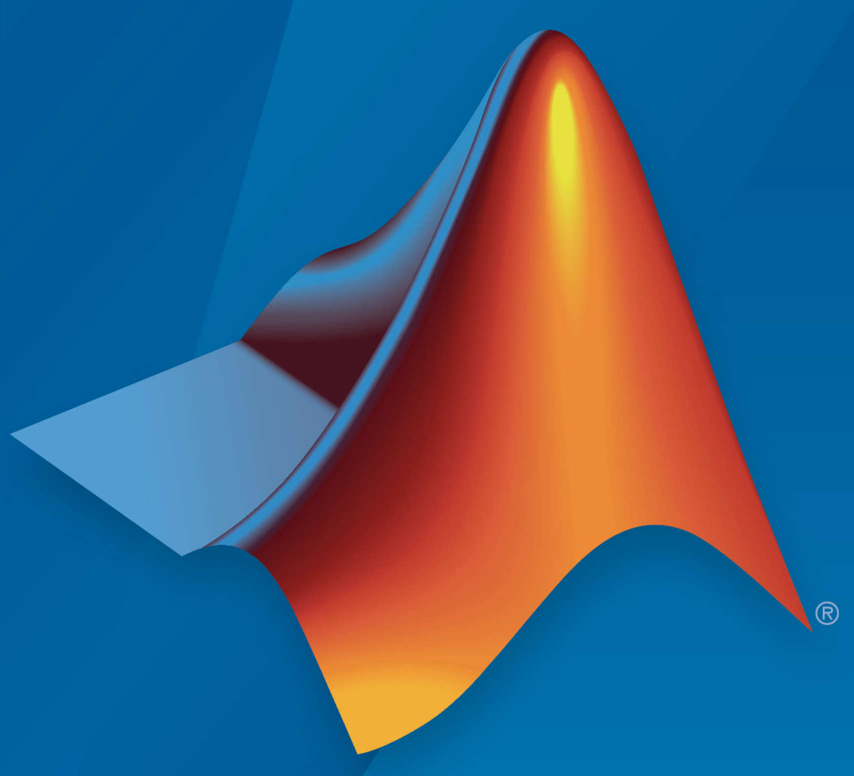
Automated Driving Toolbox™

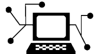
User's Guide



MATLAB®

R2019a

How to Contact MathWorks



Latest news: www.mathworks.com

Sales and services: www.mathworks.com/sales_and_services

User community: www.mathworks.com/matlabcentral

Technical support: www.mathworks.com/support/contact_us

Phone: 508-647-7000

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

Automated Driving Toolbox™ User's Guide

© COPYRIGHT 2017–2019 by The MathWorks, Inc.

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

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

Trademarks

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

Patents

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

Revision History

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

Contents

1 Sensor Configuration and Coordinate System Transformations

Coordinate Systems in Automated Driving Toolbox **1-2**
World Coordinate System **1-2**
Vehicle Coordinate System **1-2**
Sensor Coordinate System **1-4**
Spatial Coordinate System **1-7**

Calibrate a Monocular Camera **1-8**
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**

2 Ground Truth Labeling and Verification

Get Started with the Ground Truth Labeler **2-2**
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 **2-18**
Save App Session **2-22**

Keyboard Shortcuts and Mouse Actions for Ground Truth Labeler **2-24**
Label Definitions **2-24**
Frame Navigation and Time Interval Settings **2-24**

Labeling Window **2-25**
Polyline Drawing **2-25**
Polygon Drawing **2-26**
Zooming **2-27**
App Sessions **2-27**

3 Tracking and Sensor Fusion

Visualize Sensor Data and Tracks in Bird's-Eye Scope **3-2**
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**

Linear Kalman Filters **3-11**
State Equations **3-11**
Measurement Models **3-13**
Linear Kalman Filter Equations **3-13**
Filter Loop **3-14**
Constant Velocity Model **3-15**
Constant Acceleration Model **3-16**

Extended Kalman Filters **3-18**
State Update Model **3-18**
Measurement Model **3-19**
Extended Kalman Filter Loop **3-19**
Predefined Extended Kalman Filter Functions **3-20**

4 Driving Scenario Generation and Sensor Models

Build a Driving Scenario and Generate Synthetic Detections
. . . . **4-2**
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 Scenario . . . 4-17

Generate Synthetic Detections from a Prebuilt Driving Scenario . . . 4-18
Choose a Prebuilt Scenario . . . 4-18
Modify Scenario . . . 4-37
Generate Synthetic Detections . . . 4-38
Save Scenario . . . 4-39

Generate Synthetic Detections from a Euro NCAP Scenario . . . 4-40
Choose a Euro NCAP Scenario . . . 4-40
Modify Scenario . . . 4-56
Generate Synthetic Detections . . . 4-57
Save Scenario . . . 4-58

Add OpenDRIVE Roads to Driving Scenario . . . 4-60
Import OpenDRIVE File . . . 4-60
Inspect Roads . . . 4-61
Add Actors and Sensors to Scenario . . . 4-67
Generate Synthetic Detections . . . 4-68
Save Scenario . . . 4-69

Test Open-Loop ADAS Algorithm Using Driving Scenario . . . 4-72

Test Closed-Loop ADAS Algorithm Using Driving Scenario . . . 4-78

5 Planning, Mapping, and Control

Access HERE HD Live Map Data . . . 5-2
Step 1: Enter Credentials . . . 5-2
Step 2: Create Reader Configuration . . . 5-3
Step 3: Create Reader . . . 5-4

Step 4: Read and Visualize Data **5-5**

Enter HERE HD Live Map Credentials **5-9**

Create Configuration for HERE HD Live Map Reader **5-11**
Create Configuration for Specific Catalog **5-12**
Create Configuration for Specific Version **5-15**
Configure Reader **5-15**

Create HERE HD Live Map Reader **5-17**
Create Reader from Specified Driving Route **5-17**
Create Reader from Specified Map Tile IDs **5-20**

Read and Visualize Data Using HERE HD Live Map Reader . **5-21**
Create Reader **5-22**
Read Map Layer Data **5-24**
Visualize Map Layer Data **5-30**

HERE HD Live Map Layers **5-34**
Road Centerline Model **5-35**
HD Lane Model **5-37**
HD Localization Model **5-39**

Control Vehicle Velocity **5-40**

Sensor Configuration and Coordinate System Transformations

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

Coordinate Systems in Automated Driving Toolbox

Automated Driving Toolbox uses these coordinate systems:

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

World Coordinate System

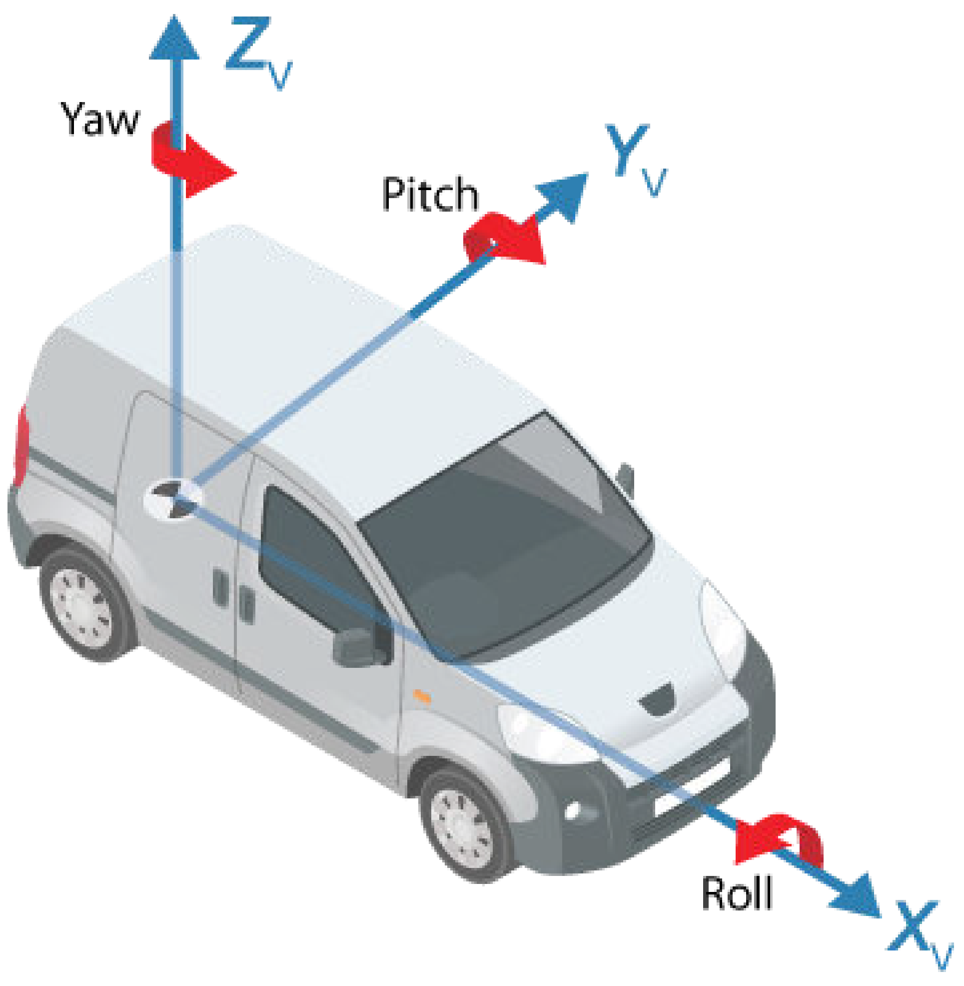
All vehicles, sensors, and their related coordinate systems are placed in the world coordinate system. A world coordinate system is important in global path planning, localization, mapping, and driving scenario generation. Units are typically in meters.

Vehicle Coordinate System

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

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

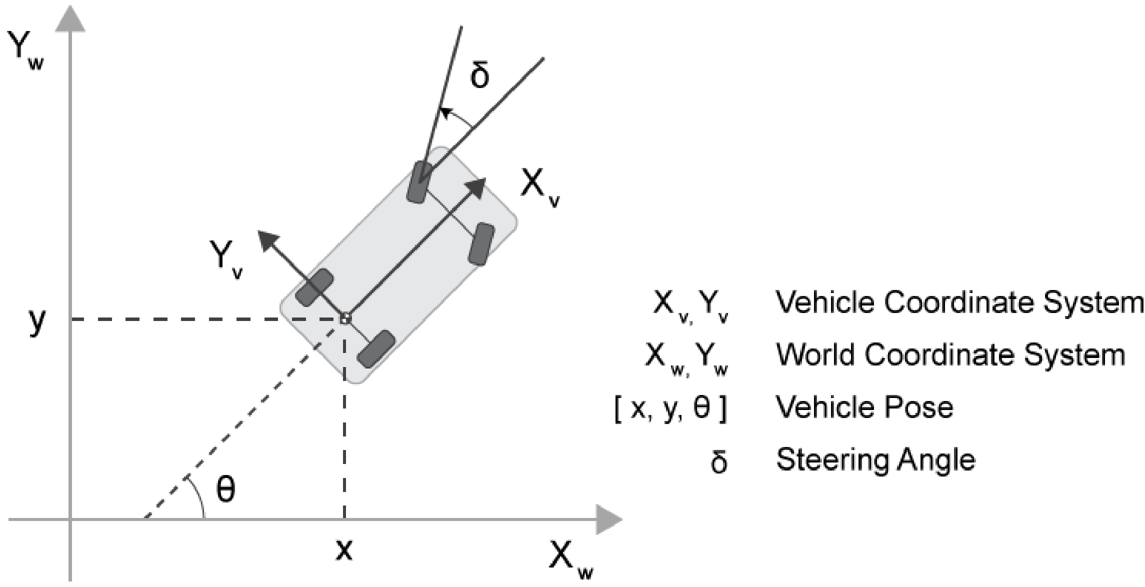
The vehicle coordinate system follows the ISO convention for rotation. Each axis is positive in the clockwise direction, when looking in the positive direction of that axis.



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.

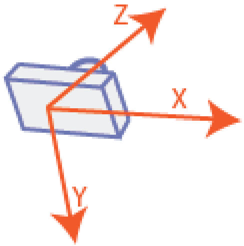
Values returned by individual sensors are transformed into the vehicle coordinate system so that they can be placed in a unified frame of reference.

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.

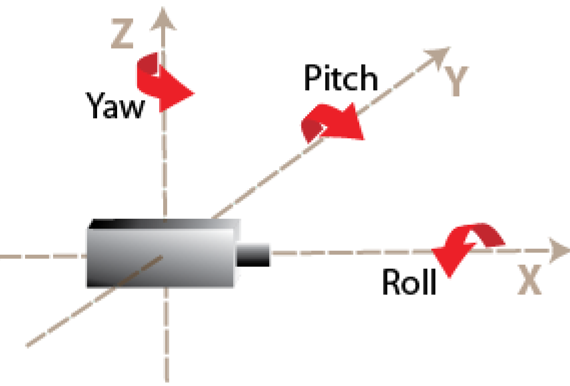


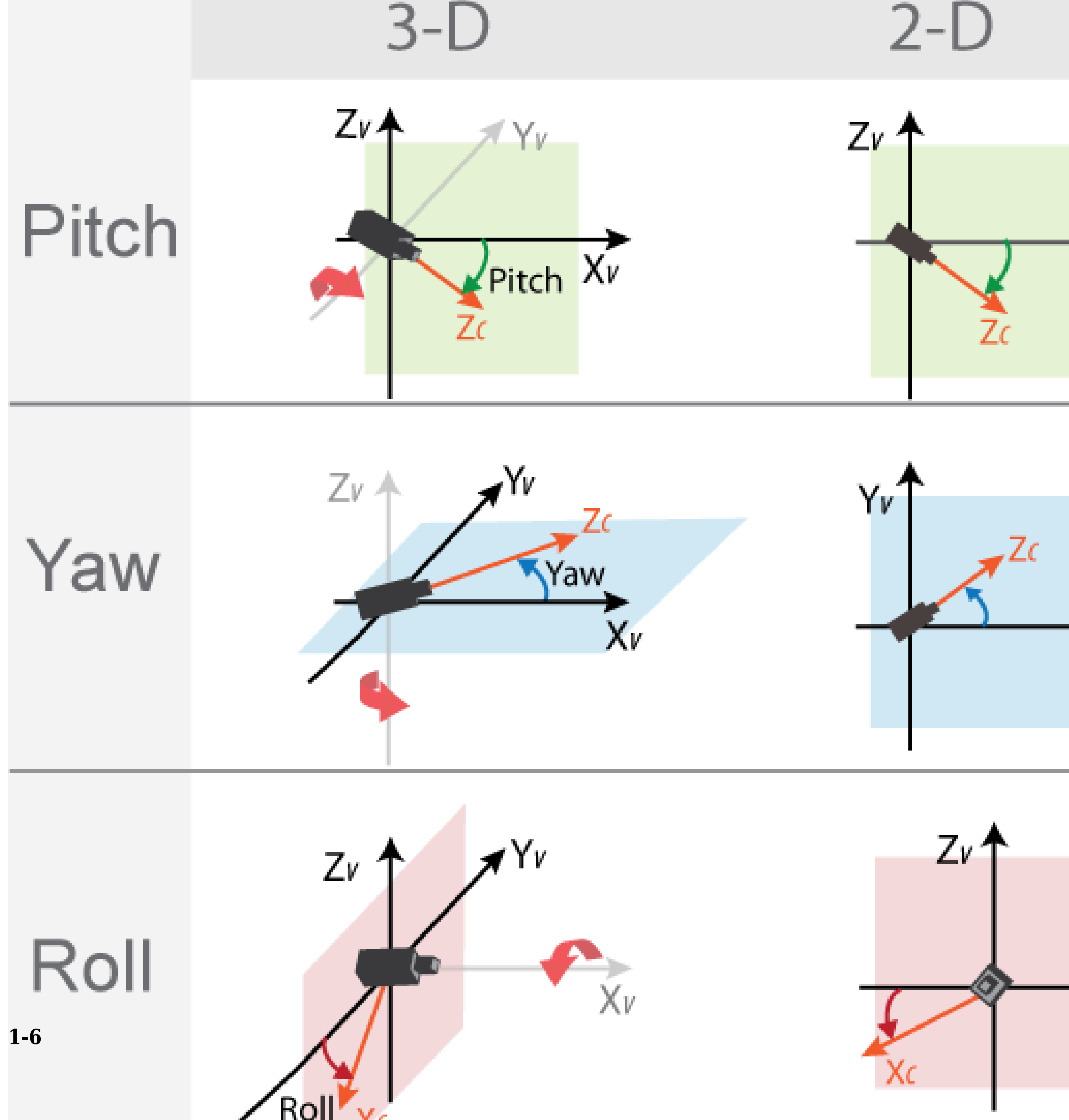
Sensor Coordinate System

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 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.



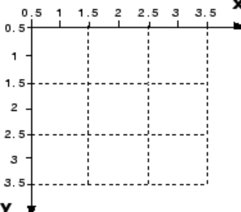
The yaw, pitch, and roll angles of sensors follow an ISO convention. These angles have positive clockwise directions when looking in the positive direction of the *Z*-, *Y*-, and *X*-axes, respectively.




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

Spatial Coordinate System

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



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

Calibrate a Monocular Camera

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

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

Estimate Intrinsic Parameters

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

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

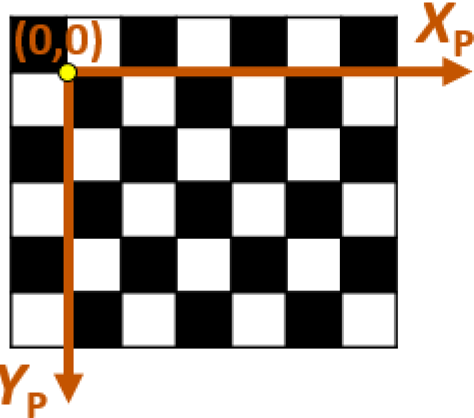
Alternatively, to better visualize the results, use the **Camera Calibrator** app. For information on setting up the camera, preparing the checkerboard pattern, and calibration techniques, see “Single Camera Calibrator App” (Computer Vision Toolbox).

Place Checkerboard for Extrinsic Parameter Estimation

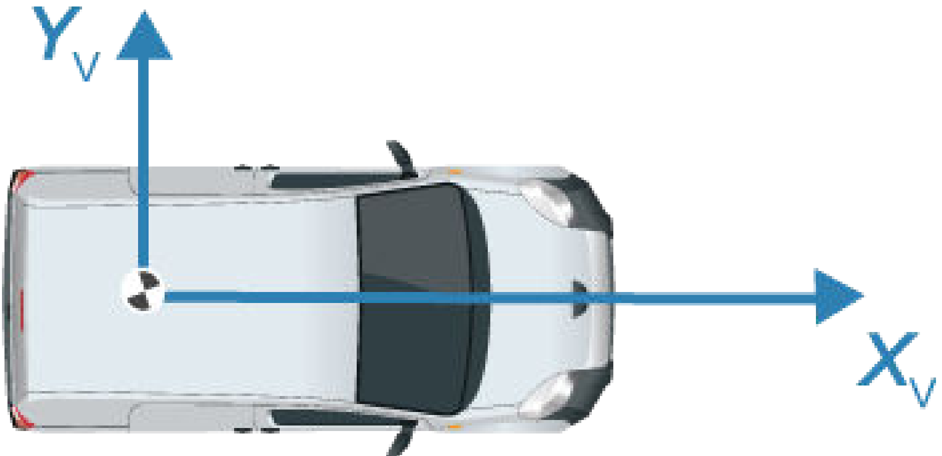
For a monocular camera mounted on a vehicle, the *extrinsic parameters* define the mounting position of that camera. These parameters include the rotation angles of the camera with respect to the vehicle coordinate system, and the height of the camera above the ground.

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

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



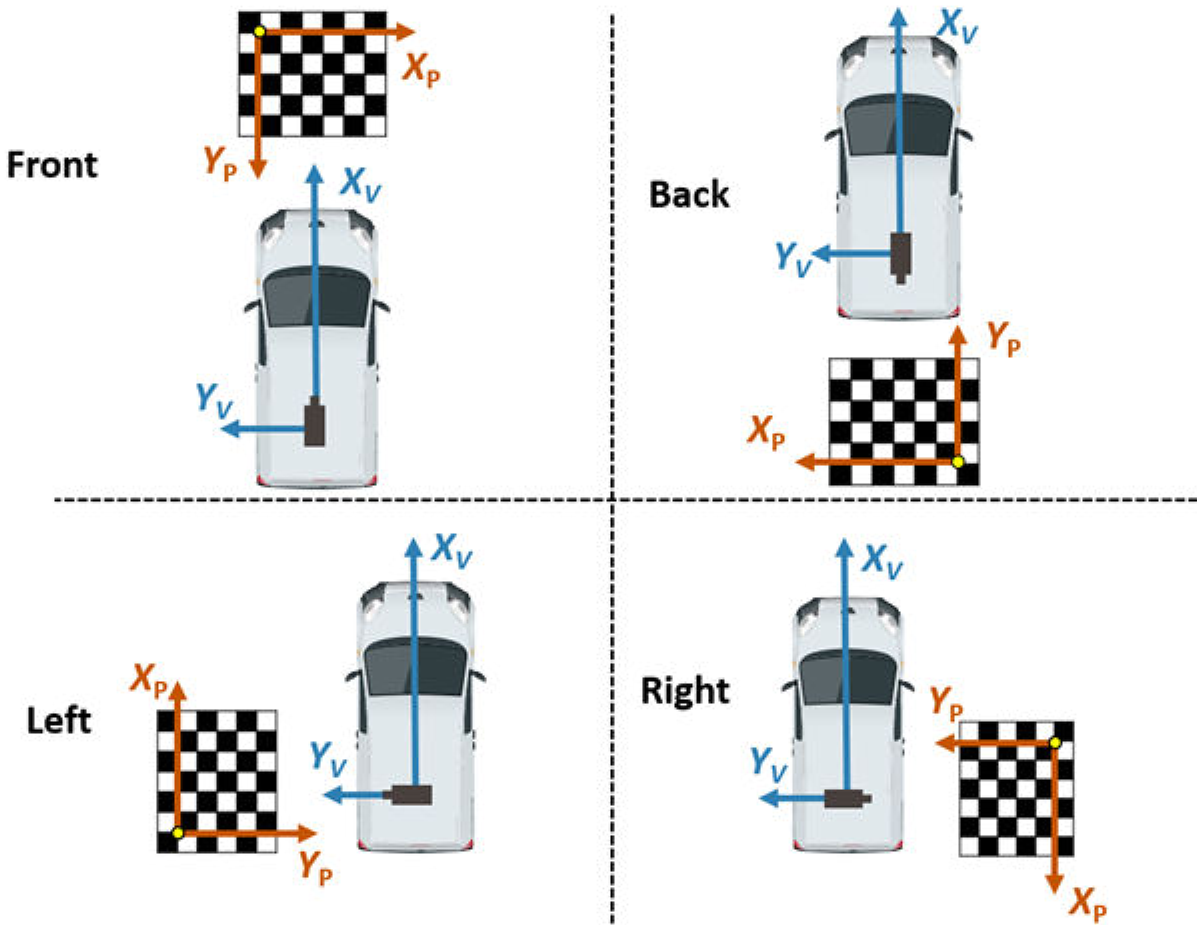
When placing the checkerboard pattern in relation to the vehicle, the X_P- and Y_P-axes must align with the X_V- and Y_V-axes of the vehicle. In the vehicle coordinate system, the X_V-axis points forward from the vehicle and the Y_V-axis points to the left, as viewed when facing forward. The origin is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



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

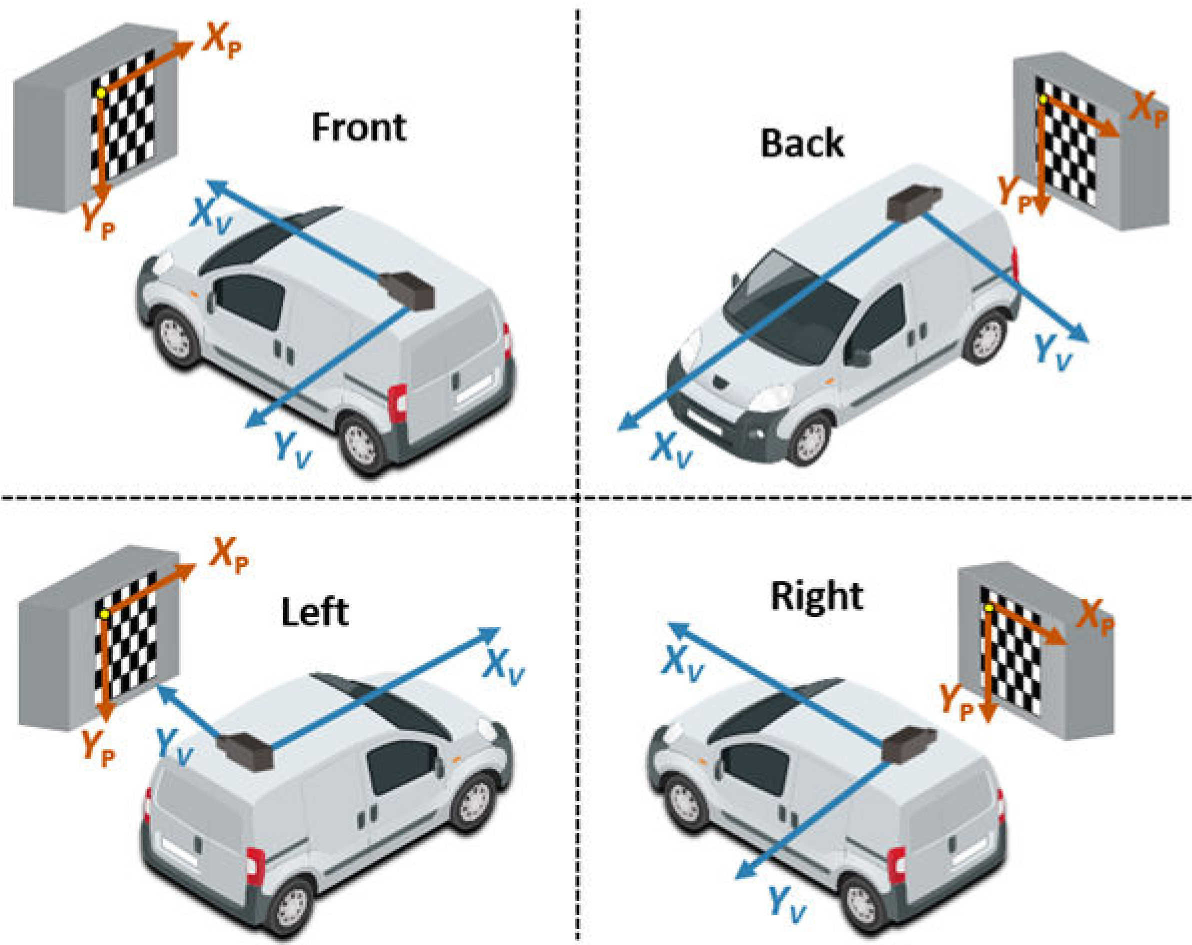
Horizontal Orientation

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



Vertical Orientation

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



Estimate Extrinsic Parameters

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

- The intrinsic parameters of the camera
- The key points detected in the image, in this case the corners of the checkerboard squares

- The world points of the checkerboard
- The height of the checkerboard pattern's origin above the ground

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

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

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

Configure Camera Using Intrinsic and Extrinsic Parameters

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

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

See Also

Apps
Camera Calibrator

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

Objects
`monoCamera`

More About

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

Ground Truth Labeling and Verification

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

Get Started with the Ground Truth Labeler

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

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

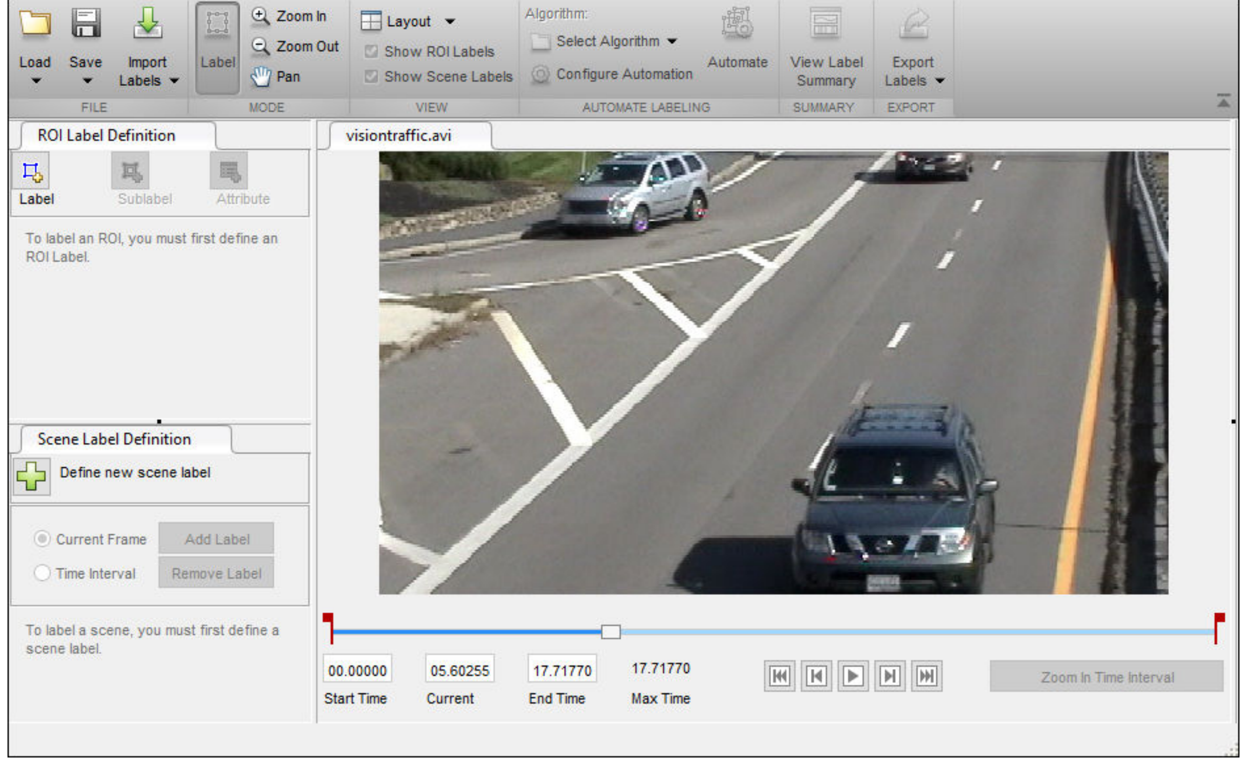
Load Unlabeled Data

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

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

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

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



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

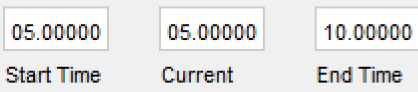
To load a custom data source that is readable by `VideoReader` or `imread`, see “Use Custom Data Source Reader for Ground Truth Labeling” (Computer Vision Toolbox).

