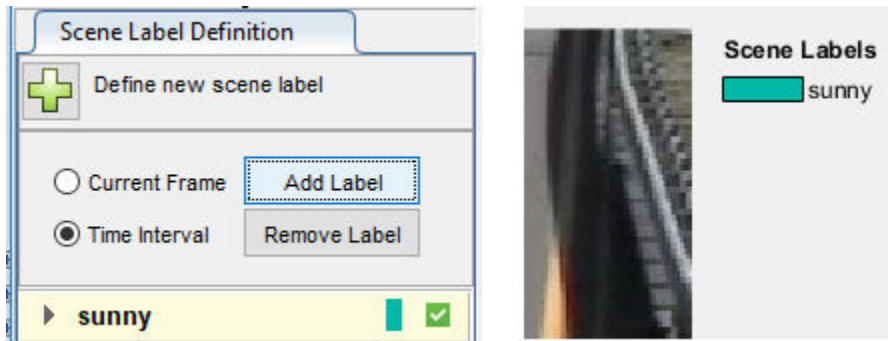
The **Scene Label Definition** pane shows the scene label definition. The scene labels that are applied to the current frame appear in the **Scene Labels** pane on the right. The **sunny** scene label is empty (white), because the scene label has not yet been applied to the frame.



- 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.

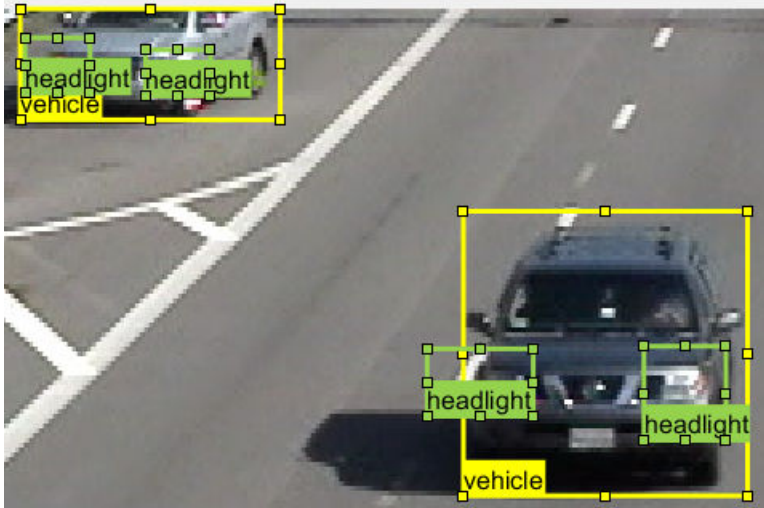
Advance frame by frame and draw the label and sublabel ROIs manually. Also update the attribute information for these ROIs.

### 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 System Toolbox) and “Temporal Automation Algorithms” (Computer Vision System 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 **vehicle** 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 **vehicle** label annotations. The labels are highlighted in yellow.



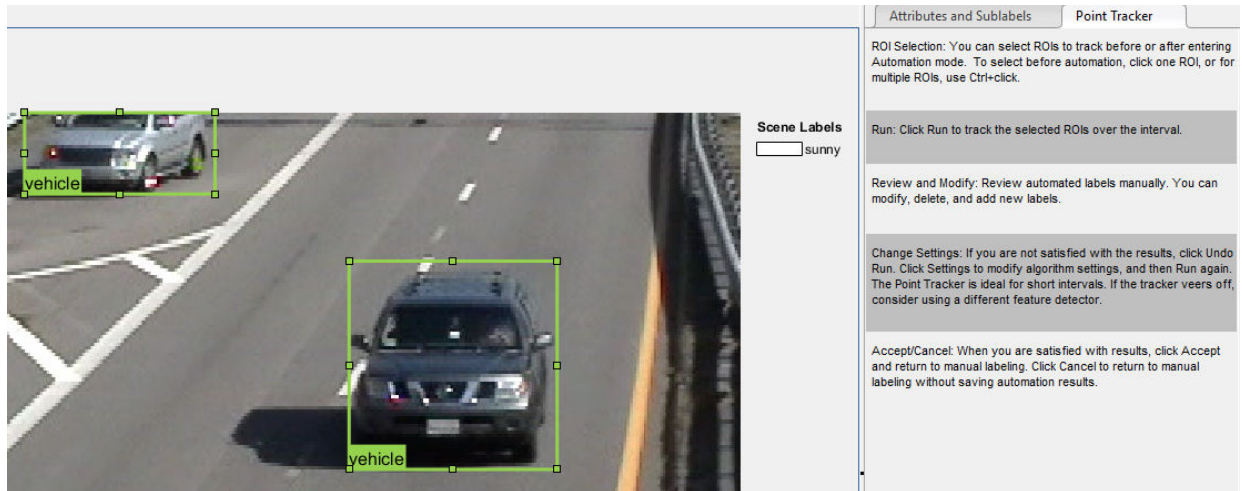
- 2 From the app toolstrip, select **Select Algorithm > Point Tracker**. This algorithm tracks one or more rectangle ROIs over short intervals using the Kanade-Lucas-Tomasi (KLT) algorithm.
- 3 (optional) Configure the automation settings. Click **Configure Automation**. By default, the automation algorithm 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 algorithm from	Example
Forward	Start time to End time	
	Current time to End time	
Reverse	End time to Start time	

Direction of automation	Run algorithm from	Example
	Current time to Start time	

Leave **Import selected ROIs** selected so that the **vehicle** labels you selected can be imported into the automation session.

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



- Click **Run** to track the selected ROIs over the interval.
- 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.

- 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 toolstrip, 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 System 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 “Train Object Detector or Semantic Segmentation Network from Ground Truth Data” (Computer Vision System 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 System 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: [2x4 table]  
LabelData: [531x2 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 =
```

```
2x4 table
```

Name	Type	Description	Hierarchy
'vehicle'	Rectangle	''	[1x1 struct]
'sunny'	Scene	''	[]

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 `vehicle` label.

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

```
ans =
  struct with fields:
    vehicleType: [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 `vehicleType` attribute.

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

```
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 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.

```
gTruth.LabelData
```

```
ans =
```

```
531x2 timetable

      Time          vehicle          sunny
      -----          -
0 sec          [1x0 struct]         false
0.033367 sec   [1x0 struct]         false
0.066733 sec   [1x0 struct]         false
...
```

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

```
gTruthInterval = gTruth.LabelData(timerange('00:00:05','00:00:10'),:)
```

```
gTruthInterval =
```

```
150x2 timetable

      Time          vehicle          sunny
      -----          -
5.005 sec      [1x2 struct]         true
5.0384 sec     [1x2 struct]         true
5.0717 sec     [1x2 struct]         true
...
```

For each `vehicle` 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.

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

```
ans =
```

```
1x4 single row vector

415.5744    89.7500   120.2985   120.2985
```

```
ans =
```

```
1x4 single row vector

230.0450     1.4422   109.7325    47.2849
```



## 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 System 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 System Toolbox)
- “Use Sublabels and Attributes to Label Ground Truth Data” (Computer Vision System Toolbox)

- “Label Pixels for Semantic Segmentation” (Computer Vision System Toolbox)
- “Create Automation Algorithm for Labeling” (Computer Vision System Toolbox)
- “View Summary of Ground Truth Labels” (Computer Vision System Toolbox)
- “Share and Store Labeled Ground Truth Data” (Computer Vision System Toolbox)
- “Train Object Detector or Semantic Segmentation Network from Ground Truth Data” (Computer Vision System Toolbox)

## Use Custom Data Source Reader for Ground Truth Labeling

### In this section...

“Import Data Source Using Custom Reader Dialog Box” on page 2-23

“Import Data Source Using Custom Reader Function” on page 2-24

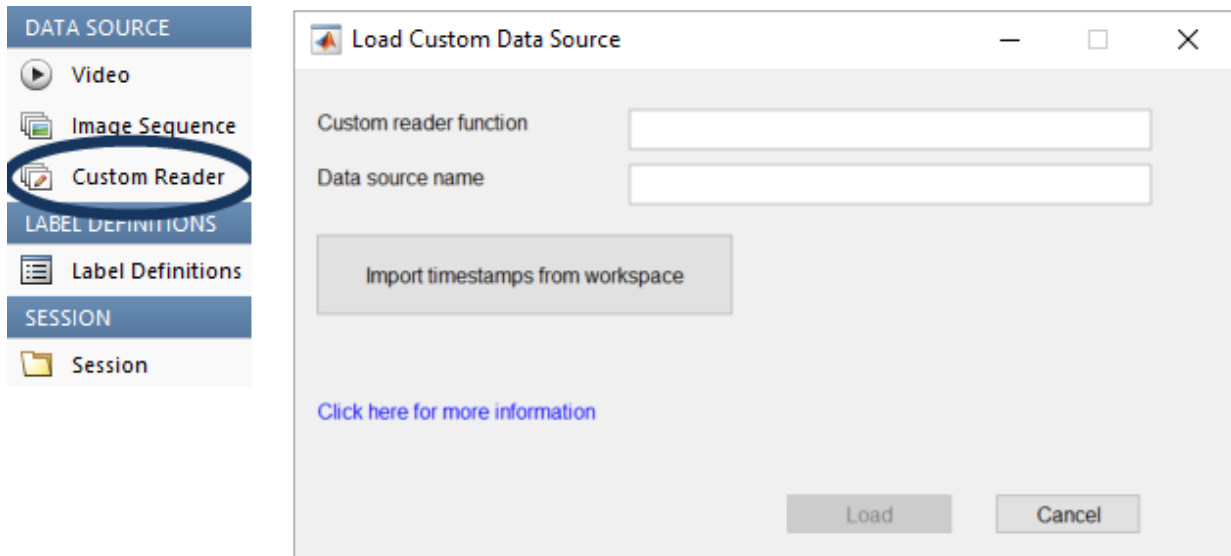
The **Ground Truth Labeler** (requires Automated Driving System Toolbox) and **Video Labeler** apps enable you to label ground truth data in a video or in a sequence of images.

You can use a custom reader to import any video or sequence of images that is supported by `VideoReader` or `imread`. You can either use the custom reader dialog box in the app or open the app and specify a custom reader source.

The **Image Labeler** app does not support custom data source readers.

### Import Data Source Using Custom Reader Dialog Box

In your app, **Load > Custom Reader** to load your data by using a custom reader function. You must provide the **Custom reader function** handle and the **Data source name**. In addition, you must import corresponding timestamps from the MATLAB workspace.



## Import Data Source Using Custom Reader Function

### Specify the Custom Reader

Specify a custom reader as a function handle. The custom reader must have the syntax:

```
outputImage = readerFcn(sourceName,currentTimeStamp)
```

where `readerFcn` is the name of your custom reader function.

The custom reader function loads an image from `sourceName`, which corresponds to the current timestamp specified by `currentTimeStamp`.

```
currentTimeStamp = timestamps(currIdx);
```

The `outputImage` from the custom function must be a grayscale or RGB image in any format supported by `imshow`. `currentTimeStamp` is a scalar value that corresponds to the current frame that the algorithm is executing.

### Read Ground Truth Data Using Custom Reader

Use the `groundTruthDataSource` function to read the custom source data with the custom reader function handle:

```
gtSource = groundTruthDataSource(sourceName, readerFcn, timeStamps)
```

The syntax returns a `groundTruthDataSource` object with the custom reader function handle, `readerFcn`. The app uses the handle to load the custom data source specified by `sourceName`. The custom reader function loads an image from `sourceName` that corresponds to the current timestamp specified by the indexed value in the `timeStamps` vector.

The syntax returns a `groundTruthDataSource` object, which the app uses to read data from the custom source.

### Read Ground Truth Data Using Custom Reader

Use the `groundTruthDataSource` function to read the custom source data with the custom reader function handle:

```
gtSource = groundTruthDataSource(sourceName, readerFcn, timeStamps)
```

The syntax returns a `groundTruthDataSource` object with the custom reader function handle, `readerFcn`. The app uses the handle to load the custom data source specified by `sourceName`. The custom reader function loads an image from `sourceName` that corresponds to the current timestamp specified by the indexed value in the `timeStamps` vector.

The syntax returns a `groundTruthDataSource` object, which the app uses to read data from the custom source.

### Import Ground Truth Data into App

You can import the returned `groundTruthDataSource` object into the **Ground Truth Labeler** or **Video Labeler** app. For example:

```
groundTruthLabeler(gtSource)
```

```
videoLabeler(gtSource)
```

## See Also

### Apps

**Ground Truth Labeler** | **Video Labeler**

### **Functions**

groundTruth | groundTruthDataSource

### **More About**

- “Get Started with the Ground Truth Labeler” on page 2-2
- “Get Started with the Video Labeler” (Computer Vision System Toolbox)

# Tracking and Sensor Fusion

---

- “Visualize Sensor Data and Tracks in Bird's-Eye Scope” on page 3-2
- “Linear Kalman Filters” on page 3-11
- “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:

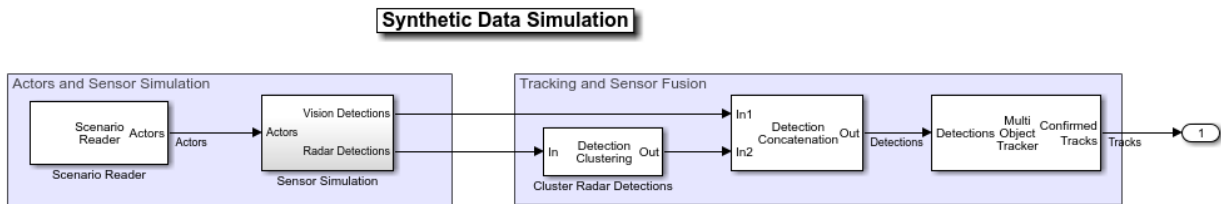
- Sensor coverages of vision and radar sensors
- Sensor detections of actors and lane boundaries
- Tracks of moving objects in the scenario

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 for tracks. This model is used in the “Sensor Fusion Using Synthetic Radar and Vision Data in Simulink” example.