Set Time Interval to Label

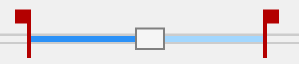
You can label the entire video or start with a portion of the video. In this example, you label a five-second time interval within the loaded video. In the text boxes below the video, enter these times in seconds:

1. In the **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.

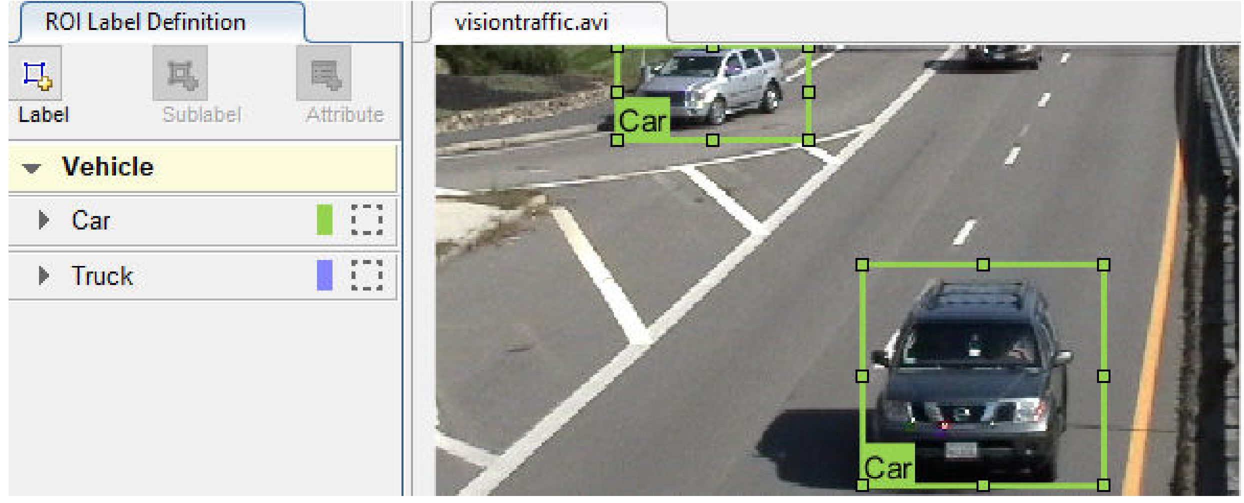
ROI Label	Description	Example: Driving Scene
`Rectangle`	Draw rectangular ROI labels (bounding boxes) around objects.	Vehicles, pedestrians, road signs
`Line`	Draw linear ROI labels to represent lines. To draw a polyline ROI, use two or more points.	Lane boundaries, guard rails, road curbs

ROI Label	Description	Example: Driving Scene
`Pixel label`	Assign labels to pixels for semantic segmentation. See "Label Pixels for Semantic Segmentation" (Computer Vision Toolbox).	Vehicles, road surface, trees, pavement

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

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

   The **Vehicle** group name appears in the **ROI Label Definition** pane with the label **Car** created. You can move a labels to a different position or group by left-clicking and dragging the label.
5. Add a second label. Click **Label**. Name the label **Truck** and make sure the **Vehicle** group is selected. Click **OK**.
6. In the first video frame within the time interval, use the mouse to draw rectangular **Car** ROIs around the two vehicles.



Create Sublabels

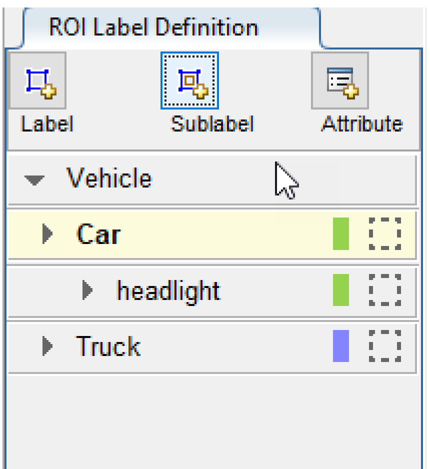
A sublabel is a type of ROI label that corresponds to a parent ROI label. Each sublabel must belong to, or be a child of, a specific label defined in the **ROI Label Definition** pane. For example, in a driving scene, a vehicle label might have sublabels for headlights, license plates, or wheels.

Define a sublabel for headlights.

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

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

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

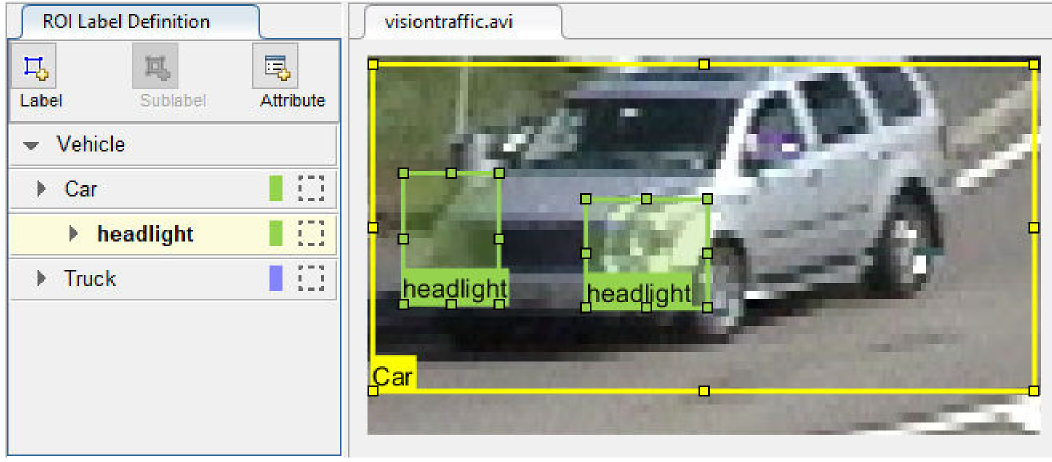


4 In the **ROI Label Definition** pane, select the **headlight** sublabel.

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

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

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

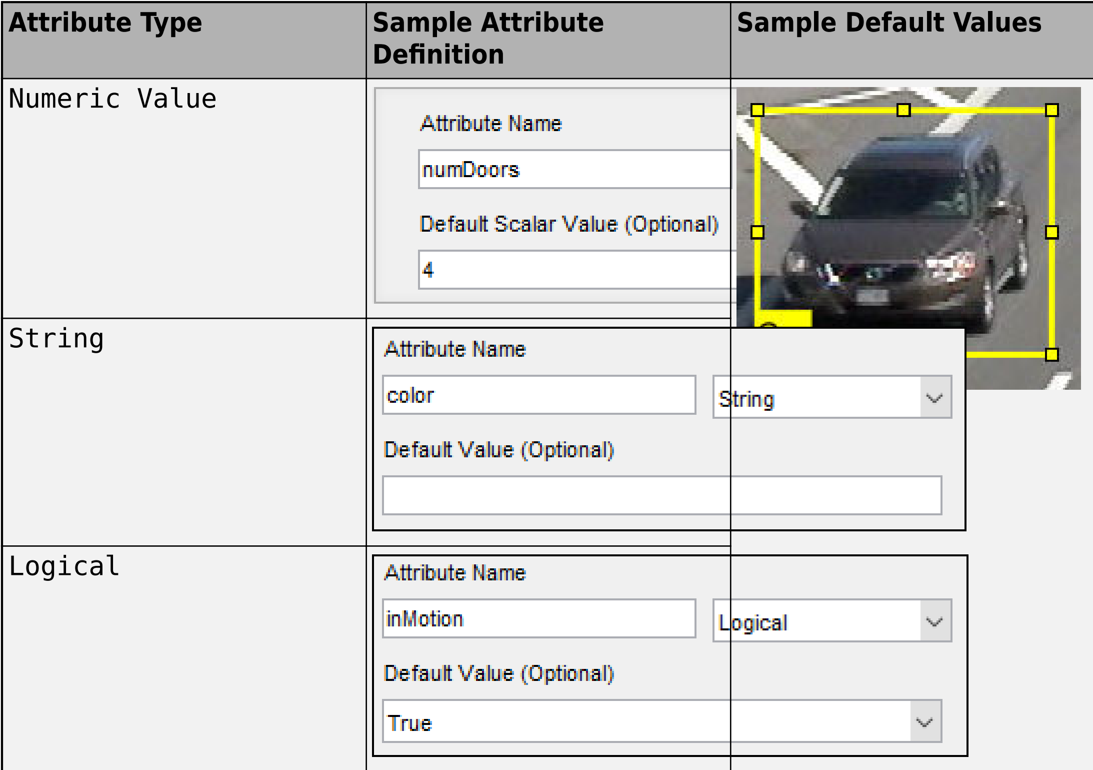


Sublabels can only be used with rectangular or polyline ROI labels and cannot have their own sublabels. For more details on working with sublabels, see “Use Sublabels and Attributes to Label Ground Truth Data” (Computer Vision Toolbox).

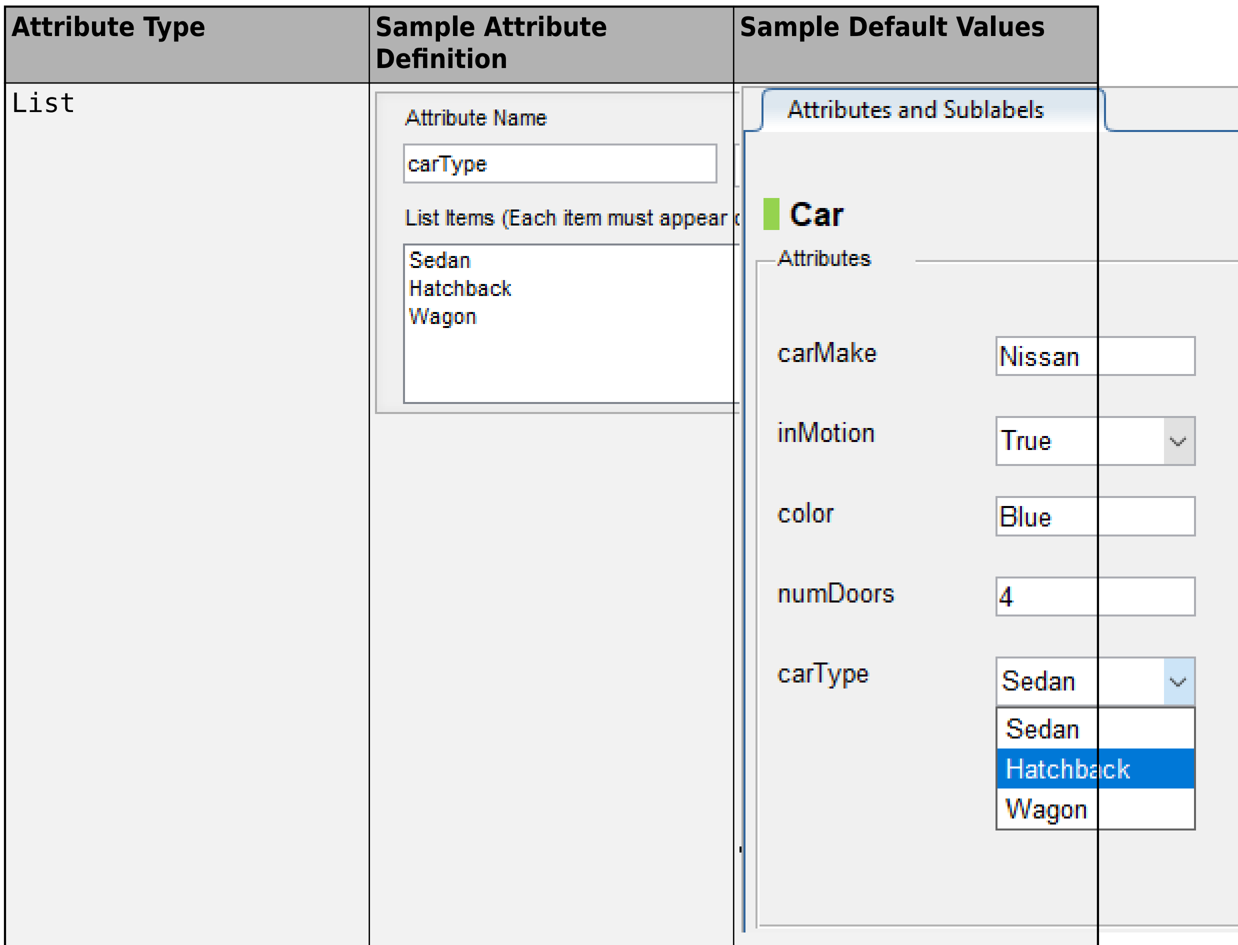
Create Attributes

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

You can define these types of attributes.



Attribute Type	Sample Attribute Definition	Sample Default Values
`Numeric Value`	Attribute Name: numDoors Default Scalar Value (Optional): 4	
`String`	Attribute Name: color; String Default Value (Optional):	
`Logical`	Attribute Name: inMotion; Logical Default Value (Optional): True	

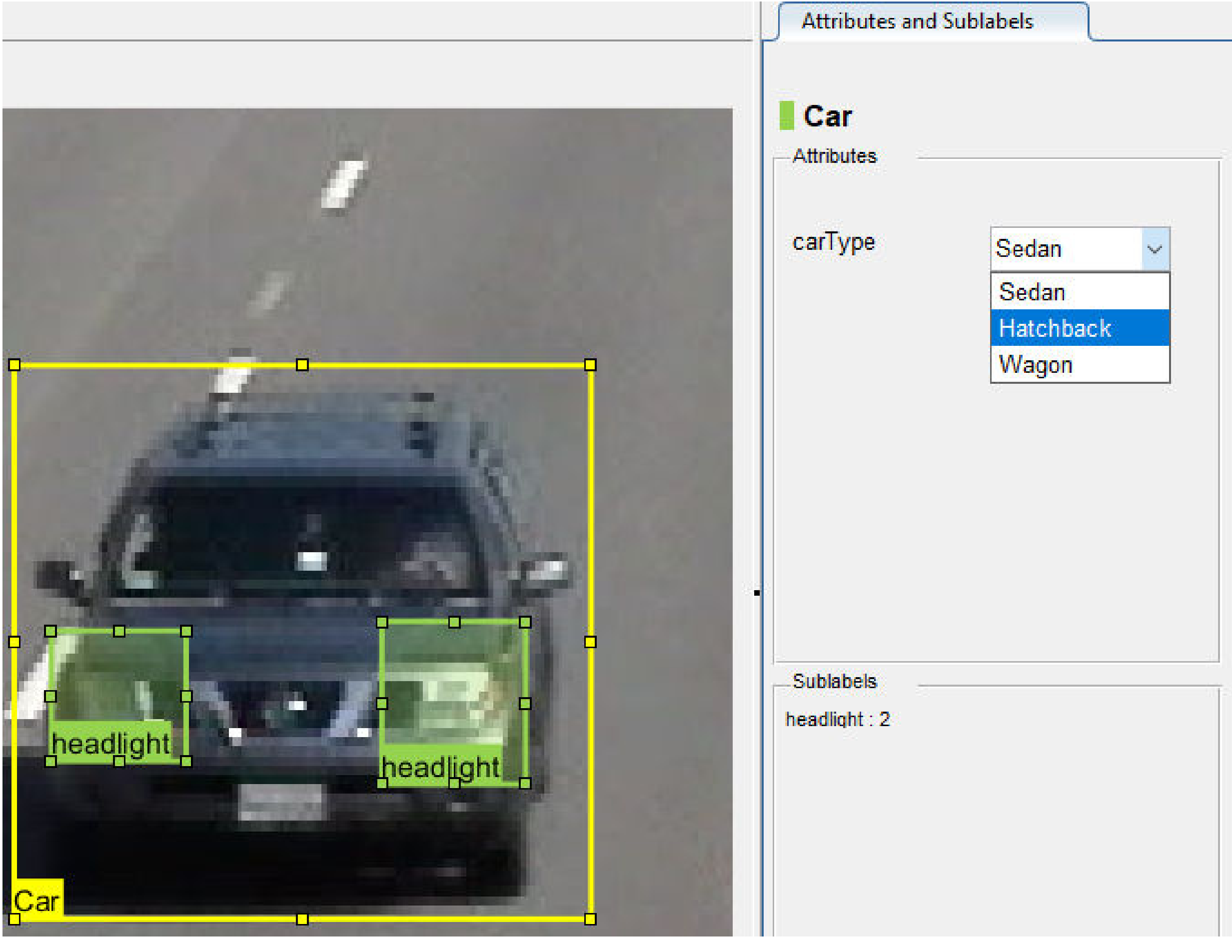


Attribute Type	Sample Attribute Definition	Sample Default Values
`List`		

Add an attribute for the vehicle type.

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

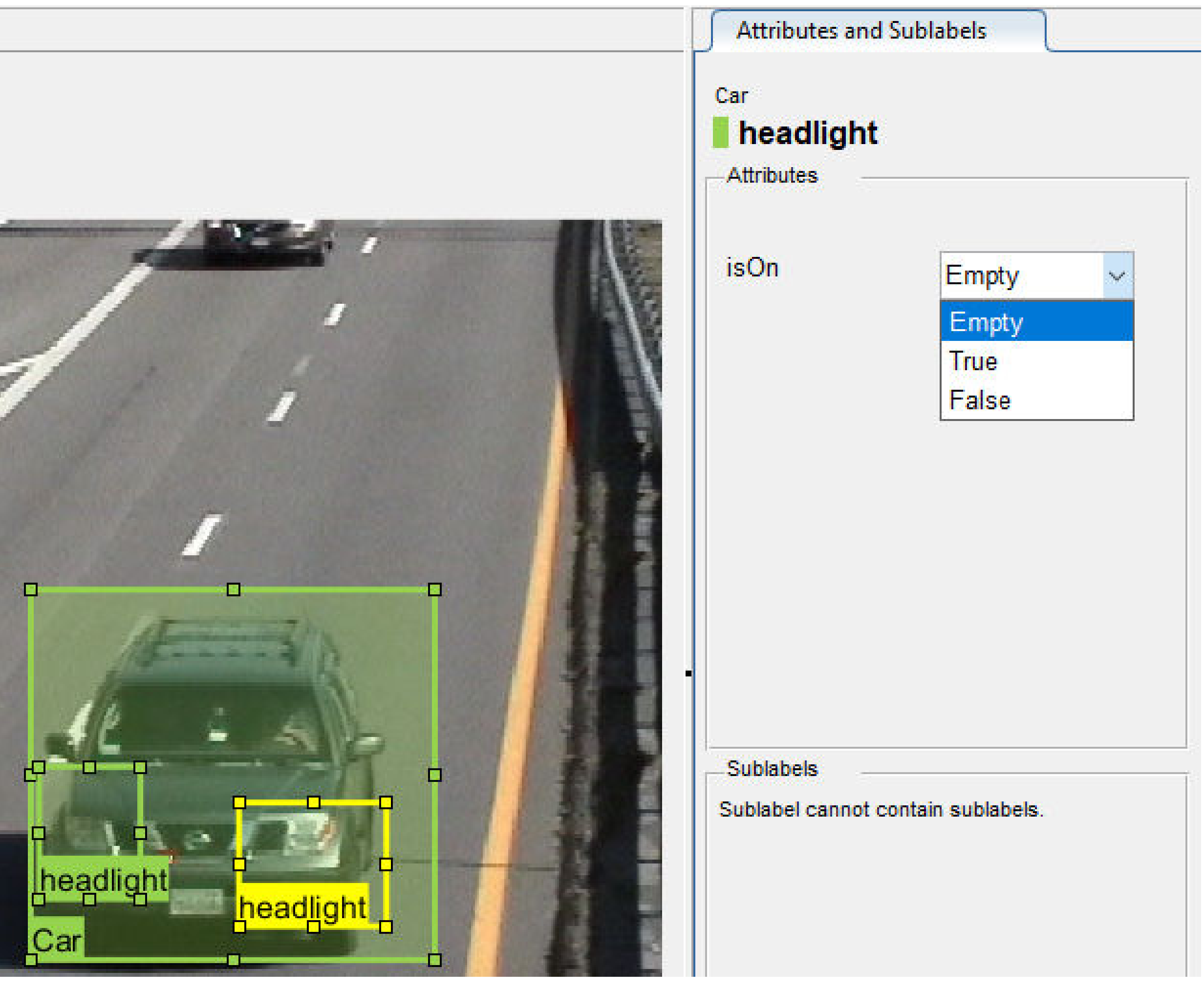
4. In the first frame of the video, select a **Car** ROI label. In the **Attributes and Sublabels** pane, select the appropriate **carType** attribute value for that vehicle.
5. Repeat the previous step to assign a **carType** attribute to the other vehicle.



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

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

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



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

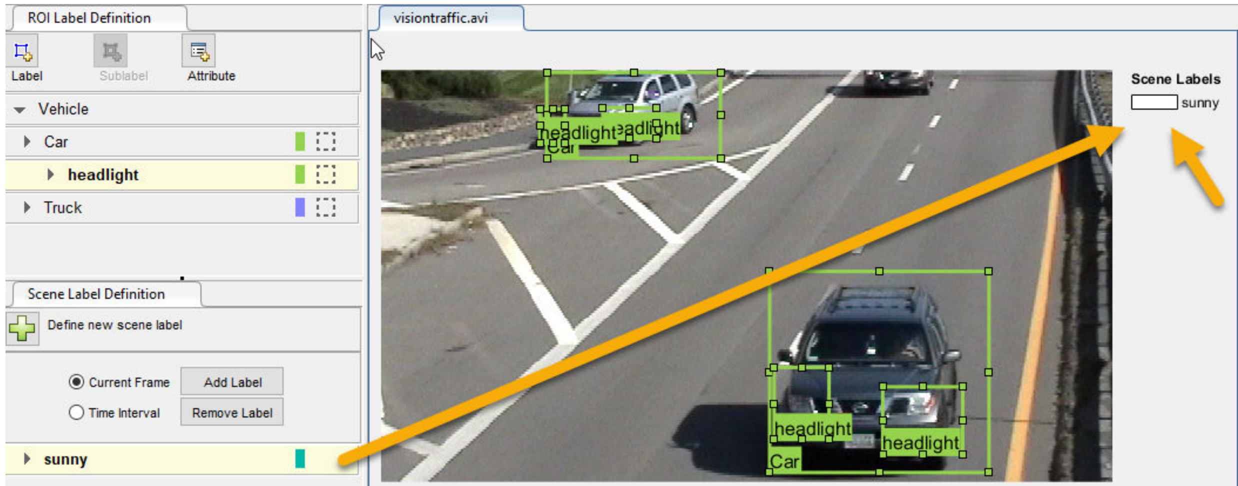
Create Scene Labels

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

Create a scene label to use in the video.

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

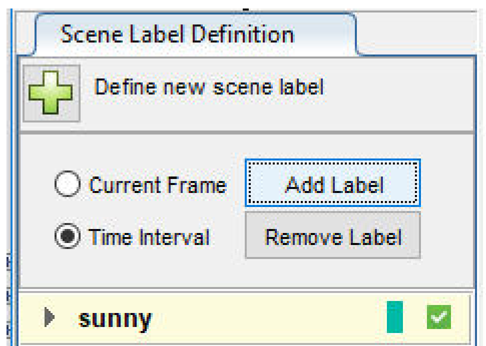
   The **Scene Label Definition** pane shows the scene label definition. The scene labels that are applied to the current frame appear in the **Scene Labels** pane on the right. The **sunny** scene label is empty (white), because the scene label has not yet been applied to the frame.

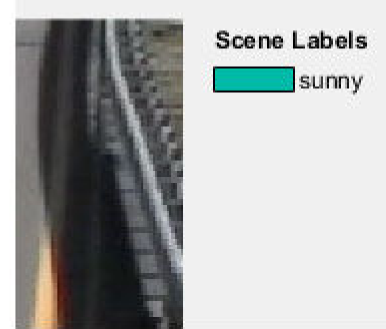


2 The entire scene is sunny, so specify to apply the **sunny** scene label over the entire time interval. With the **sunny** scene label definition still selected in the **Scene Label Definition** pane, select **Time Interval**.

3 Click **Add Label**.

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





Label Ground Truth

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

Label Ground Truth Manually

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

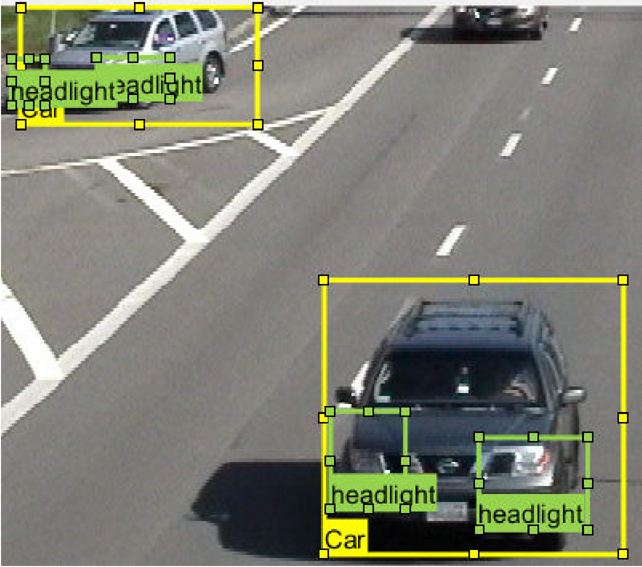
Advance frame by frame and draw the label and sublabel ROIs manually. Also update the attribute information for these ROIs.

Label Ground Truth Using Automation Algorithm

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

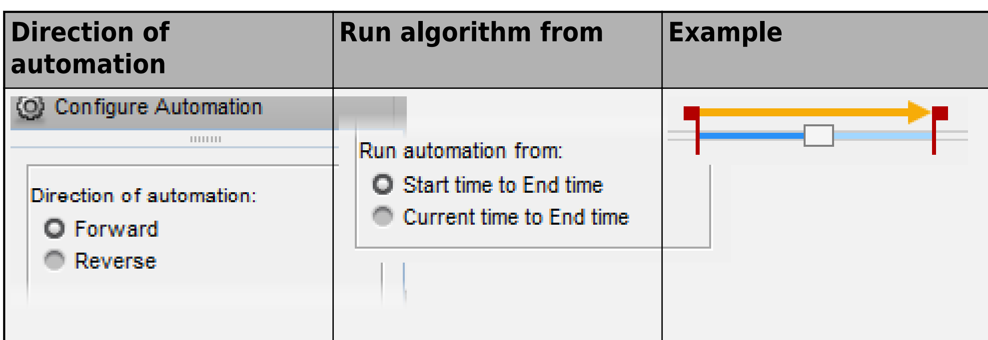
In this example, you automate the labeling of only the **Car** ROI labels. The built-in automation algorithms do not support sublabel and attribute automation.

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

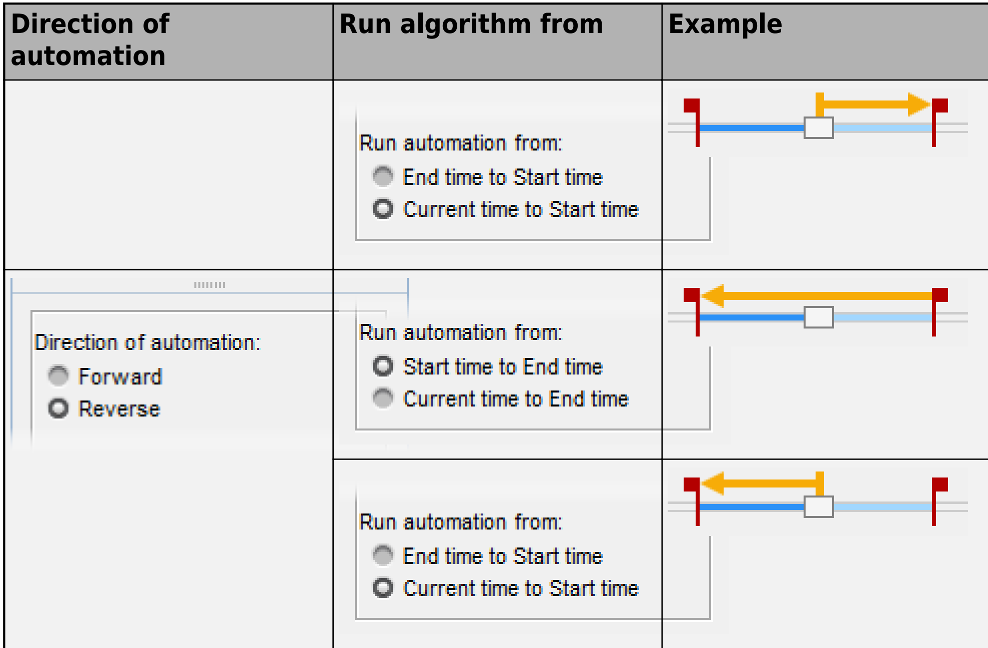


2 From the app toolstrip, select **Select Algorithm** > **Point Tracker**. This algorithm tracks one or more rectangle ROIs over short intervals using the Kanade-Lucas-Tomasi (KLT) algorithm.

3 (optional) Configure the automation settings. Click **Configure Automation**. By default, the automation algorithm applies labels from the start of the time interval to the end. To change the direction and start time of the algorithm, choose one of the options shown in this table.

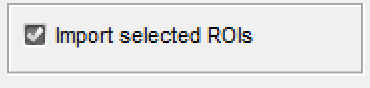


Direction of automation	Run algorithm from	Example
Configure Automation Direction of automation: Forward Reverse	Run automation from: Start time to End time Current time to End time	

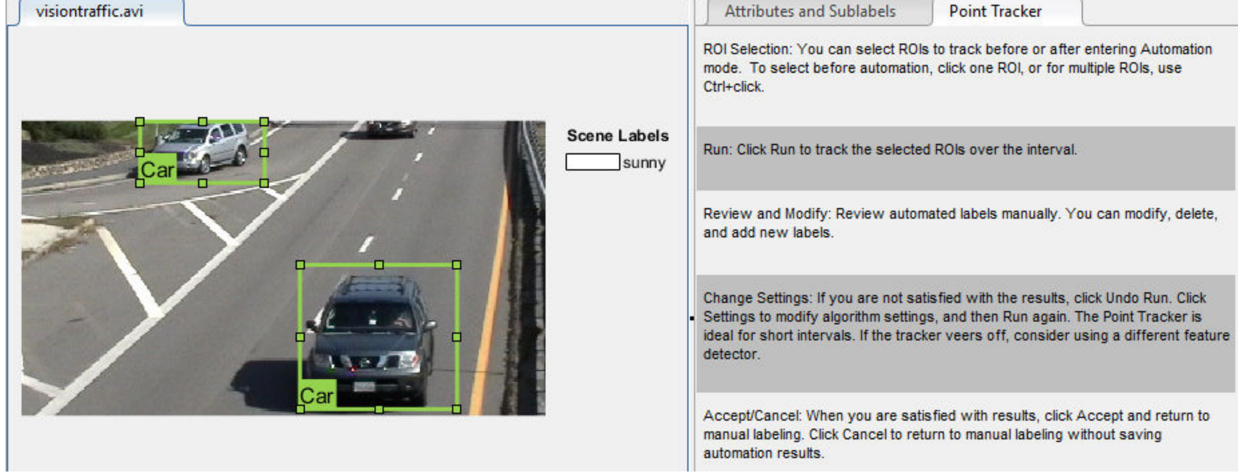


Direction of automation	Run algorithm from	Example
	Run automation from: End time to Start time; Current time to Start time	
Direction of automation: Forward; Reverse	Run automation from: Start time to End time; Current time to End time	
	Run automation from: End time to Start time; Current time to Start time	

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



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



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

   The vehicles that enter the scene later are unlabeled. The unlabeled vehicles did not have an initial ROI label, so the algorithm did not track them. Click **Undo Run**. Use the slider to find the frames where each vehicle first appears. Draw **vehicle** ROIs around each vehicle, and then click **Run** again.
7. Advance frame by frame and manually move, resize, delete, or add ROIs to improve the results of the automation algorithm.

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

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

To further evaluate your labels, you can view a visual summary of the labeled ground truth. From the app 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 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 Toolbox).

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

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

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

```
gTruth
```

```
gTruth =

  groundTruth with properties:

          DataSource: [1×1 groundTruthDataSource]
    LabelDefinitions: [3×5 table]
           LabelData: [531×3 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: [531×1 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 =

  3×5 table

     Name        Type         Group       Description     Hierarchy
    _______    _________    _________    ___________    ____________

    'Car'      Rectangle    'Vehicle'        ''         [1×1 struct]
    'Truck'    Rectangle    'Vehicle'        ''         []
    'sunny'    Scene        'None'           ''         []
```

Within `LabelDefinitions`, the `Hierarchy` column stores information about the sublabel and attribute definitions of a parent ROI label.

Display the sublabel and attribute information for the `Car` label.

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

```
ans =

  struct with fields:

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

Display information about the `headlight` sublabel.

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

```
ans = 

  struct with fields:

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

Display information about the `carType` attribute.

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

```
ans = 

  struct with fields:

      ListItems: {3×1 cell}
    Description: ''
```

Label Data

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

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

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

```
ans = 

  8×3 timetable

       Time           Car             Truck         sunny
    __________    ____________    ____________    _____

    5.005 sec     [1×2 struct]    [1×0 struct]    true 
    5.0384 sec    [1×2 struct]    [1×0 struct]    true 
    5.0717 sec    [1×2 struct]    [1×0 struct]    true 
    5.1051 sec    [1×2 struct]    [1×0 struct]    true 
    5.1385 sec    [1×2 struct]    [1×0 struct]    true 
    5.1718 sec    [1×2 struct]    [1×0 struct]    true 
```

```
    5.2052 sec    [1×2 struct]    [1×0 struct]    true
    5.2386 sec    [1×2 struct]    [1×0 struct]    true
    ...
```

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

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

```
ans =

  8×3 timetable

       Time           Car             Truck          sunny
    __________    ____________    ____________    _____

    5.005 sec     [1×2 struct]    [1×0 struct]    true
    5.0384 sec    [1×2 struct]    [1×0 struct]    true
    5.0717 sec    [1×2 struct]    [1×0 struct]    true
    5.1051 sec    [1×2 struct]    [1×0 struct]    true
    5.1385 sec    [1×2 struct]    [1×0 struct]    true
    5.1718 sec    [1×2 struct]    [1×0 struct]    true
    5.2052 sec    [1×2 struct]    [1×0 struct]    true
    5.2386 sec    [1×2 struct]    [1×0 struct]    true
```

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

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

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

```
ans =

  1×4 single row vector

  415.8962   82.4737  130.8474  129.3805


ans =

  1×4 single row vector

  235.2182    1.0000  117.0611   55.3500
```

Save App Session

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

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

See Also

Apps
Ground Truth Labeler

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

Related Examples

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

More About

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

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

Keyboard Shortcuts and Mouse Actions for Ground Truth Labeler

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

Label Definitions

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

Frame Navigation and Time Interval Settings

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

Task	Action
Go to the next frame	Right arrow
Go to the previous frame	Left arrow
Go to the last frame	• PC: **End** • Mac: Hold **Fn** and press the right arrow
Go to the first frame	• PC: **Home** • Mac: Hold **Fn** and press the left arrow
Navigate through time interval boxes and frame navigation buttons	**Tab**

Task	Action
Commit time interval settings	Press **Enter** within the active time interval box (**Start Time**, **Current**, or **End Time**)

Labeling Window

Perform labeling actions, such as adding, moving, and deleting regions of interest (ROIs), on the current image or video frame.

Task	Action
Undo labeling action	**Ctrl+Z**
Redo labeling action	**Ctrl+Y**
Select all rectangle and line ROIs	**Ctrl+A**
Select specific rectangle and line ROIs	Hold **Ctrl** and click the ROIs you want to select
Cut selected rectangle and line ROIs	**Ctrl+X**
Copy selected rectangle and line ROIs to clipboard	**Ctrl+C**
Paste copied rectangle and line ROIs • If a sublabel was copied, both the sublabel and its parent label are pasted. • If a parent label was copied, only the parent label is pasted, not its sublabels. For more details, see “Use Sublabels and Attributes to Label Ground Truth Data” (Computer Vision Toolbox).	**Ctrl+V**
Delete selected rectangle and line ROIs	**Delete**

Polyline Drawing

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

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

Polygon Drawing

Draw polygons to label pixels on a frame.

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

Zooming

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

App Sessions

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

See Also

Ground Truth Labeler

More About

- "Get Started with the Ground Truth Labeler" on page 2-2

3

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 3-18

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:

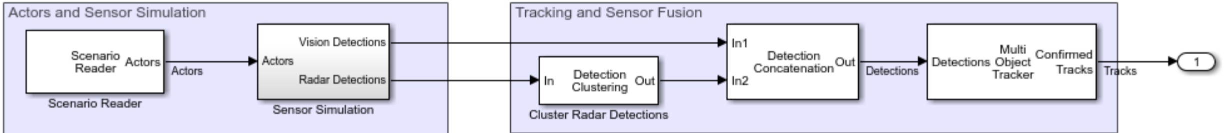
- Sensor coverages of vision and radar sensors
- Sensor detections of actors and lane boundaries
- Tracks of moving objects in the scenario

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

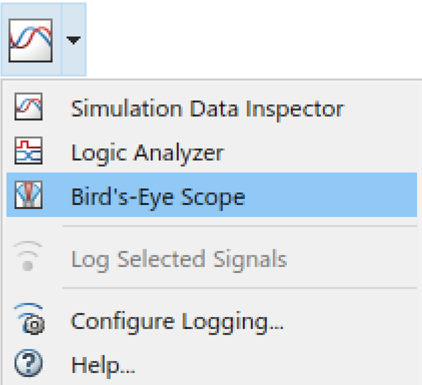
Open Model and Scope

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)
```

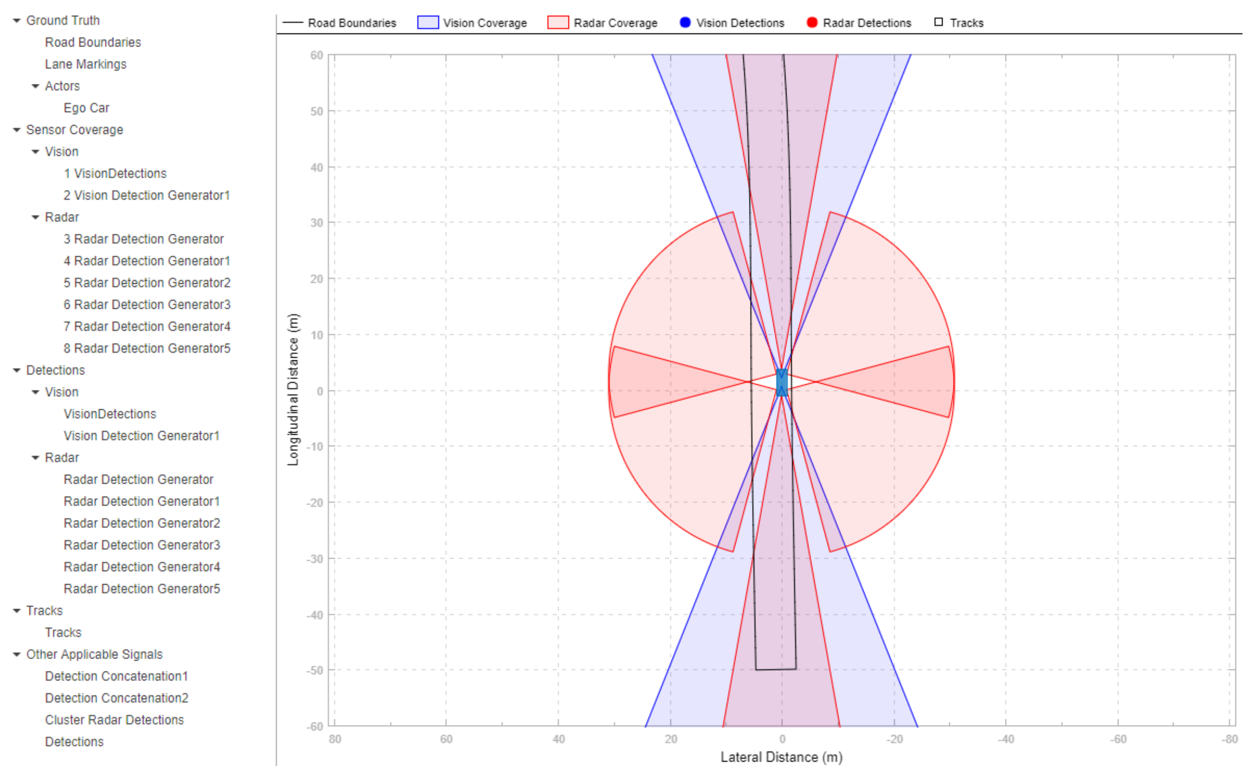


Open the scope. From the Simulink model toolbar, click the **Bird's-Eye Scope** button . If instead you see a button for a different model visualization tool, such as the **Simulation Data Inspector** or **Logic Analyzer** , click the arrow next to the displayed button and select **Bird's-Eye Scope**.



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.



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

Signal Group	Description	Signal Sources

Ground Truth	Road boundaries, lane markings, and actors in the scenario, including the ego vehicle
	
	You cannot modify this group or any of the signals within it.
		• Scenario Reader block
		• Vision Detection Generator and Radar Detection Generator blocks (for actor profile information only, such as the length, width, and height of actors)
		  • If actor profile information is not set or is inconsistent between blocks, the scope sets the actor profiles to the block defaults.
		  • The profile of the ego vehicle is always set to the block defaults.

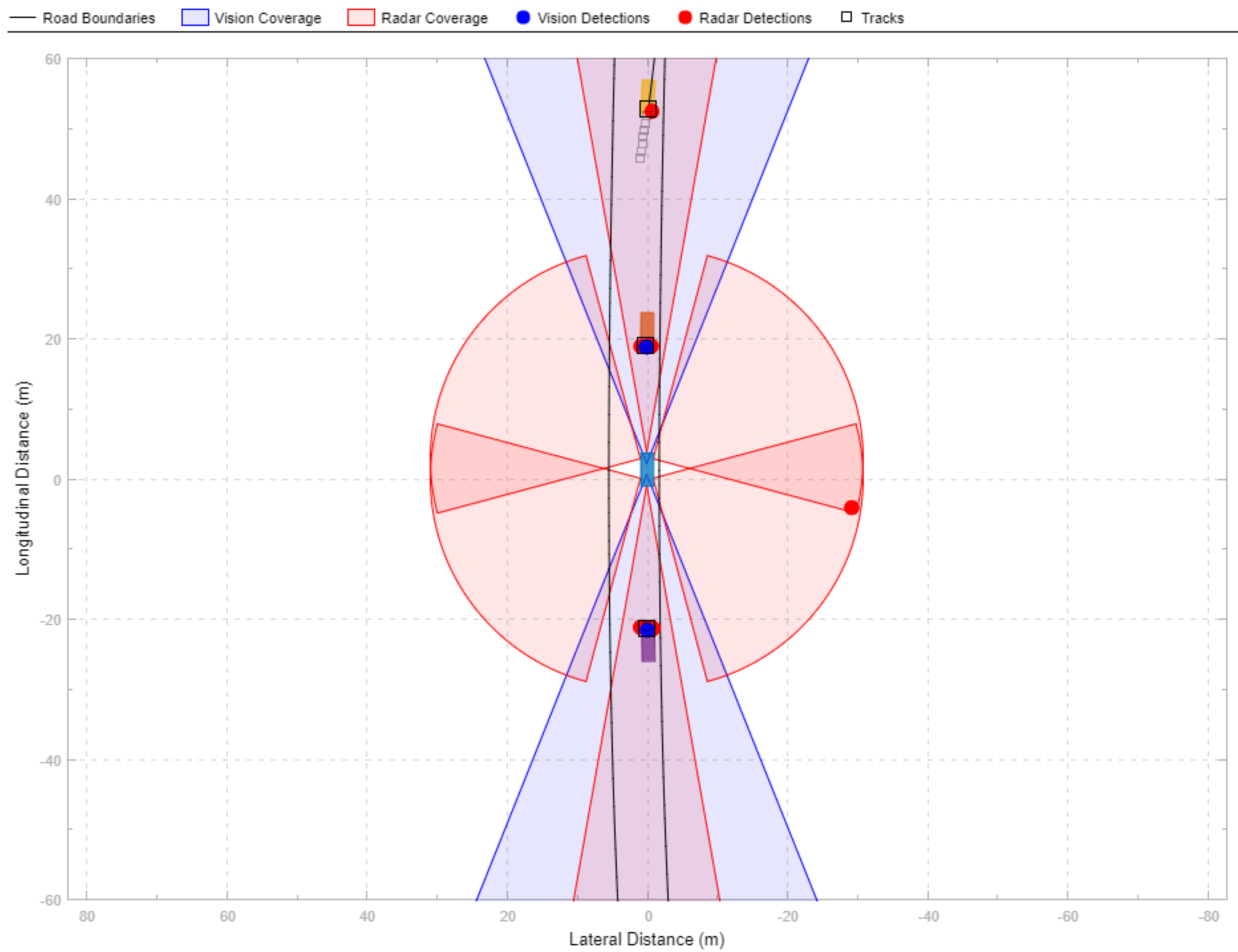
Sensor Coverage	Coverage areas of your vision and radar sensors, sorted into **Vision** and **Radar** subgroups You can move or modify these subgroups and their signals. You cannot move or modify the top-level **Sensor Coverage** group.	• Vision Detection Generator block • Radar Detection Generator block

Signal Group	Description	Signal Sources
Detections	Detections obtained from your vision and radar sensors, sorted into **Vision** and **Radar** subgroups You can move or modify these subgroups and their signals. You cannot move or modify the top-level **Detections** group.	• Vision Detection Generator block • Radar Detection Generator block
Tracks	Tracks of objects in the scenario	• Multi Object Tracker block
Other Applicable Signals	Signals that the scope cannot automatically group, such as ones that combine information from multiple sensors Signals in this group do not display during simulation.	• Blocks that combine or cluster signals (such as the Detection Concatenation block) • Nonvirtual Simulink buses containing position and velocity information for detections and tracks

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 toolstrip, 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.

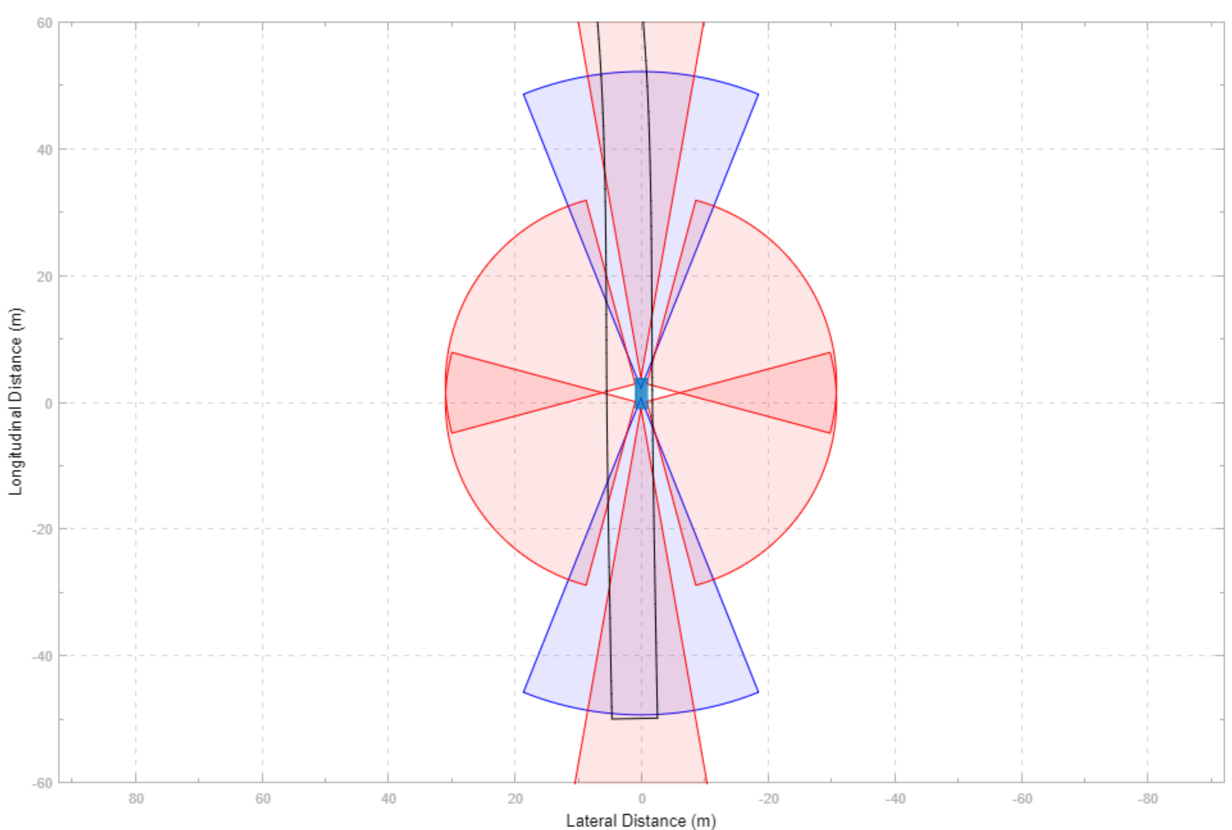
When you modify block parameters, you can rerun the simulation without having to find signals again. If you add or remove blocks, ports, or signal lines, then you must click **Find Signals** again before rerunning the simulation.

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 | Scenario Reader | Vision Detection Generator

Related Examples

- “Sensor Fusion Using Synthetic Radar and Vision Data in Simulink”

- “Lateral Control Tutorial”
- “Autonomous Emergency Braking with Sensor Fusion”
- “Test Open-Loop ADAS Algorithm Using Driving Scenario” on page 4-72
- “Test Closed-Loop ADAS Algorithm Using Driving Scenario” on page 4-78

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 3-15
"Constant Acceleration Model" on page 3-16

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

State Equations

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

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

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

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

If you define the state as

$$x_1 = x$$

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

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

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

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

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

v_k 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 differential equations. A state-space representation is a mathematical model of a physical system where the inputs, outputs, and state variables are related by first-order coupled equations.

Measurement Models

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

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

$$z_k = H_k x_k + w_k$$

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

Linear Kalman Filter Equations

Without noise, the dynamic equations are

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

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

$$z_k = H_k x_k$$

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

Filter Loop

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

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

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

   Propagate the covariance matrix as well.

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

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

   Then predict the measurement at the updated time.

   $$z_{k+1|k} = H_{k+1} x_{k+1|k}$$
2. Use the difference between the actual measurement and predicted measurement to correct the state at the updated time. The correction requires computing the Kalman gain. To do this, first compute the measurement prediction covariance (innovation)

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

   Then the Kalman gain is

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

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

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

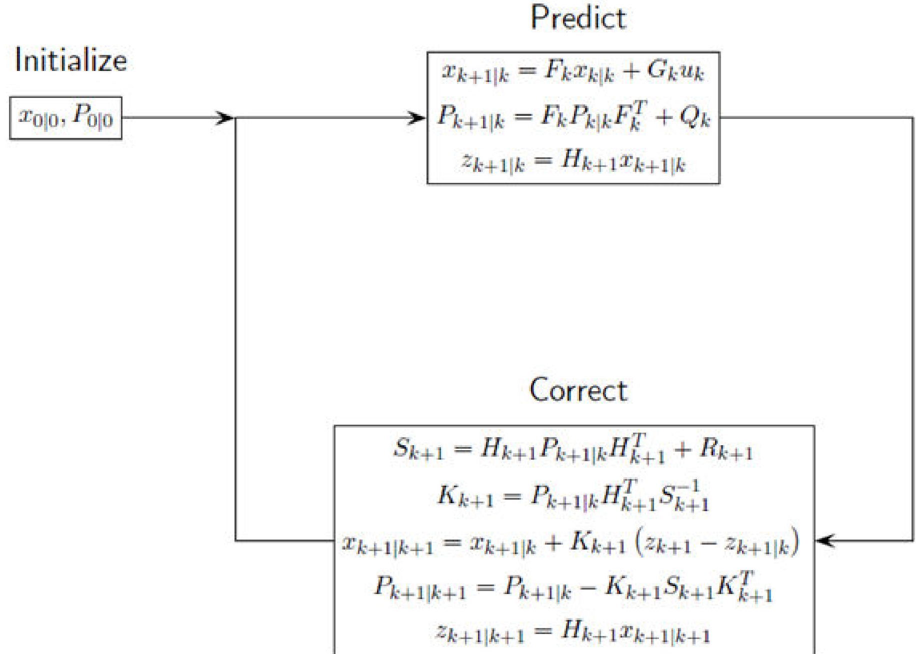
   where K_{k+1} is the Kalman gain. The corrected state is often called the *a posteriori* estimate of the state because it is derived after the measurement is included.

   Correct the state covariance matrix

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

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

This figure summarizes the Kalman loop operations.



Constant Velocity Model

The linear Kalman filter contains a built-in linear constant-velocity motion model. Alternatively, you can specify the transition matrix for linear motion. The state update at the next time step is a linear function of the state at the present time. In this filter, the

measurements are also linear functions of the state described by a measurement matrix. For an object moving in 3-D space, the state is described by position and velocity in the *x*-, *y*-, and *z*-coordinates. The state transition model for the constant-velocity motion is

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

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

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

Constant Acceleration Model

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

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

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

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

Extended Kalman Filters

In this section...

"State Update Model" on page 3-18
"Measurement Model" on page 3-19
"Extended Kalman Filter Loop" on page 3-19
"Predefined Extended Kalman Filter Functions" on page 3-20

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:

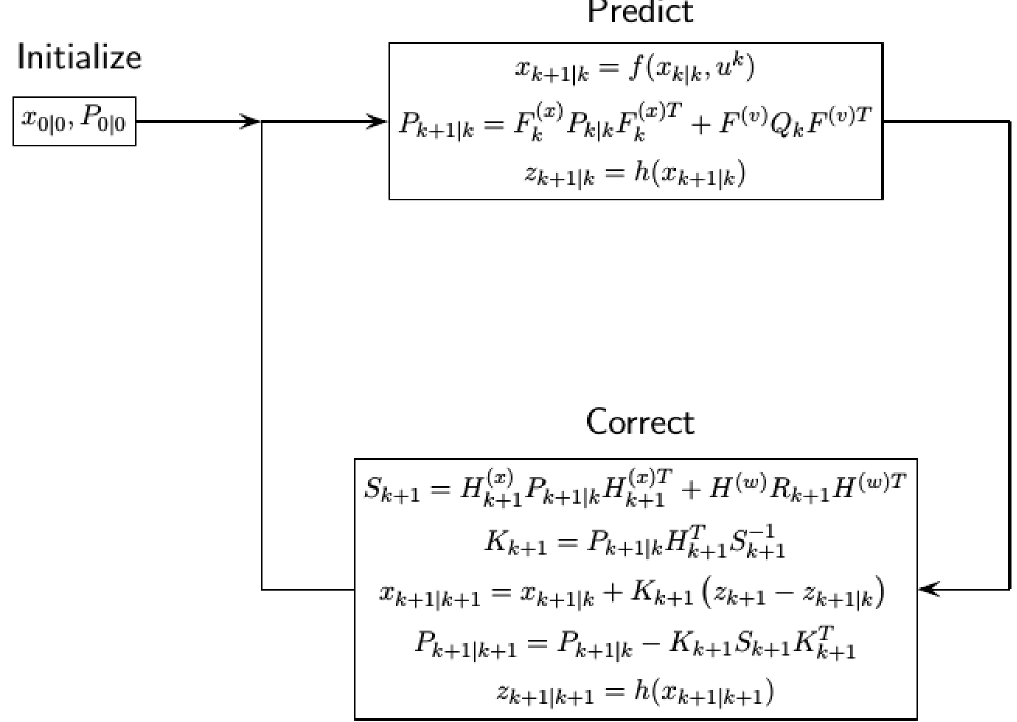
$$z_k = h(x_k, t) + v_k$$

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

Extended Kalman Filter Loop

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

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



Predefined Extended Kalman Filter Functions

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

Motion Model	Function Name	Function Purpose
Constant velocity	`constvel`	Constant-velocity state update model
	`constveljac`	Constant-velocity state update Jacobian

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

4

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.

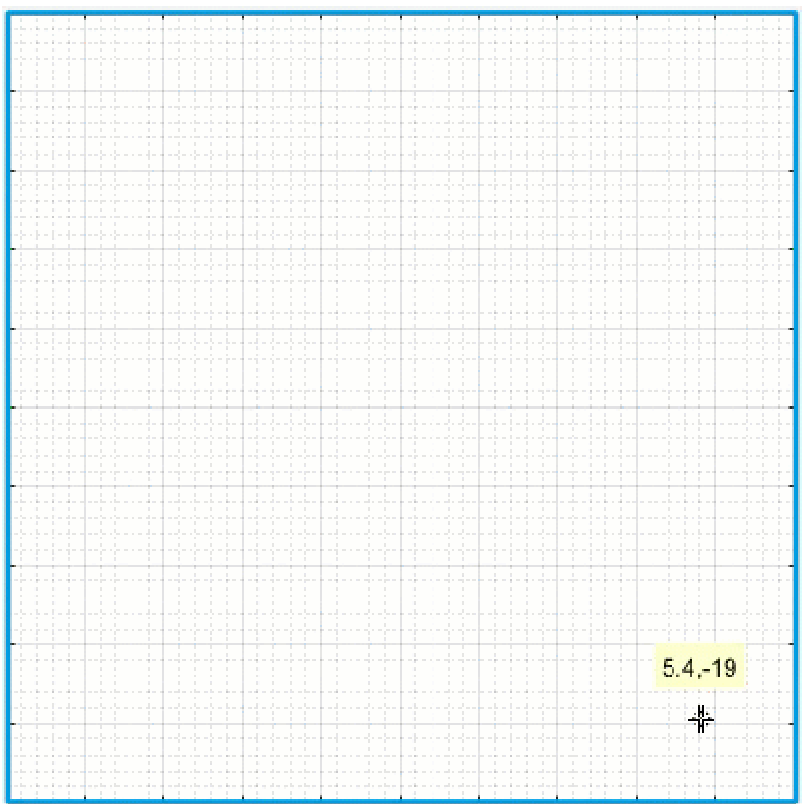
This example covers the entire workflow for creating a scenario and generating synthetic detections. Alternatively, you can generate detections from prebuilt scenarios. For more details, see “Generate Synthetic Detections from a Prebuilt Driving Scenario” on page 4-18.

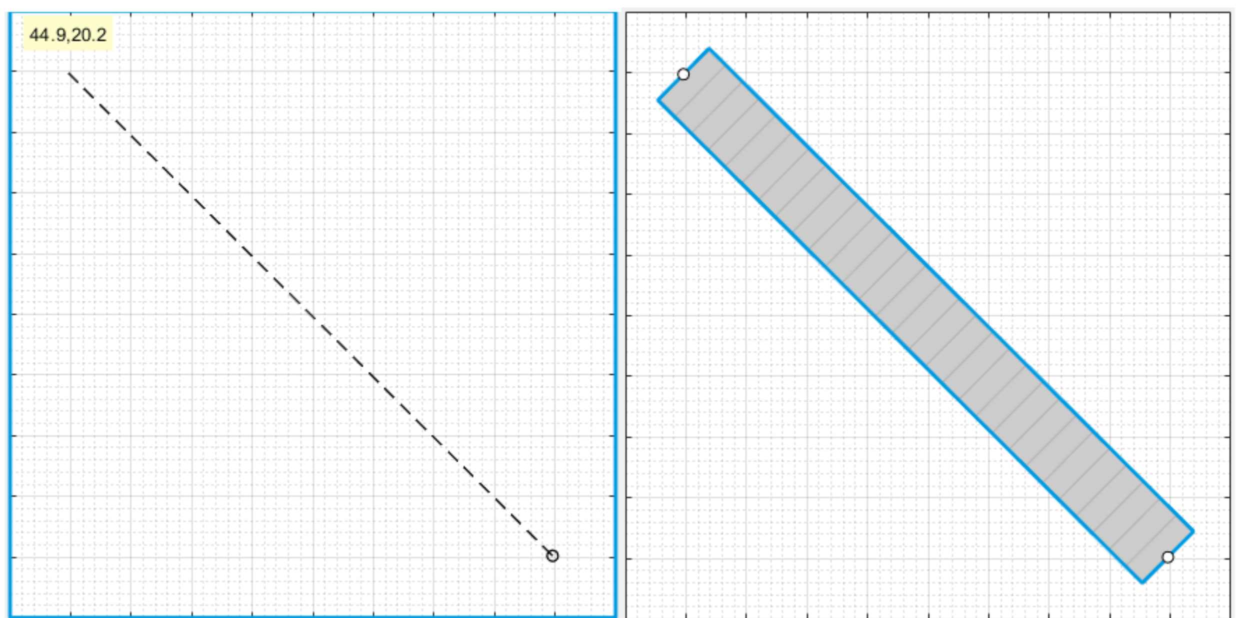
Create a New Driving Scenario

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

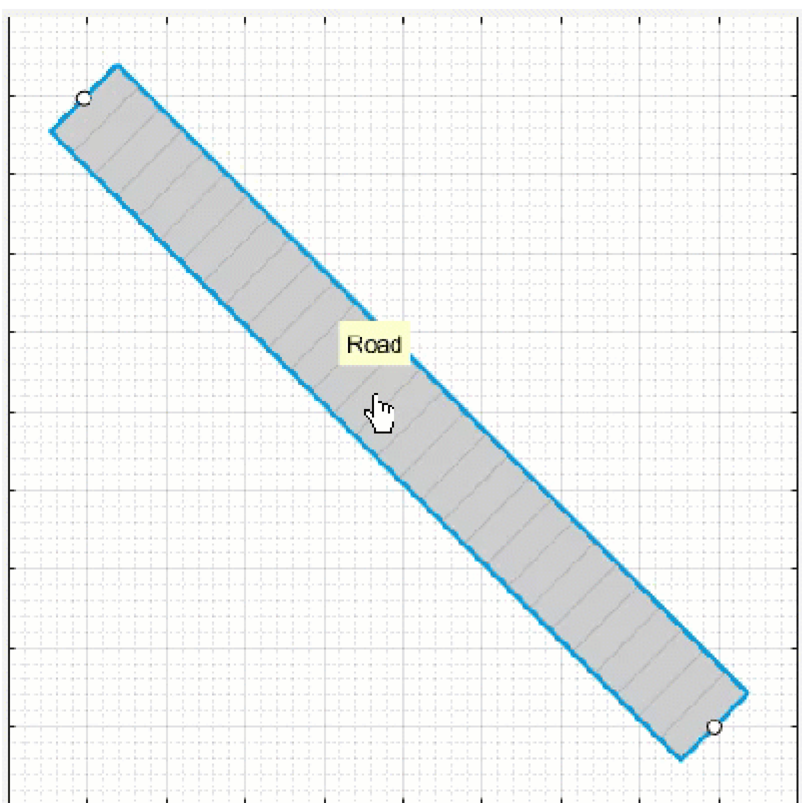
Add a Road

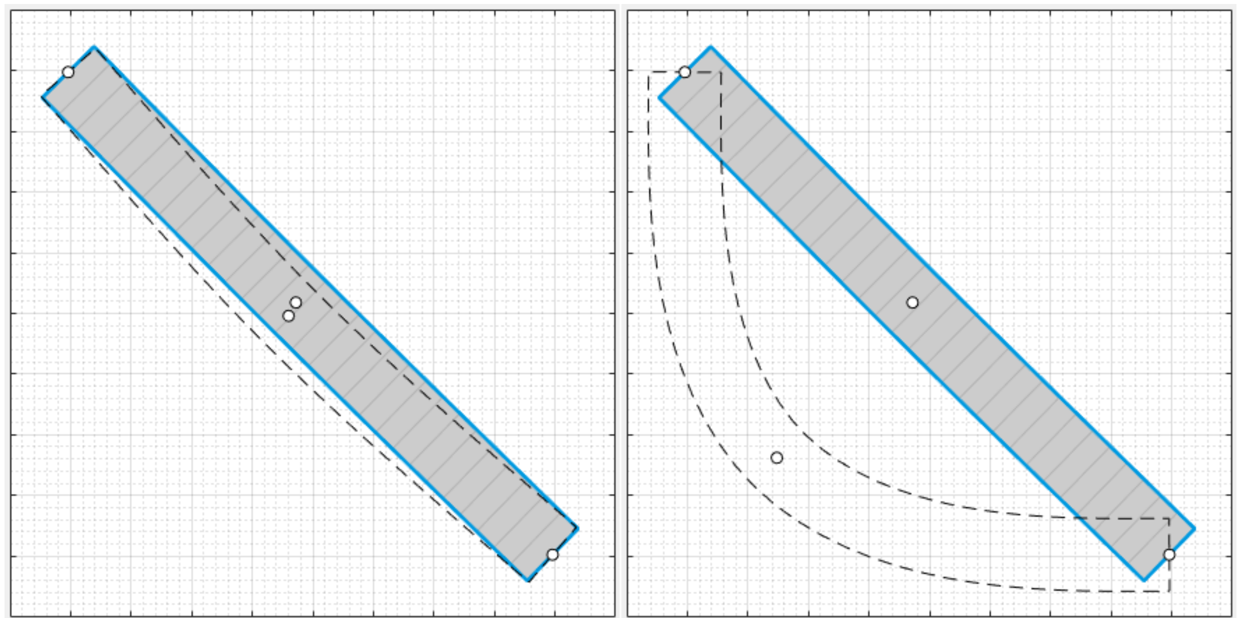
Add a curved road to the scenario canvas. From the app toolstrip, click **Add Road**. Then click one corner of the canvas, extend the road to the opposite corner, and double-click to create the road.


5.4,-19



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.

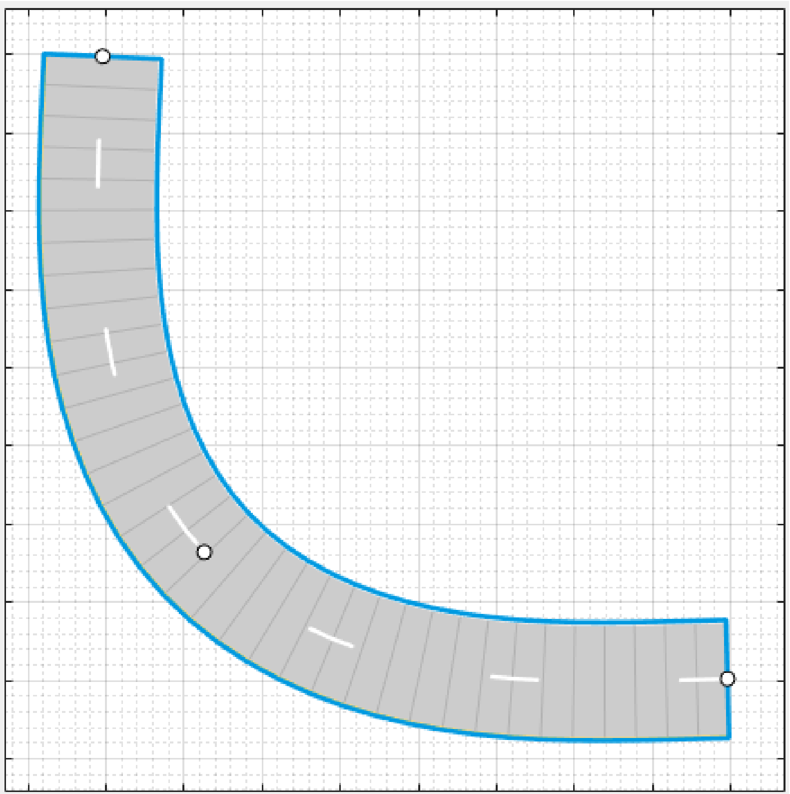

Road



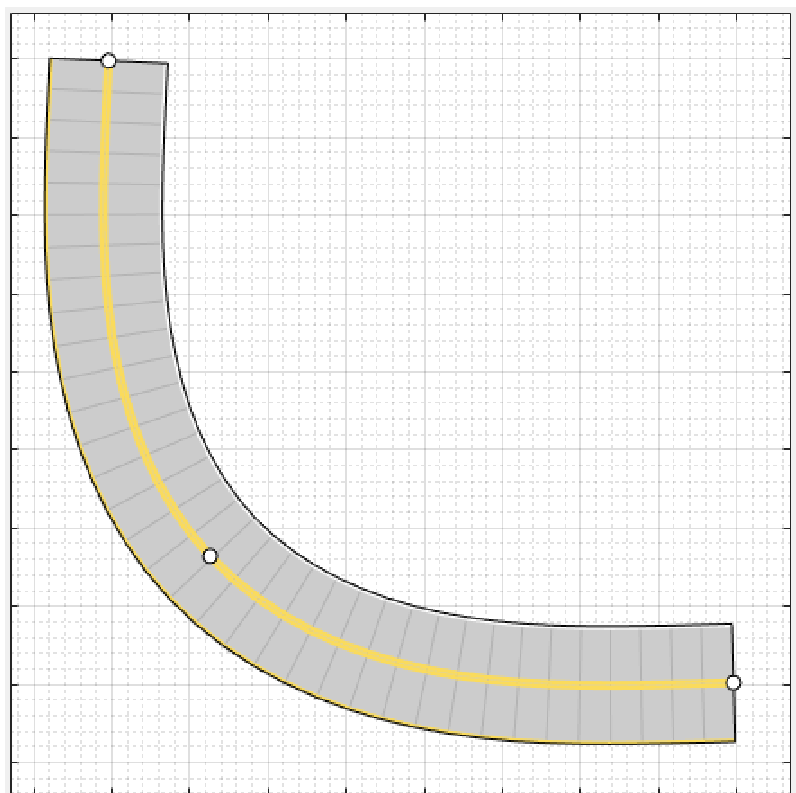
To adjust the road further, you can click and drag any of the road centers. To create more complex curves, add more road centers.

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`.



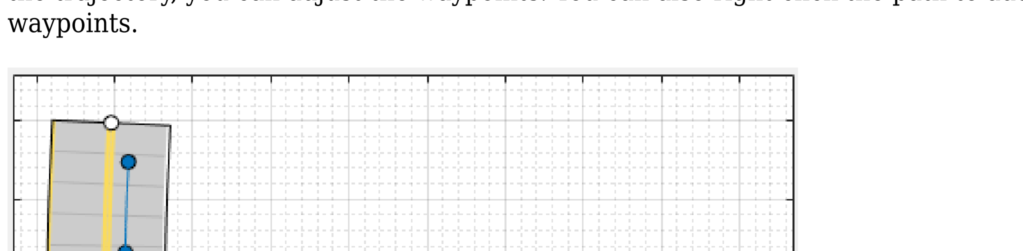
Add Vehicles

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

Add Ego Vehicle

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

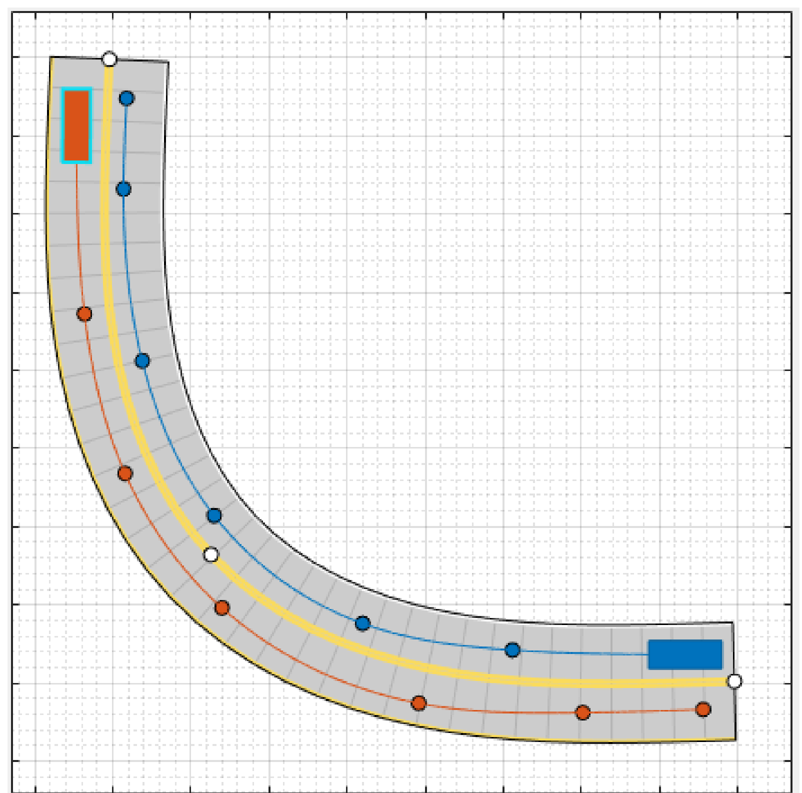
the trajectory, you can adjust the waypoints. You can also right-click the path to add new waypoints.



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

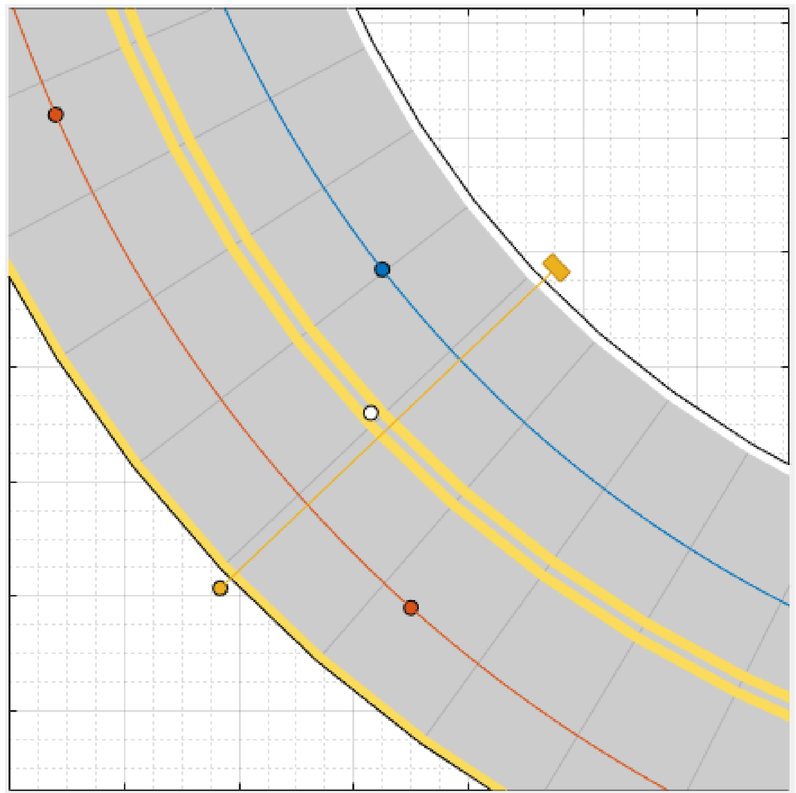
Add Second Car

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

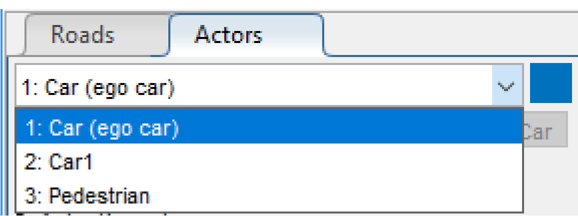


Add a Pedestrian

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



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



Add Sensors

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

Add Camera

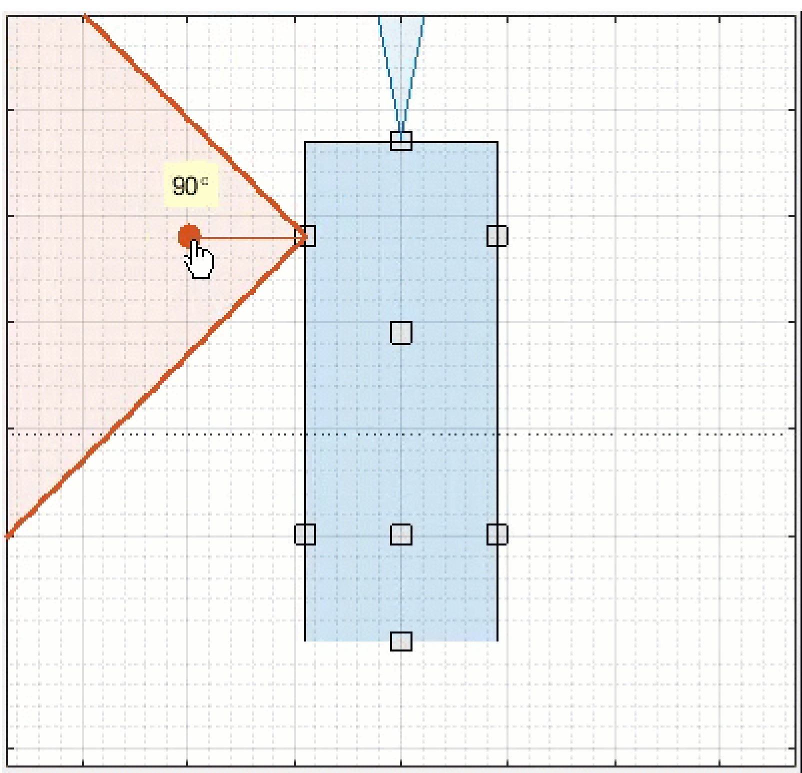
From the app toolstrip, click **Add Camera**. The sensor canvas shows standard locations at which to place sensors. Click the front-most predefined sensor location to add a camera sensor to the front bumper of the ego vehicle. To place sensors more precisely, you can disable snapping options. In the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

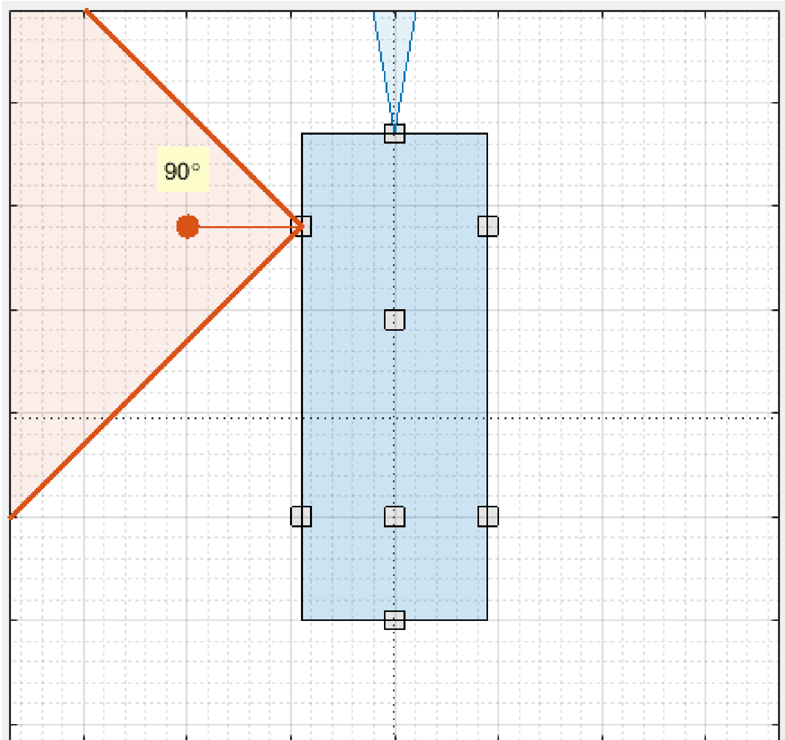
By default, the camera detects only actors and not lanes. To enable lane detections, on the **Sensors** tab in the left pane, expand the **Detection Parameters** section and set **Detection Type** to `Objects & Lanes`. Then expand the **Lane Settings** section and update the settings as needed.

Add Radar

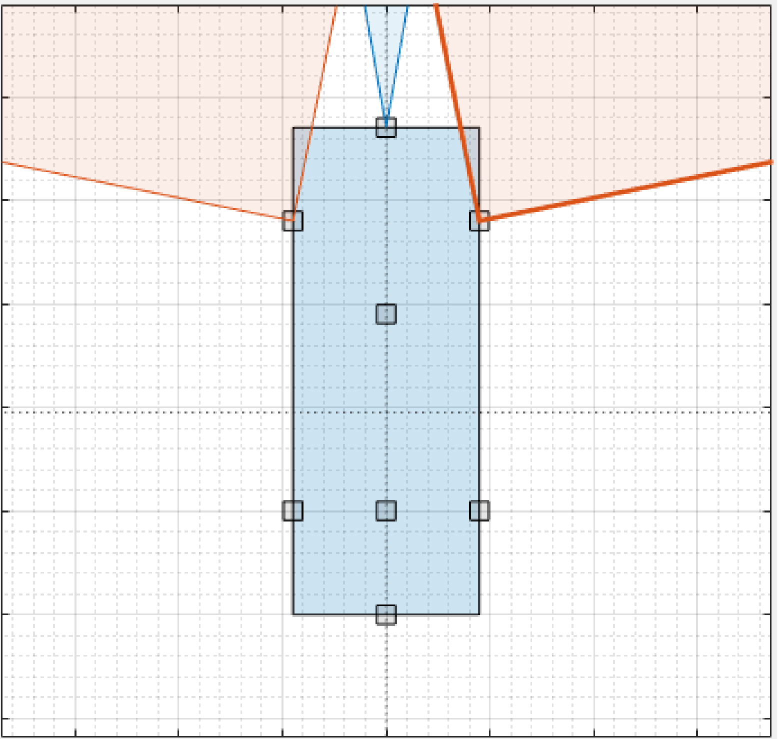
Snap a radar sensor to the front-left wheel. Right-click the predefined sensor location for the wheel and select **Add Radar**. By default, sensors added to the wheels are short range.

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


90°



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

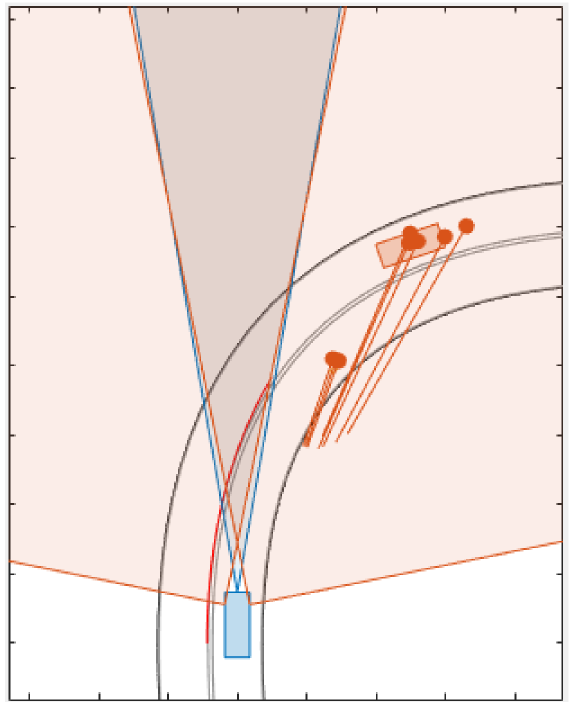


The camera and radar sensors now provide overlapping coverage of the front of the ego vehicle.

Generate 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 vehicle. The **Bird's-Eye Plot** displays the detections.



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

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

Export Sensor 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**. 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 Scenario

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

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