```
model = fullfile(matlabroot, 'examples', 'driving', 'SyntheticDataSimulinkExample');
open_system(model)
```



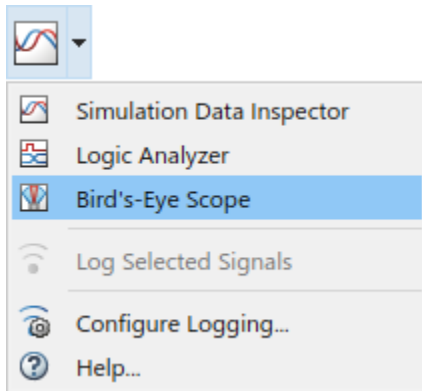
Open the scope. From the Simulink model toolbar, click the **Bird's-Eye Scope** button



. If instead you see a button for a different model visualization tool, such as the

**Simulation Data Inspector**  or **Logic Analyzer**  , click the arrow next to the displayed button and select **Bird's-Eye Scope**.

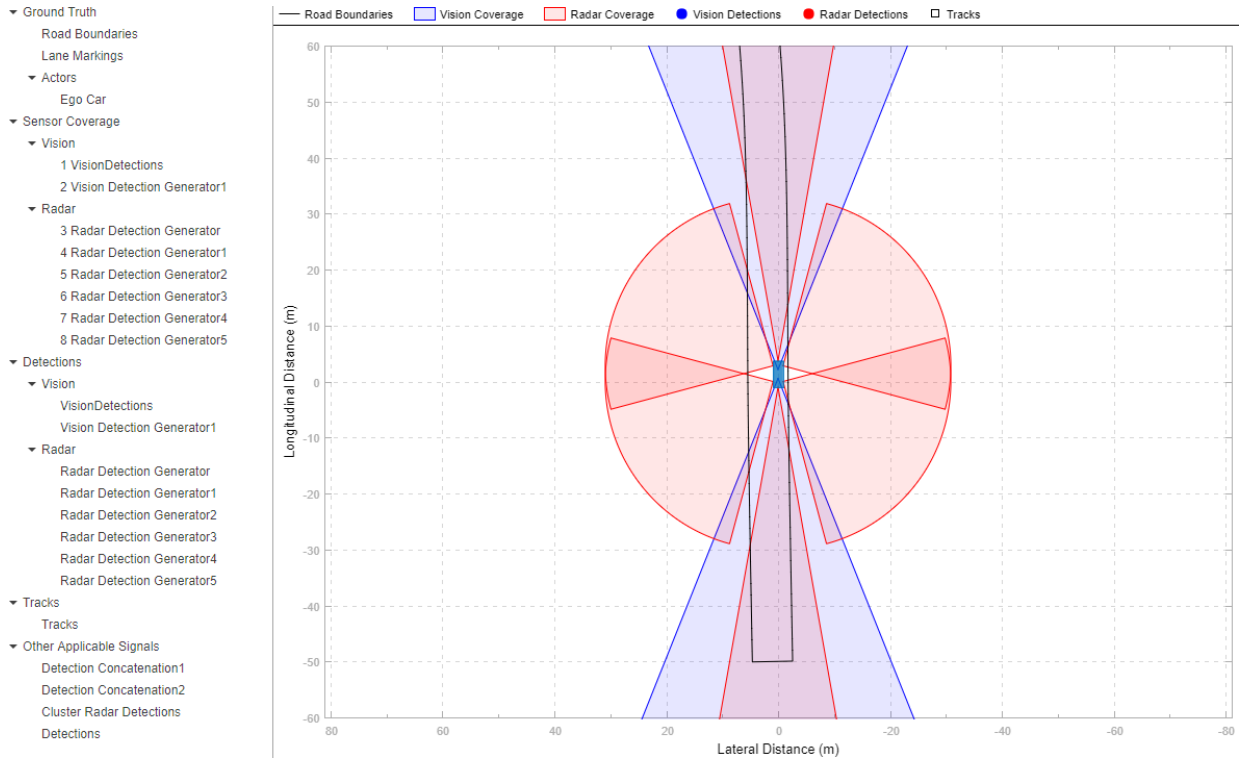




## Find Signals

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

### 3 Tracking and Sensor Fusion



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

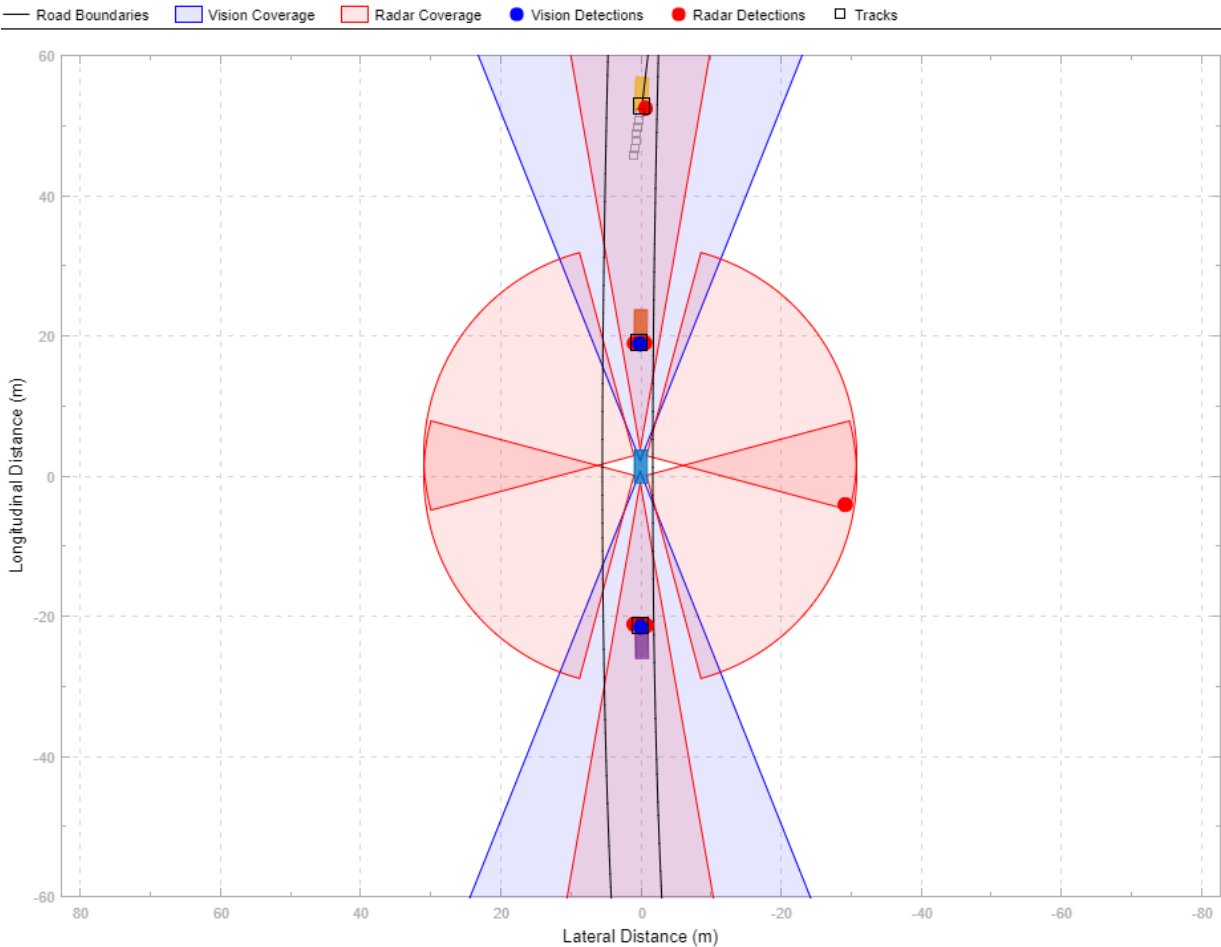
Signal Group	Description	Signal Sources
<p><b>Ground Truth</b></p>	<p>Road boundaries, lane markings, and actors in the scenario, including the ego vehicle</p> <p>You cannot modify this group or any of the signals within it.</p>	<ul style="list-style-type: none"> <li>• Scenario Reader block (such as the one used in the “Sensor Fusion Using Synthetic Radar and Vision Data in Simulink” example)</li> <li>• Vision Detection Generator and Radar Detection Generator blocks (for actor profile information only, such as the length, width, and height of actors)                             <ul style="list-style-type: none"> <li>• If actor profile information is not set or is inconsistent between blocks, the scope sets the actor profiles to the block defaults.</li> <li>• The profile of the ego vehicle is always set to the block defaults.</li> </ul> </li> </ul>
<p><b>Sensor Coverage</b></p>	<p>Coverage areas of your vision and radar sensors, sorted into <b>Vision</b> and <b>Radar</b> subgroups</p> <p>You can move or modify these subgroups and their signals. You cannot move or modify the top-level <b>Sensor Coverage</b> group.</p>	<ul style="list-style-type: none"> <li>• Vision Detection Generator block</li> <li>• Radar Detection Generator block</li> </ul>


Signal Group	Description	Signal Sources
<b>Detections</b>	<p>Detections obtained from your vision and radar sensors, sorted into <b>Vision</b> and <b>Radar</b> subgroups</p> <p>You can move or modify these subgroups and their signals. You cannot move or modify the top-level <b>Detections</b> group.</p>	<ul style="list-style-type: none"> <li>• Vision Detection Generator block</li> <li>• Radar Detection Generator block</li> </ul>
<b>Tracks</b>	Tracks of objects in the scenario	<ul style="list-style-type: none"> <li>• Multi Object Tracker block</li> </ul>
<b>Other Applicable Signals</b>	<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"> <li>• Blocks that combine or cluster signals (such as the Detection Concatenation block)</li> <li>• Nonvirtual Simulink buses containing position and velocity information for detections and tracks</li> </ul>

The scope canvas displays the signals grouped in **Ground Truth** and **Sensor Coverage** only. The signals in **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.



During simulation, the scope canvas remains centered on the ego vehicle. You can pan and zoom to inspect other parts of the model during simulation. To center on the ego vehicle again, in the upper right corner of the scope canvas, click the home button .

You can update the properties of signals during simulation. To access the properties of a signal, first select the signal from the left pane. Then, from the scope toolbar, click **Properties**. For example, with these properties, you can show and hide coverages or detections. You can also change the color or transparency of certain coverages to highlight them.

Under **Settings**, you can change the axis limits and the display of the signal names during simulation. You cannot change the **Track position selector** and **Track velocity selector** parameters during simulation. For more details on these parameters, see the parameters section on the **Bird's-Eye Scope** reference page.

To prevent signals from displaying during the next simulation, first right-click the signal. Then, select **Move to Other Applicable** to move that signal into the **Other Applicable Signals** group.

### 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 Vision Detection Generator blocks. Then, on the **Measurements** tab, reduce the **Maximum detection range (m)** parameter to 50. To see how the sensor coverage changes, rerun the simulation.

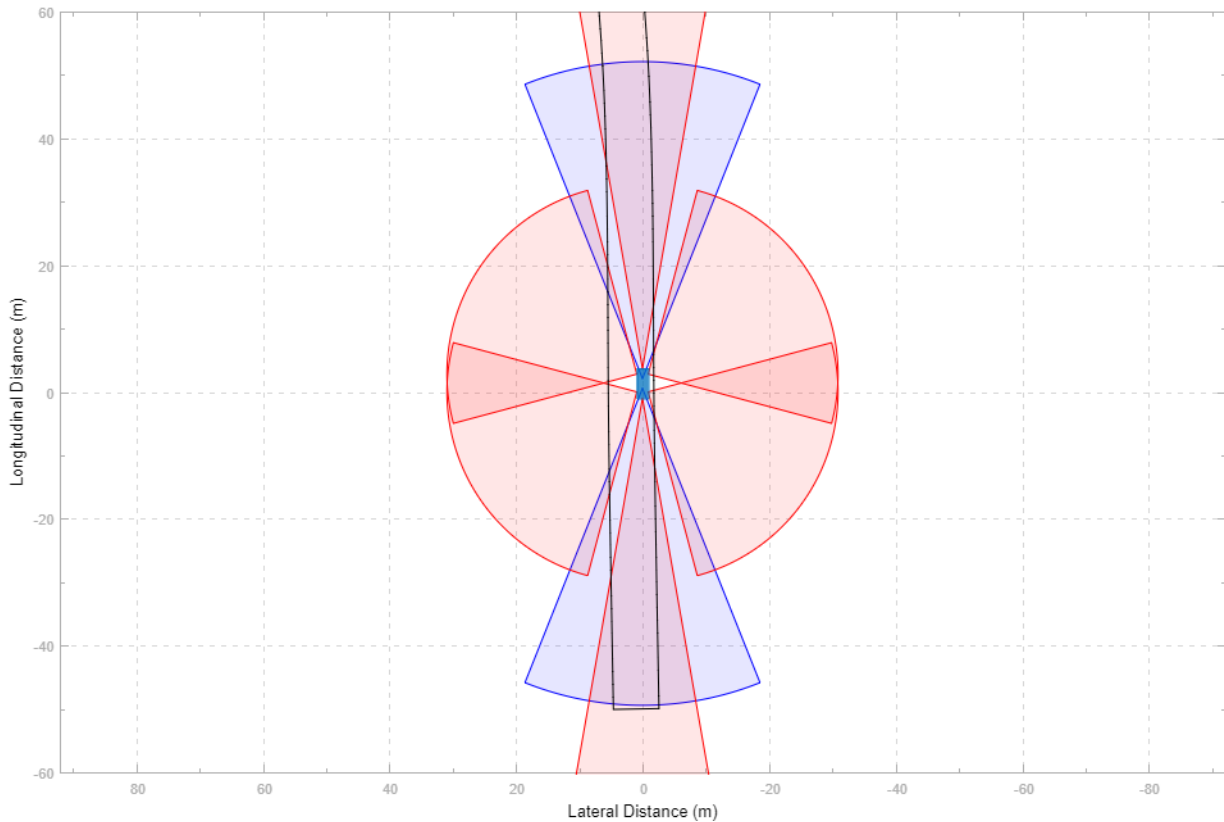
When you modify block parameters, you can rerun the simulation without having to find signals again. If you add or remove blocks, ports, or signal lines, then you must click **Find Signals** again before rerunning the simulation.

### 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.



## See Also

[Bird's-Eye Scope](#) | [Detection Concatenation](#) | [Multi Object Tracker](#) | [Radar Detection Generator](#) | [Vision Detection Generator](#)

## Related Examples

- [“Sensor Fusion Using Synthetic Radar and Vision Data in Simulink”](#)

- “Lane Keeping Assist with Lane Detection”
- “Adaptive Cruise Control with Sensor Fusion”
- “Lateral Control Tutorial”
- “Automatic Emergency Braking with Sensor Fusion”



# Linear Kalman Filters

## In this section...

“State Equations” on page 3-11

“Measurement Models” on page 3-13

“Linear Kalman Filter Equations” on page 3-13

“Filter Loop” on page 3-14

“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 System 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.

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

$v_k$  and 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