```
drivingScenarioDesigner(scenarioFileName)
```

If you are developing a driving algorithm in Simulink, you can use a Scenario Reader block to read the roads and actors from this file into your model. However, because the block does not support reading in sensor data, the sensors you created are ignored. You must instead create the sensors within your model, using blocks such as Radar Detection Generator and Vision Detection Generator.

See Also

Apps
Driving Scenario Designer

Blocks
Radar Detection Generator | Scenario Reader | Vision Detection Generator

Objects
`drivingScenario` | `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
- "Test Open-Loop ADAS Algorithm Using Driving Scenario" on page 4-72
- "Test Closed-Loop ADAS Algorithm Using Driving Scenario" on page 4-78

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

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

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

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

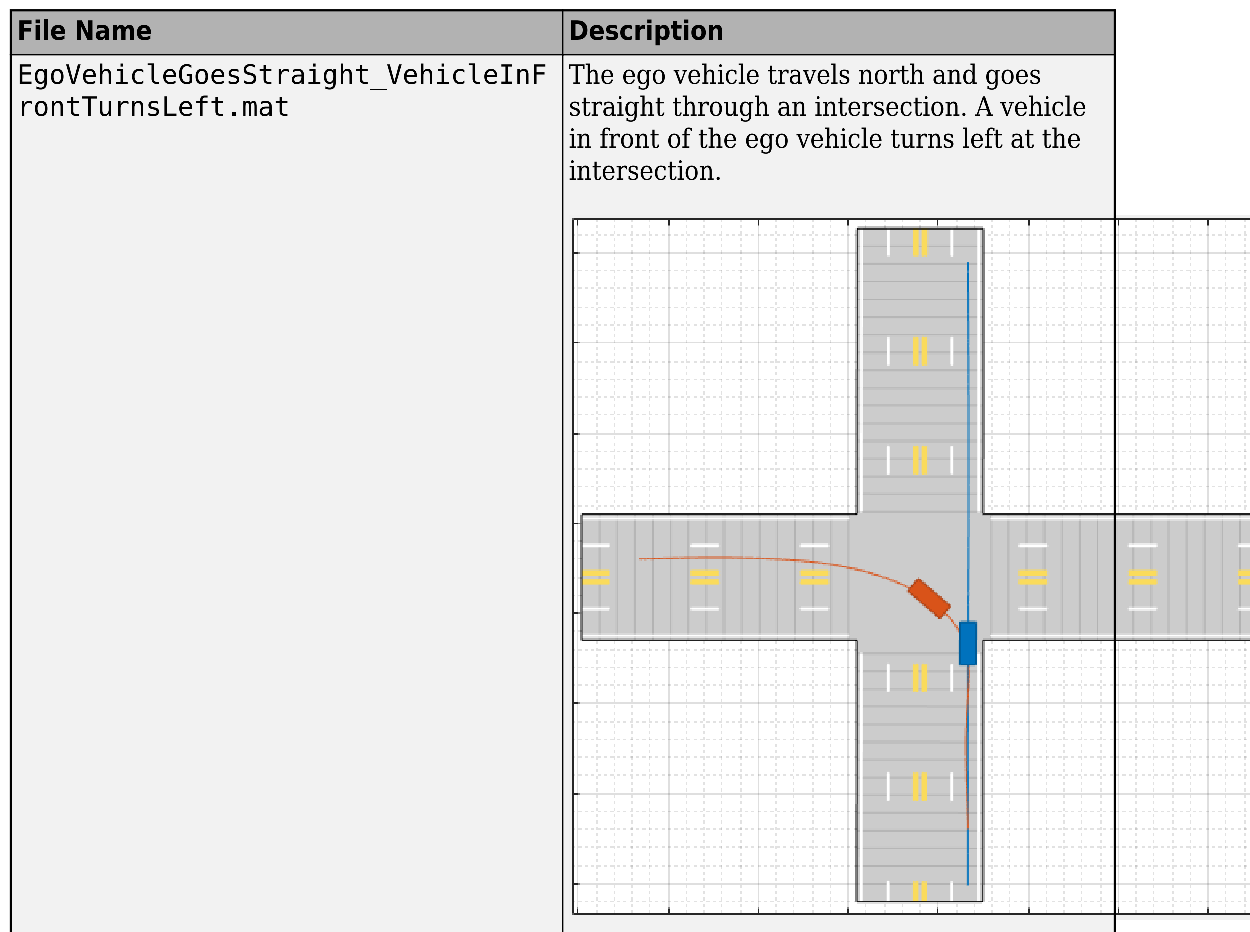
File Name	Description
Roundabout.mat	The ego vehicle travels north and crosses the path of a pedestrian while entering a roundabout. The ego vehicle then passes a truck as both vehicles drive through the roundabout.

Turns

These scenarios involve turns at four-way intersections.

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

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

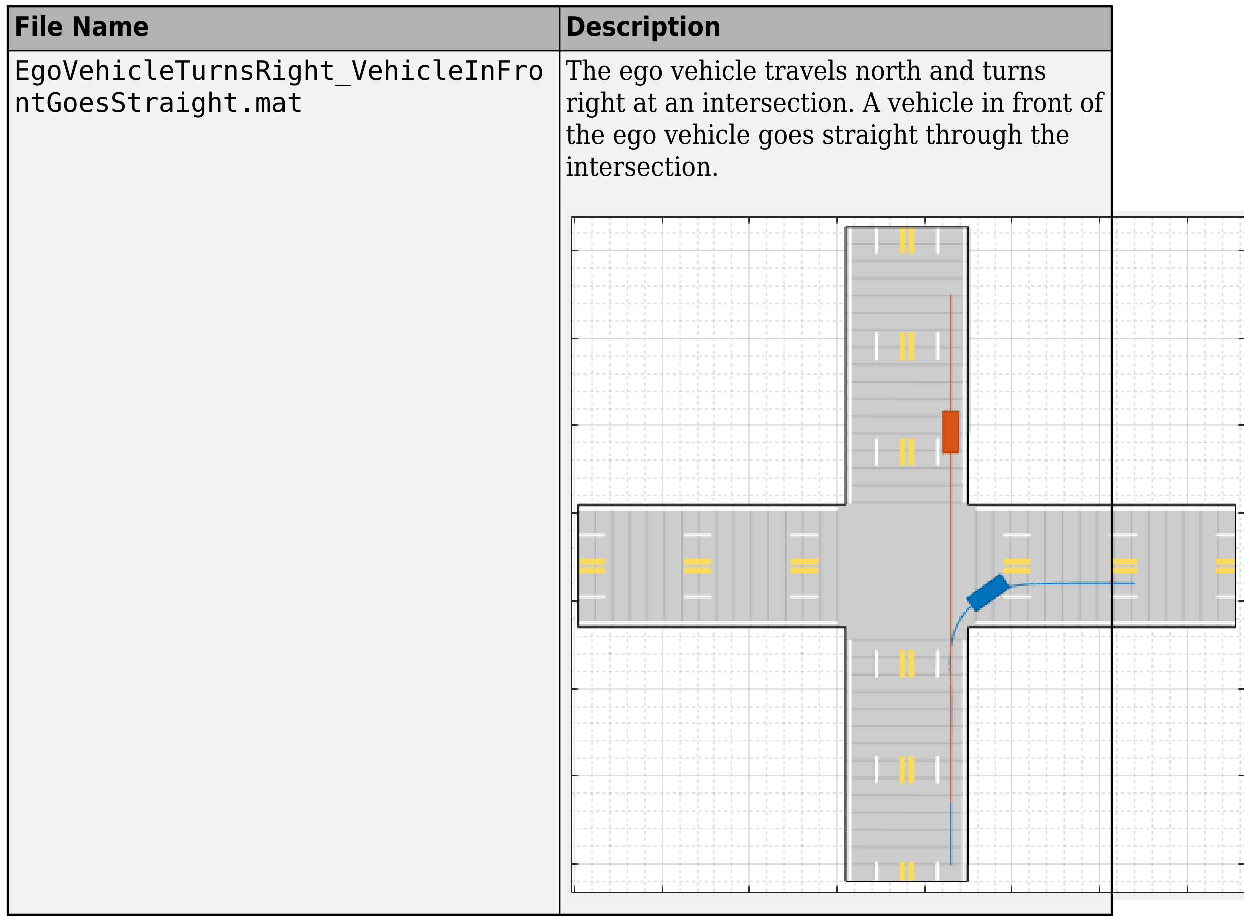
File Name	Description
EgoVehicleGoesStraight_VehicleInFrontTurnsLeft.mat	The ego vehicle travels north and goes straight through an intersection. A vehicle in front of the ego vehicle turns left at the intersection. 

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

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

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

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

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

U-Turns

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

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

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

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

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

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

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

Modify Scenario

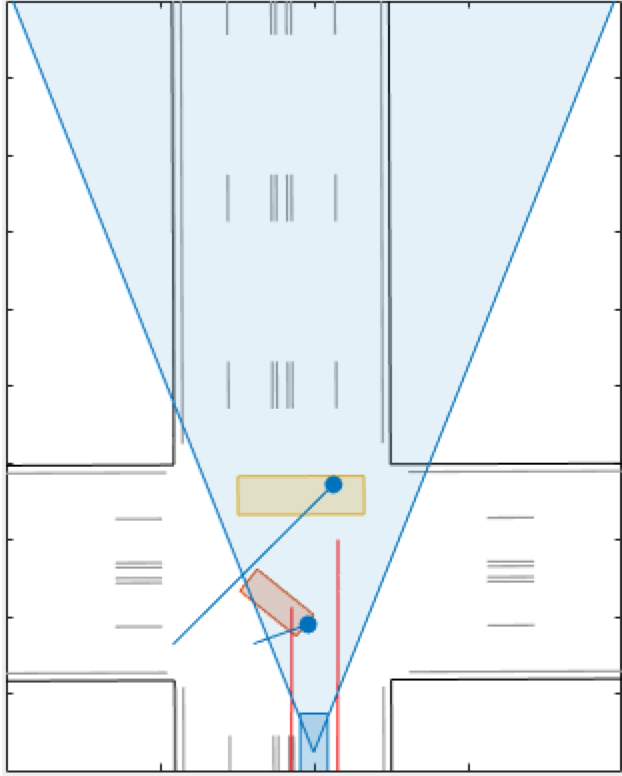
After you choose a scenario, you can modify the parameters of the roads and actors. For example, from the **Actors** tab on the left, you can change the position or velocity of the ego vehicle or other actors. From the **Roads** tab, you can change the width of the lanes or the type of lane markings.

You can also add or modify sensors. For example, from the **Sensors** tab, you can change the detection parameters or the positions of the sensors. By default, in Euro NCAP

scenarios, the ego vehicle does not contain sensors. All other prebuilt scenarios have at least one front-facing camera or radar sensor, set to detect lanes and objects.

Generate Synthetic Detections

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



Export the detections.

- To export 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** > **Scenario File As** to save the scenario file.

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

```
drivingScenarioDesigner(scenarioFileName)
```

If you are developing a driving algorithm in Simulink, you can use a Scenario Reader block to read the roads and actors from this file into your model. However, because the block does not support reading in sensor data, the sensor you created is ignored. You must instead create the sensors within your model, using blocks such as Radar Detection Generator and Vision Detection Generator.

See Also

Apps
Driving Scenario Designer | Radar Detection Generator | Vision Detection Generator

Classes
`drivingScenario` | `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
- “Test Open-Loop ADAS Algorithm Using Driving Scenario” on page 4-72
- “Test Closed-Loop ADAS Algorithm Using Driving Scenario” on page 4-78

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 autonomous emergency braking (AEB), emergency lane keeping (ELK), and lane keep assist (LKA) systems.

Choose a Euro NCAP Scenario

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

In the app, the Euro NCAP scenarios are stored as MAT-files and organized into folders. To open a Euro NCAP file, 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.

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

Autonomous Emergency Braking

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

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

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

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

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

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

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

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

Emergency Lane Keeping

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

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

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

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

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

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

Lane Keep Assist

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

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

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

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

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

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

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

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

Modify Scenario

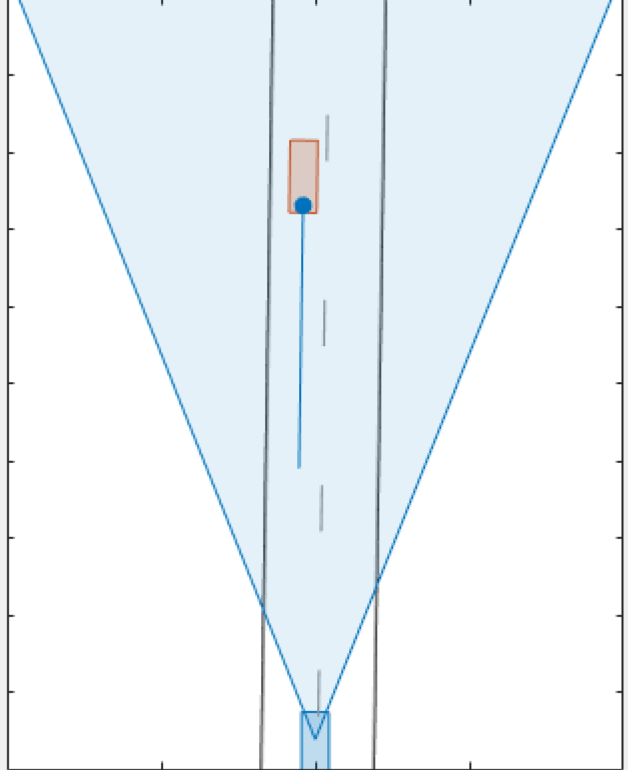
By default, in Euro NCAP scenarios, the ego vehicle does not contain sensors. If you are testing a vehicle sensor, from the app toolstrip, click **Add Camera** or **Add Radar** to add a sensor to the ego vehicle. Then, on the **Sensor** tab, adjust the parameters of the sensors to match your sensor model.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

You can also adjust the parameters of the roads and actors in the scenario. For example, from the **Actors** tab on the left, you can change the position or velocity of the ego vehicle

or other actors. From the **Roads** tab, you can change the width of lanes or the type of lane markings.

Generate Synthetic Detections

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



Export the detections.

- To export 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** > **Scenario File As** to save the scenario file.

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

```
drivingScenarioDesigner(scenarioFileName)
```

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.

If you are developing a driving algorithm in Simulink, you can use a Scenario Reader block to read the roads and actors from this file into your model. However, because the block does not support reading in sensor data, the sensor you created is ignored. You must instead create the sensors within your model, using blocks such as Radar Detection Generator and Vision Detection Generator.

References

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

[2] European New Car Assessment Programme. *Euro NCAP AEB C2C Test Protocol*. Version 2.0.1. January 2018.

[3] European New Car Assessment Programme. *Euro NCAP LSS Test Protocol*. Version 2.0.1. January 2018.

See Also

Apps
Driving Scenario Designer

Blocks
Radar Detection Generator | Scenario Reader | Vision Detection Generator

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

More About

- "Build a Driving Scenario and Generate Synthetic Detections" on page 4-2
- "Generate Synthetic Detections from a Prebuilt Driving Scenario" on page 4-18
- "Autonomous Emergency Braking with Sensor Fusion"
- "Test Open-Loop ADAS Algorithm Using Driving Scenario" on page 4-72
- "Test Closed-Loop ADAS Algorithm Using Driving Scenario" on page 4-78

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 developed in MATLAB. Alternatively, to test driving algorithms developed in Simulink, you can use a Scenario Reader block to read the road network and actors into a model.

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

Import OpenDRIVE File

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

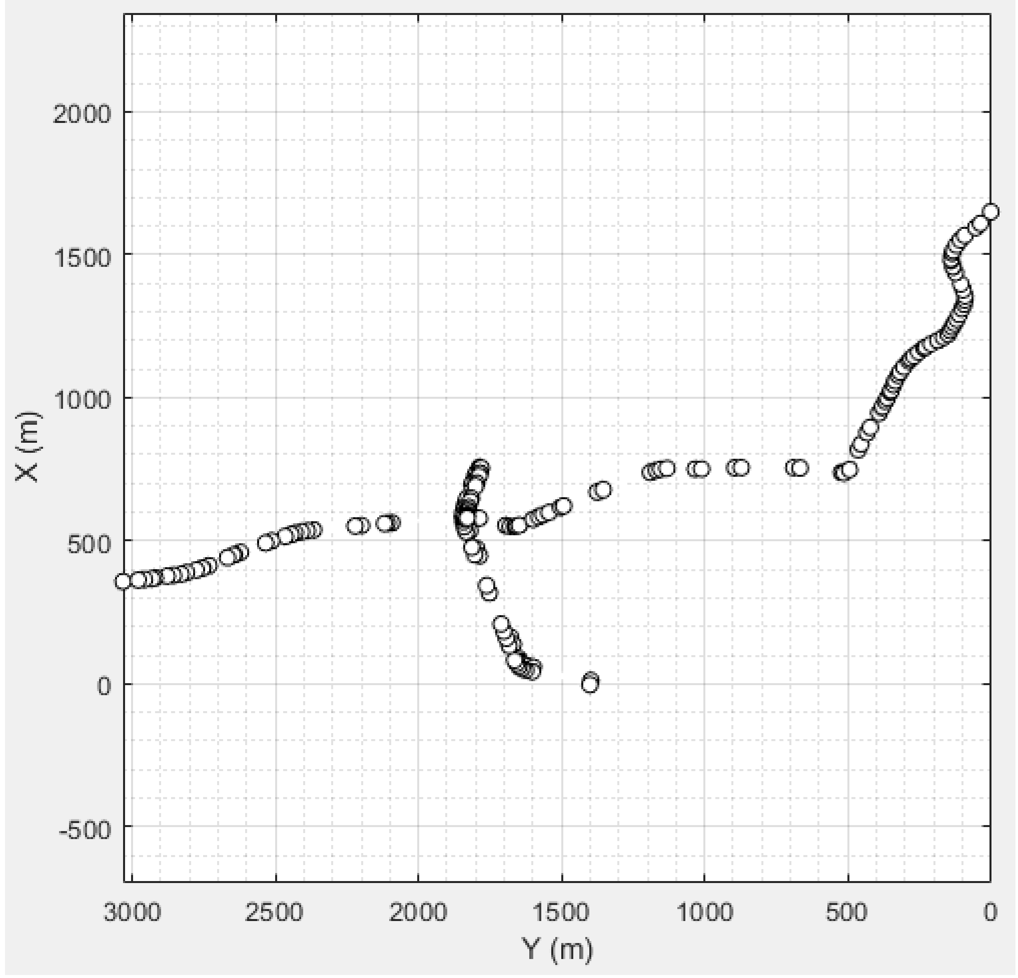
To import an OpenDRIVE file, from the app toolstrip, select **Open** > **OpenDRIVE Road Network**. The file you select must be a valid OpenDRIVE file of type `.xodr` or `.xml`. In addition, the file must conform with OpenDRIVE format specification version 1.4H.

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 scenario file, the app prompts you to save your current driving scenario.

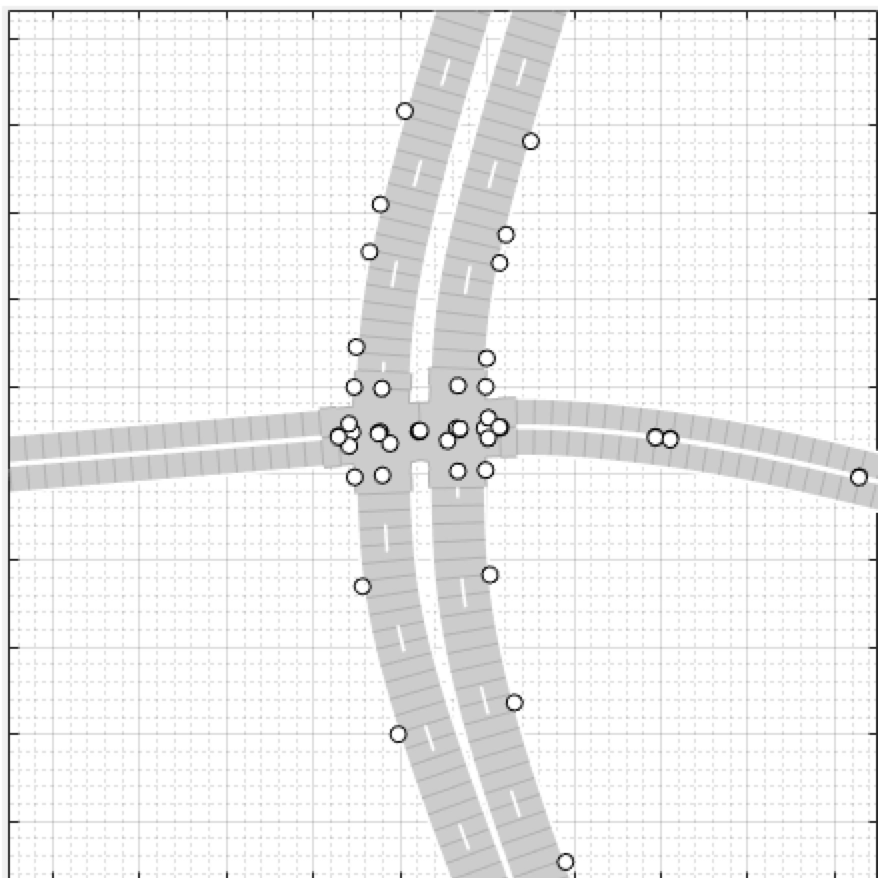
The **Scenario Canvas** of the app displays the imported road network.



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

Inspect Roads

The imported road network shows a pair of two-lane roads intersecting with a single two-lane road.



Verify that the road network imported as expected, keeping in mind the following limitations and behaviors within the app.

OpenDRIVE Import Limitations

The **Driving Scenario Designer** app does not support all components of the OpenDRIVE specification.

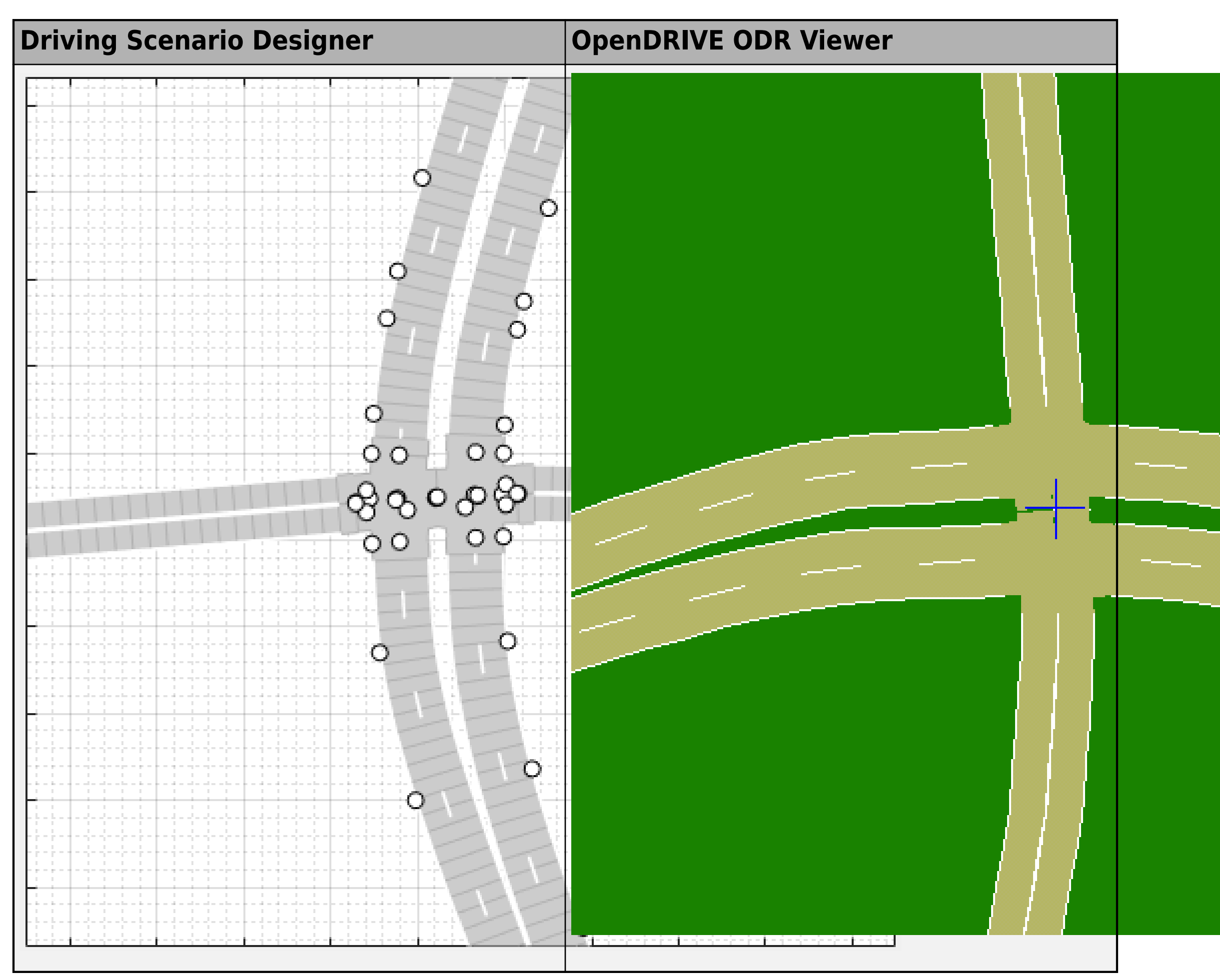
- You can import only lanes and roads. The import of road objects and traffic signals is not supported.
- OpenDRIVE files containing large road networks can take up to several minutes to load. In addition, these road networks can cause slow interactions on the app canvas.

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. Lanes with these marking styles are imported as unmarked.

Road Orientation

In the **Driving Scenario Designer** app, the orientation of roads can differ from the orientation of roads in other tools that display OpenDRIVE roads. The table shows this difference in orientation between the app and the OpenDRIVE ODR Viewer.

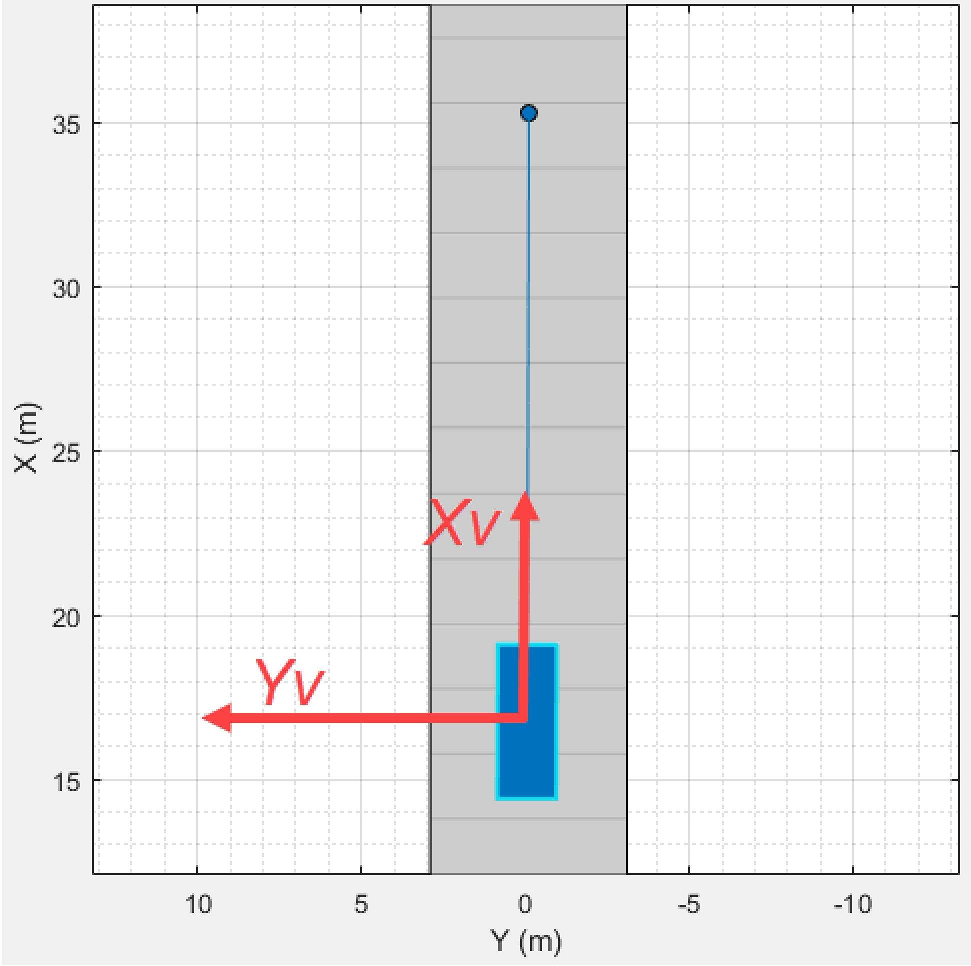


In the OpenDRIVE ODR viewer, the *X*-axis runs along the bottom of the viewer, and the *Y*-axis runs along the left side of the viewer.

In the **Driving Scenario Designer** app, the *Y*-axis runs along the bottom of the canvas, and the *X*-axis runs along the left side of the canvas. This world coordinate system in the app aligns with the vehicle coordinate system (X_V,Y_V) used by vehicles in the driving scenario, where:

- The X_V-axis (longitudinal axis) points forward from a vehicle in the scenario.

- The Y_V-axis (lateral axis) points to the left of the vehicle, as viewed when facing forward.



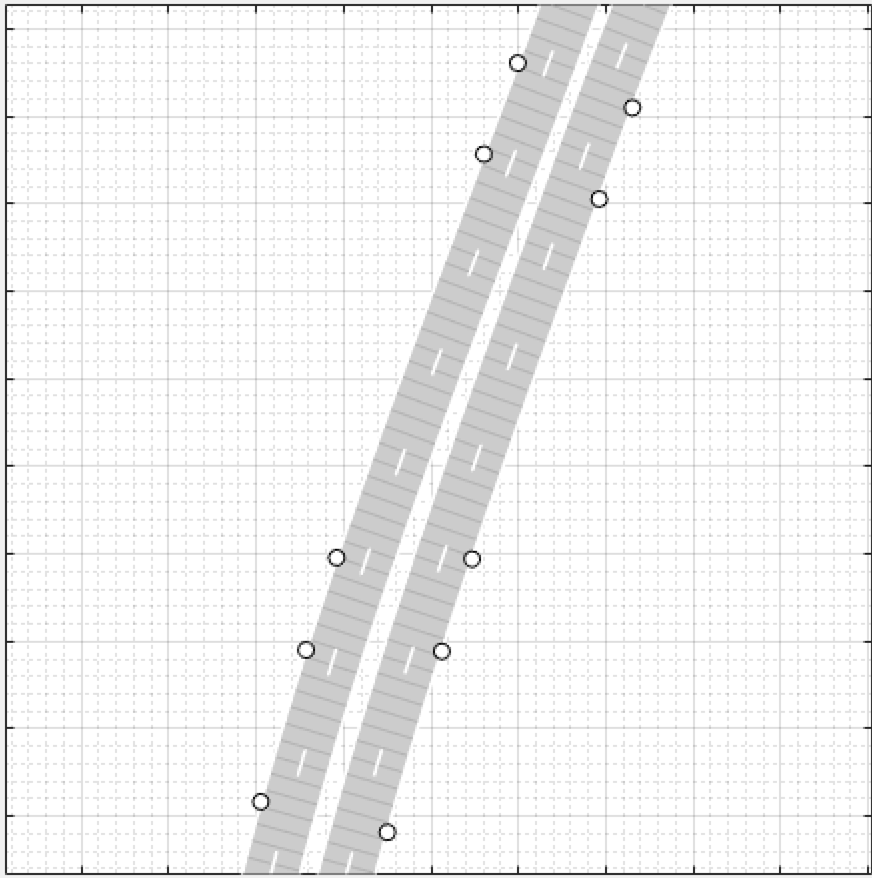
For more details about the coordinate systems, see “Coordinate Systems in Automated Driving Toolbox” on page 1-2.

Road Centers on Edges

In the **Driving Scenario Designer** app, the location and orientation of roads are defined by road centers. When you create a road in the app, the road centers are always in the middle of the road. When you import OpenDRIVE road networks into the app, however, some roads have their road centers on the road edges. This behavior occurs when the OpenDRIVE roads are explicitly specified as being right lanes or left lanes.

Consider the divided highway in the imported OpenDRIVE file.

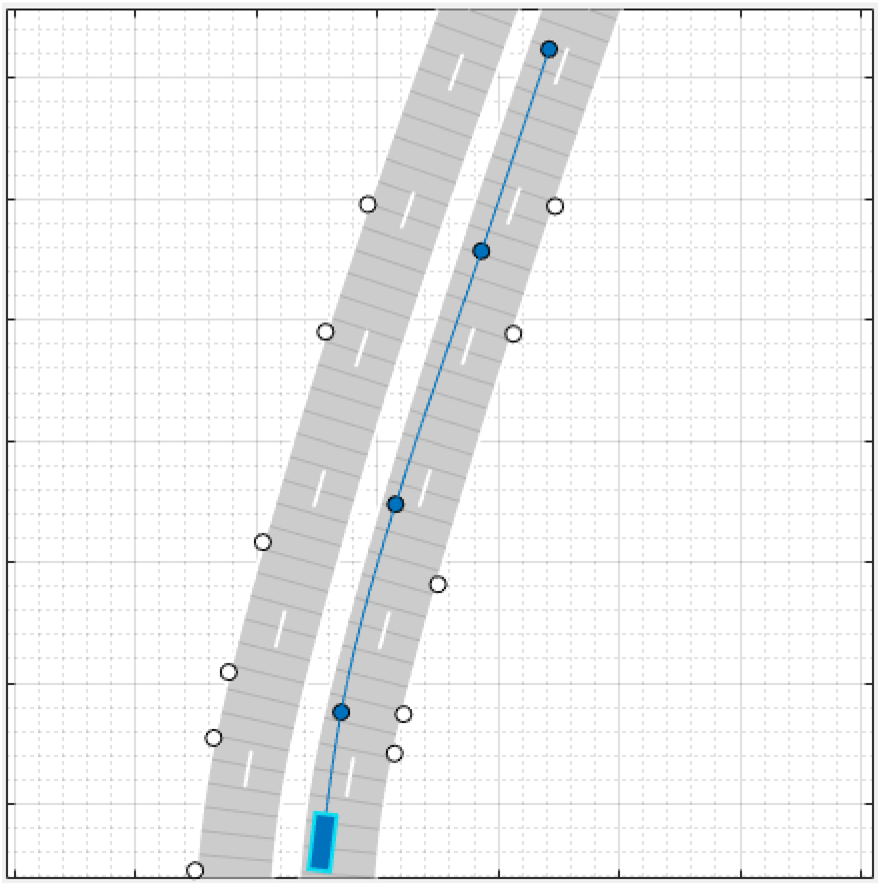
- The lanes on the right side of the highway have their road centers on the right edge.
- The lanes on the left side of the highway have their road centers on the left edge.



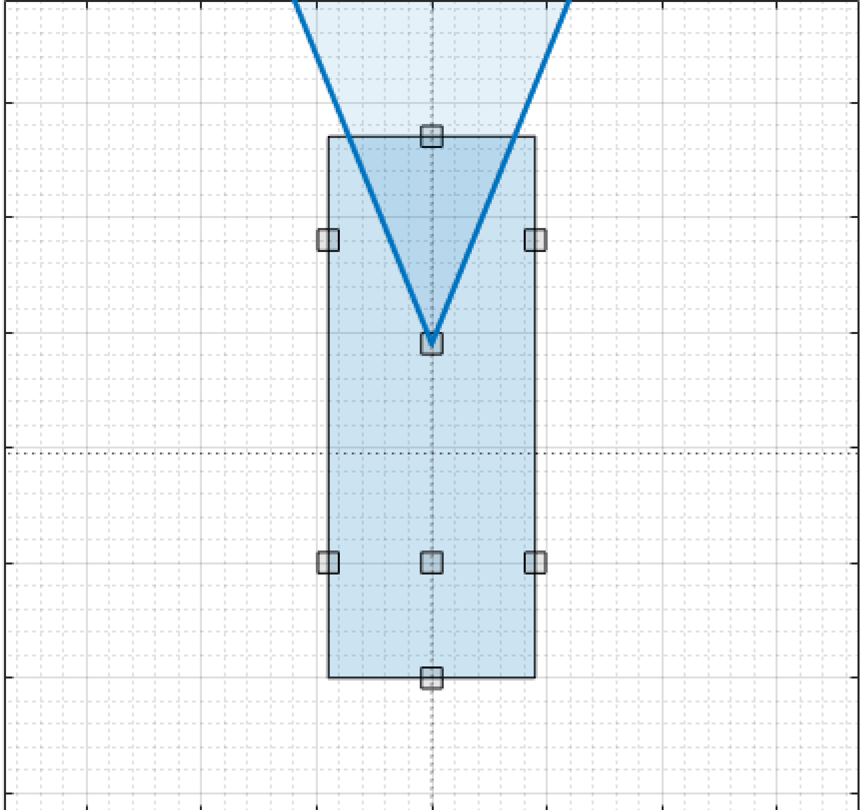
Add Actors and Sensors to Scenario

You can add actors and sensors to a scenario containing OpenDRIVE roads. However, you cannot add other roads to the scenario. If a scenario contains an OpenDRIVE road network, the **Add Road** button in the app toolstrip is disabled. In addition, you cannot import additional OpenDRIVE road networks into a scenario.

Add an ego vehicle to the scenario by right-clicking one of the roads in the canvas and selecting **Add Car**. To specify the trajectory of the car, right-click the car in the canvas, select **Add Waypoints**, and add waypoints along the road for the car to pass through. After you add the last waypoint along the road, press **Enter**. The car autorotates in the direction of the first waypoint.



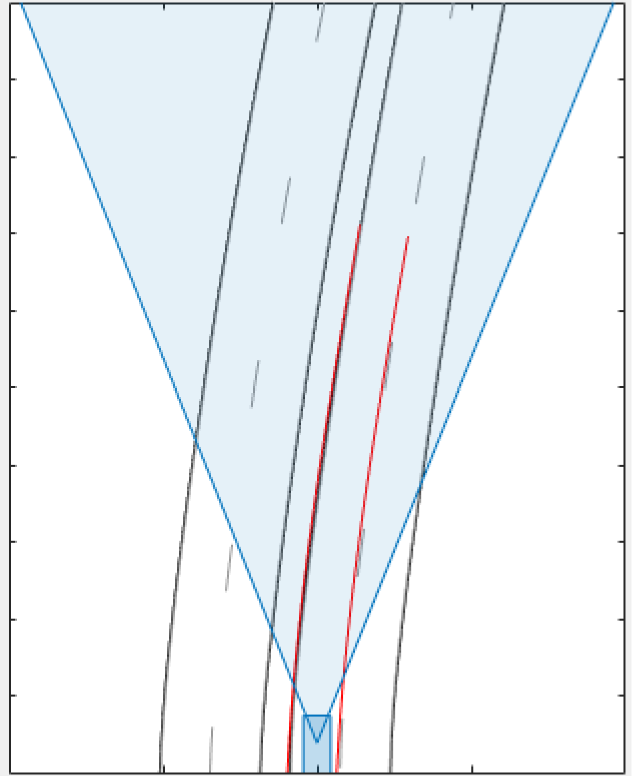
Add a camera sensor to the ego vehicle. From the app toolstrip, click **Add Camera**. Then, on the sensor canvas, add the camera to the predefined location representing the front window of the car.



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 vehicle. The **Bird's-Eye Plot** displays the left-lane and right-lane boundaries of the ego vehicle.



To export the detections to the MATLAB workspace, 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 Scenario

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

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