$$\mathbf{x}_{k+1|k+1} = \mathbf{x}_{k+1|k} + \mathbf{K}_{k+1}(z_{k+1} - z_{k+1|k})$$

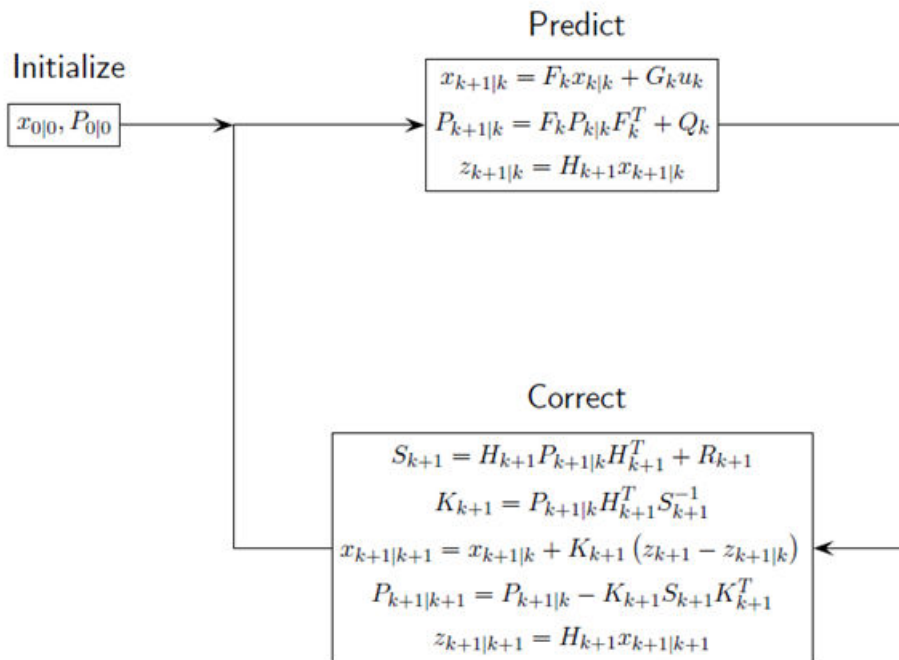
where  $\mathbf{K}_{k+1}$  is the Kalman gain. The corrected state is often called the *a posteriori* estimate of the state because it is derived after the measurement is included.

Correct the state covariance matrix

$$\mathbf{P}_{k+1|k+1} = \mathbf{P}_{k+1|k} - \mathbf{K}_{k+1}\mathbf{S}_{k+1}\mathbf{K}'_{k+1}$$

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

This figure summarizes the Kalman loop operations.



## 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.

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



## 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:

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

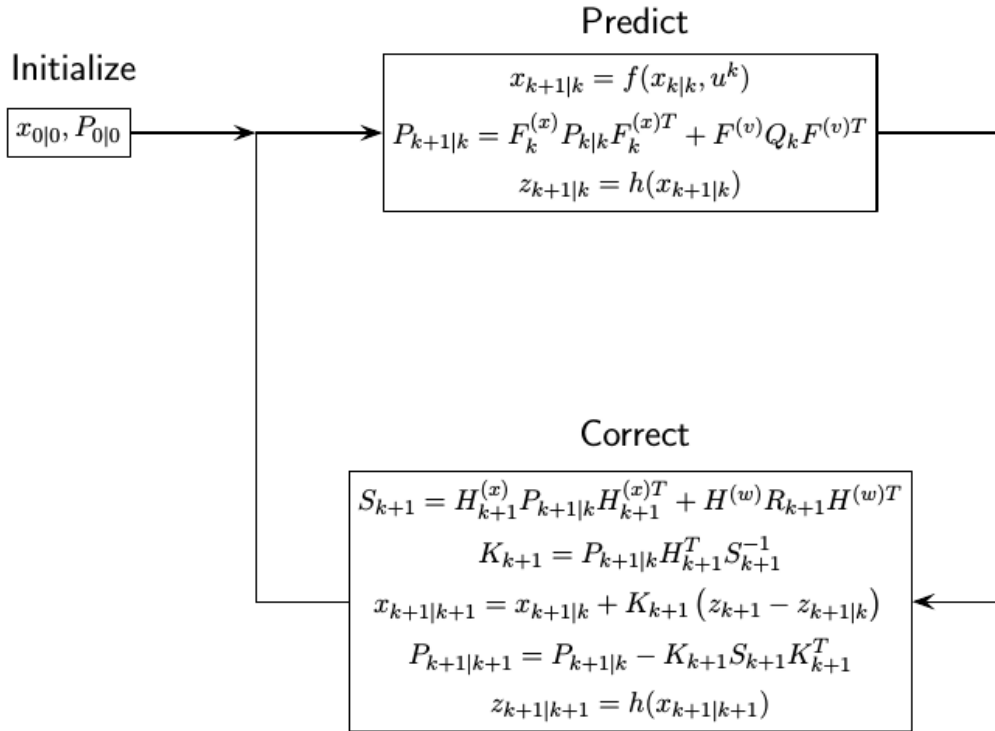
In this case,  $H^{(v)} = 1_N$ .

## 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 System 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

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

# 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.

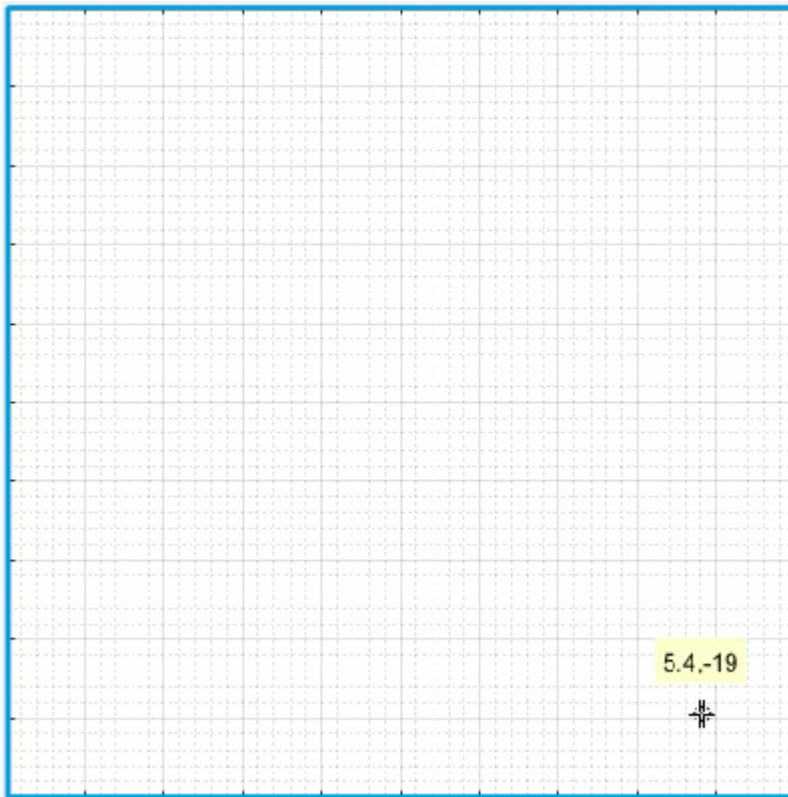
This example covers the entire workflow for creating a scenario and generating synthetic detections. Alternatively, you can generate detections from prebuilt scenarios. For more details, see “Generate Synthetic Detections from a Prebuilt Driving Scenario” on page 4-18.

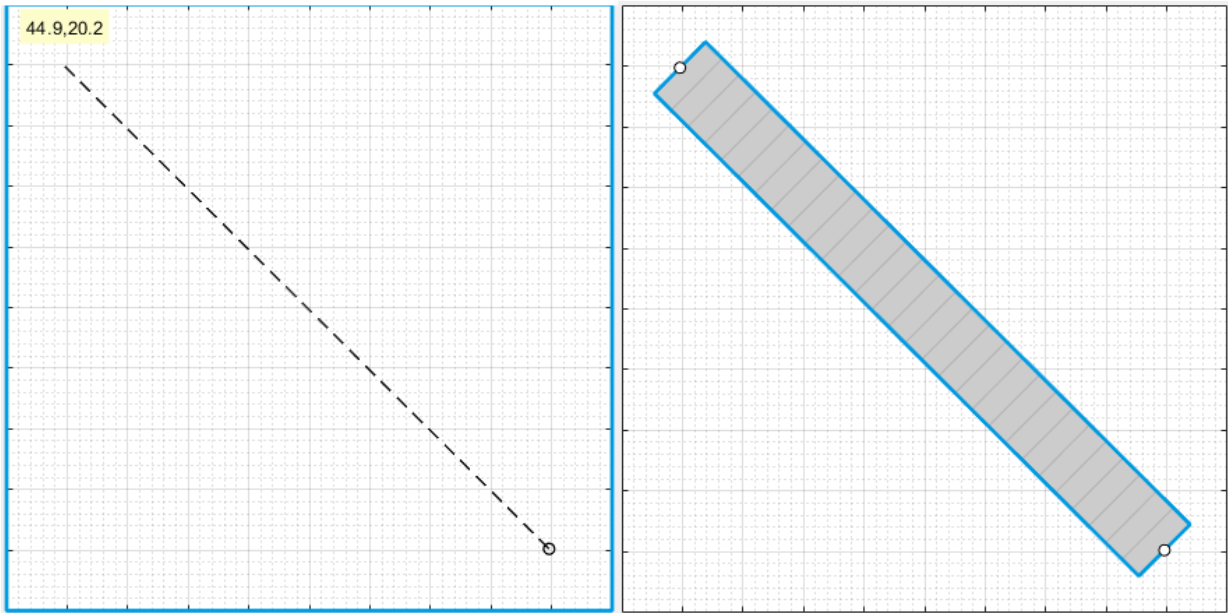
## Create a New Driving Scenario

To open a blank session of the app, at the MATLAB command prompt, enter `drivingScenarioDesigner`.

## Add a Road

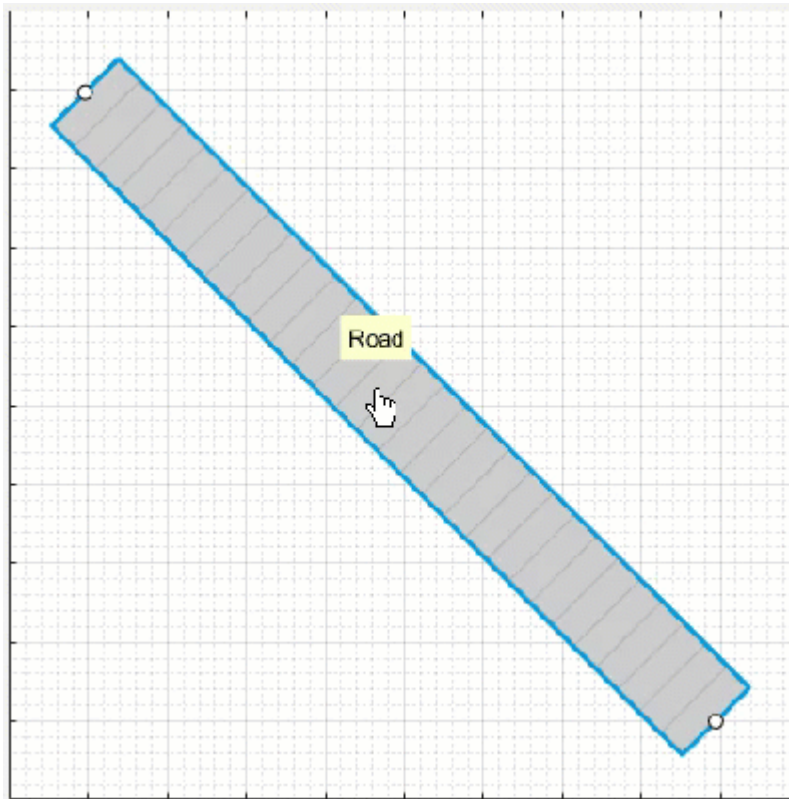
Add a curved road to the scenario canvas. From the app toolbar, click **Add Road**. Then click one corner of the canvas, extend the road to the opposite corner, and double-click to create the road.

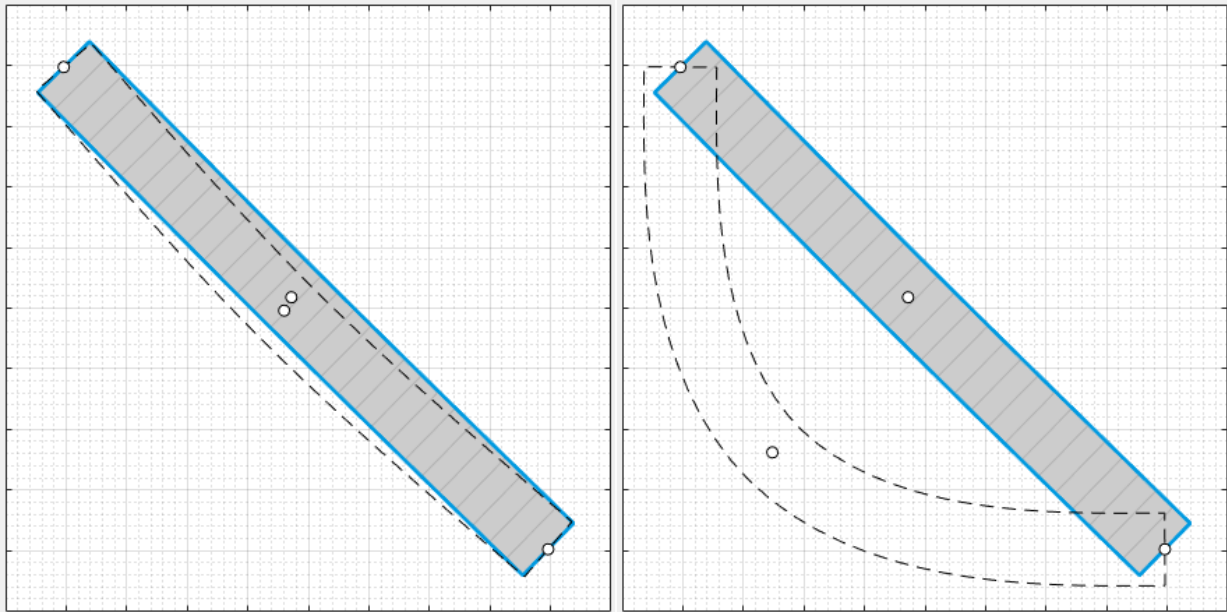




To make the road curve, add a road center around which to curve it. Right-click the middle of the road and select **Add Road Center**. Then drag the added road center to one of the empty corners of the canvas.



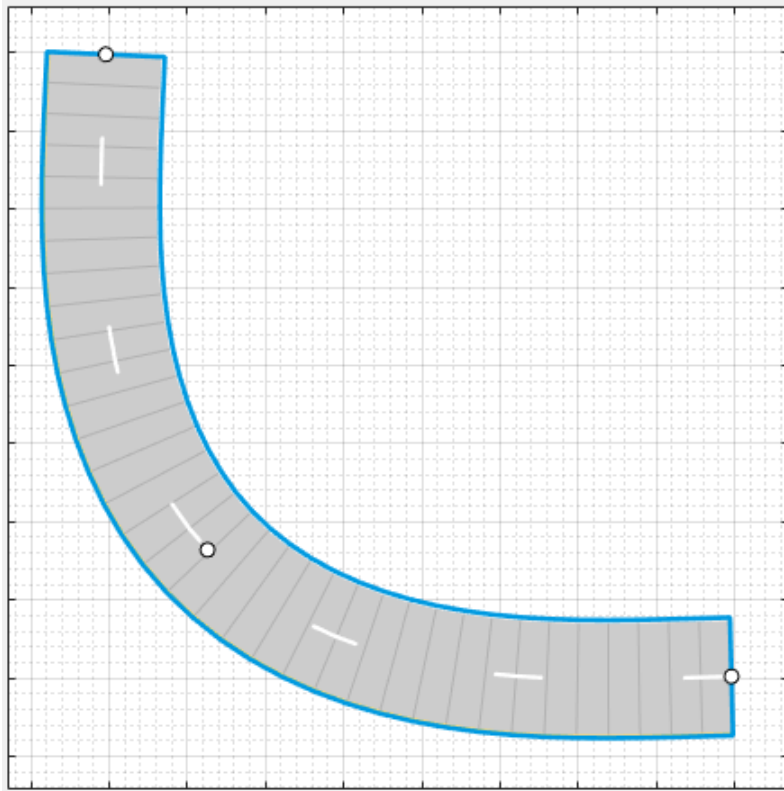




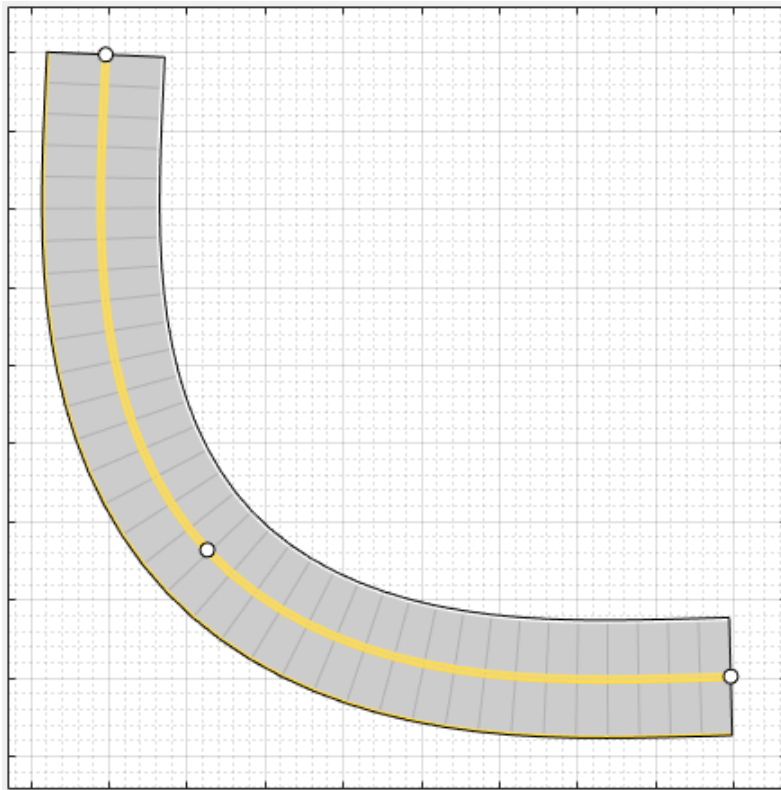
To adjust the road further, you can click and drag any of the road centers. To create more complex curves, add more road centers.

### 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 2 and the **Lane Width** to 3.6 meters, which is a typical highway lane width.



The road is now one-way and has solid lane markings on either side to indicate the shoulder. Make the road two-way by converting the center lane marking from a single dashed line to a solid double-yellow line. From the **Marking** list, select 2:Dashed. Then set the **Type** to DoubleSolid and specify the **Color** as the string yellow.



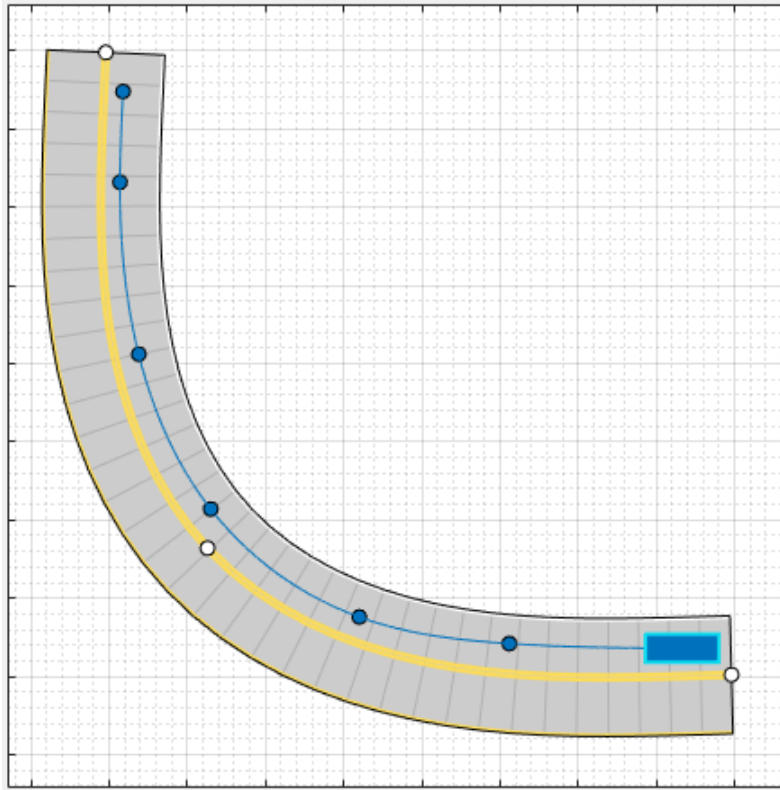
### Add Vehicles

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

### Add Ego Car

To add the ego car, 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

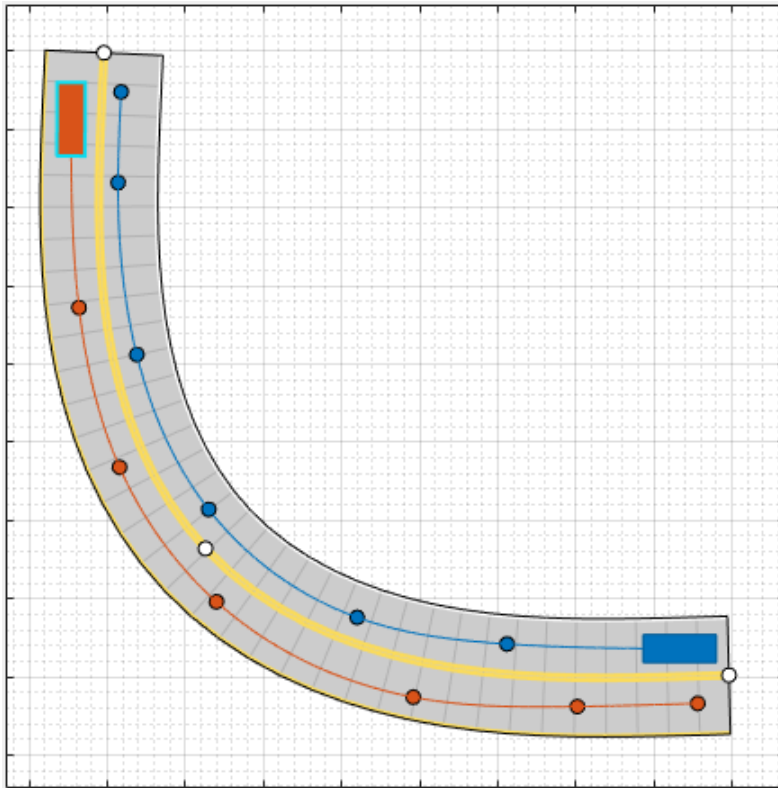
the trajectory, you can adjust the waypoints. You can also right-click the path to add new waypoints.



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.

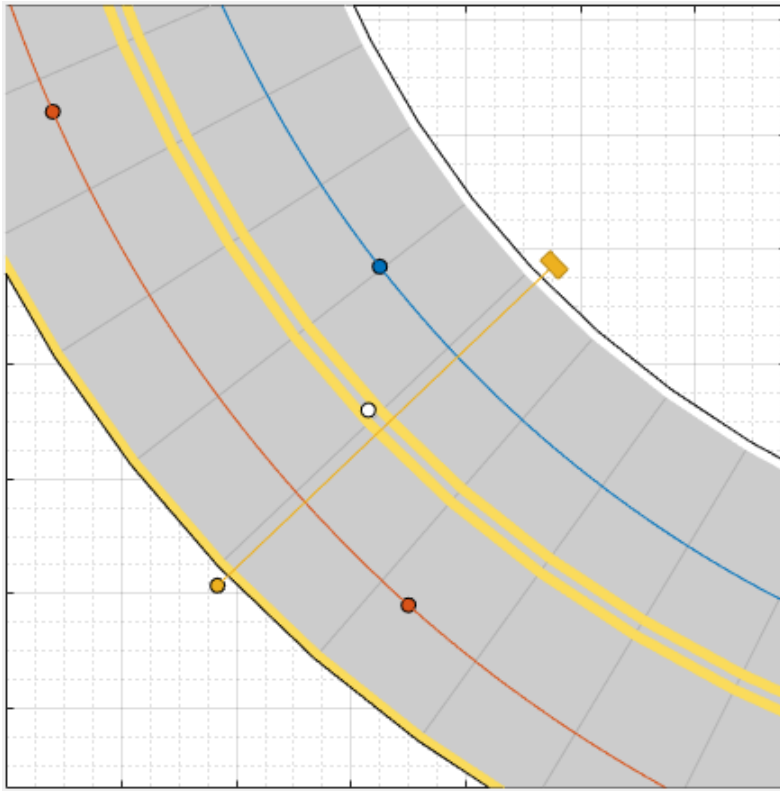
### Add Second Car

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

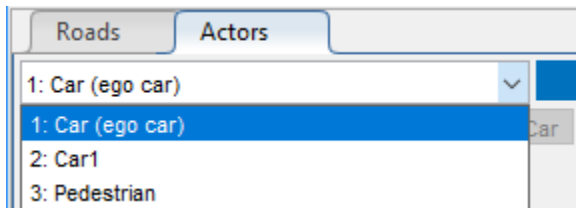


### Add a Pedestrian

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 car. Use these sensors to generate detections of the pedestrian, the lane boundaries, and the other vehicle.

#### Add Camera

From 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 car. 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 .

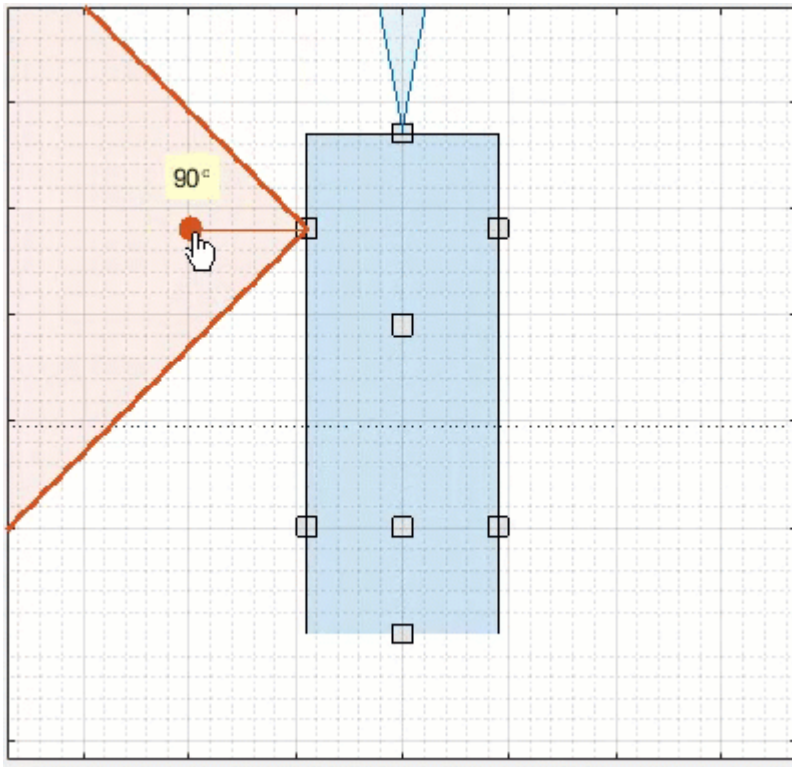
By default, the camera detects only actors and not lanes. To enable lane detections, on the **Sensors** tab in the left pane, expand the **Detection Parameters** section and set **Detection Type** to **Objects & Lanes**. Then expand the **Lane Settings** section and update the settings as needed.

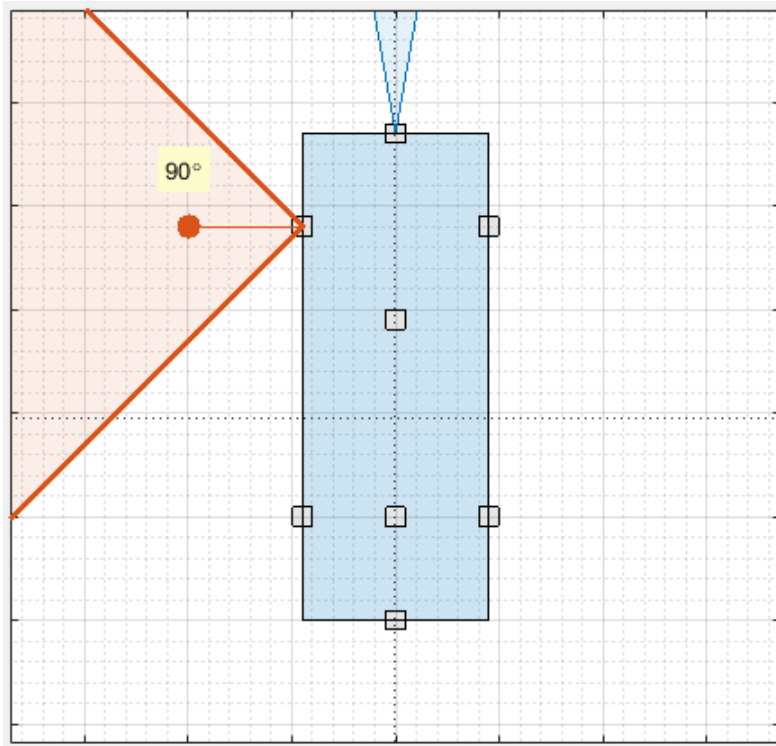
#### Add Radar

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.

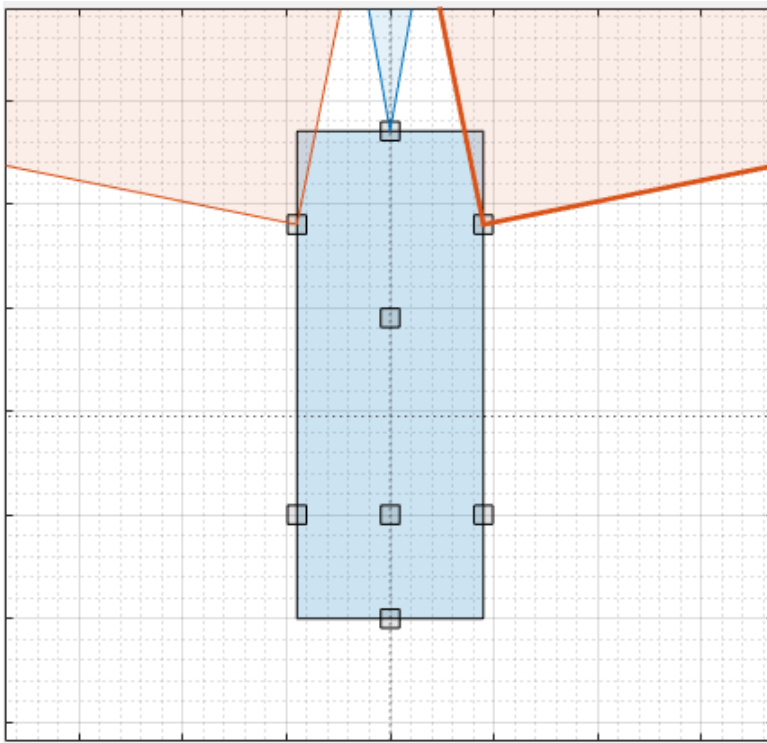
Tilt the radar sensor toward the front of the car. Move your cursor over the coverage area, then click and drag the angle marking.







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.

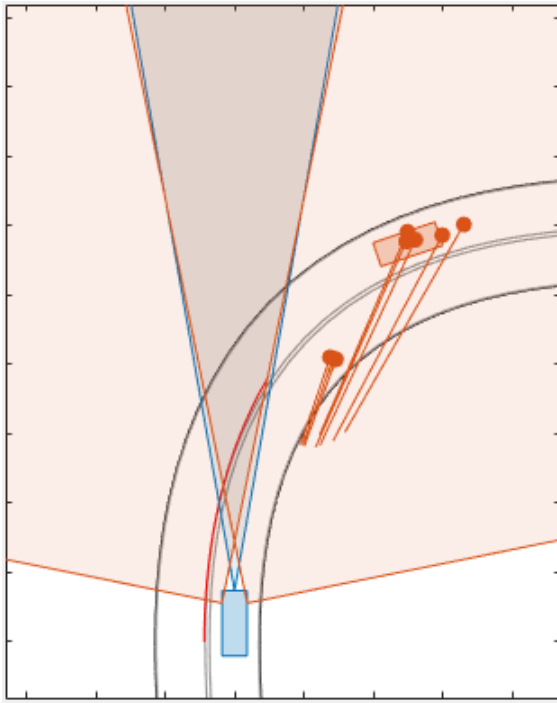



The camera and radar sensors now provide overlapping coverage of the front of the ego car.

## Generate Sensor 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 car. 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 have the scenario run for a set time instead, from the app toolbar, click **Settings** and change the stop condition.

### Export Sensor Detections

To export the detections to the MATLAB workspace, from the app toolbar, click **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, click **Export** > **Export MATLAB Function**. The scenario is a `drivingScenario` object. The sensor detections are generated by `visionDetectionGenerator` and `radarDetectionGenerator` System objects. To adjust the parameters of the scenario,

you can update the code in the exported function directly. To generate new detections, call the exported function.

## Save Session

After you generate the detections, click **Save** to save the app session to a MAT-file. In addition, you can save the sensor models separately. You can also save the road and actor models into a separate file.

You can reopen this session from within the app or by using this syntax at the MATLAB command prompt:

```
drivingScenarioDesigner(sessionFileName)
```

## See Also

### Apps

**Driving Scenario Designer**

### Classes

`drivingScenario`

### System Objects

`radarDetectionGenerator` | `visionDetectionGenerator`

## More About

- “Generate Synthetic Detections from a Prebuilt Driving Scenario” on page 4-18
- “Generate Synthetic Detections from a Euro NCAP Scenario” on page 4-40
- “Add OpenDRIVE Roads to Driving Scenario” on page 4-60

# Generate Synthetic Detections from a Prebuilt Driving Scenario

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. You can generate synthetic vision and radar detections from these prebuilt scenarios. You can then use these detections to test your vehicle controllers or sensor fusion algorithms.

## 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.

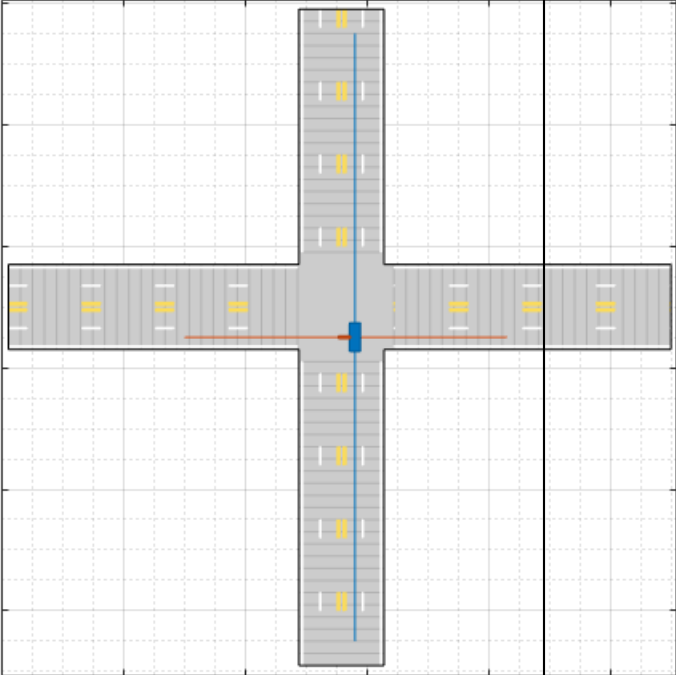
- “Euro NCAP” on page 4-18
- “Intersections” on page 4-18
- “Turns” on page 4-23
- “U-Turns” on page 4-31

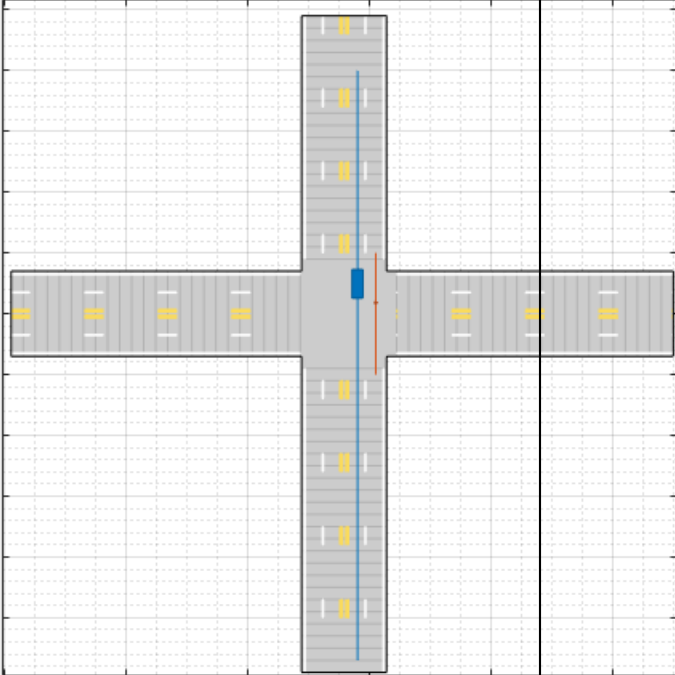
### Euro NCAP

These scenarios represent Euro NCAP test protocols. The app includes scenarios for testing automatic emergency braking, emergency lane keeping, and lane keep assist systems. For more details, see “Generate Synthetic Detections from a Euro NCAP Scenario” on page 4-40.

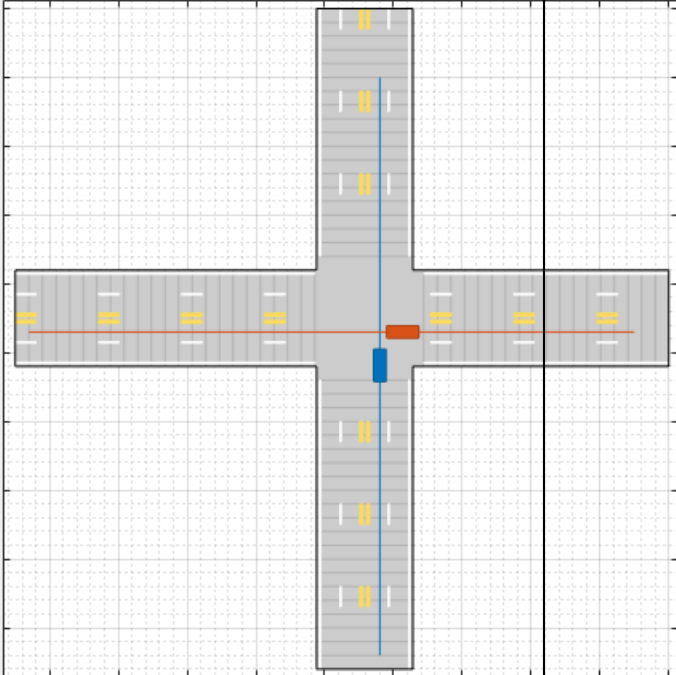
### Intersections

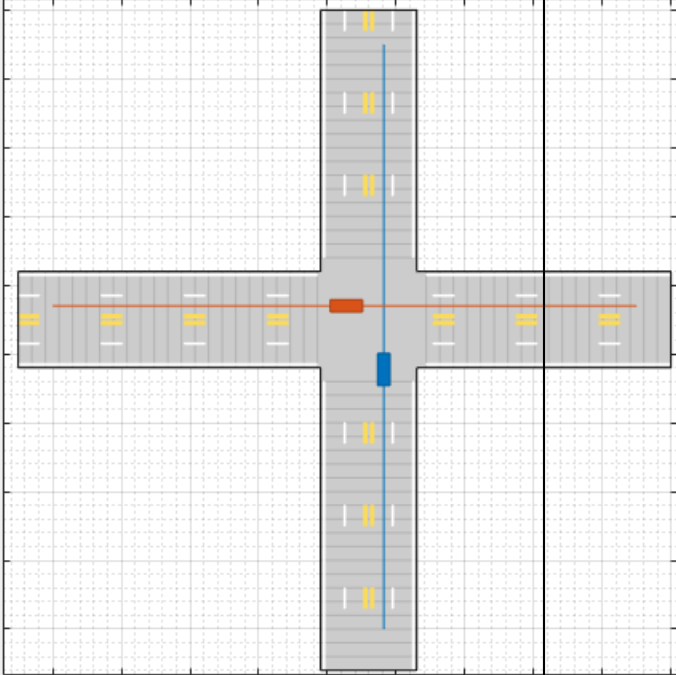
These scenarios involve common traffic patterns at four-way intersections and roundabouts.

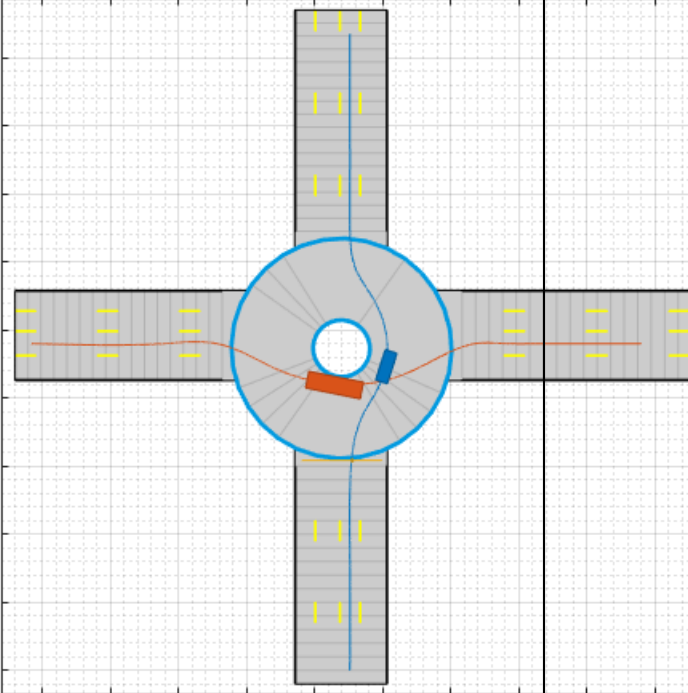
File Name	Description
<p>EgoCarGoesStraight_BicycleFromLeftGoesStraight_Collision.mat</p>	<p>The ego car travels north and goes straight through an intersection. A bicycle coming from the left side of the intersection goes straight and collides with the ego vehicle.</p> 

File Name	Description
<p>EgoCarGoesStraight_PedestrianToRightGoesStraight.mat</p>	<p>The ego car travels north and goes straight through an intersection. A pedestrian in the lane to the right of the ego car also travels north and goes straight through the intersection. The pedestrian travels at a slower pace than the ego car.</p> 



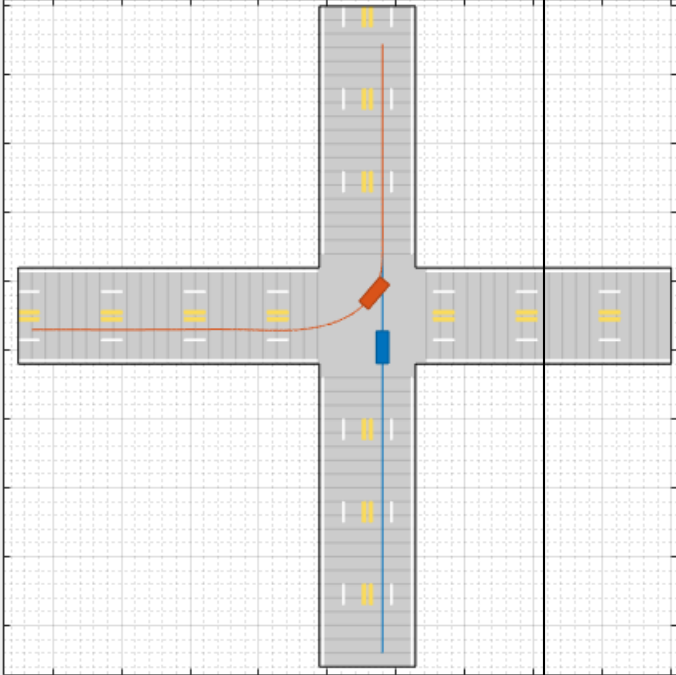
File Name	Description
<p>EgoCarGoesStraight_VehicleFromLeftGoesStraight.mat</p>	<p>The ego car travels north and goes straight through an intersection. A vehicle coming from the left side of the intersection also goes straight and crosses through the intersection first.</p> 

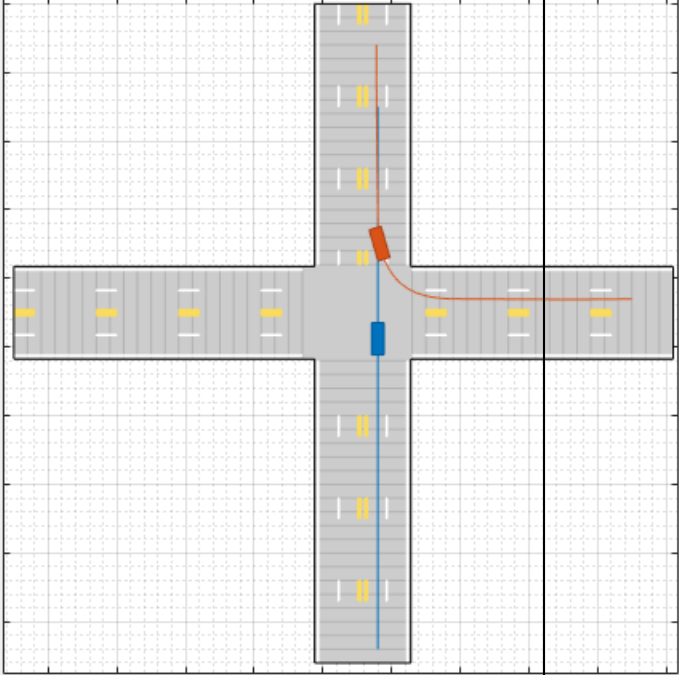
File Name	Description
<p>EgoCarGoesStraight_VehicleFromRightGoesStraight.mat</p>	<p>The ego car travels north and goes straight through an intersection. A vehicle coming from the right side of the intersection also goes straight and crosses through the intersection first.</p> 

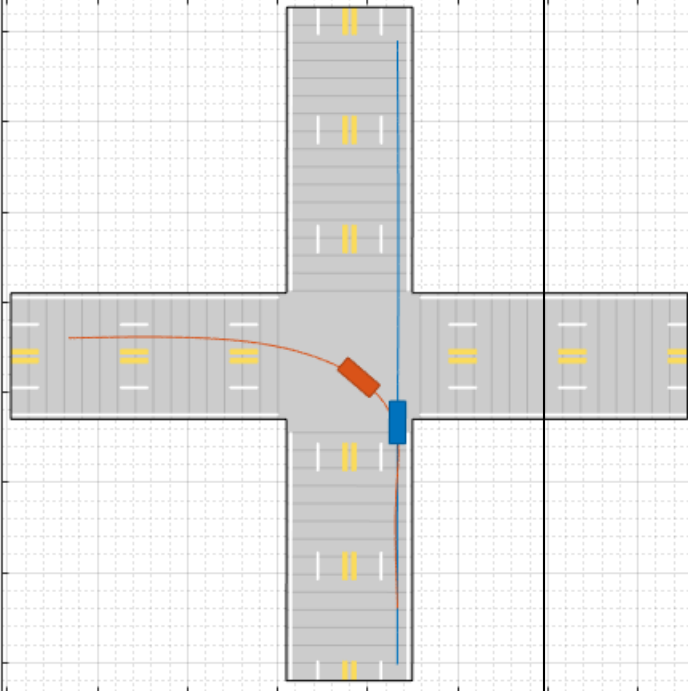
File Name	Description
Roundabout .mat	<p>The ego car travels north and crosses the path of a pedestrian while entering a roundabout. The ego car then passes a truck as both vehicles drive through the roundabout.</p> 

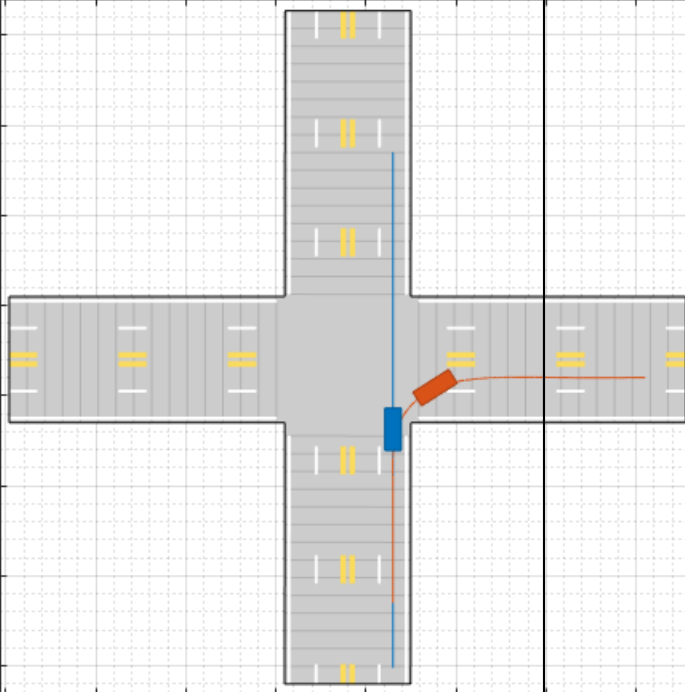
### Turns

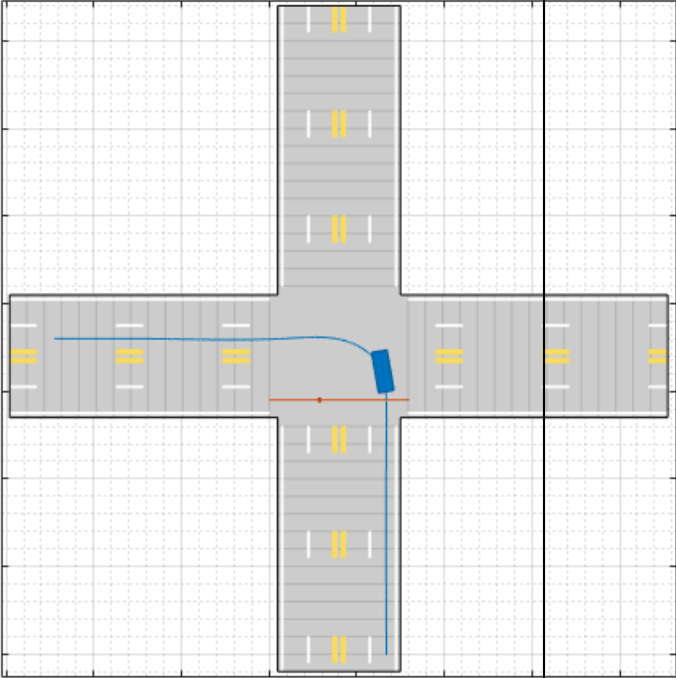
These scenarios involve turns at four-way intersections.

File Name	Description
EgoCarGoesStraight_VehicleFromLeftTurnsLeft.mat	<p data-bbox="795 305 1329 430">The ego car 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 car.</p> 

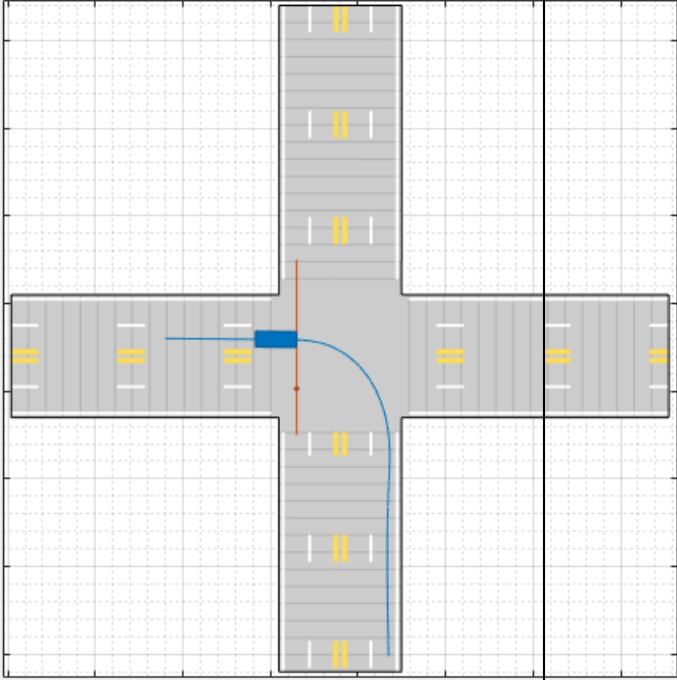
File Name	Description
<p>EgoCarGoesStraight_VehicleFromRightTurnsRight.mat</p>	<p>The ego car travels north and goes straight through an intersection. A vehicle coming from the right side of the intersection turns right and ends up in front of the ego car.</p> 

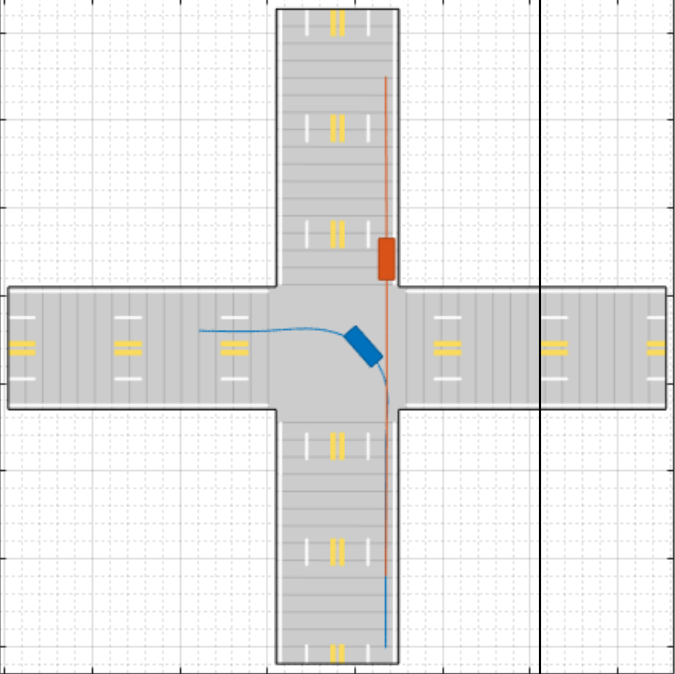
File Name	Description
EgoCarGoesStraight_VehicleInFrontTurnsLeft.mat	<p>The ego car travels north and goes straight through an intersection. A vehicle in front of the ego car turns left at the intersection.</p> 

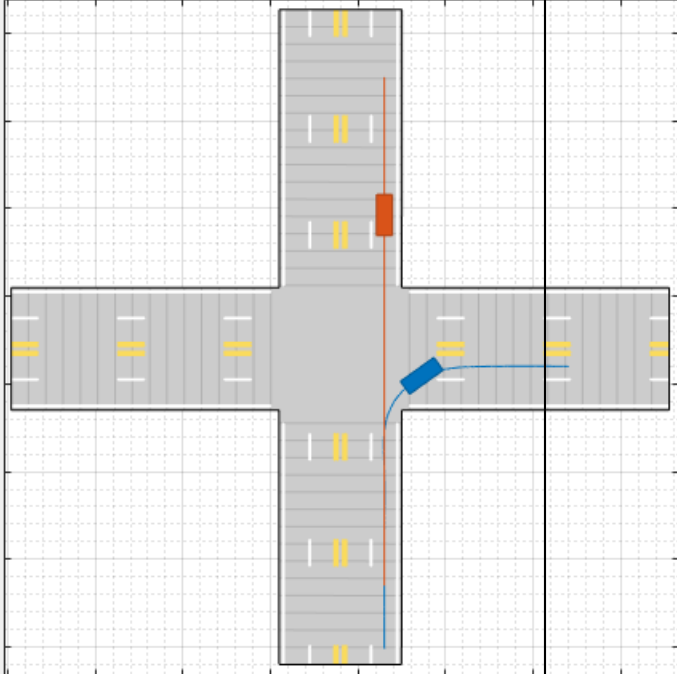
File Name	Description
<p>EgoCarGoesStraight_VehicleInFrontTurnsRight.mat</p>	<p>The ego car travels north and goes straight through an intersection. A vehicle in front of the ego car turns right at the intersection.</p> 

File Name	Description
EgoCarTurnsLeft_PedestrianFromLeftGoesStraight.mat	<p>The ego car travels north and turns left at an intersection. A pedestrian coming from the left side of the intersection goes straight. The ego car completes its turn before the pedestrian crosses the intersection.</p> 



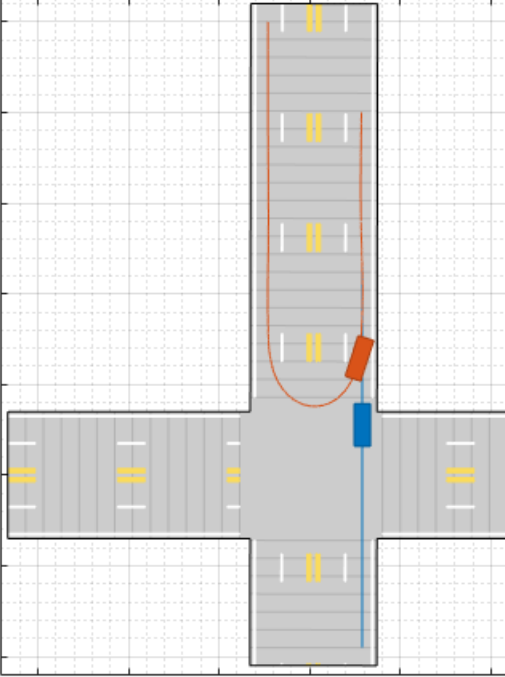
File Name	Description
<p>EgoCarTurnsLeft_PedestrianInOppLaneGoesStraight.mat</p>	<p>The ego car travels north and turns left at an intersection. A pedestrian in the opposite lane goes straight through the intersection. The ego car completes its turn before the pedestrian crosses the intersection.</p> 