```
drivingScenarioDesigner(scenarioFileName)
```

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

If you are developing a driving algorithm in Simulink, you can use a Scenario Reader block to read the road network and actors from this file into your model. However, because the Scenario Reader block does not support reading in sensor data, the sensor you created is ignored. You must instead create the sensors within your model, using blocks such as Radar Detection Generator and Vision Detection Generator.

Scenario files containing large OpenDRIVE road networks can take up to several minutes to read into models.

References

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

See Also

Apps
Driving Scenario Designer

Blocks
Scenario Reader

Objects
`drivingScenario`

Functions
`roadNetwork`

More About

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

See Also

External Websites

- opendrive.org

Test Open-Loop ADAS Algorithm Using Driving Scenario

This example shows how to test an open-loop ADAS (advanced driver assistance system) algorithm in Simulink®. To test the scenario, you use a driving scenario that was saved from the Driving Scenario Designer app. In this example, you read in a scenario using a Scenario Reader block, and then visually verify the performance of a simple sensor fusion algorithm on the Bird's-Eye Scope.

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

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

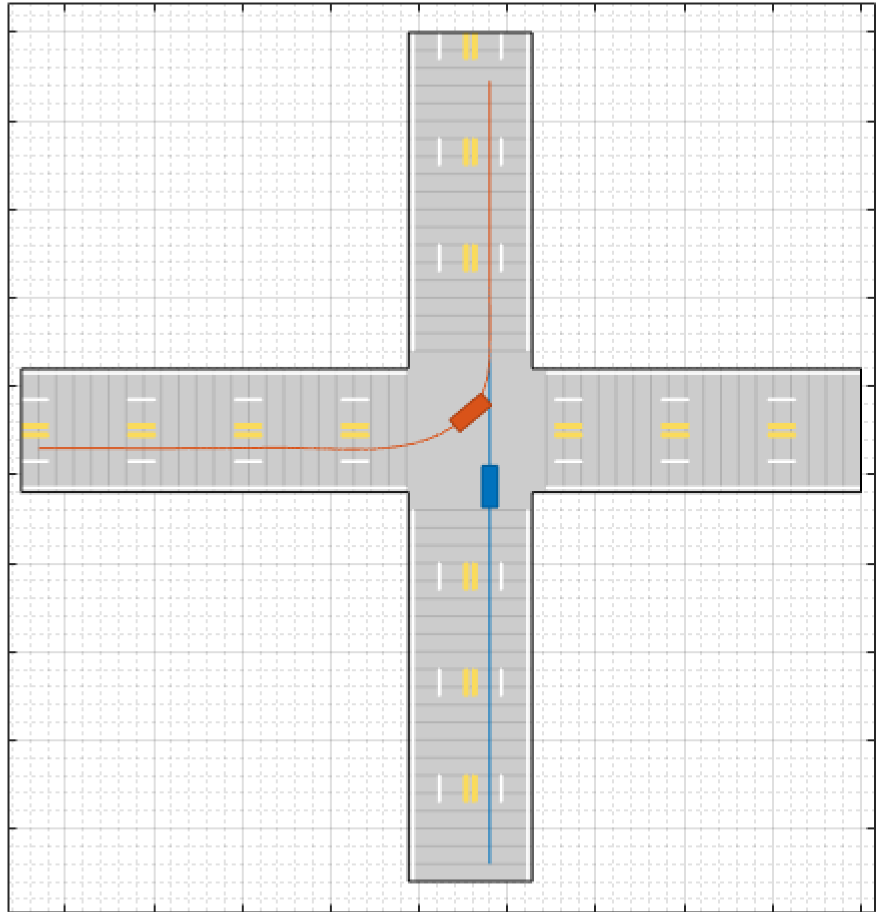
Inspect Driving Scenario

This example uses a driving scenario that is based on one of the prebuilt scenarios that you can access through the Driving Scenario Designer app. For more details on these scenarios, see “Generate Synthetic Detections from a Prebuilt Driving Scenario” on page 4-18.

Open the scenario file in the app.

```
drivingScenarioDesigner('LeftTurnScenario.mat')
```

Click **Run** to simulate the scenario. In this scenario, the ego vehicle travels north and goes straight through an intersection. Meanwhile, a vehicle coming from the left side of the intersection turns left and ends up in front of the ego vehicle.

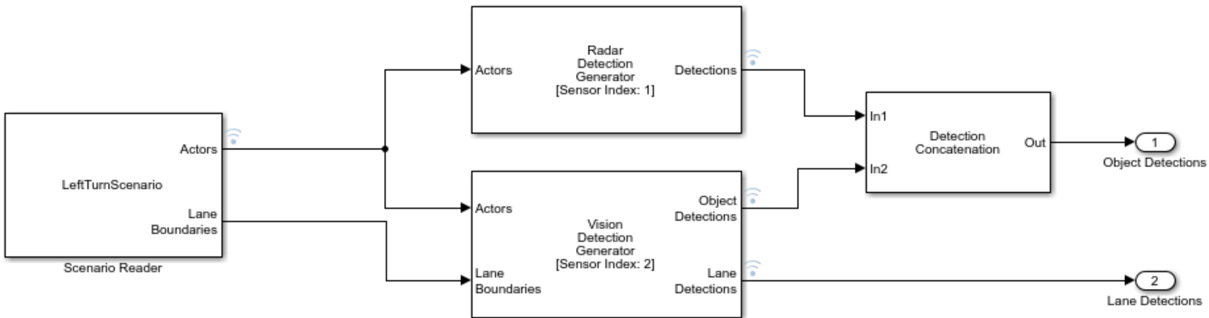


In the model used in this example, you generate detections of the other vehicle and the lane boundaries of the ego vehicle. Although this scenario includes a vision sensor defined in the app, the Scenario Reader block does not support reading sensor data from scenarios. Therefore, sensors must be defined in the model. If you read a scenario file containing sensor data, the block ignores this sensor data.

Inspect Model

In the model, a Scenario Reader block reads the actors and roads from the scenario file and outputs the non-ego actors and lane boundaries. Open the model.

```
open_system('OpenLoopWithScenarios.slx')
```



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

The Scenario Reader block outputs the poses of the non-ego actors in the scenario and the left-lane and right-lane boundaries of the ego vehicle. To output all lane boundaries of the road on which the ego vehicle is traveling, select the corresponding option for the **Lane boundaries to output** parameter.

The actors are passed to a Radar Detection Generator and a Vision Detection Generator block, and the lane boundaries are passed to the Vision Detection Generator block. These sensor blocks produce synthetic detections from the scenario. The outputs are in vehicle coordinates, where:

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

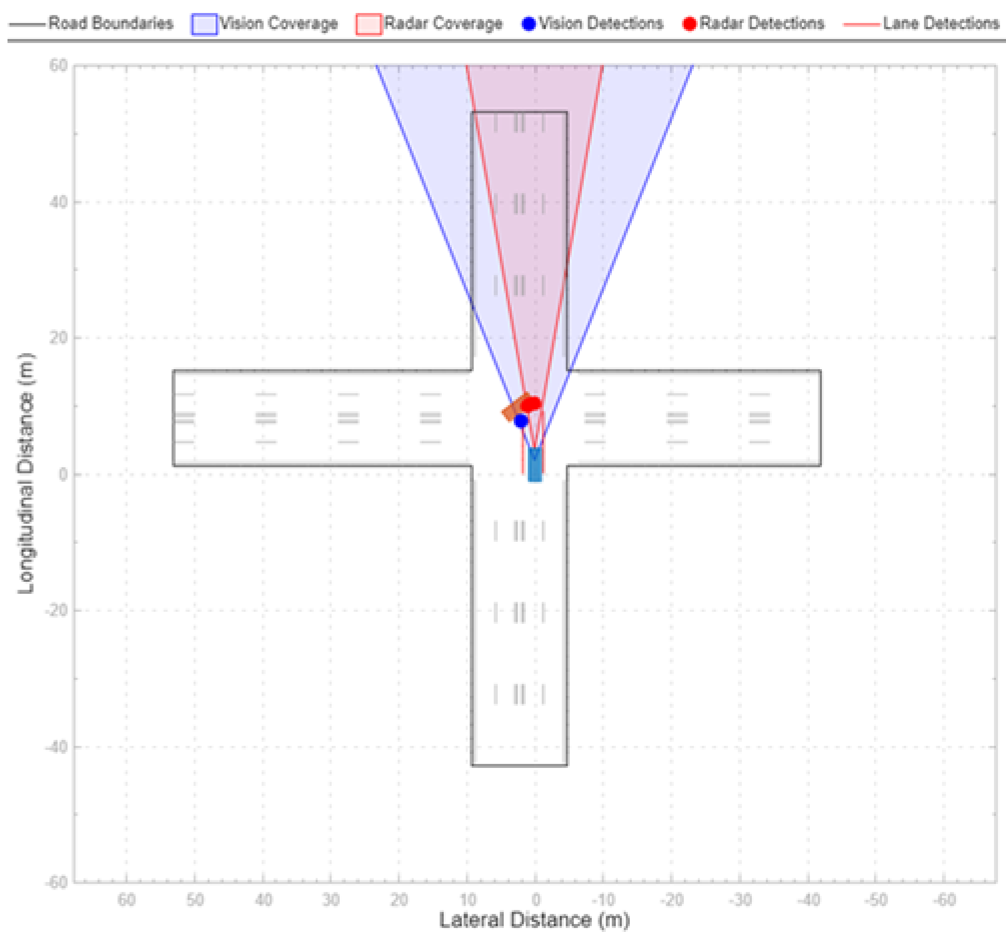
If a scenario has multiple ego vehicles, in the Scenario Reader block, set the **Coordinate system of outputs** parameter to `World coordinates` instead of `Vehicle Coordinates`. In the world coordinate system, the actors and lane boundaries are in the world coordinates of the driving scenario. The Bird's-Eye Scope does not support visualization of world coordinates.

In this model, the Scenario Reader block reads the ego vehicle from the scenario file (the **Source of ego vehicle** parameter is set to `Scenario file`). Because this algorithm is *open-loop*, the ego vehicle behavior does not change as the simulation advances.

Therefore, the **Source of ego vehicle** parameter is set to `Scenario file`, and the block reads the predefined ego vehicle pose and trajectory from the scenario. For vehicle controllers and other closed-loop algorithms, set the **Source of ego vehicle** parameter to `Input port`. With this option, you specify an ego vehicle that is defined in the model as an input to the Scenario Reader block. For an example, see "Test Closed-Loop ADAS Algorithm Using Driving Scenario" on page 4-78.

Visually Verify Algorithm

To visualize the scenario and the object and lane boundary detections, use the Bird's-Eye Scope. From the Simulink model toolbar, click the Bird's-Eye Scope button. Then, click **Find Signals**, and run the simulation. The vision sensor correctly generates detections for the non-ego actor and the lane boundaries.



Update Simulation Settings

This model uses the default simulation stop time of 10 seconds. However, because the scenario is only about 5 seconds long, the simulation continues to run in the Bird's-Eye Scope even after the scenario has ended. To synchronize the simulation and scenario stop times, in the Simulink model toolbar, set the simulation stop time to `5.2` seconds, which is the exact stop time of the app scenario. After you run the simulation, the app displays this value in the bottom-right corner of the scenario canvas.

If the simulation runs too fast in the Bird's-Eye Scope, you can slow down the simulation by using simulation pacing. From the Simulink model toolbar, select **Simulation > Pacing Options**. Select the **Enable pacing to slow down simulation** check box and decrease the simulation time to slightly less than 1 second per wall-clock second, such as 0.8 seconds. Then, rerun the simulation in the Bird's-Eye Scope.

When you are done with this example, remove the example file folder from the MATLAB search path.

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

See Also

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

More About

- “Sensor Fusion Using Synthetic Radar and Vision Data in Simulink”
- “Test Closed-Loop ADAS Algorithm Using Driving Scenario” on page 4-78

Test Closed-Loop ADAS Algorithm Using Driving Scenario

This model shows how to test a closed-loop ADAS (advanced driver assistance system) algorithm in Simulink®. To test the scenario, you use a driving scenario that was saved from the Driving Scenario Designer app. In this model, you read in a scenario using a Scenario Reader block, and then visually verify the performance of the algorithm, an autonomous emergency braking (AEB) system, on the Bird's-Eye Scope.

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

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

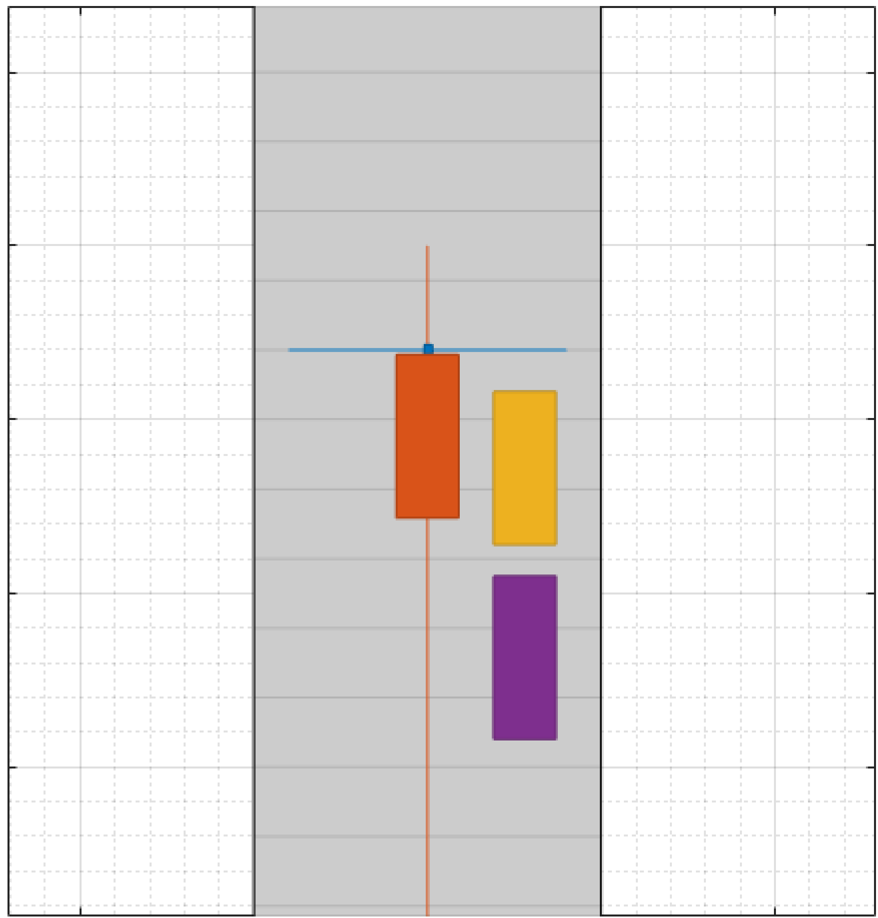
Inspect Driving Scenario

This example uses a driving scenario that is based on one of the prebuilt Euro NCAP test protocol scenarios that you can access through the Driving Scenario Designer app. For more details on these scenarios, see “Generate Synthetic Detections from a Euro NCAP Scenario” on page 4-40.

Open the scenario file in the app.

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

Click **Run** to simulate the scenario. In this scenario, the ego vehicle collides with a pedestrian child who is crossing the street.

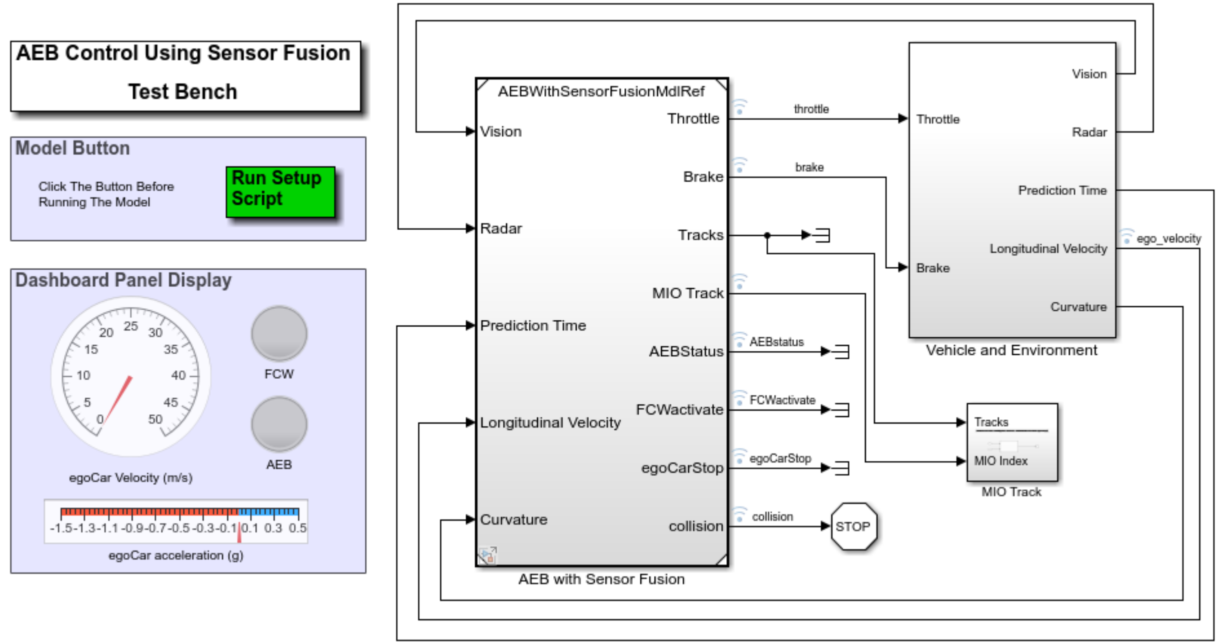


In the model used in this example, you use an AEB sensor fusion algorithm to detect the pedestrian child and test whether the ego vehicle brakes in time to avoid a collision.

Inspect Model

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

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



A Scenario Reader block reads the actors and roads from the specified scenario file and outputs the non-ego actors. This block is located in the **Vehicle Environment > Actors and Sensor Simulation** subsystem. Open this subsystem.

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

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

The Scenario Reader block outputs the poses of the non-ego actors in the scenario. These poses are passed to vision and radar sensors, whose detections are used to determine the behavior of the AEB controller.

The actor poses are output in vehicle coordinates, where:

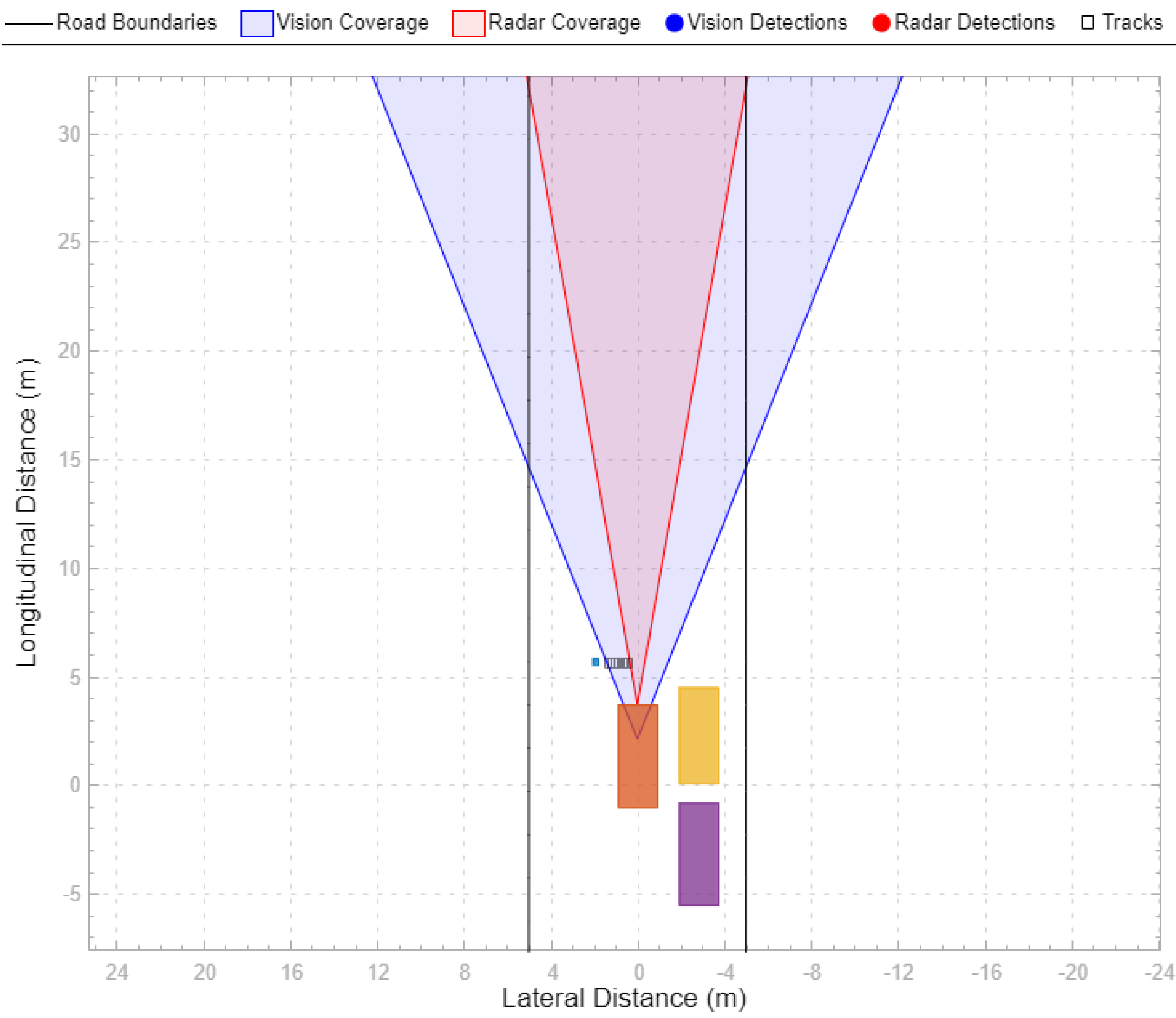
- The *X*-axis points forward from the ego vehicle.
- The *Y*-axis points to the left of the ego vehicle.
- The origin is located at the center of the rear axle of the ego vehicle.

If a scenario has multiple ego vehicles, in the Scenario Reader block, set the **Coordinate system of outputs** parameter to `World coordinates` instead of `Vehicle Coordinates`. In the world coordinate system, the actors and lane boundaries are in the world coordinates of the driving scenario. The Bird's-Eye Scope does not support visualization of world coordinates.

Although this scenario includes a predefined ego vehicle, the Scenario Reader block is configured to ignore this ego vehicle definition. Instead, the ego vehicle is defined in the model and specified as an input to the Scenario Reader block (the **Source of ego vehicle** parameter is set to `Input port`). As the simulation advances, the AEB algorithm determines the pose and trajectory of the ego vehicle. If you are developing an open-loop algorithm, where the ego vehicle is predefined in the driving scenario, set the **Source of ego vehicle** parameter to `Scenario file`. For an example, see “Test Open-Loop ADAS Algorithm Using Driving Scenario” on page 4-72.

Visually Verify Algorithm

To visualize the scenario, use the Bird's-Eye Scope. From the Simulink model toolbar, click the Bird's-Eye Scope button. Then, click **Find Signals**, and run the simulation. With the AEB algorithm, the ego vehicle brakes in time to avoid a collision.



When you are done verifying the algorithm, remove the example file folder from the MATLAB search path.

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

See Also

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

More About

- “Autonomous Emergency Braking with Sensor Fusion”
- “Lateral Control Tutorial”
- “Test Open-Loop ADAS Algorithm Using Driving Scenario” on page 4-72

5

Planning, Mapping, and Control

- “Access HERE HD Live Map Data” on page 5-2
- “Enter HERE HD Live Map Credentials” on page 5-9
- “Create Configuration for HERE HD Live Map Reader” on page 5-11
- “Create HERE HD Live Map Reader” on page 5-17
- “Read and Visualize Data Using HERE HD Live Map Reader” on page 5-21
- “HERE HD Live Map Layers” on page 5-34
- “Control Vehicle Velocity” on page 5-40

Access HERE HD Live Map Data

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

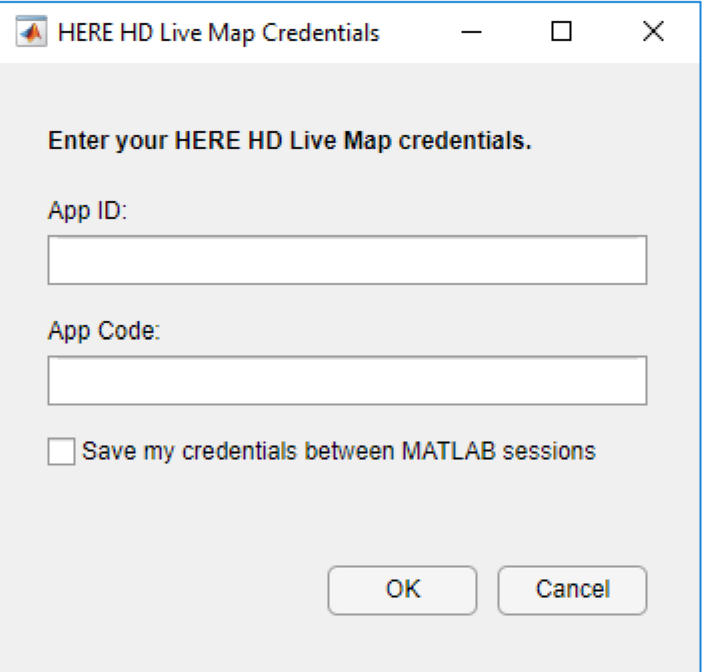
Using Automated Driving Toolbox functions and objects, you can create a HERE HDLM reader, read map data from the HERE HDLM web service, and then visualize the data from certain layers.

Step 1: Enter Credentials

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

```
hereHDLMCredentials setup
```

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

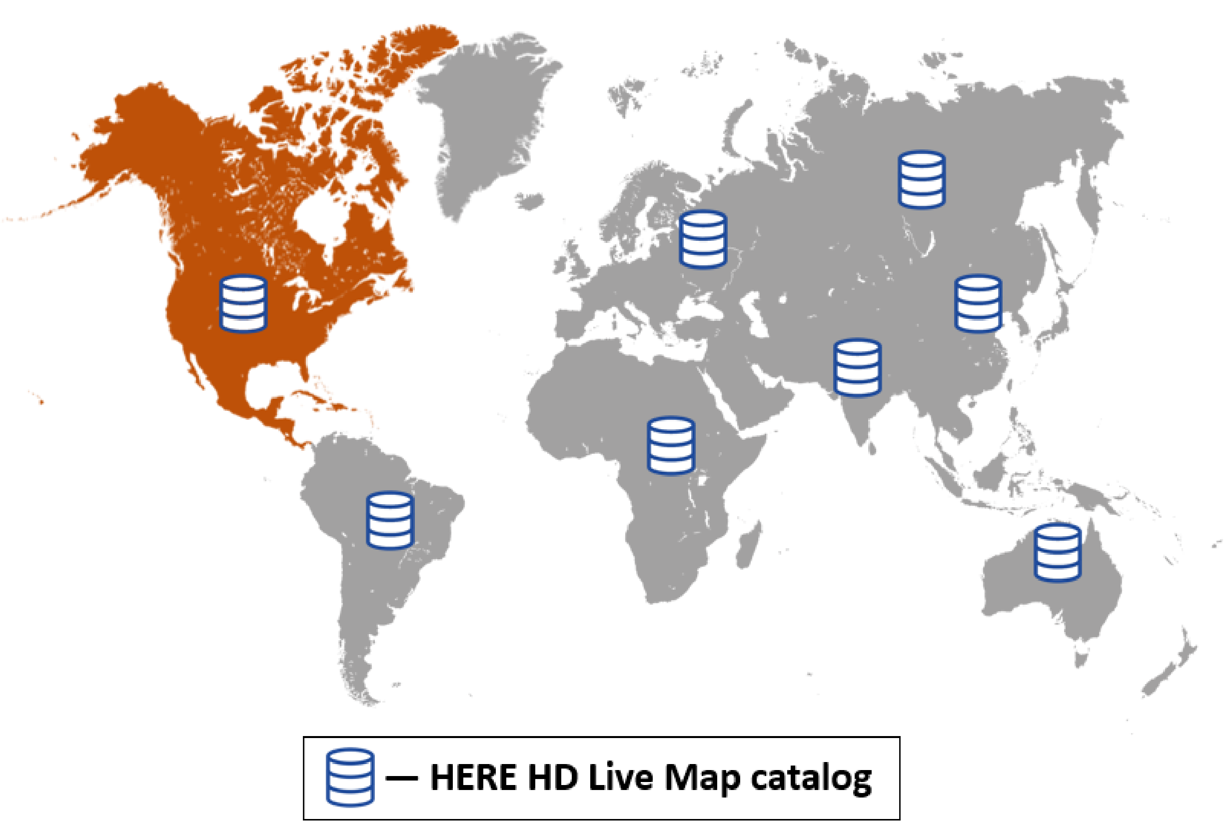


For more details, see "Enter HERE HD Live Map Credentials" on page 5-9.

Step 2: Create Reader Configuration

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

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

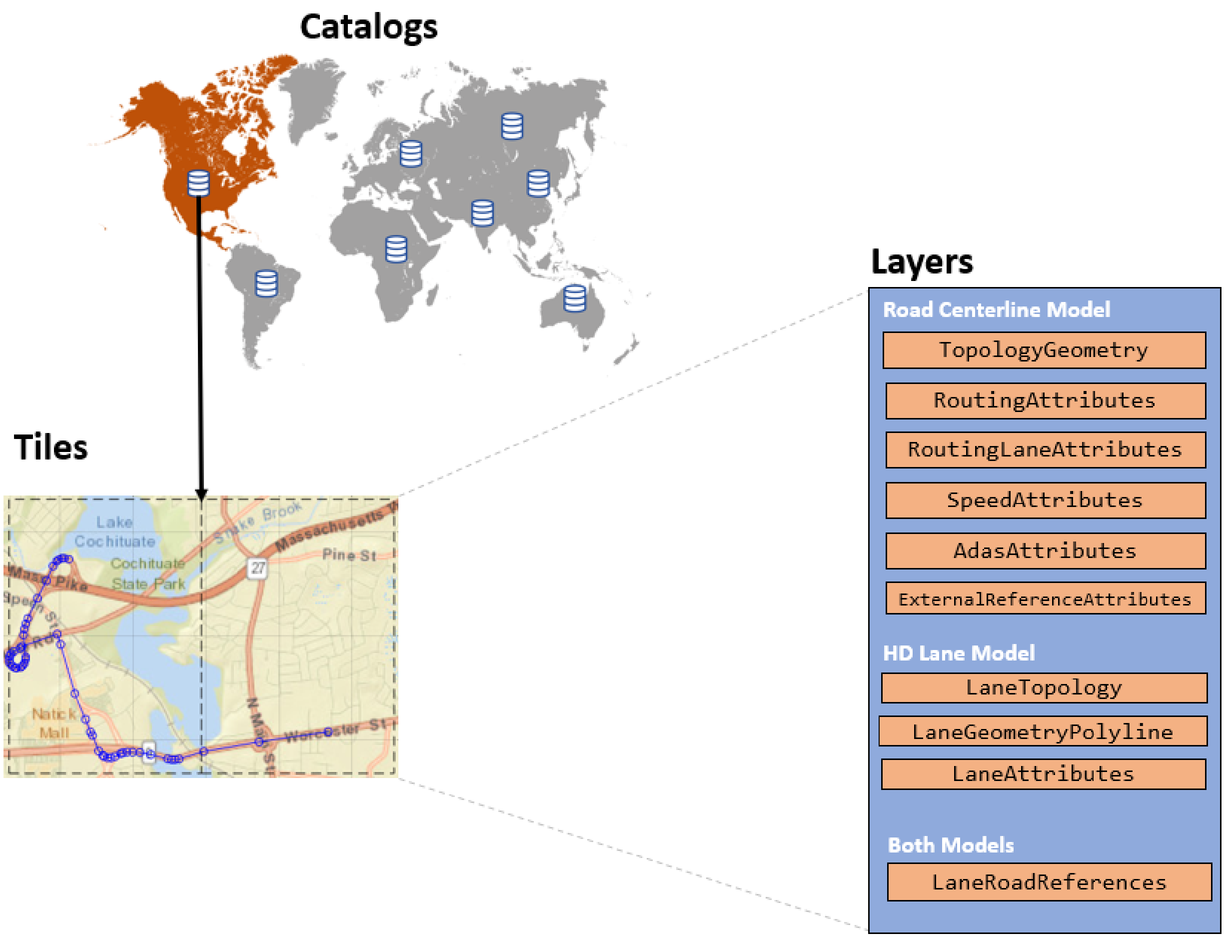


For more details, see “Create Configuration for HERE HD Live Map Reader” on page 5-11.

Step 3: Create Reader

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

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



For more details, see “Create HERE HD Live Map Reader” on page 5-17.

Step 4: Read and Visualize Data

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

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

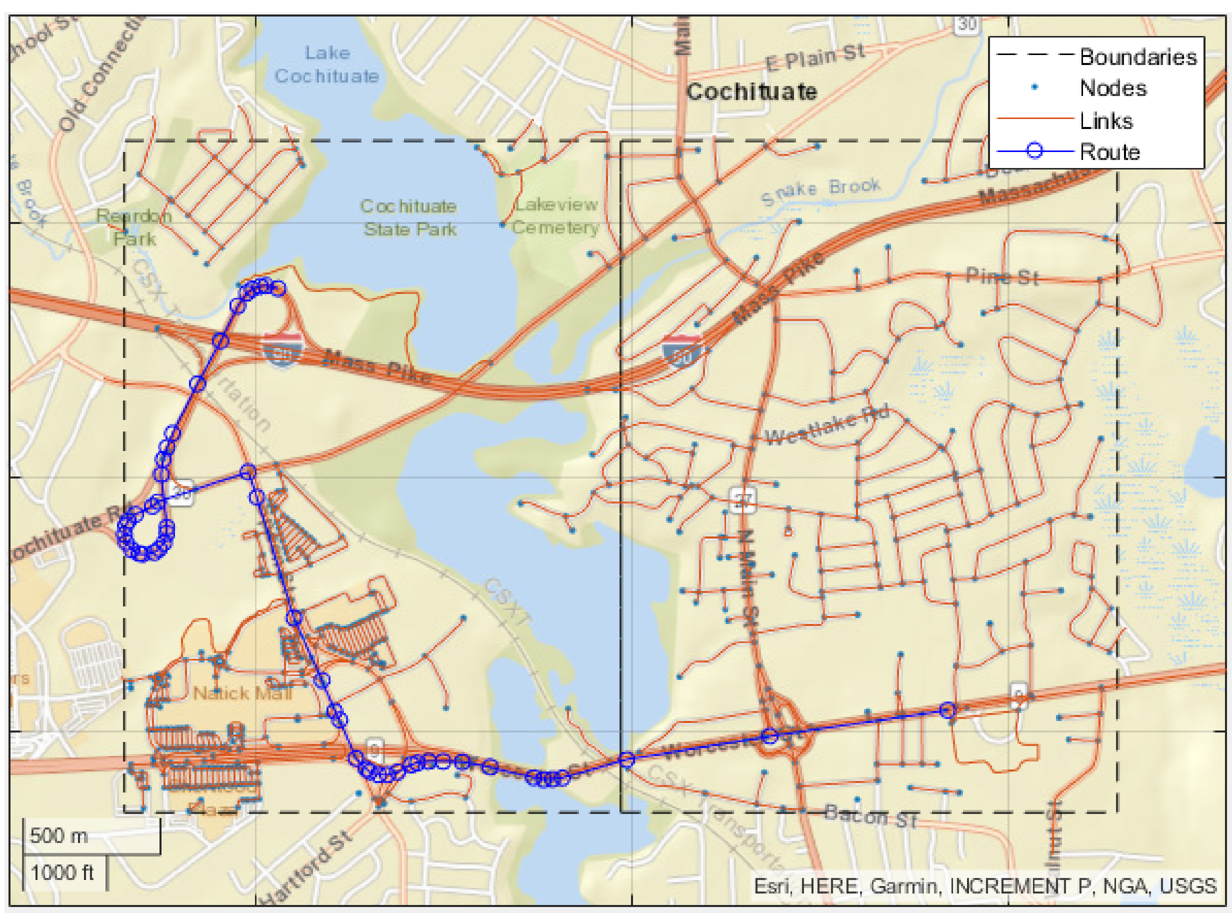
```
topology = 

  2×1 TopologyGeometry array with properties:

   Data:
    HereTileId
    IntersectingLinkRefs
    LinksStartingInTile
    NodesInTile
    TileCenterHere2dCoordinate

   Metadata:
    Catalog
    CatalogVersion
```

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



For more details, see “Read and Visualize Data Using HERE HD Live Map Reader” on page 5-21.

See Also

hereHDLMConfiguration | hereHDLMCredentials | hereHDLMReader | plot | read

More About

- “Enter HERE HD Live Map Credentials” on page 5-9

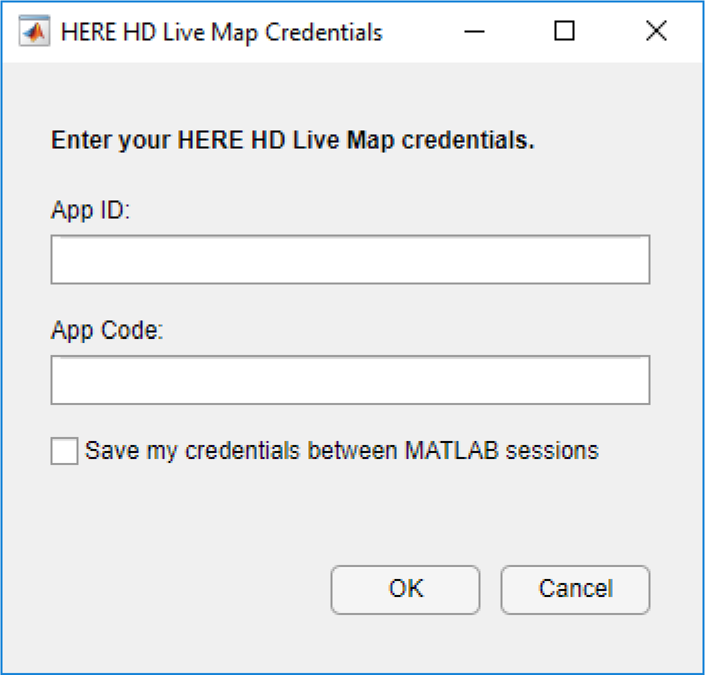
- “Create Configuration for HERE HD Live Map Reader” on page 5-11
- “Create HERE HD Live Map Reader” on page 5-17
- “Read and Visualize Data Using HERE HD Live Map Reader” on page 5-21
- “HERE HD Live Map Layers” on page 5-34
- “Use HERE HD Live Map Data to Verify Lane Configurations”

External Websites

- HD Live Map Data Specification

Enter HERE HD Live Map Credentials

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



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

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

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

```
hereHDLMCredentials setup
```

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

```
hereHDLMCredentials delete
```

After you enter your credentials, you can then configure your HERE HDLM reader to search for map data in only a specific geographic region. See “Create Configuration for HERE HD Live Map Reader” on page 5-11. Alternatively, you can create the reader without specifying a configuration. See “Create HERE HD Live Map Reader” on page 5-17.

See Also

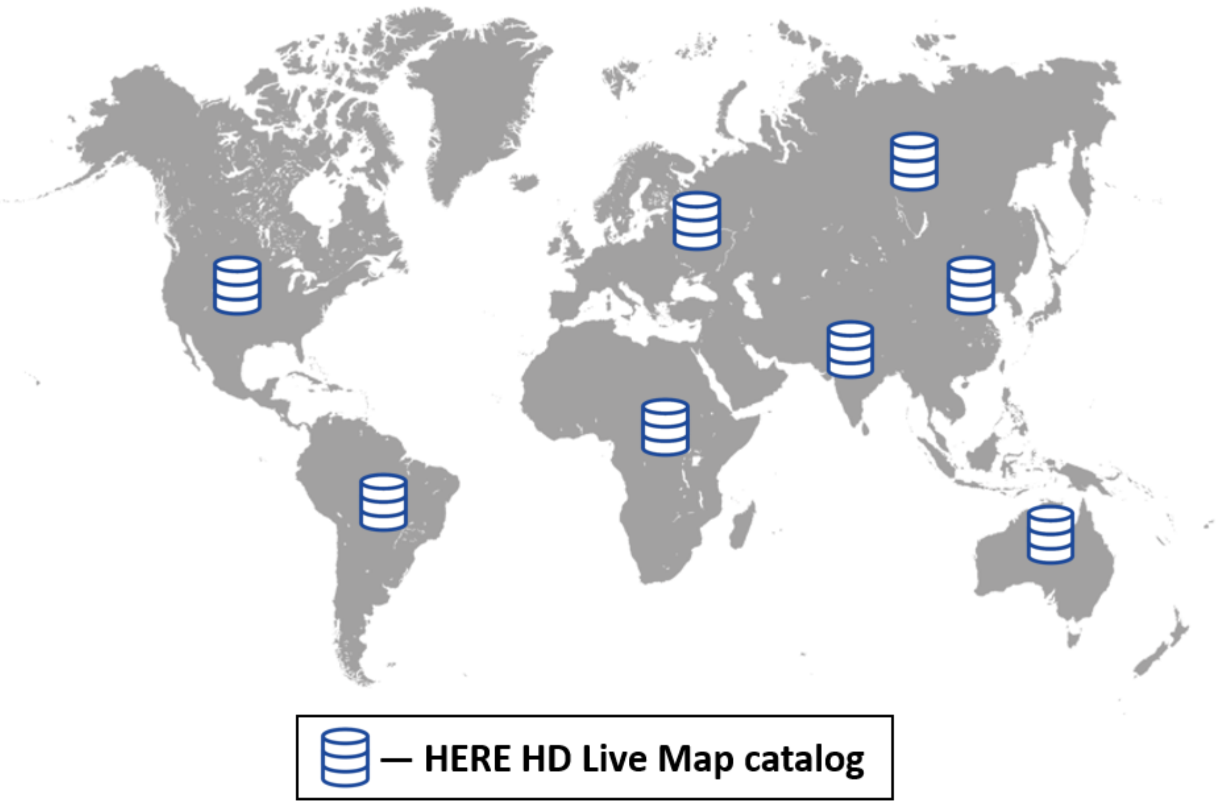
`hereHDLMConfiguration` | `hereHDLMCredentials` | `hereHDLMReader`

More About

- “Create Configuration for HERE HD Live Map Reader” on page 5-11
- “Create HERE HD Live Map Reader” on page 5-17

Create Configuration for HERE HD Live Map Reader

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



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

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

map data. You can also configure a reader to search from only a specific version of a catalog.

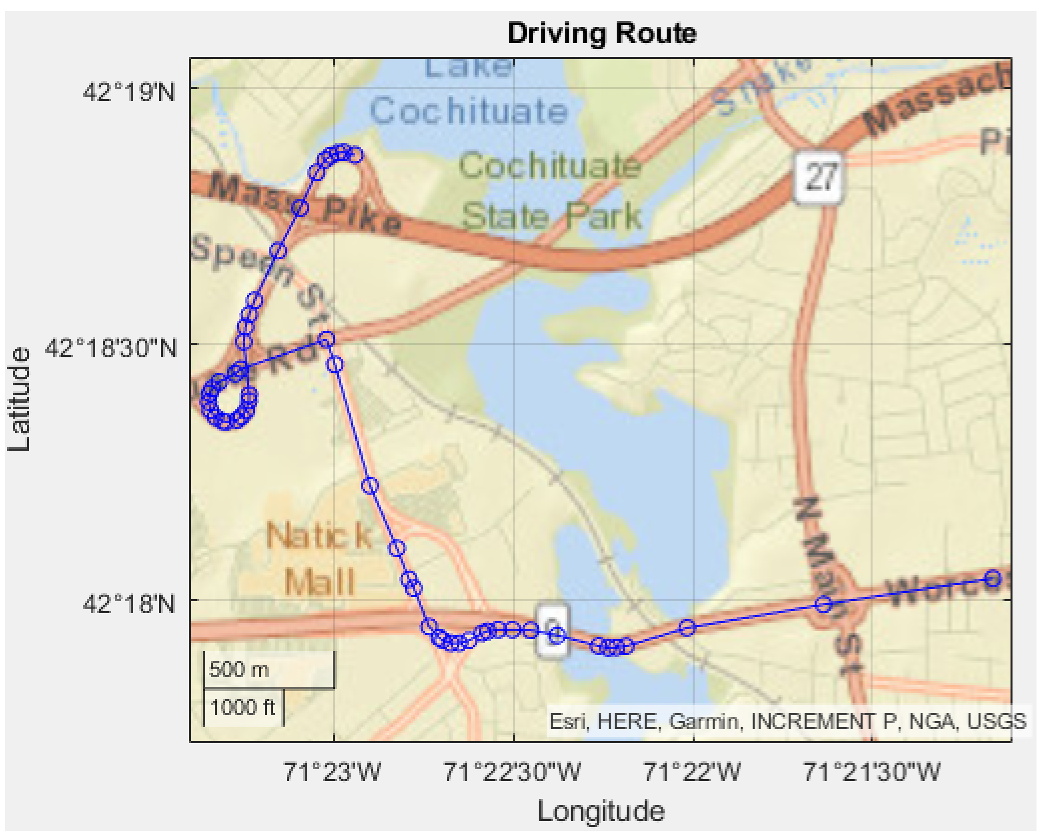
Configuring a HERE HDLM reader using a `hereHDLMConfiguration` object is optional. If you do not specify a configuration, by default, the reader searches for map tiles across all catalogs and returns map data from the latest version of that catalog.

Create Configuration for Specific Catalog

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

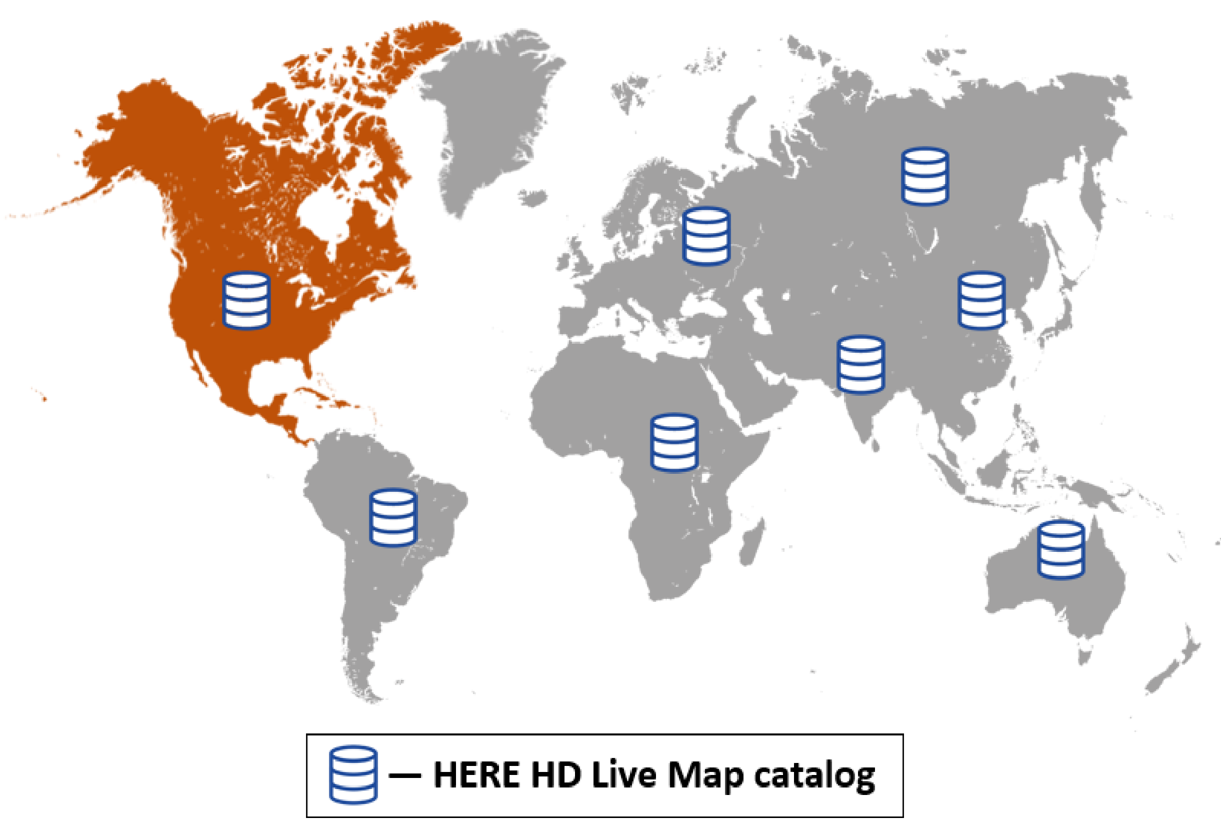
Consider a driving route located in North America.

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



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

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



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

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

Region	Catalog
'Asia Pacific'	'here-hdmap-ext-apac-1'
'Eastern Europe'	'here-hdmap-ext-eeu-1'
'India'	'here-hdmap-ext-rn-1'
'Middle East And Africa'	'here-hdmap-ext-mea-1'

Region	Catalog
'North America'	'here-hdmap-ext-na-1'
'Oceania'	'here-hdmap-ext-au-1'
'South America'	'here-hdmap-ext-sam-1'
'Western Europe'	'here-hdmap-ext-weu-1'

Create Configuration for Specific Version

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

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

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

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

Configure Reader

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

For example, create a HERE HDLM reader using the configuration and latitude-longitude coordinates that you created in the “Create Configuration for Specific Catalog” on page 5-12 section. Your catalog version might differ from the one shown here. This reader is configured for the latest catalog version, but the HERE HDLM service is continually updated and frequently produces new map versions.

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

```
  hereHDLMConfiguration with properties:
```

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

For details about creating HERE HDLM readers, see “Create HERE HD Live Map Reader” on page 5-17.

See Also

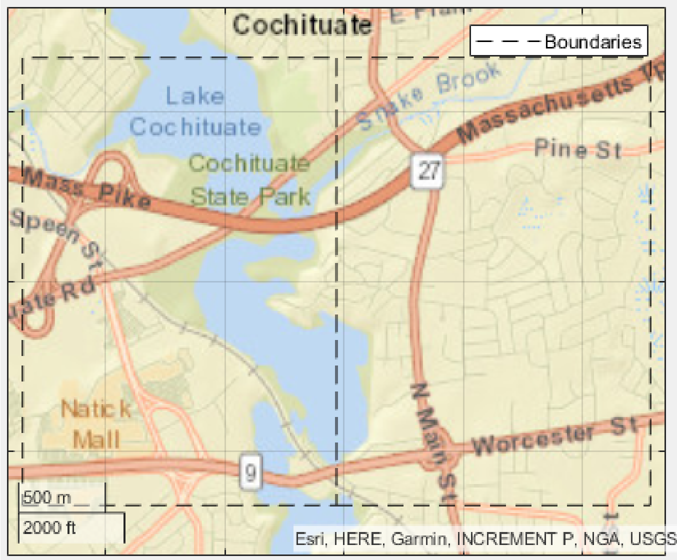
`hereHDLMConfiguration` | `hereHDLMReader`

More About

- “Create HERE HD Live Map Reader” on page 5-17

Create HERE HD Live Map Reader

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



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

Create Reader from Specified Driving Route

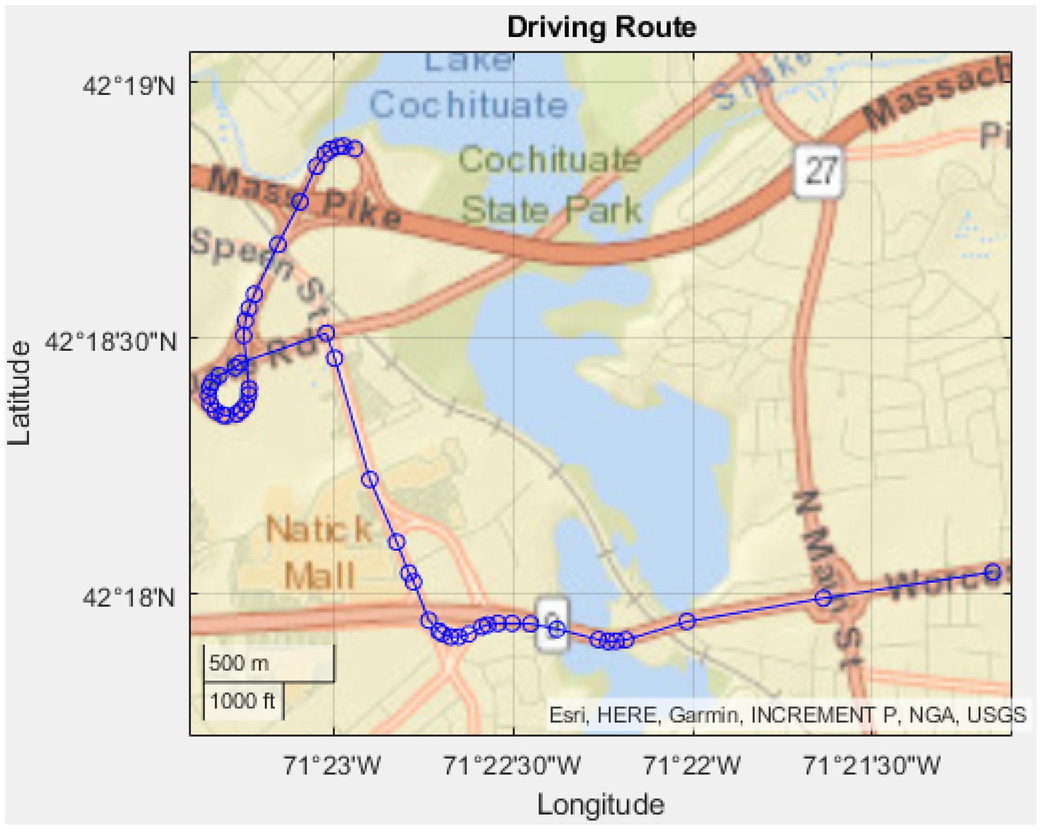
If you have a driving route stored as a vector of latitude-longitude coordinates, you can use these coordinates to select map tiles from which to read data.

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

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

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

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



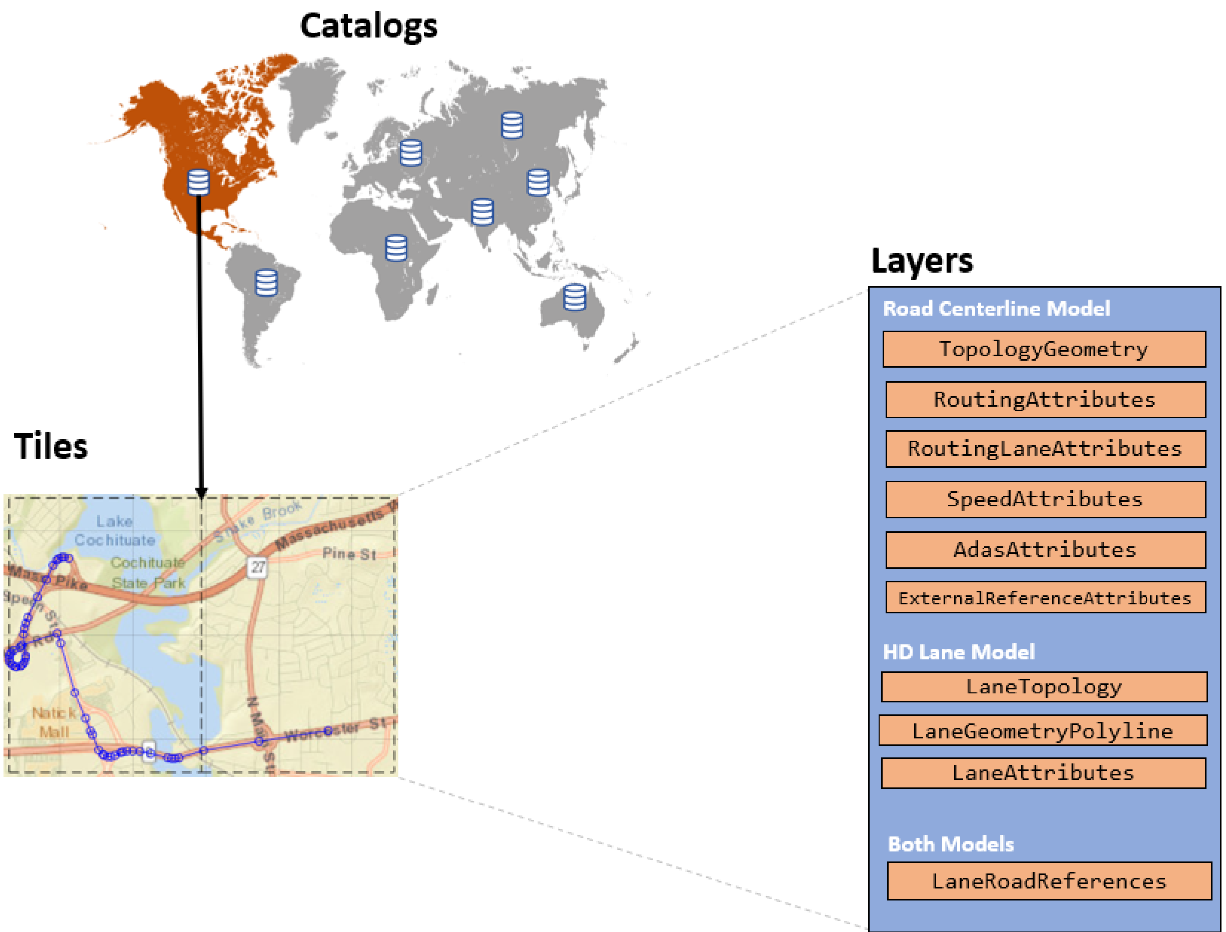
Create a `hereHDLMConfiguration` object for reading data from only the North America catalog. For more details about configuring HERE HDLM readers, see “Create Configuration for HERE HD Live Map Reader” on page 5-11. If you have not previously set up HERE HDLM credentials, a dialog box prompts you to enter them.

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

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

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

This HERE HDLM reader enables you to read map data for the tiles that the driving route is on. The map data is stored in a set of layers containing detailed information about various aspects of the map. The reader supports reading data from the map layers for the Road Centerline Model and HD Lane Model. For more details on the layers in these models, see “HERE HD Live Map Layers” on page 5-34.



If you call the `read` function with the HERE HDLM reader, you can read the map tile data for a specific layer. If the layer supports visualization, you can also plot the layer. For more details, see "Read and Visualize Data Using HERE HD Live Map Reader" on page 5-21.

Create Reader from Specified Map Tile IDs

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

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

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

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

To learn more about reading and plotting data from map tiles, see "Read and Visualize Data Using HERE HD Live Map Reader" on page 5-21.

See Also

`hereHDLMConfiguration` | `hereHDLMReader` | `read`

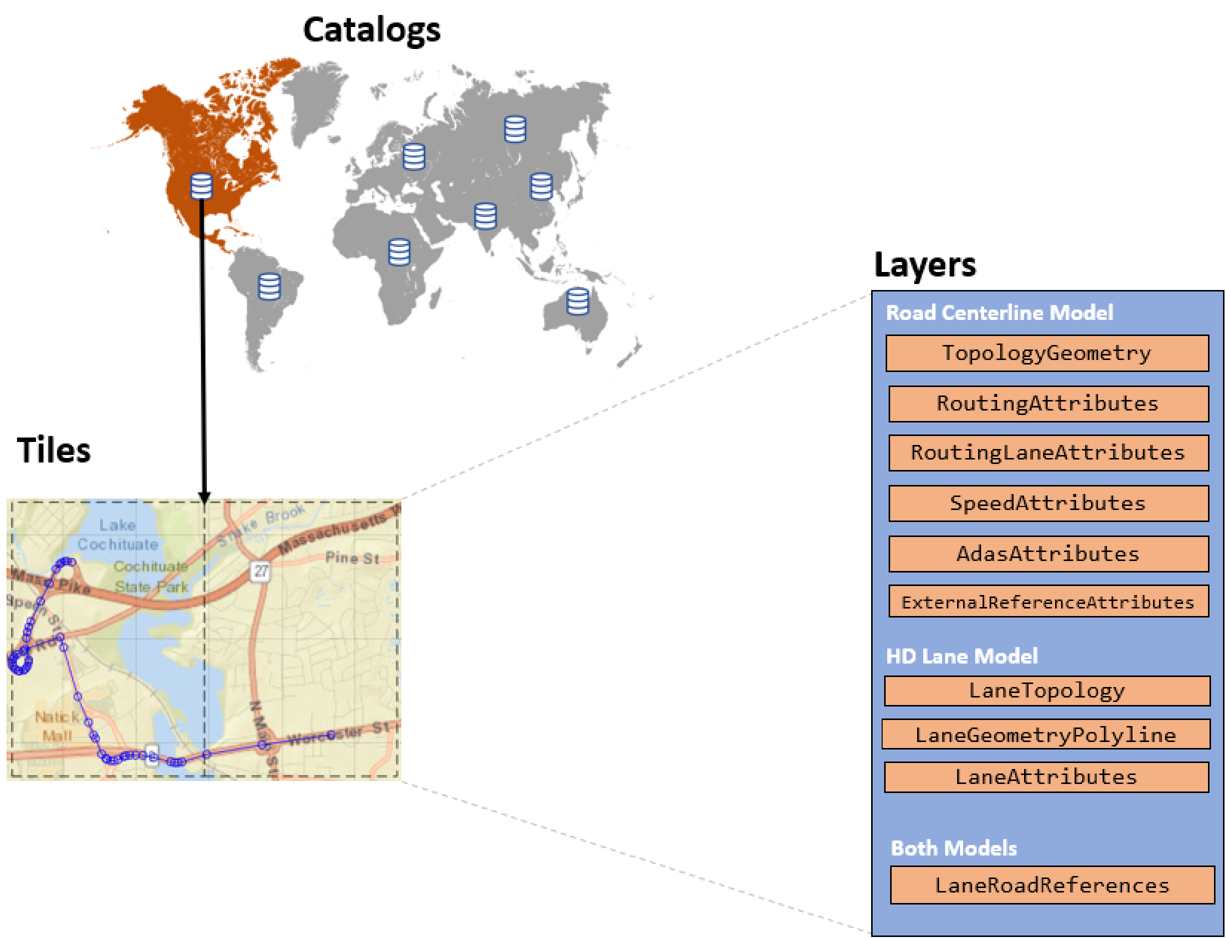
More About

- "Read and Visualize Data Using HERE HD Live Map Reader" on page 5-21
- "HERE HD Live Map Layers" on page 5-34

Read and Visualize Data Using HERE HD Live Map Reader

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

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



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

Create Reader

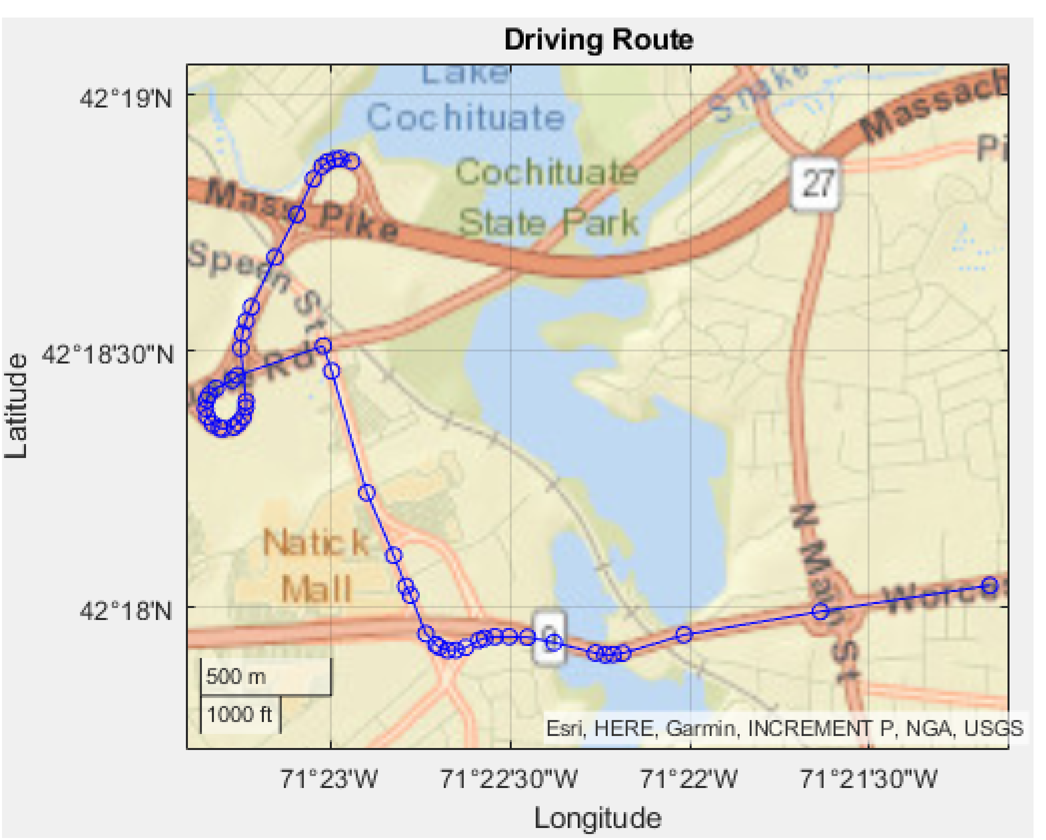
To read map data using the `read` function, you must specify a `hereHDLMReader` object as an input argument. This object specifies the map tiles from which you want to read data.

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

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

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

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



For more details about configuring a HERE HDLM reader, see “Create Configuration for HERE HD Live Map Reader” on page 5-11. For more details about creating a reader, see “Create HERE HD Live Map Reader” on page 5-17.

Read Map Layer Data

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

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