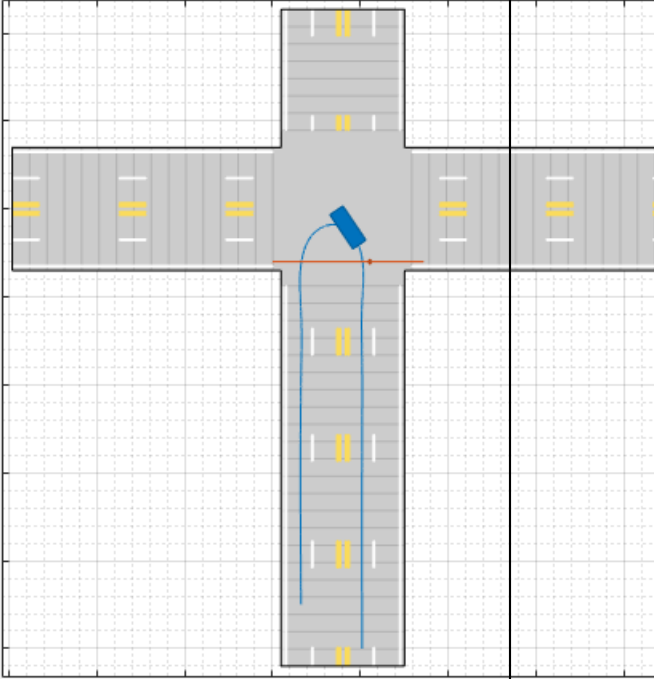
File Name	Description
EgoCarTurnsLeft_VehicleInFrontGoesStraight.mat	<p>The ego car travels north and turns left at an intersection. A vehicle in front of the ego car goes straight through the intersection.</p> 

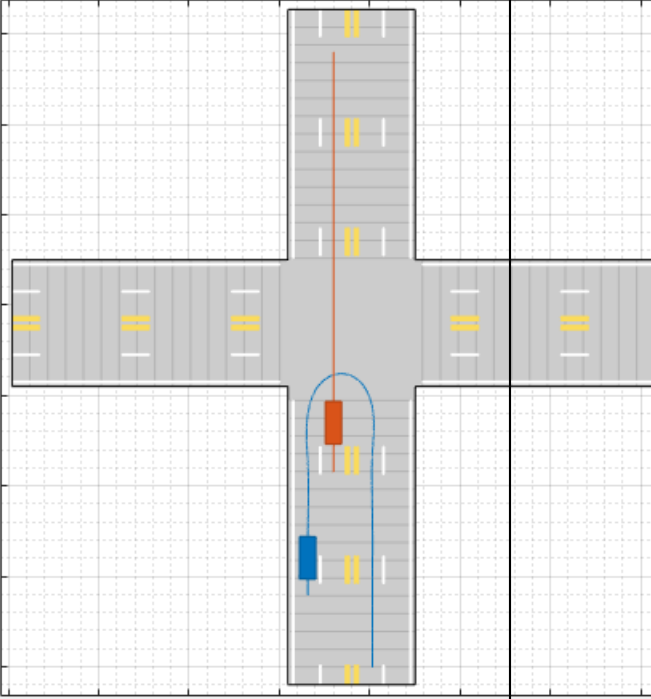
File Name	Description
EgoCarTurnsRight_VehicleInFrontGoesStraight.mat	<p>The ego car travels north and turns right at an intersection. A vehicle in front of the ego car goes straight through the intersection.</p> 

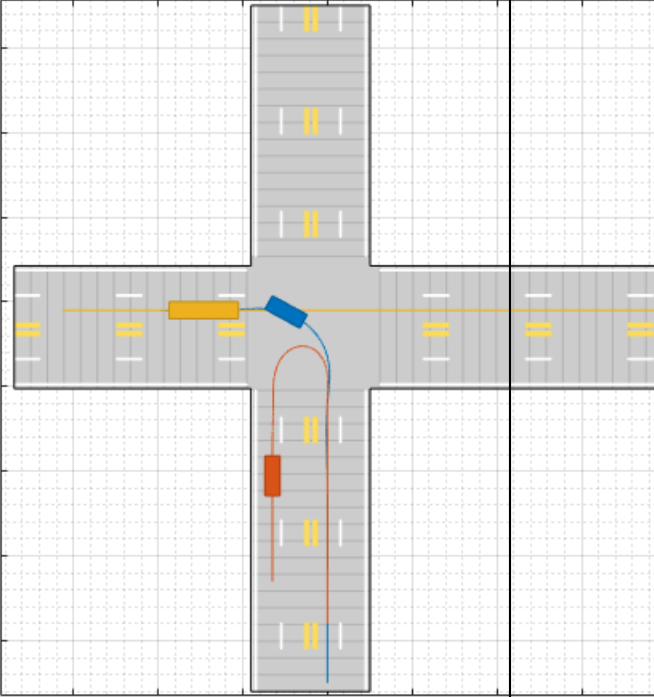
### U-Turns

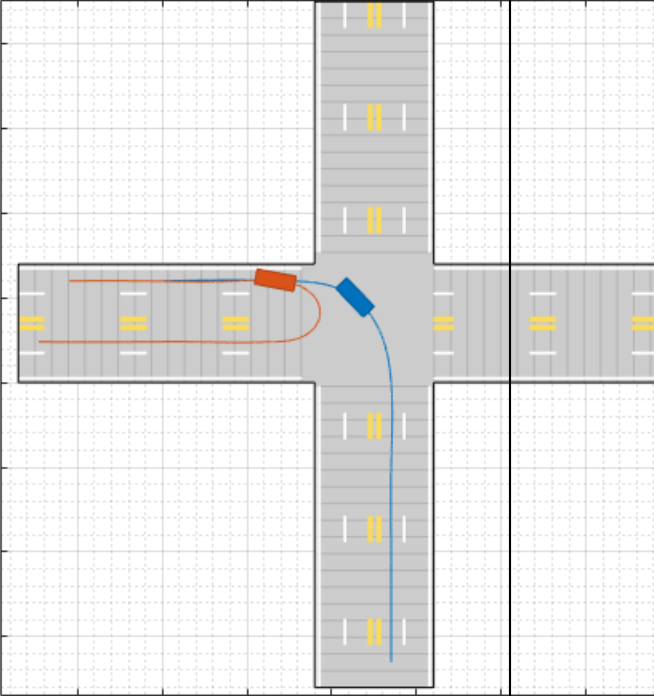
These scenarios involve U-turns at four-way intersections.

File Name	Description
EgoCarGoesStraight_VehicleInOppLaneMakesUTurn.mat	<p>The ego car travels north and goes straight through an intersection. A vehicle in the opposite lane makes a U-turn. The ego car ends up behind the vehicle.</p>  <p>The diagram illustrates a driving scenario at a four-way intersection. The ego car, represented by a blue rectangle, is moving north through the intersection. A vehicle in the opposite lane, represented by an orange rectangle, is making a U-turn from the southbound lane into the northbound lane. The ego car ends up behind the vehicle. The intersection is shown with a grid background and lane markings. The ego car's path is indicated by a blue line, and the U-turning vehicle's path is indicated by an orange line.</p>

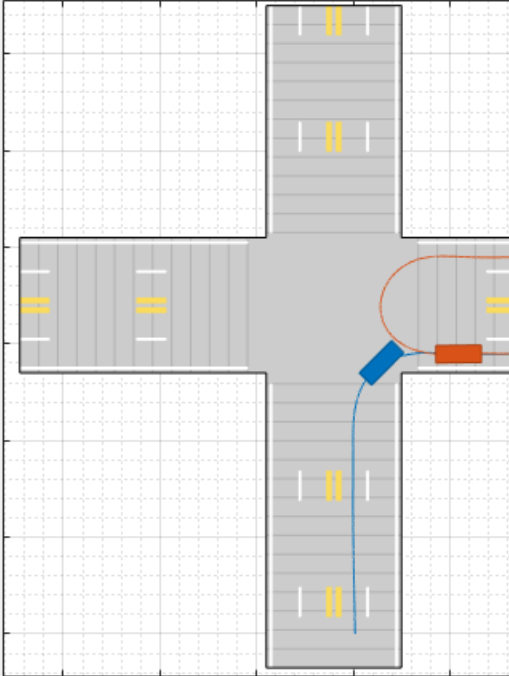
File Name	Description
EgoCarMakesUTurn_PedestrianFromRightGoesStraight.mat	<p data-bbox="825 305 1328 461">The ego car travels north and makes a U-turn at an intersection. A pedestrian coming from the right side of the intersection goes straight and crosses the path of the U-turn.</p>  <p>The diagram illustrates a driving scenario at a four-way intersection. A blue car is shown in the center of the intersection, performing a U-turn from the northbound lane to the southbound lane. A red pedestrian is shown walking straight across the intersection from the eastbound side to the westbound side. The road is marked with lane lines and yellow dashed lines. The background is a grid.</p>

File Name	Description
EgoCarMakesUTurn_VehicleInOppLaneGoesStraight.mat	<p>The ego car travels north and makes a U-turn at an intersection. A vehicle traveling south in the opposite lane goes straight and crosses the path of the U-turn.</p> 

File Name	Description
<p>EgoCarTurnsLeft_Vehicle1MakesUTurn_Vehicle2GoesStraight.mat</p>	<p>The ego car travels north and turns left at an intersection. A vehicle in front of the ego car makes a U-turn at the intersection. A second vehicle, a truck, comes from the right side of the intersection and goes in front of the ego car.</p> 

File Name	Description
EgoCarTurnsLeft_VehicleFromLeftMake sUTurn.mat	<p data-bbox="826 298 1329 456">The ego car travels north and turns left at an intersection. A vehicle coming from the left side of the intersection makes a U-turn. The ego car ends up behind the vehicle.</p> 



File Name	Description
EgoCarTurnsRight_VehicleFromRightMakesUTurn.mat	<p>The ego car travels north and turns right at an intersection. A vehicle coming from the right side of the intersection makes a U-turn. The ego car ends up behind the vehicle.</p> 

## Modify Scenario

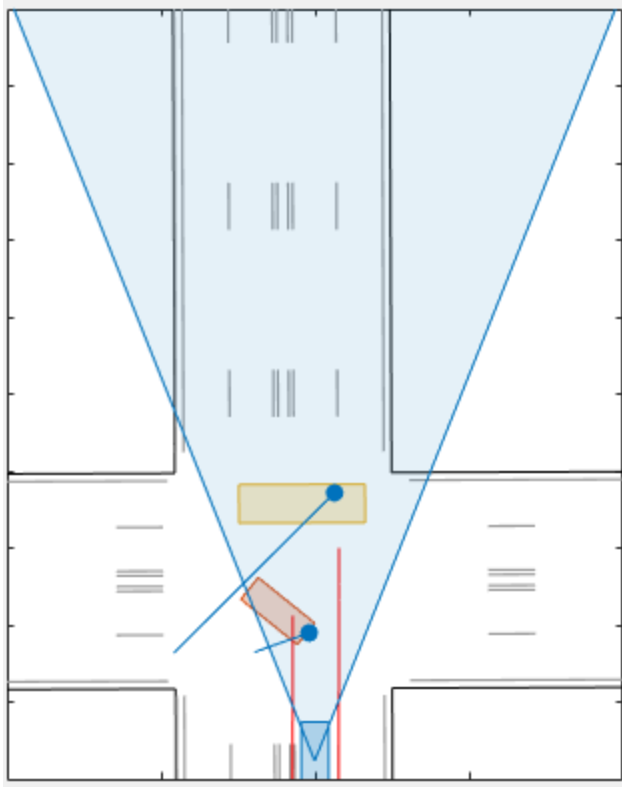
After you choose a scenario, you can modify the parameters of the roads and actors. For example, from the **Actors** tab on the left, you can change the position or velocity of the ego car or other actors. From the **Roads** tab, you can change the width of the lanes or the type of lane markings.

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



Export the detections.

- To export the detections to the MATLAB workspace, from the app toolstrip, click **Export > Export Sensor Data**. Name the workspace variable and click **OK**.

- To export a MATLAB function that generates the scenario and its detections, click **Export > Export MATLAB Function**. The scenario is a `drivingScenario` object. The sensor detections are generated by `visionDetectionGenerator` and `radarDetectionGenerator` System objects. To adjust the parameters of the scenario, you can update the code in the exported function directly. To generate new detections, call the exported function.

## Save Scenario

Because prebuilt scenarios are read-only, save a copy of the driving scenario to a new folder. From the app toolstrip, select **Save > Session As** to save the app session to a MAT-file.

You can reopen this session from within the app or by using this syntax at the MATLAB command prompt:

```
drivingScenarioDesigner(sessionFileName)
```

## See Also

### Apps

**Driving Scenario Designer**

### Classes

`drivingScenario`

### System Objects

`radarDetectionGenerator` | `visionDetectionGenerator`

## More About

- “Build a Driving Scenario and Generate Synthetic Detections” on page 4-2
- “Generate Synthetic Detections from a Euro NCAP Scenario” on page 4-40

# Generate Synthetic Detections from a Euro NCAP Scenario

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 automatic 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, from the app toolstrip, select **Open > Prebuilt Scenario**. The `PrebuiltScenarios` folder opens, which includes subfolders for all prebuilt scenarios available in the app (see also “Generate Synthetic Detections from a Prebuilt Driving Scenario” on page 4-18).

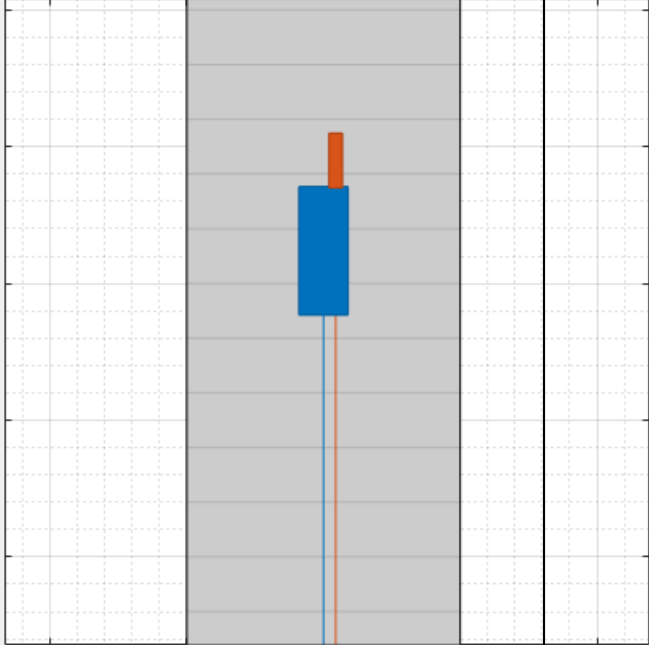
Double-click the **EuroNCAP** folder, and then choose a Euro NCAP scenario from one of these subfolders.

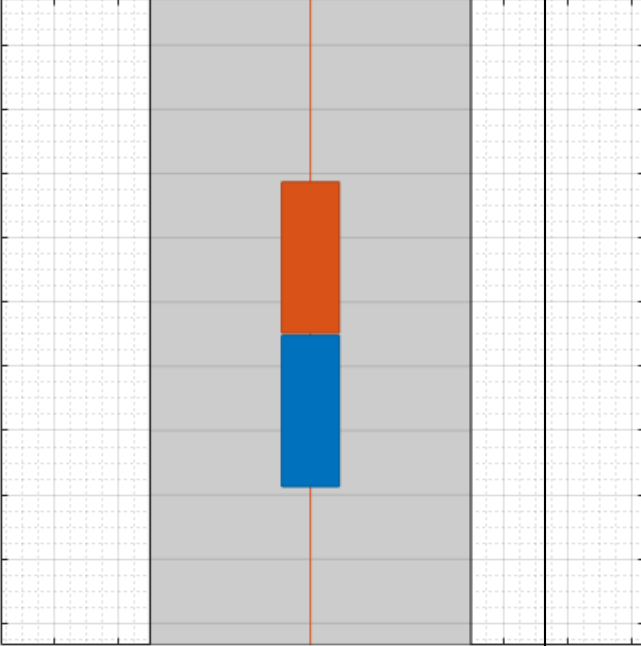
- “Automatic Emergency Braking” on page 4-40
- “Emergency Lane Keeping” on page 4-46
- “Lane Keep Assist” on page 4-50

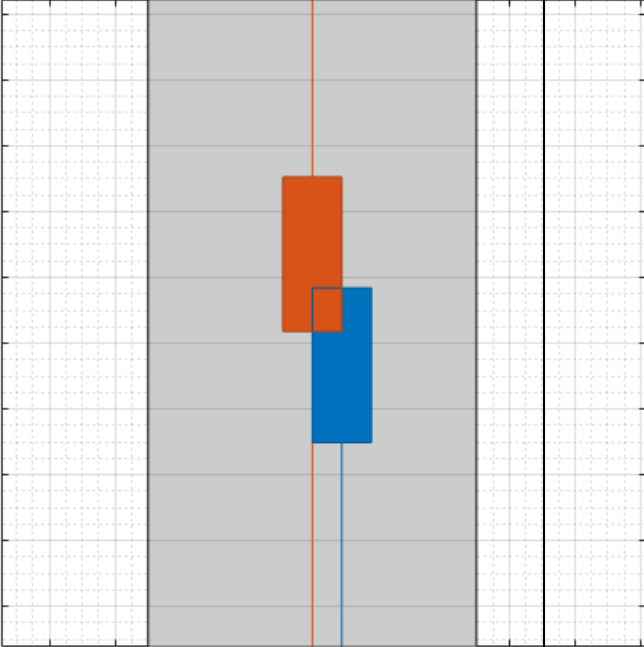
## Automatic Emergency Braking

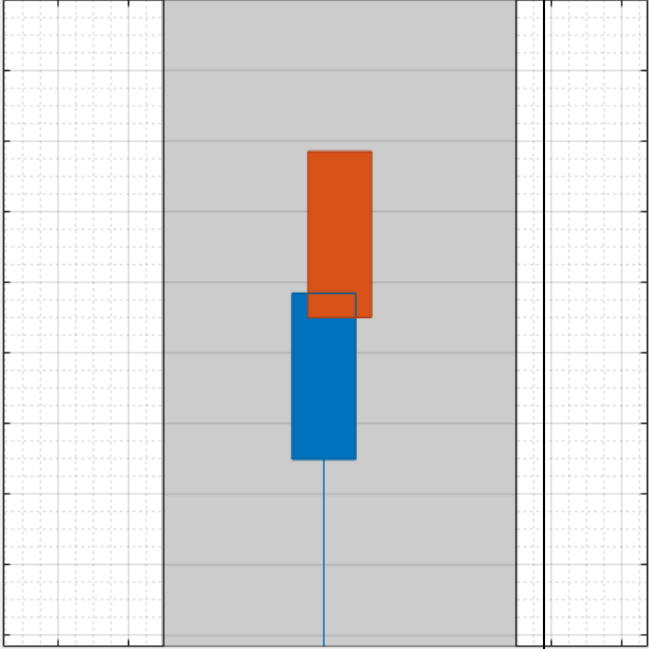
These scenarios are designed to test automatic emergency braking (AEB) systems. AEB systems warn drivers of impending collisions and automatically apply brakes to prevent collisions or reduce the impact of collisions. Some AEB systems prepare the vehicle and restraint systems for impact.

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.

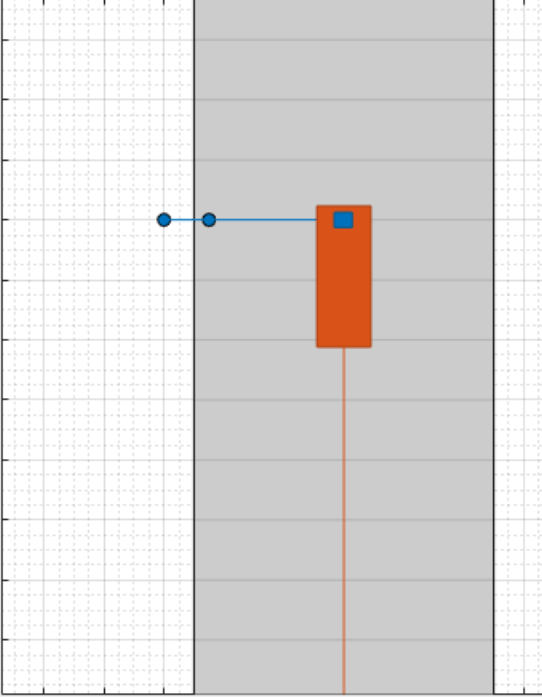
File Name	Description
<p>AEB_Bicyclist_Longitudinal_25width h.mat</p>	<p>The ego car collides with the bicyclist that is in front of it. Before the collision, the bicyclist and ego car are traveling in the same direction along the longitudinal axis. At collision time, the bicycle is 25% of the way across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

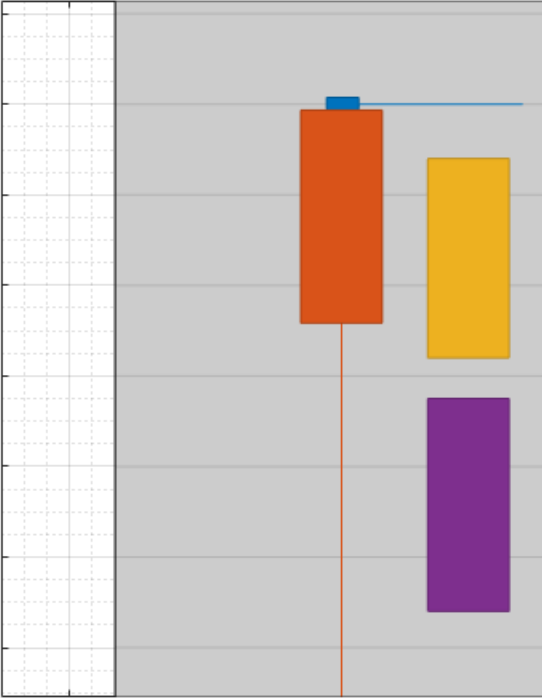
File Name	Description
<p>AEB_CCRb_2_initialGap_12m.mat</p>	<p>A car-to-car rear braking (CCRb) scenario, where the ego car rear-ends a braking vehicle. The braking vehicle begins to decelerate at <math>2 \text{ m/s}^2</math>. The initial gap between the ego car and the braking vehicle is 12 m.</p> 

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

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



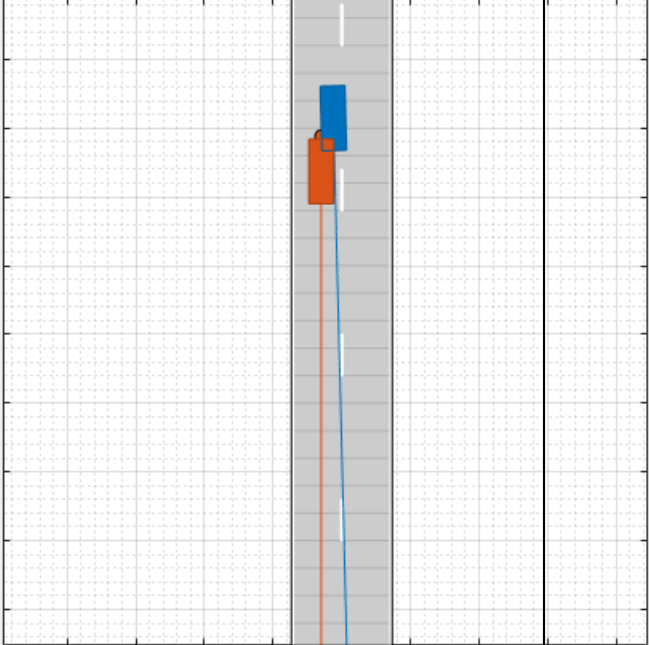
File Name	Description
<p>AEB_Pedestrian_Farside_50width.mat</p>	<p>The ego car collides with a pedestrian who is traveling from the left side of the road, which Euro NCAP test protocols refer to as the far side. These protocols assume that vehicles travel on the right side of the road. Therefore, the left side of the road is the side farthest from the ego car. At collision time, the pedestrian is 50% of the way across the width of the ego car.</p>  <p>The diagram shows a top-down view of a collision scenario on a grid. An orange rectangle represents the ego car, positioned in the center of the road. A blue dot represents the pedestrian, positioned to the left of the car. A horizontal line connects the pedestrian to the left side of the car, indicating the pedestrian's position relative to the car's width. The pedestrian is positioned at the 50% mark across the width of the ego car.</p>

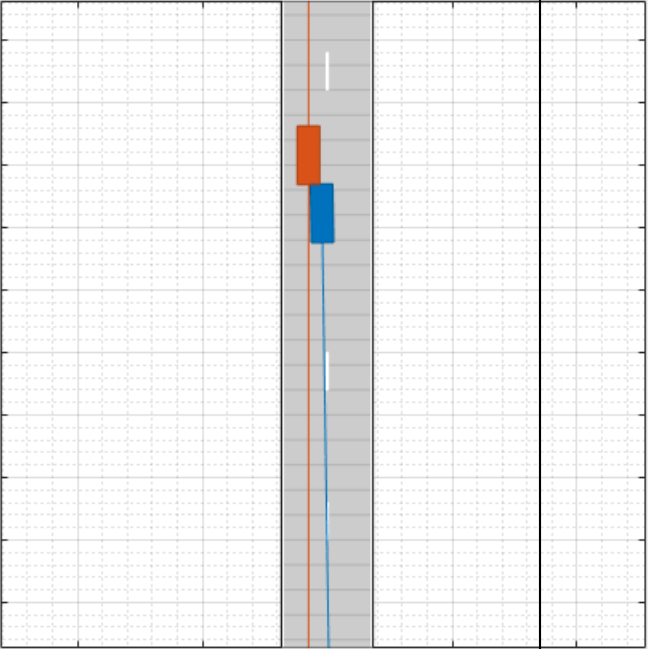
File Name	Description
AEB_PedestrianChild_Nearside_50width.mat	<p>The ego car collides with a pedestrian who is traveling from the right side of the road, which Euro NCAP test protocols refer to as the near side. These protocols assume that vehicles travel on the right side of the road. Therefore, the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

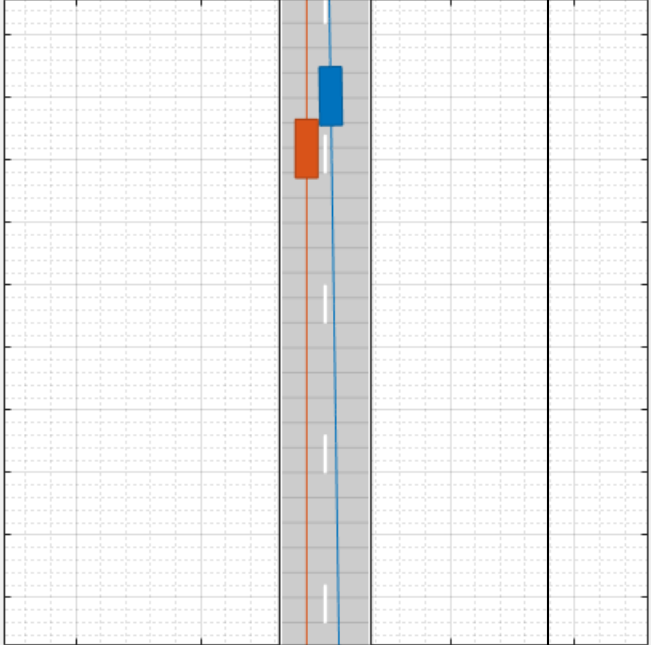
### Emergency Lane Keeping

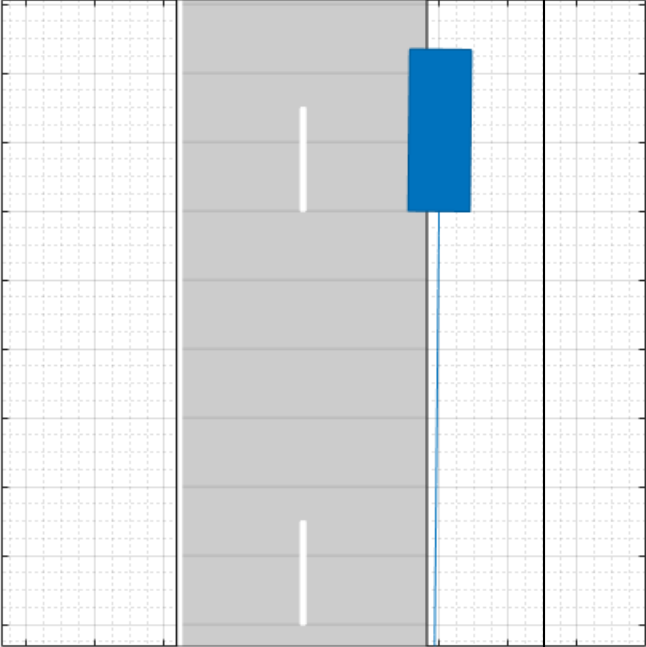
These scenarios are designed to test emergency lane keeping (ELK) systems. ELK systems prevent collisions by warning drivers of impending, unintentional lane departures.

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 car intentionally changes lanes and collides with a faster, overtaking vehicle that is in the other lane. The ego car travels at a lateral velocity of 0.5 m/s.  The diagram illustrates a two-lane road with a central dashed line and solid outer lines. A blue rectangular vehicle (the ego car) is positioned in the right lane, moving downwards. An orange rectangular vehicle (the overtaking vehicle) is positioned in the left lane, also moving downwards but slightly ahead of the blue car. A vertical line extends from the front of the orange car towards the blue car, indicating a collision point. The background is a light gray grid.

File Name	Description
ELK_OncomingVeh_Vlat_0.3.mat	<p>The ego car unintentionally changes lanes and collides with an oncoming vehicle that is in the other lane. The ego car travels at a lateral velocity of 0.3 m/s.</p>  <p>The diagram illustrates a collision scenario on a road. A vertical orange line represents the lane boundary. An orange rectangular vehicle is positioned in the left lane, moving towards the right. A blue rectangular vehicle is positioned in the right lane, moving towards the left. The two vehicles are shown overlapping, indicating a collision. The background is a light gray grid.</p>

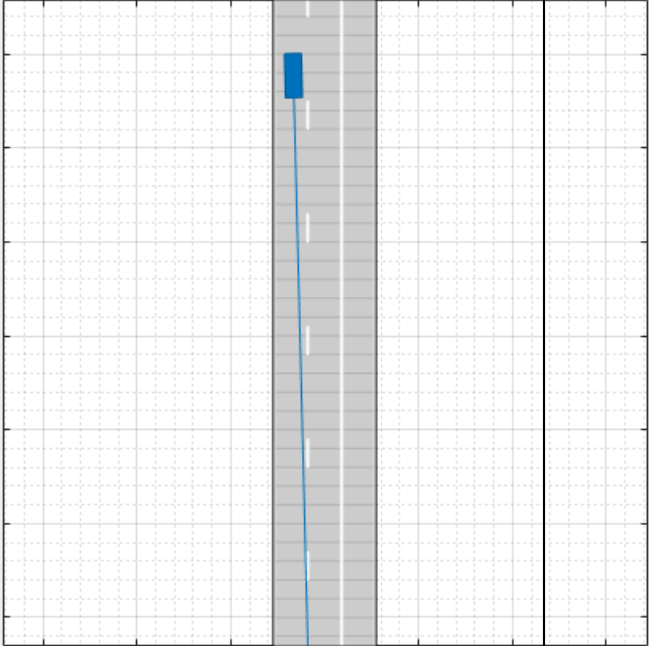
File Name	Description
<p>ELK_OvertakingVeh_Unintent_Vlat_0 .3.mat</p>	<p>The ego car unintentionally changes lanes, overtakes a vehicle in the other lane, and collides with that vehicle. The ego car travels at a lateral velocity of 0.3 m/s.</p> 

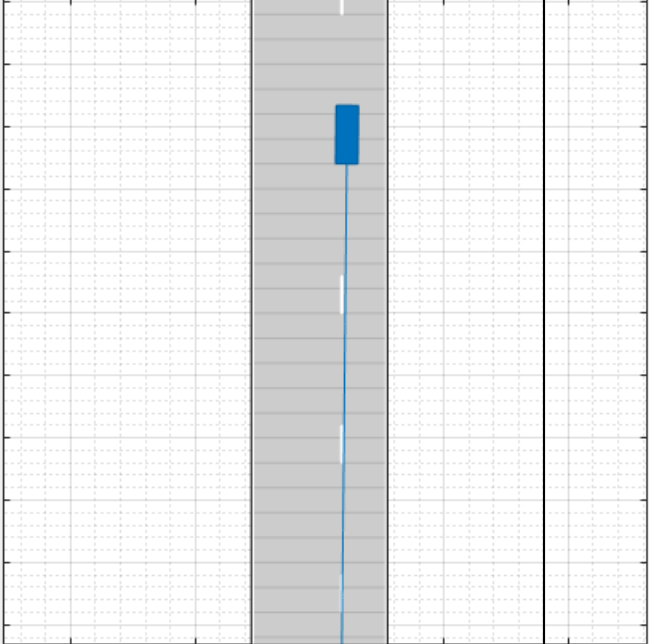
File Name	Description
ELK_RoadEdge_NoBndry_Vlat_0.2.mat	<p>The ego car unintentionally changes lanes and ends up on the road edge. The road edge has no lane boundary markings. The ego car travels at a lateral velocity of 0.2 m/s.</p> 

**Lane Keep Assist**

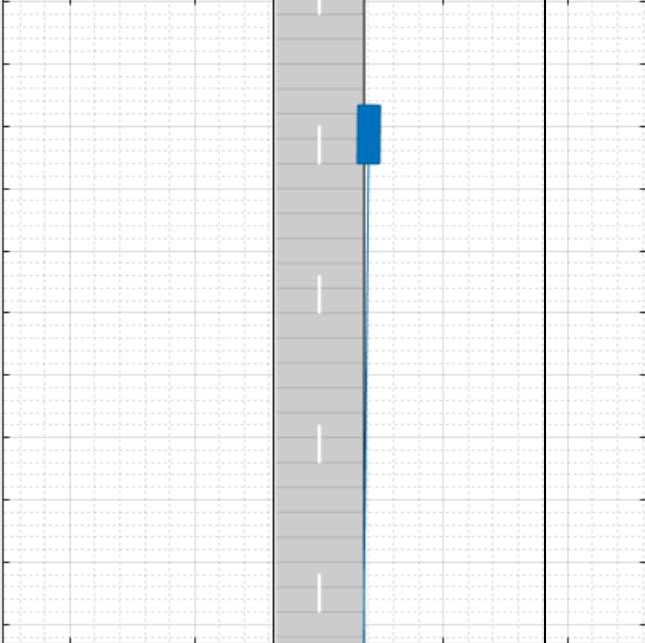
These scenarios are designed to test lane keep assist (LKA) systems. LKA systems detect unintentional lane departures and automatically adjust the steering angle of the vehicle to stay within the lane boundaries.

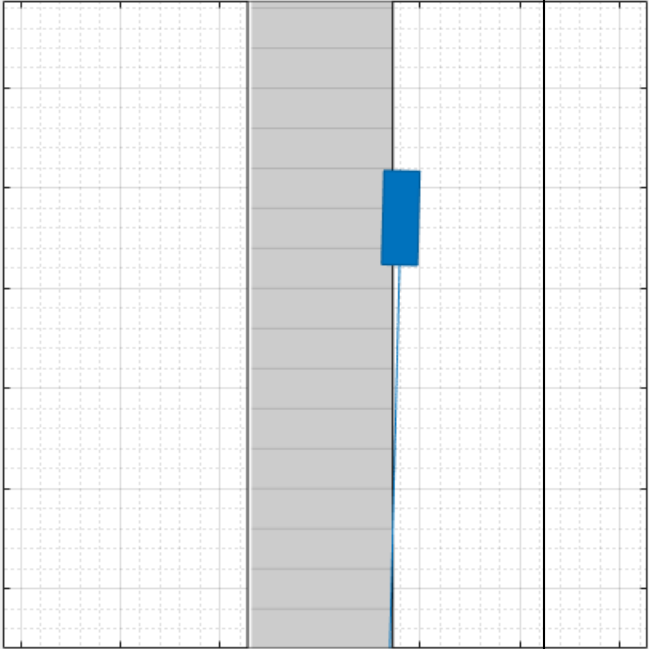
The table lists a subset of the available LKA scenarios. Other LKA scenarios in the folder vary the lateral velocity of the ego vehicle and the lane marking types.

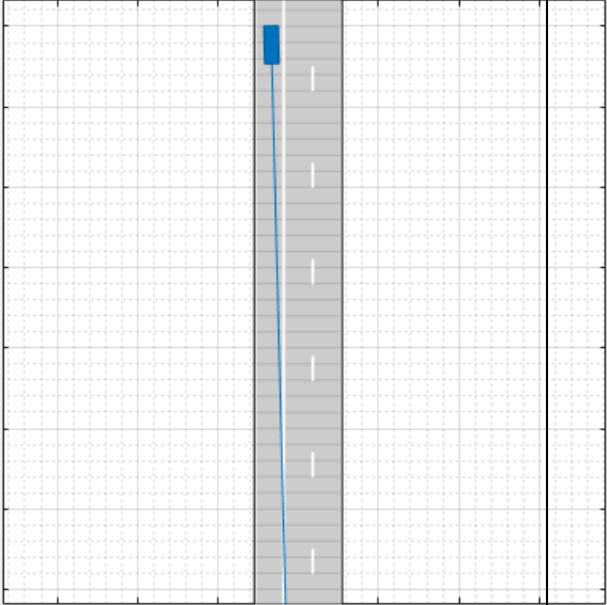
File Name	Description
<p>LKA_DashedLine_Solid_Left_Vlat_0.5.mat</p>	<p>The ego car unintentionally departs from a lane that is dashed on the left and solid on the right. The car departs the lane from the left (dashed) side, traveling at a lateral velocity of 0.5 m/s.</p> 

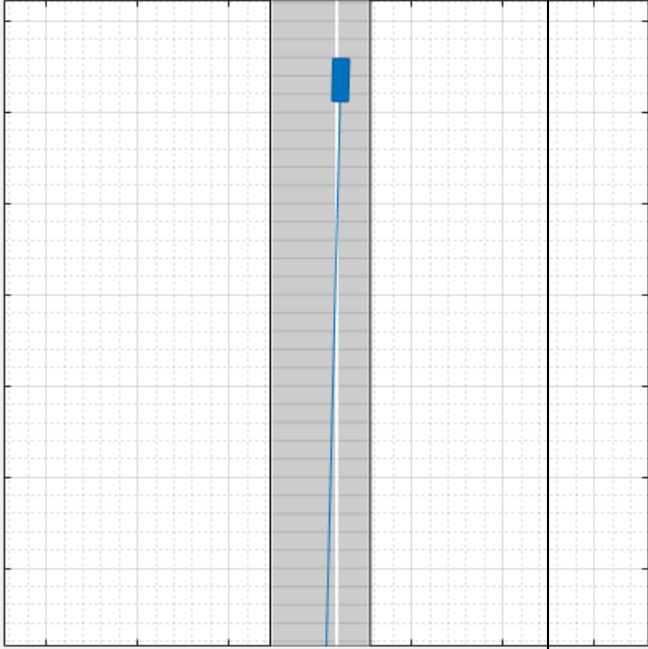
File Name	Description
LKA_DashedLine_Unmarked_Right_Vla t_0.5.mat	<p data-bbox="795 305 1334 461">The ego car unintentionally departs from a lane that is dashed on the right and unmarked on the left. The car departs the lane from the right (dashed) side, traveling at a lateral velocity of 0.5 m/s.</p> 



File Name	Description
<p>LKA_RoadEdge_NoBndry_Vlat_0.5.mat</p>	<p>The ego car 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>The ego car 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 car 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 car 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

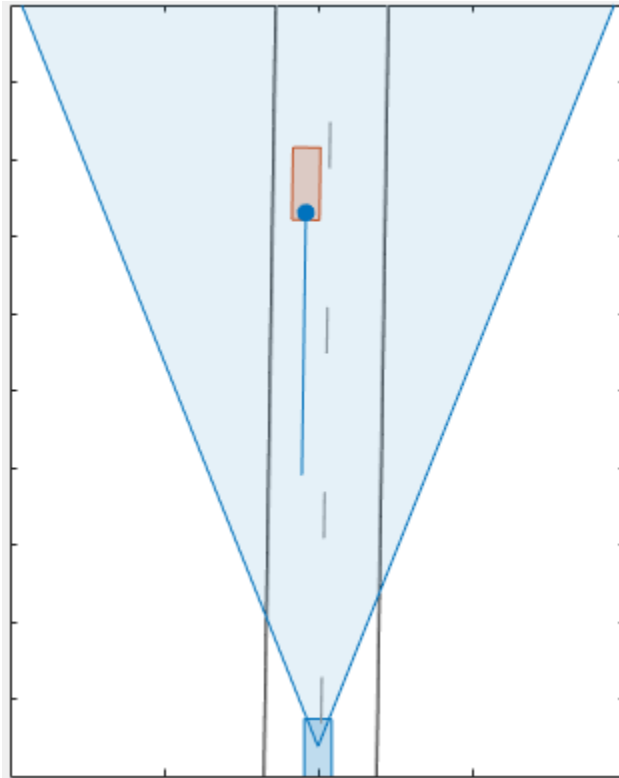
By default, in Euro NCAP scenarios, the ego car does not contain sensors. If you are testing a vehicle sensor, from the app toolstrip, click **Add Camera** or **Add Radar** to add a sensor to the ego car. Then, on the **Sensor** tab, adjust the parameters of the sensors to match your sensor model. If 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 car 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 car. The **Bird's-Eye Plot** displays the detections.



Export the detections.

- To export the detections to the MATLAB workspace, from the app toolstrip, click **Export > Export Sensor Data**. Name the workspace variable and click **OK**.
- To export a MATLAB function that generates the scenario and its detections, click **Export > Export MATLAB Function**. The scenario is a `drivingScenario` object.

The sensor detections are generated by `visionDetectionGenerator` and `radarDetectionGenerator` System objects. To adjust the parameters of the scenario, you can update the code in the exported function directly. To generate new detections, call the exported function.

### Save Scenario

Because Euro NCAP scenarios are read-only, save a copy of the driving scenario to a new folder. From the app toolstrip, select **Save > Session As** to save the app session to a MAT-file.

You can reopen this session from within the app or by using the following syntax at the MATLAB command prompt:

```
drivingScenarioDesigner(sessionFileName)
```

You can modify one or more scenario parameters and save multiple variations of the same scenario. For example, you can adjust the velocity of the ego vehicle or the type of lane markings on the road. Then you can save an altered version of the scenario.

You can now use the scenario and generated detections to test your driving algorithms. For an example, see “Automatic Emergency Braking with Sensor Fusion”.

### 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

#### Classes

`drivingScenario`

### **System Objects**

radarDetectionGenerator | visionDetectionGenerator

### **More About**

- “Build a Driving Scenario and Generate Synthetic Detections” on page 4-2
- “Generate Synthetic Detections from a Prebuilt Driving Scenario” on page 4-18
- “Automatic Emergency Braking with Sensor Fusion”

### **External Websites**

- Euro NCAP Safety Assist Protocols

# Add OpenDRIVE Roads to 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.

To import OpenDRIVE roads and lanes into a `drivingScenario` object instead of into the app, use the `roadNetwork` method.

## Import OpenDRIVE File

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

To import an OpenDRIVE file, from 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.

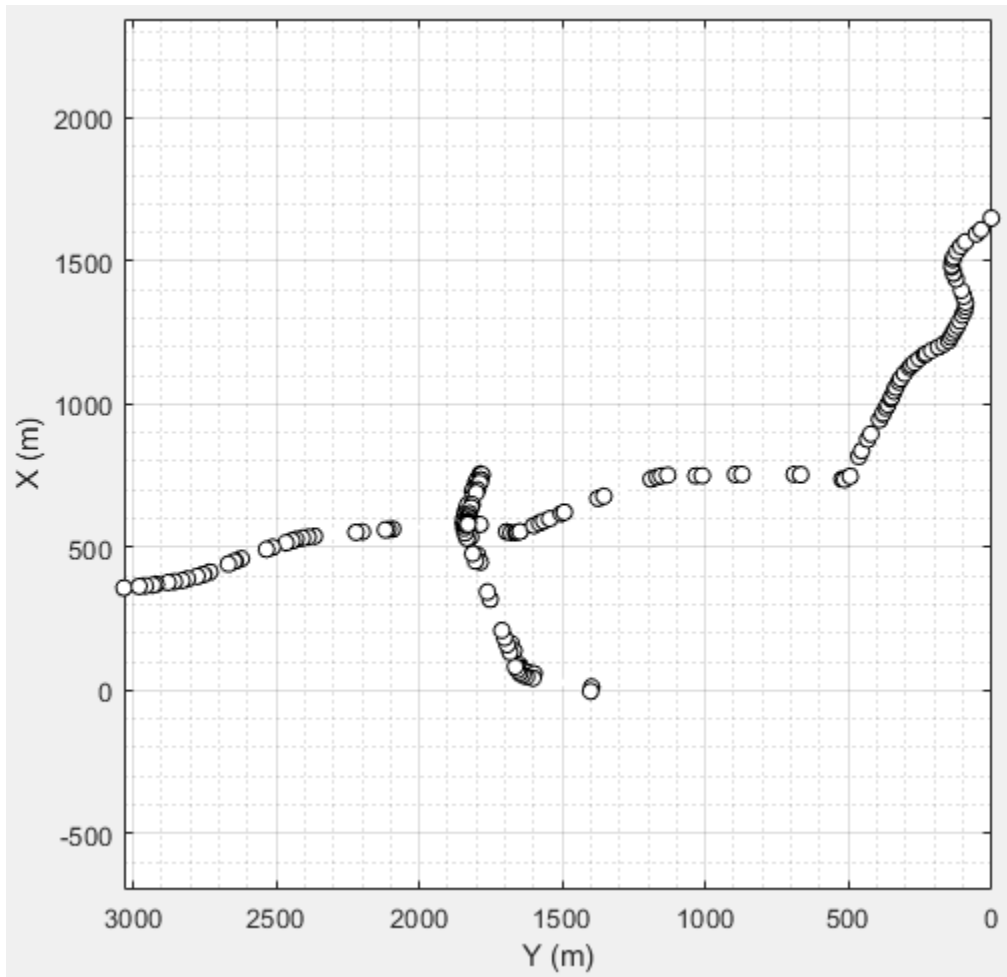
From your MATLAB root folder, navigate to and open this file:

```
matlabroot/toolbox/driving/drivingdata/intersection.xodr
```

Because you cannot import an OpenDRIVE road network into an existing app session, the app prompts you to save your current app session.

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

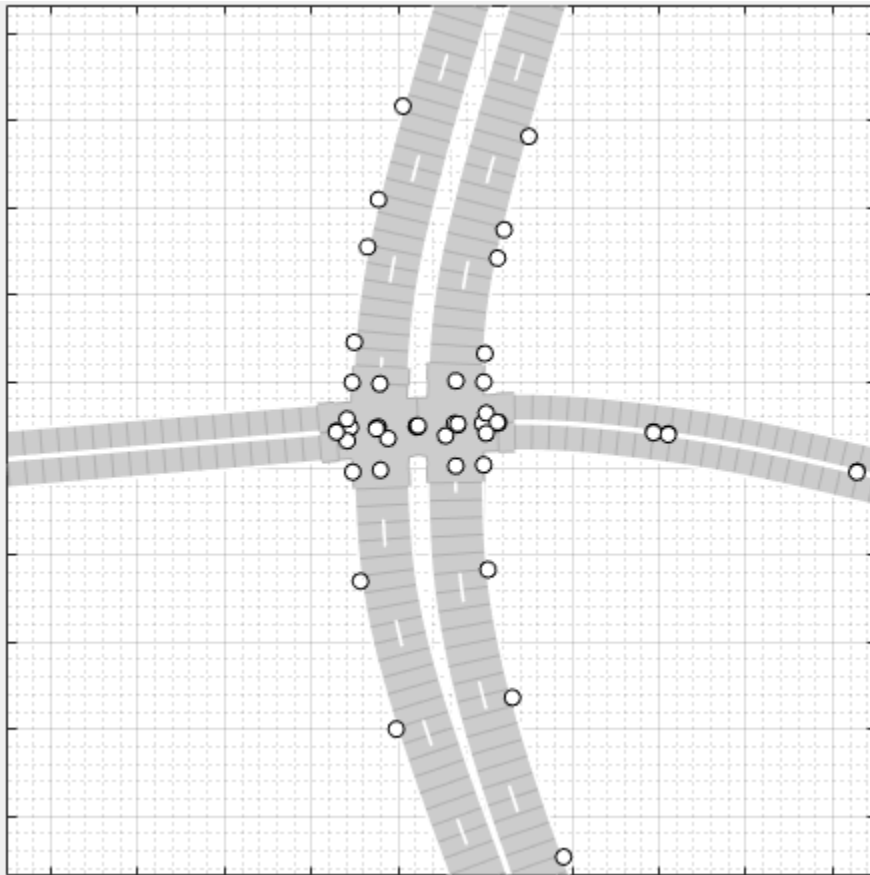




The roads in this network are thousands of meters long. You can zoom in (press **Ctrl + Plus**) on the road to inspect it more closely.

## Inspect Roads

The imported road network is an intersection of a two-lane undivided highway and a two-lane by two-lane divided highway.



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.

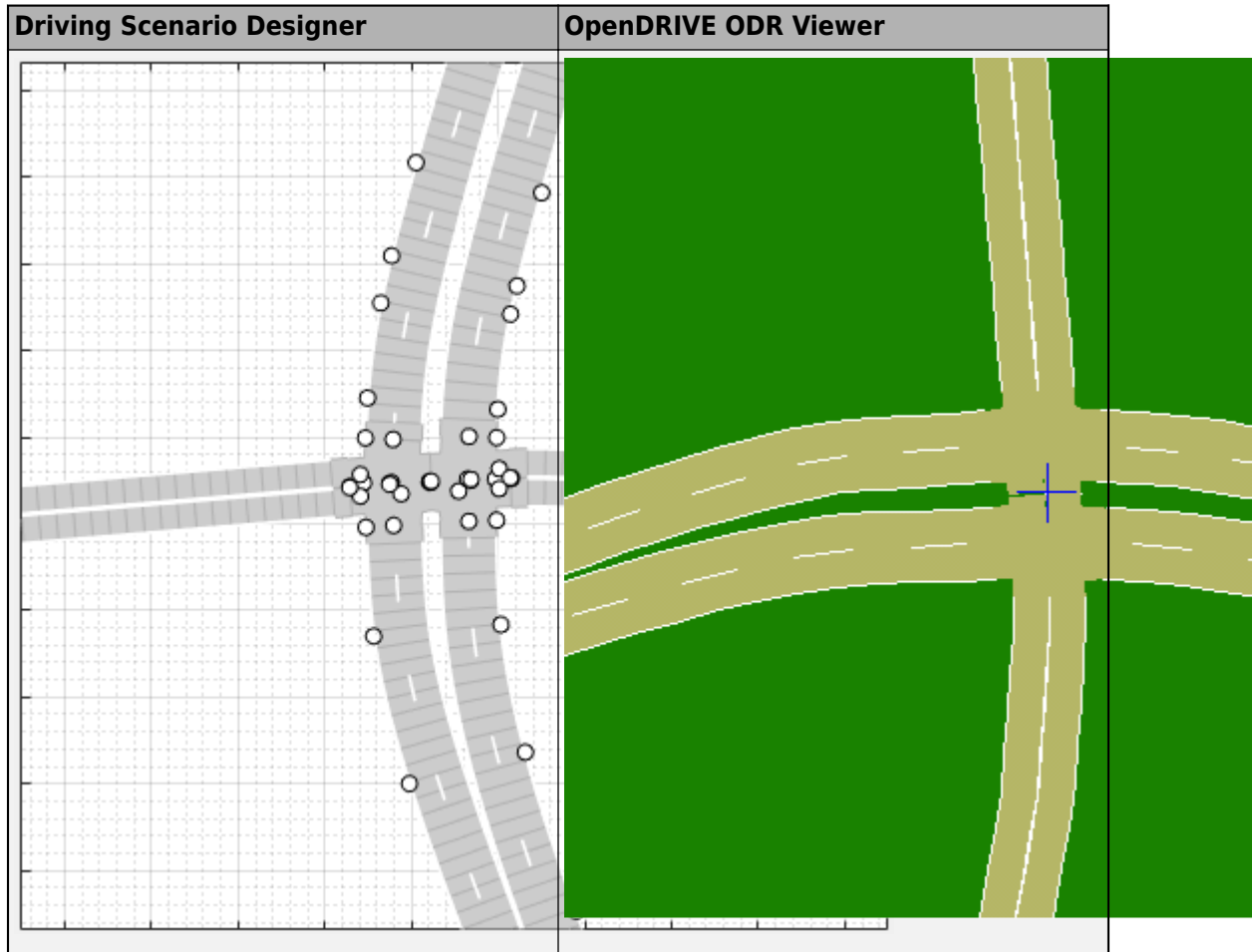
- You can import only lanes and roads. The import of road objects and traffic signals is not supported.
- OpenDRIVE files containing large road networks can take up to several minutes to load. In addition, these road networks can cause slow interactions on the app canvas.

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 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. If imported roads have these lane marking styles, the app sets their lane markings to the default style, as determined by the number of lanes in the road.

### **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.

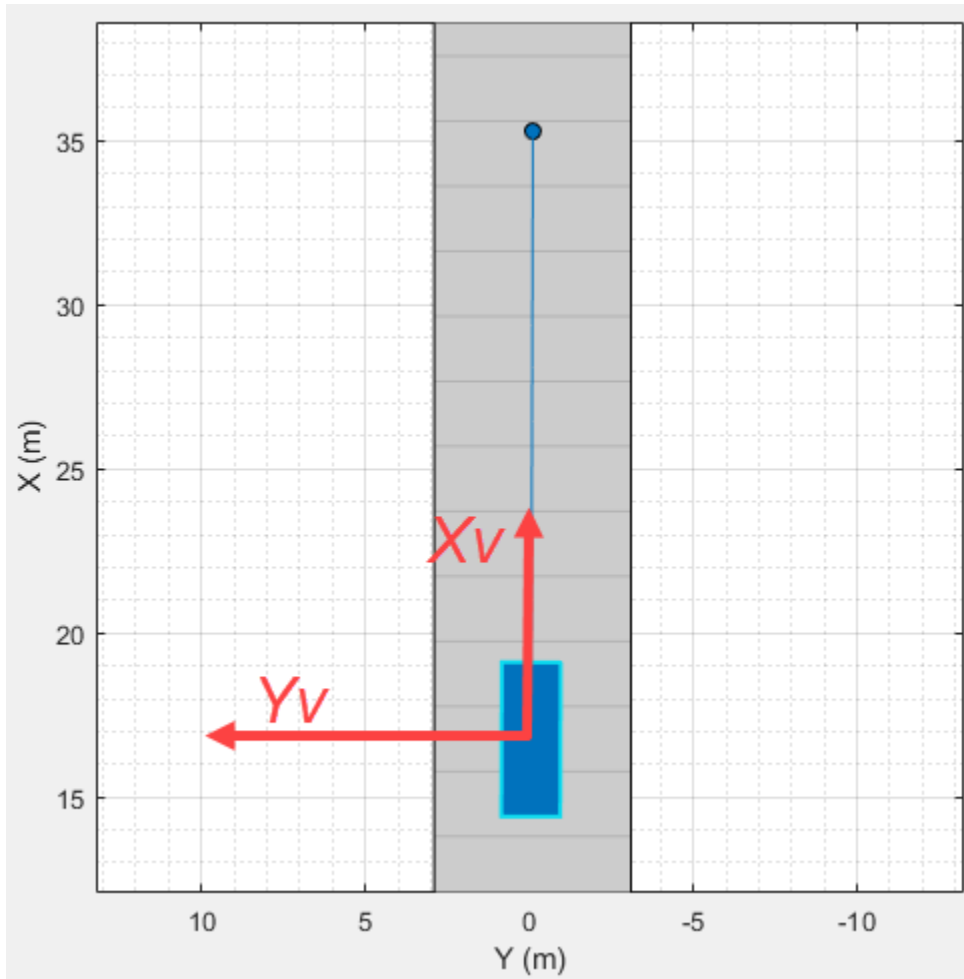


In the OpenDRIVE ODR viewer, the  $X$ -axis runs along the bottom of the viewer, and the  $Y$ -axis runs along the left side of the viewer.

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.



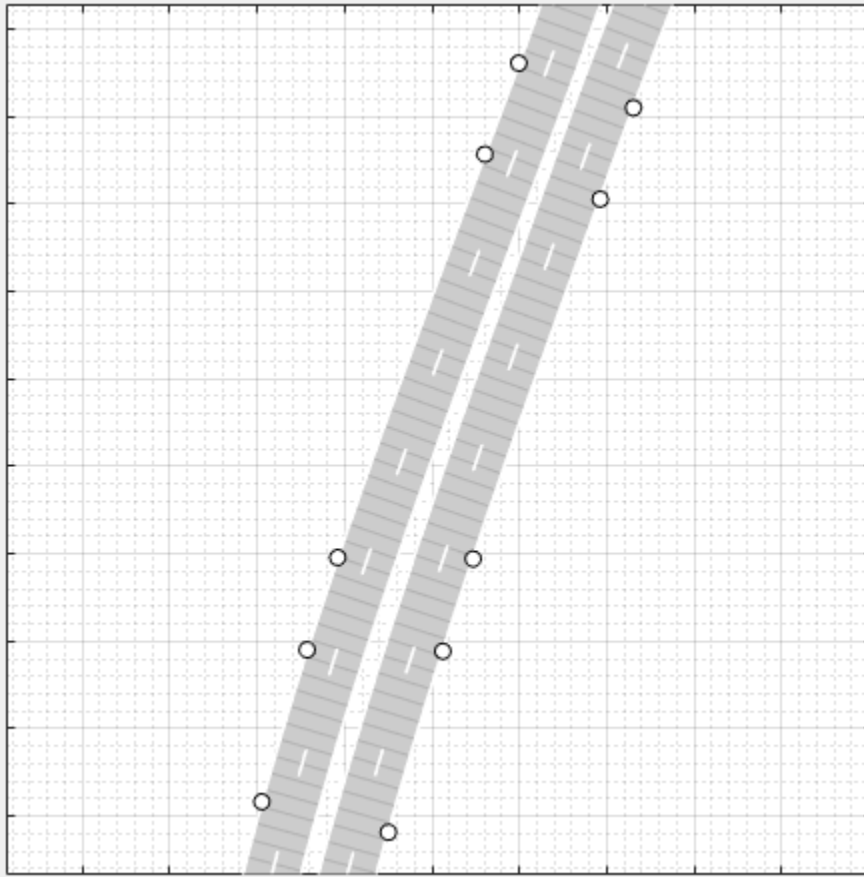
For more details about the coordinate systems, see “Coordinate Systems in Automated Driving System Toolbox” on page 1-2.

### 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.

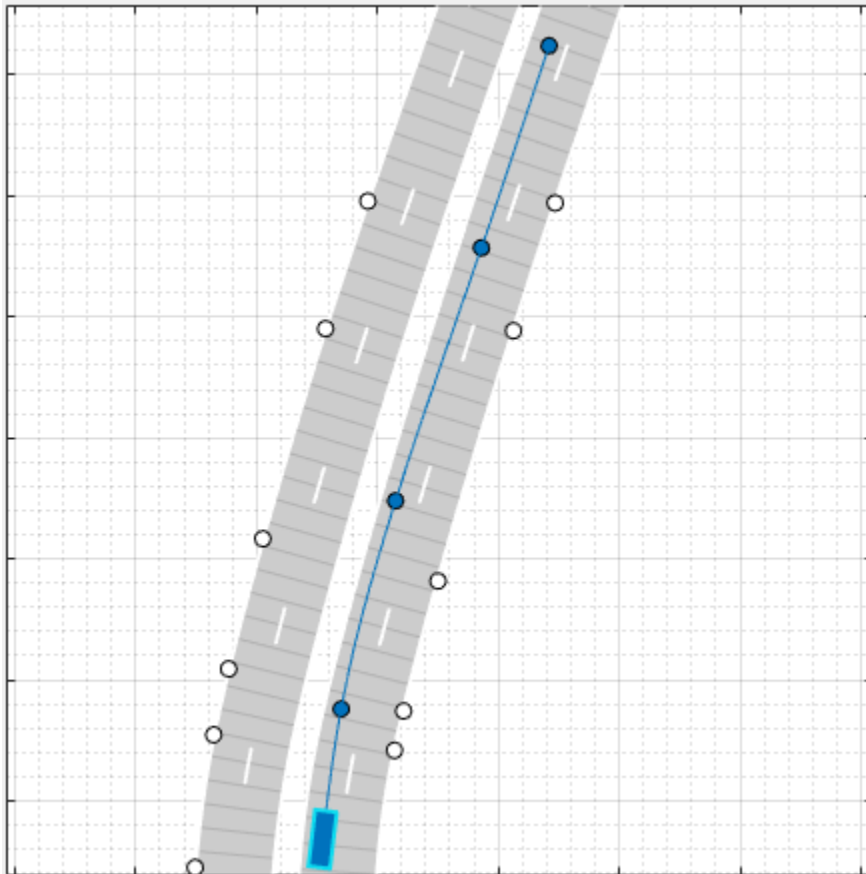
- The lanes on the right side of the highway have their road centers on the right edge.
- The lanes on the left side of the highway have their road centers on the left edge.



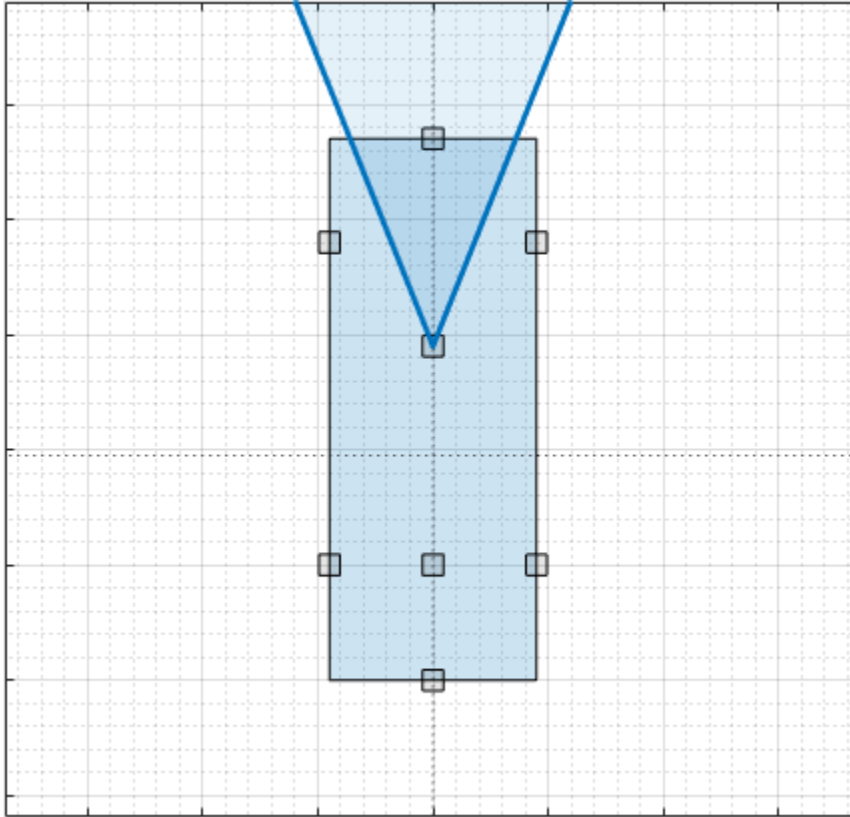
## Add Actors and Sensors to Scenario

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 car 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 car. From the app toolstrip, click **Add Camera**. Then, on the sensor canvas, add the camera to the predefined location representing the front window of the car.

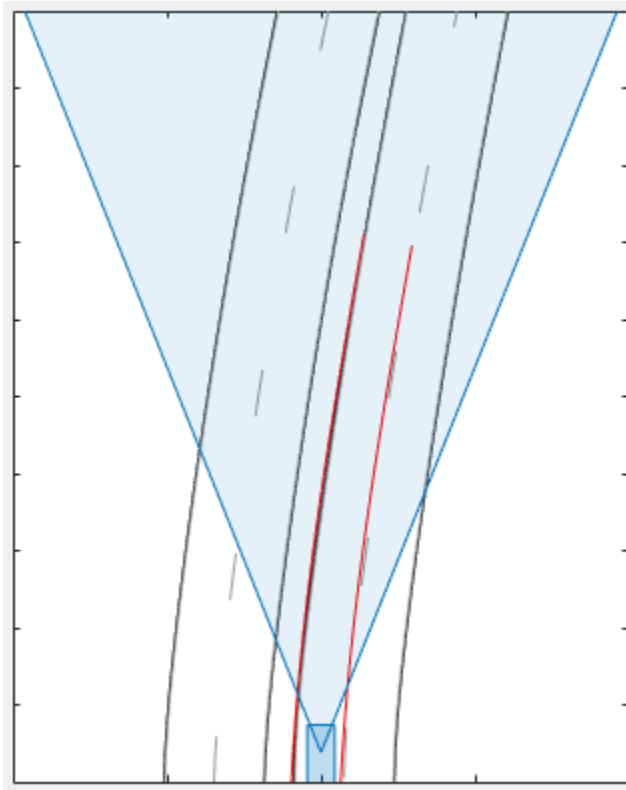


Configure the camera to detect lanes. In the left pane, on the **Sensors** tab, expand the **Detection Parameters** section. Then, set the **Detection Type** parameter to Lanes.

### Generate Synthetic Detections

To generate lane detections from the camera, from the app toolstrip, click **Run**. As the scenario runs, the **Ego-Centric View** displays the scenario from the perspective of the ego car. The **Bird's-Eye Plot** displays the left-lane and right-lane boundaries of the ego car.





To export the detections to the MATLAB workspace, from 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 Session

After you generate the detections, click **Save** to save the app session to a MAT-file. In addition, you can save the sensor models separately. You can also save the road and actor models into a separate file.

You can reopen this session from within the app or by using this syntax at the MATLAB command prompt:

`drivingScenarioDesigner(sessionFileName)`

When you reopen this session, the **Add Road** button remains disabled.

## References

- [1] Dupuis, Marius, et al. *OpenDRIVE Format Specification*. Revision 1.4, Issue H, Document No. VI2014.106. Bad Aibling, Germany: VIRES Simulationstechnologie GmbH, November 4, 2015.

## See Also

### Apps

**Driving Scenario Designer**

### Classes

`drivingScenario`

### Functions

`roadNetwork`

## More About

- “Build a Driving Scenario and Generate Synthetic Detections” on page 4-2
- “Generate Synthetic Detections from a Prebuilt Driving Scenario” on page 4-18
- “Coordinate Systems in Automated Driving System Toolbox” on page 1-2

## See Also

### External Websites

- [opendrive.org](http://opendrive.org)