```
topology = 

  2×1 TopologyGeometry array with properties:

   Data:
    HereTileId
    IntersectingLinkRefs
    LinksStartingInTile
    NodesInTile
    TileCenterHere2dCoordinate

   Metadata:
    Catalog
    CatalogVersion
```

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

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

```
topology(1)
```

```
ans = 

  TopologyGeometry with properties:

   Data:
                    HereTileId: 321884279
```

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

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

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

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

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

```
layerData.NodeAttribution.NodeId
```

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

Road Centerline Model and HD Lane Model. For an overview of HERE HDLM layers and the models that they belong to, see “HERE HD Live Map Layers” on page 5-34. For a full description of the fields, see HD Live Map Data Specification on the HERE Technologies website.

Layer Object	Description	Top-Level Data Fields (Layer Object Properties)	Plot Support
AdasAttributes	Precision geometry measurements, such as slope, elevation, and curvature of roads. Use this data to develop advanced driver assistance systems (ADAS).	• HereTileId • LinkAttribution • NodeAttribution	Not available
ExternalReferenceAttributes	References to external map links, nodes, and topologies for other HERE maps.	• HereTileId • LinkAttribution • NodeAttribution	Not available
LaneAttributes	Lane-level attributes, such as direction of travel and lane type.	• HereTileId • LaneGroupAttribution	Not available

Layer Object	Description	Top-Level Data Fields (Layer Object Properties)	Plot Support
LaneGeometryPolyline	3-D lane geometry composed of a set of 3-D points joined into polylines.	• HereTileId • TileCenterHere3dCoordinate • LaneGroupGeometries	Available — Use the plot function.
LaneRoadReferences	Road and lane group references and range information. Use this data to translate positions between the Road Centerline Model and the HD Lane Model.	• HereTileId • LaneGroupLinkReferences • LinkLaneGroupReferences	Not available

Layer Object	Description	Top-Level Data Fields (Layer Object Properties)	Plot Support
`LaneTopology`	Topologies of the HD Lane model, including lane group, lane group connector, lane, and lane connector topologies. This layer also contains the simplified 2-D boundary geometry of the lane model for determining map tile affinity and overflow.	• `HereTileId` • `TileCenterHere2dCoordinate` • `LaneGroupsStartingInTile` • `LaneGroupConnectorsInTile` • `IntersectingLaneGroupRefs`	Available — Use the `plot` function.
`RoutingAttributes`	Road attributes related to navigation and conditions. These attributes are mapped parametrically to the 2-D polyline geometry in the topology layer.	• `HereTileId` • `LinkAttribution` • `NodeAttribution` • `StrandAttribution` • `AttributionGroupList`	Not available

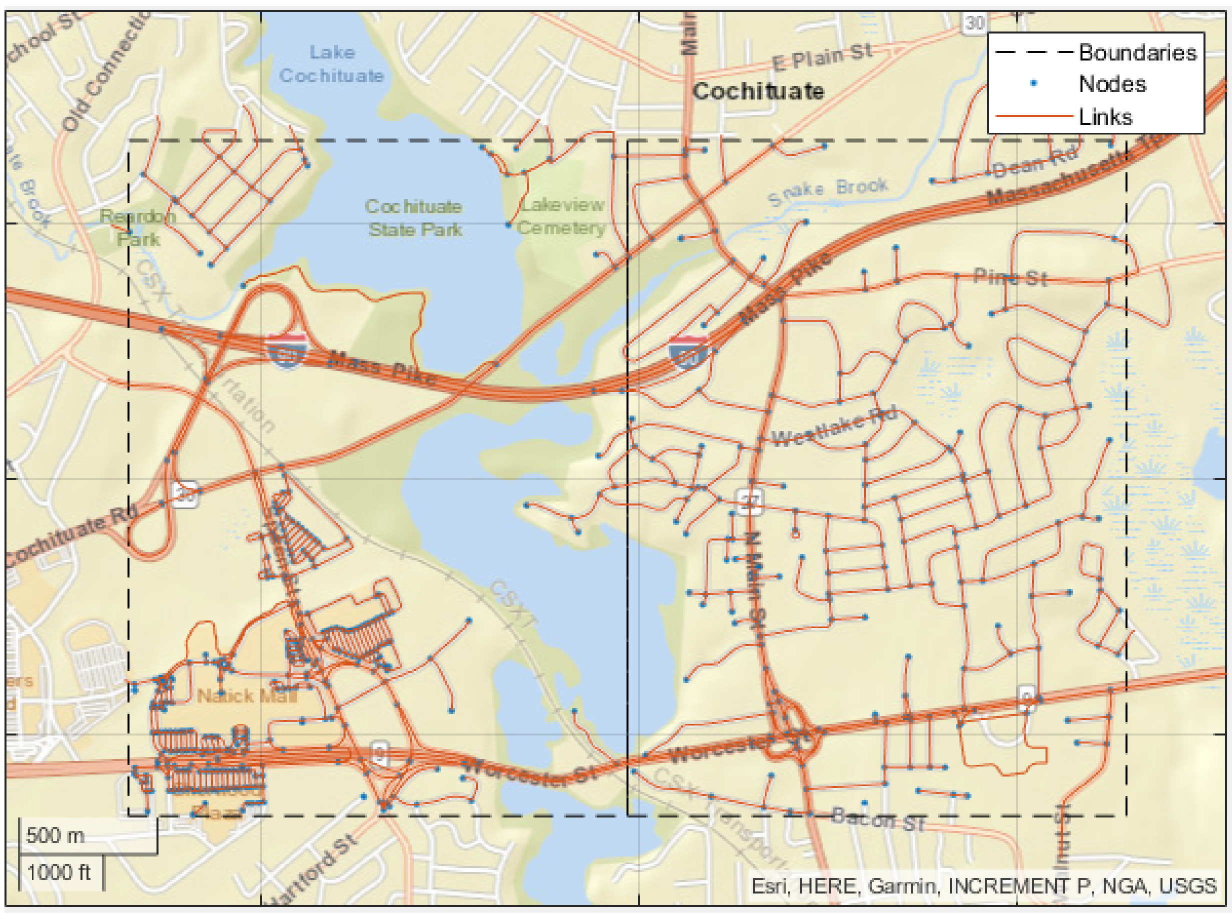
Layer Object	Description	Top-Level Data Fields (Layer Object Properties)	Plot Support
`RoutingLaneAttributes`	Core navigation lane attributes and conditions, such as the number of lanes in a road. These values are mapped parametrically to 2-D polylines along the road links.	• `HereTileId` • `LinkAttribution`	Not available
`SpeedAttributes`	Speed-related road attributes, such as speed limits. These attributes are mapped to the 2-D polyline geometry of the topology layer.	• `HereTileId` • `LinkAttribution`	Not available

Layer Object	Description	Top-Level Data Fields (Layer Object Properties)	Plot Support
`TopologyGeometry`	Topology and 2-D line geometry of the road. This layer also contains definitions of the nodes and links in the map tile.	• `HereTileId` • `TileCenterHere2dCoordinate` • `NodesInTile` • `LinksStartingInTile` • `IntersectingLinkRefs`	Available — Use the `plot` function.

Visualize Map Layer Data

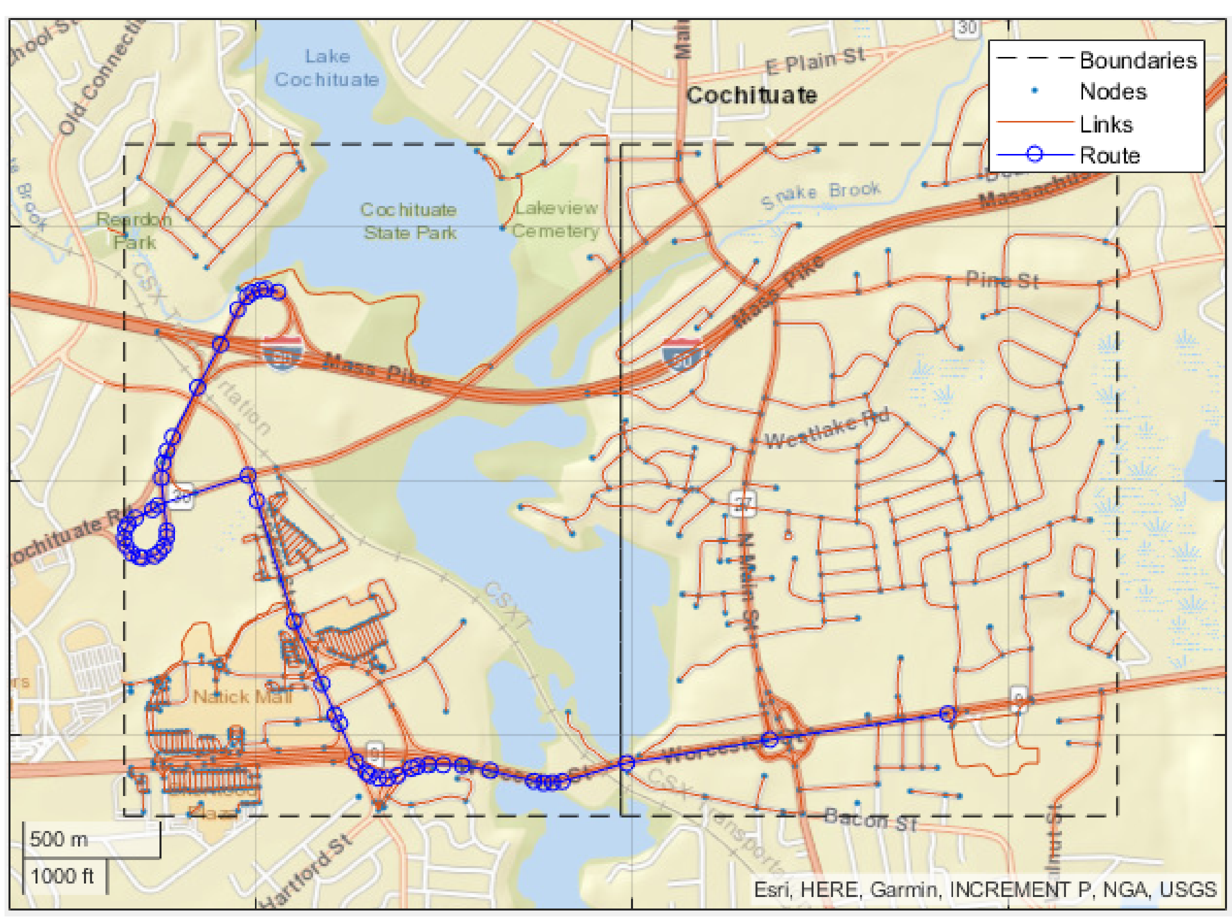
You can visualize the data of certain map layers. To visualize these layers, use the `plot` function. Plot the topology geometry of the returned map layers. The plot shows the boundaries, nodes (intersections and dead-ends), and links (streets) within the map tiles. If a link extends past the tile boundary,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

```
plot(topology)
```



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

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



See Also

`hereHDLMReader` | `plot` | `read`

More About

- “HERE HD Live Map Layers” on page 5-34
- “Use HERE HD Live Map Data to Verify Lane Configurations”

External Websites

- HD Live Map Data Specification

HERE HD Live Map Layers

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

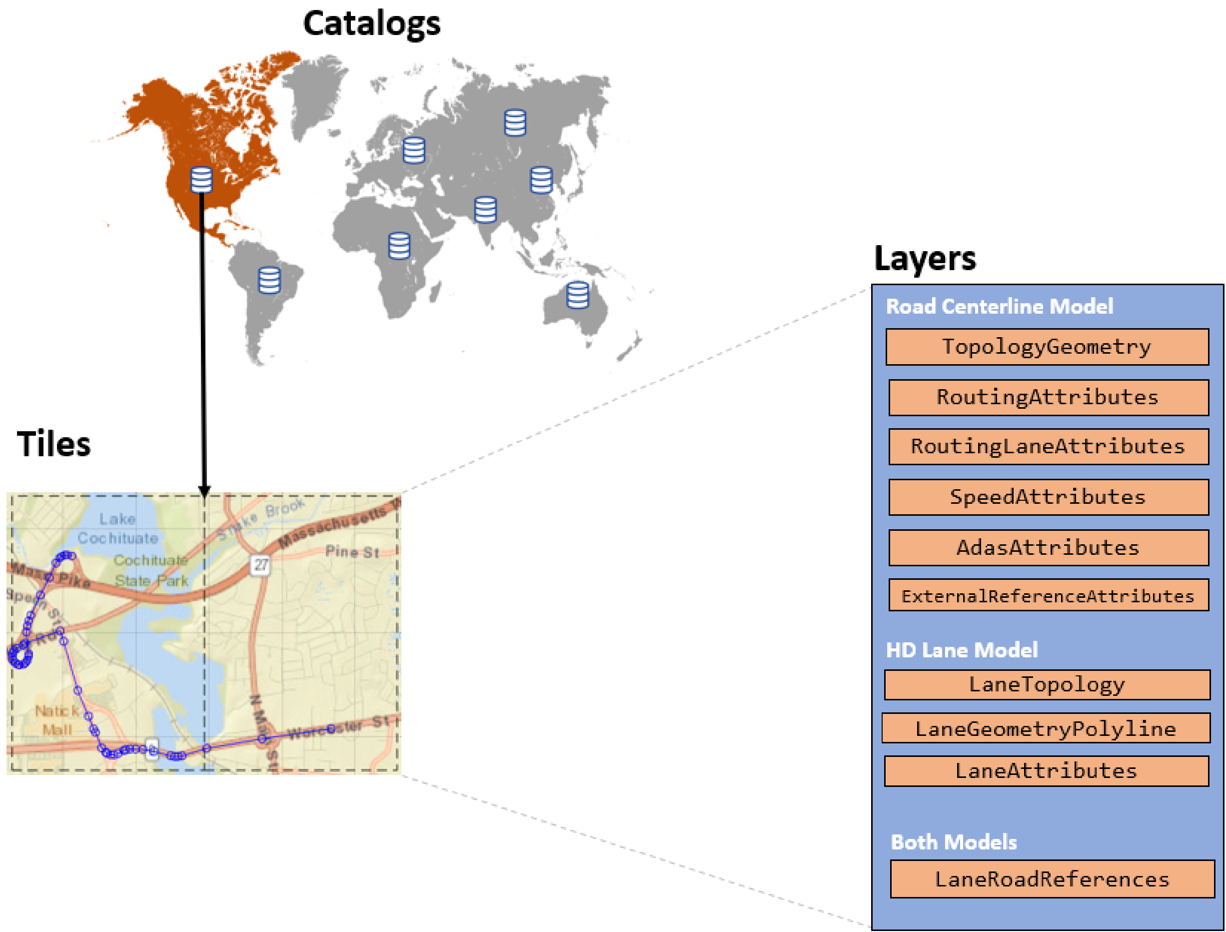
To access layer data for a selection of map tiles, use a `hereHDLMReader` object. For information on the `hereHDLMReader` workflow, see "Access HERE HD Live Map Data" on page 5-2.

The layers are grouped into these models:

- "Road Centerline Model" on page 5-35 — Provides road topology, shape geometry, and other road-level attributes
- "HD Lane Model" on page 5-37 — Contains lane topology, highly accurate geometry, and lane-level attributes
- "HD Localization Model" on page 5-39 — Includes multiple features, such as road signs, to support localization strategies

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

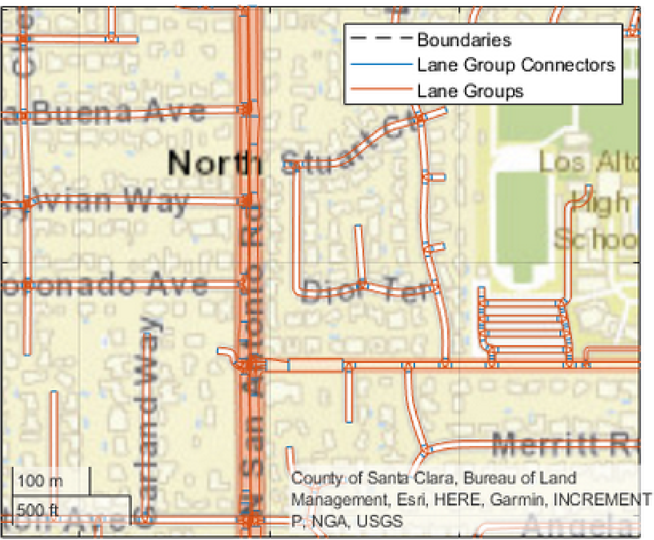
6. You need to enter into a separate agreement with HERE in order to gain access to the HDLM services and to get the required credentials (app_id and app_code) for using the HERE Service.



Road Centerline Model

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

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



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

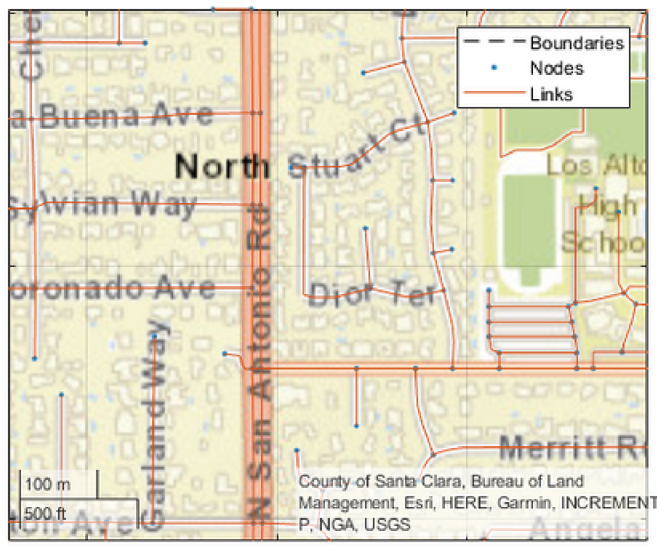
Road Centerline Model Layers	Description
`TopologyGeometry`	Topology and 2-D line geometry of the road. This layer also contains definitions of the links (streets) and nodes (intersections and dead-ends) in the map tile.
`RoutingAttributes`	Road attributes related to navigation and conditions. These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

Road Centerline Model Layers	Description
`RoutingLaneAttributes`	Core navigation lane attributes and conditions, such as the number of lanes in a road. These values are mapped parametrically to 2-D polylines along the road links.
`SpeedAttributes`	Speed-related road attributes, such as speed limits. These attributes are mapped to the 2-D polyline geometry of the topology layer.
`AdasAttributes`	Precision geometry measurements such as slope, elevation, and curvature of roads. Use this data to develop advanced driver assistance systems (ADAS).
`ExternalReferenceAttributes`	References to external links, nodes, and topologies for other HERE maps.
`LaneRoadReferences` (also part of HD Lane Model)	Road and lane group references and range information. Use this data to translate positions between the Road Centerline Model and the HD Lane Model.

HD Lane Model

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

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



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

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

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

HD Localization Model

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

See Also

hereHDLMReader | plot | read

More About

- "Access HERE HD Live Map Data" on page 5-2
- "Use HERE HD Live Map Data to Verify Lane Configurations"

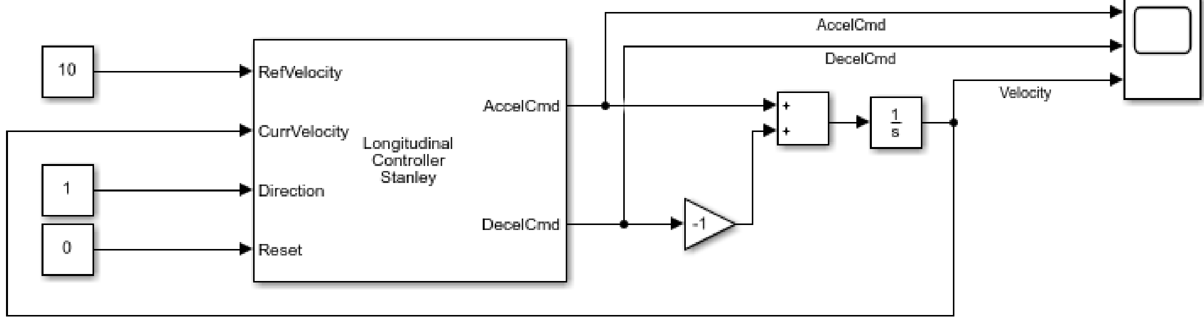
External Websites

- HD Live Map Data Specification

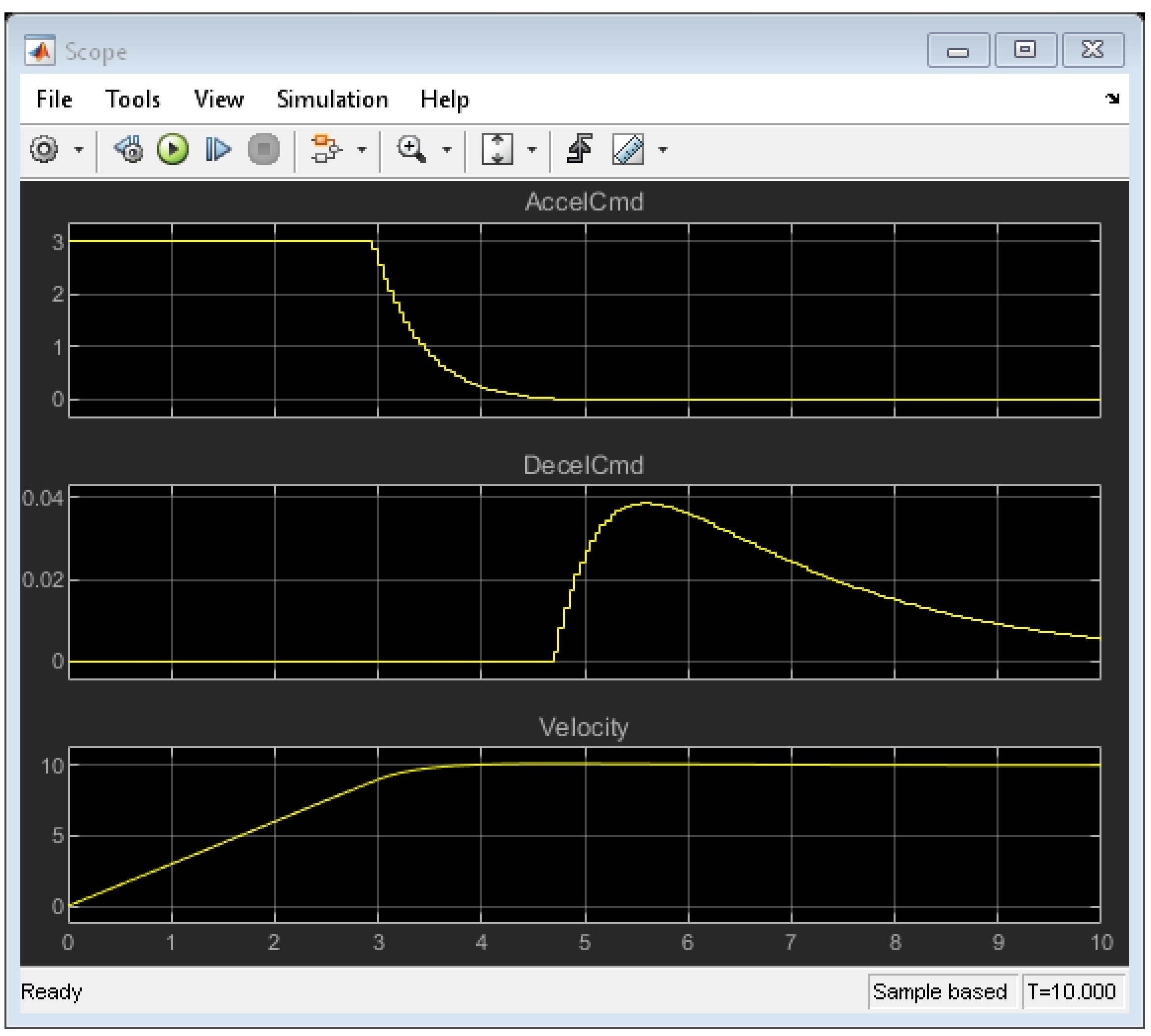
Control Vehicle Velocity

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

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



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



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

integral components of the block, using the **Proportional gain, Kp** and **Integral gain, Ki** parameters, respectively.

See Also

Lateral Controller Stanley | Longitudinal Controller Stanley

More About

- "Automated Parking Valet in Simulink"